Verifying finite-state properties of large-scale programs

by

Eric Bodden

School of Computer Science
McGill University, Montréal

June 2009

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Designers of software components can use finite-state properties to denote behavioral interface specifications which enforce client-side programming rules that state how the components ought to be used. This allows users of these components to check their client code for compliance with these rules, both statically and at runtime.

In this dissertation we explain the design and implementation of Clara, a framework for specifying and verifying finite-state properties of large-scale programs. With Clara, programmers specify finite-state properties together with runtime monitors, using a syntactic extension to the aspect-oriented programming language AspectJ. Clara then uses a sequence of three increasingly detailed static analyses to determine if the program satisfies the finite-state properties, i.e., is free of property violations.

Clara produces a list of program points at which the program may violate the properties, ranked by a confidence value. If violations are possible, Clara also instruments the program with the supplied runtime monitor, which will capture property violations when the program executes. Due to its static analyses, Clara can omit the instrumentation at program locations which the analyses proved safe, and so optimize the instrumented program. When much instrumentation remains, Clara partitions the instrumentation into subsets, so that one can distribute multiple partially instrumented program versions that each run with a low overhead.

We validated the approach by applying Clara to finite-state properties denoted in multiple formalisms over several large-scale Java programs. Clara proved that most of the programs fulfill our example properties. For most other programs, Clara could remove the monitoring overhead to below 10%. We also found multiple property violations by manually inspecting the top entries in Clara's ranked result list.

Résumé

Les concepteurs des différentes composantes logicielles peuvent utiliser les propriétés des automates finis pour fixer les spécifications de l'interface comportementale qui contrôleront les règles de programmations définissant l'utilisation des composantes. Ceci permet aux utilisateurs de ces composantes de vérifier le respect de ses règles par leurs codes sources, à la fois lors d'une analyse statique qu'à l'exécution.

Dans cette dissertation, nous montrerons la conception de Clara, une structure qui permet de spécifier et de vérifier les propriétés des automates finis dans des programmes étendus, puis expliquerons son implantation. Le programmeur, à l'aide de Clara, peut définir les propriétés des automates finis en complément aux processus de vérification à l'exécution, en utilisant une extension de la syntaxe d'AspectJ, un langage de programmation orienté aspect. Clara utilise alors, en séquence, trois analyses statiques de précision croissante pour déterminer si le programme respecte les propriétés des automates finis.

Clara produit une liste des positions dans le code source où il y a risque de violation de ces «propriétés», en ordre décroissant de certitude d'une violation. Quand cela est possible, Clara ajoute au programme des processus de vérification permettant d'étudier la violation de «propriétés» lors de son exécution. Grâce à son analyse statique, Clara n'ajoute pas au code ces processus dans les portions de code qui n'ont pas la possibilité de violer les propriétés des automates finis, ce qui limite les ralentissements dus aux processus de vérification. Lorsque ses ajouts restent considérables, Clara organise les processus de vérification à l'exécution en sous-groupe, de sorte qu'il soit possible de distribuer différentes versions du programme contenant seulement une partie de ceux-ci, limitant ainsi l'utilisation des ressources système à l'exécution.

Nous avons validé cette approche en soumettant à Clara les propriétés des automates finis sous différents modèles à appliquer sur différents programmes Java. Clara a permis de prouver que la plupart de ces programmes respectaient déjà les propriétés définies. Dans les autres cas, Clara a pu réduire le coût des processus de vérification à moins de 10%. De plus, nous avons pu localiser de nombreuses violations de propriété manuellement, en inspectant les entrées en importance dans la liste produite par Clara.

Acknowledgments

First, I would like to thank my advisor, Laurie Hendren. Laurie's support and advice was priceless. Laurie's expertise and suggestions contributed significantly to this work. Equally valuable was the advice of Patrick Lam, much beyond his time as a post-doctoral fellow at McGill. I thank Hans Vangheluwe and Clark Verbrugge for supporting my work and for being part of my Ph.D. progress committee. Much gratitude goes to Olivier Savary and Maxime Chevalier-Boisvert for translating my abstract into French.

This work was initially motivated by Volker Stolz's runtime verification of Haskell programs. My implementation extends the AspectBench Compiler, and much gratitude goes to Oege de Moor and the members of the Programming Tools Group at Oxford University for maintaining this tool. In particular, I wish to thank Julian Tibble and Pavel Avgustinov for maintaining the tracematch implementation. My work would not have been possible without their efforts. CLARA's program analyses build on top of Soot, and I am deeply grateful to my colleagues at the Sable Research Group, and all other contributors to Soot for providing this wonderful framework. I am particularly grateful to Ondřej Lhoták and Nomair Naeem, as their tracematch analyses inspired parts of the solution presented here.

My work would not have been possible without Manu Sridharan's points-to analysis. Manu generously contributed his implementation and hours of discussion. I am grateful to Grigore Roşu and Feng Chen for encouraging me to open up CLARA to support their monitoring tool. It came as a shock to all of us when we learned that Feng passed away unexpectedly in August 2009. Feng's work on JavaMOP was invaluable for parts of this thesis and will not be forgotten.

Discussions with Kevin Bierhoff greatly improved my understanding of modular static typestate analyses. David Shepherd helped me identify and classify comments in Java source code. The idea for collaborative runtime verification arose through a discussion with Benjamin Livshits at the Dagstuhl seminar on Runtime Verification. I wish to thank Klaus Havelund for introducing me to the runtime-verification community and for his constant support of my work. Jan Sinschek provided many helpful comments on a draft of this dissertation.

I am also very grateful to my defense committee: Matthew Dwyer, Jörg Kienzle, Paul Kry, Michael Rabbat, Martin Robillard and Clark Verbrugge.

A big thank you goes to Caroline, for sharing with me an unforgettable time in Montréal. Thank you for your love, encouragement and patience.

This work was supported financially by the Natural Sciences and Engineering Research Council of Canada (NSERC) through my advisor Laurie Hendren.

Dedicated to the memory of Feng Chen.

Contents

List of Figures

List of Tables

Chapter 1
Introduction and contributions

1.1 Introduction

Programmers who develop large applications in modern programming languages often face the problem that they have to obey many restrictions to guarantee the correctness of the program they are developing, even irrespective of the actual functional requirements that the program may have. For instance, it is common that programmers compose their software systems of existing third-party components, such as libraries and frameworks [GSCK04]. These components come with restrictions on how they ought to be used. Other restrictions may arise from company specific programming rules that demand, for example, that certain program parts may not interact, not share any information, or if they may interact then only at certain times in the program execution or through certain control flows. Yet other restrictions exist with respect to the programming language being used and the runtime libraries that come with it. In the programming language C, for example, one may care about de-allocating all memory that was previously allocated in order to prevent memory leaks, but at the same time one would certainly care about not accessing a part of memory that was never allocated or has already been de-allocated. In Java, the programming language that we consider here, the Java Runtime Library can be seen as a special software component that is used by every single Java program. The library comes with special restrictions on how programmers should use its application interface. For instance, a programmer should not modify an object of class `Collection` while iterating over this collection at the same time. In this case it would be unclear whether the program should iterate over the original contents or the modified contents of the collection.

Because programmers find it hard to remember all of these programming rules, and, to date, are not even been made aware of many of these rules, programming errors arise frequently during software development [Zel05]. Identifying and removing these errors can consume a large fraction of a piece of software's development cost [SHK98]. Researchers in Computer Science and Software Engineering hence develop methods that mitigate the problems mentioned above by analysing programs for potential programming mistakes. The programming language's type system can enforce some of the programming restrictions. The type system of Java, for instance, gives static error messages if a method is called with a wrong number of parameters, or with parameters that have an invalid type. For other classes of errors, like errors related to memory allocation, researchers have developed very specialized tools [GMF06].

The approach that we present in this dissertation allows programmers to check a large range of program properties, often called typestate [SY86] properties in the literature. Typestate properties are capable of checking that certain program events on a number of objects occur in a certain ordering. This comprises many of the properties that we identified as important above.

In this dissertation we explain the design and implementation of Clara (Compile-time Approximation of Runtime Analyses), a framework for verifying finite-state properties of large-scale programs. One can see Clara as an extension of a more traditional software development process that makes use of runtime verification. Figure 1.1 gives an overview of this process. At the beginning, somebody declares a program property that requires checking by defining a set of finite-state specifications. For instance the designer of a reusable component may want to define how the application interface (API) of the component ought to be used. In general, multiple different specification languages are possible. Our current implementation supports

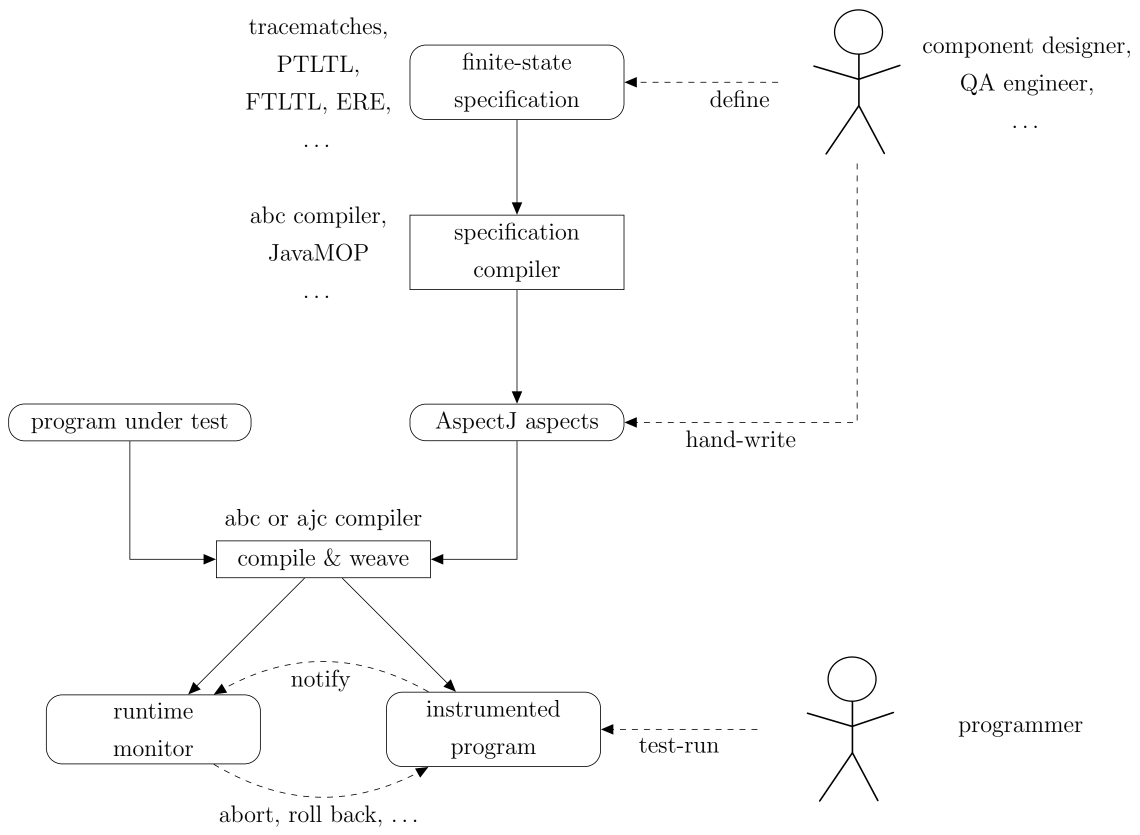


Figure 1.1: Overview of traditional runtime-verification approach (Sections 2 & 3)

```
tracematch(Iterator i) {
    sym hasNext before:
        call(* java.util.Iterator.hasNext()) && target(i);
    sym next before:
        call(* java.util.Iterator.next()) && target(i);

    next next { System.err.println("Trouble with "+i); } }
```

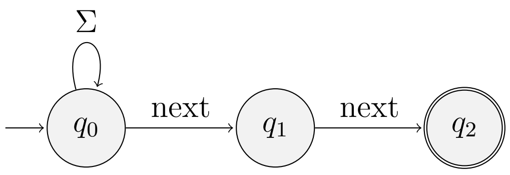


Figure 1.2: HasNext tracematch and automaton: do not call `next()` twice without an intervening call to `hasNext()`.

tracematches (a language based on regular expressions [Kle56]) through the Aspect-Bench Compiler [ACH+05a] (abc for short), as well as past-time (PTLTL) and future-time (FTLTL) linear temporal logic [Pnu77] and extended regular expressions (ERE) through the JavaMOP [CR07] specification compiler. Figure 1.2, for example, shows how a programmer could denote the "HasNext" property-specification pattern as a tracematch. This pattern states that it is an error to call the method `next()` twice on the same iterator `i` without calling `i.hasNext()` in between. In the next chapter we will explain this example in greater detail. Traditionally, tools like abc and JavaMOP compile finite-state specifications, like the one in Figure 1.2, into aspects written in the general-purpose aspect-oriented programming language AspectJ [asp03]. In special cases, the component designer may also wish to hand-write the AspectJ aspects directly, for instance if a hand-written version of the aspect would be significantly more efficient.

A programmer could then runtime-check whether her program code complies with this property by weaving the generated aspects into the program using a standard

AspectJ compiler like ajc [HH04] or abc (the compiler that we will consider in this dissertation). The weaving process results in a modified version of the original program that is instrumented with a runtime monitor. At runtime, the instrumentation in the client code notifies the runtime monitor whenever the code triggers events that are of interest to the original finite-state specification. While the exact internal workings of the runtime monitor depend on the specification formalism and specification compiler, the general evaluation scheme is always the same: the monitor essentially consists of some implementation of a finite-state machine. When the events that the client code triggers drive the monitor into a final (or accepting) state, this indicates that the current execution violated the finite-state property. In this case the runtime monitor executes some piece of error-handling code which the component designer defined along with the finite-state specification. We show the finite-state machine for the HasNext example in Figure 1.2, along with its tracematch definition. When the program under test drives this state machine into its accepting state q_2, then the tracematch will execute its action handler, printing the error message defined in line 7.

1.2 Research Motivation and Objectives

This traditional runtime-verification approach has several desirable properties. For instance, because the finite-state specifications are evaluated at runtime, the specifications can be very expressive. They can refer to runtime events, compare runtime values and can evaluate predicates over the current heap. Also, when a runtime monitor detects a property violation, it can react to this violation in many different ways. A simple monitor could issue an error message, while a more involved monitor could try to work around the effects of the detected violation or revert the program to a safe state. Another positive property is that a runtime monitor can give a safety guarantee: if a program run violates the property that the monitor describes, then the programmer has the guarantee that the monitor will detect this violation.

On the other hand, this traditional runtime-verification approach yields several drawbacks. One important drawback is that the instrumentation that is added to the program under test can yield a significant runtime overhead when test-running the program. After all, if the runtime monitor needs to monitor many events on a program run, the monitor has to consume a certain amount of execution time to update its internal state based on those events. Certain optimizations can be done and have to be done on the level of the runtime monitor itself: if the runtime monitor can compute every single state transition faster, then the instrumented program will run faster too. Avgustinov et al. [ATdM07] showed which optimizations are necessary to make runtime monitoring feasible at all. However, as we will show in our experiments, in some cases, these optimizations may not be sufficient.

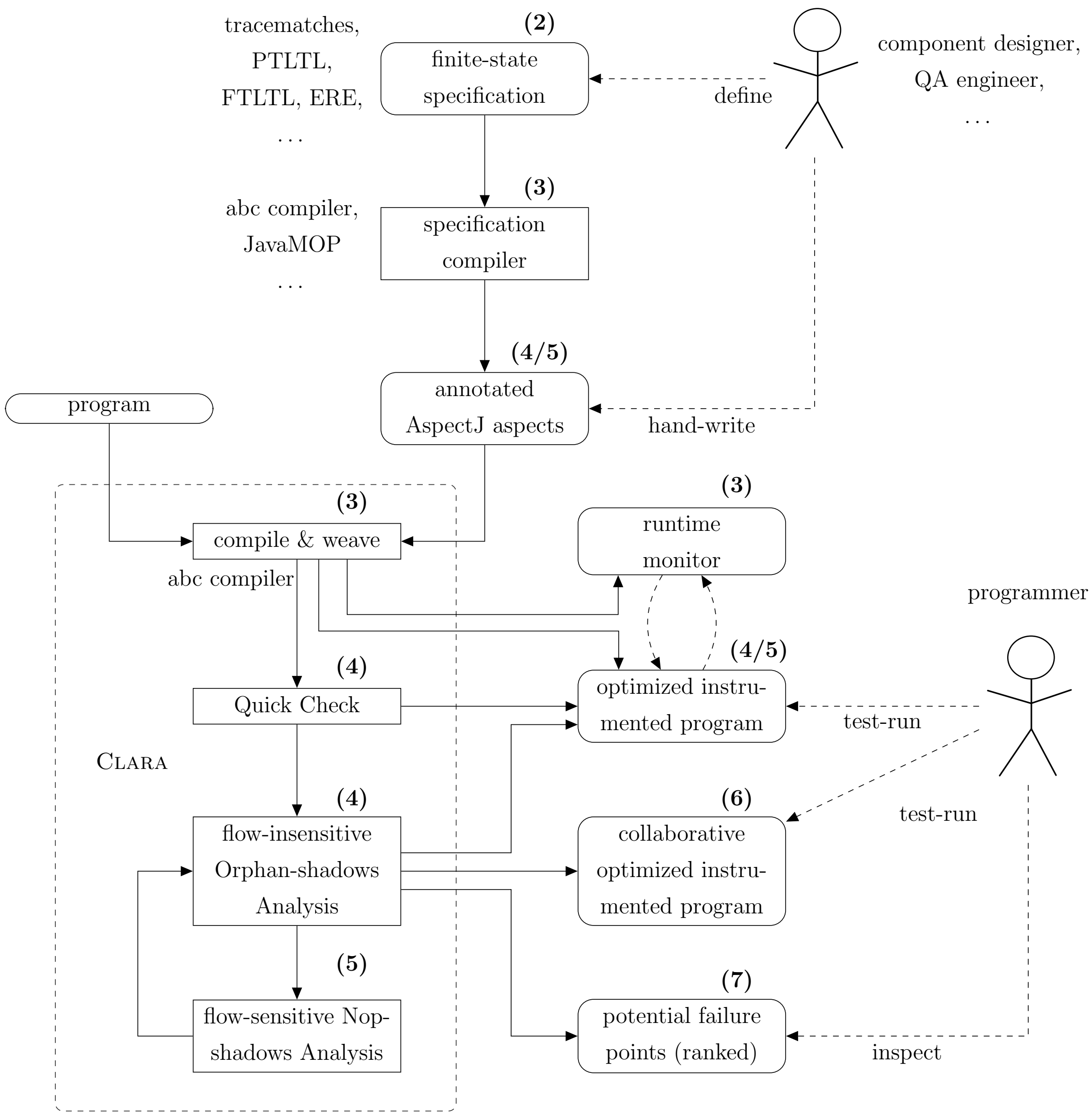
A second important drawback of runtime verification is that it gives no static guarantees. In order to detect a property violation, the programmer of the client code has to test-run the instrumented code potentially many times to achieve adequate test coverage. On the one hand this is time consuming, and on the other hand this may yield the problem that the programmer cannot say for certain when the instrumented program was tested enough. While an increasing number of different test runs can strengthen the confidence that the program will never violate the stated property, these test runs still do not constitute a proof. Therefore it would be desirable to conduct a static analysis that can prove a program safe with respect to the finite-state property already at compile-time.

1.3 Solution Overview

In this dissertation we explain the design and implementation of Clara, a framework for verifying finite-state properties of large-scale programs. The Clara framework aims at mitigating both problems mentioned above: (1) it optimizes the runtime of a program that has been instrumented with a runtime monitor, and (2) it attempts to give static guarantees about the correctness of this program with respect to the stated property. To achieve this goal, Clara conducts a set of static analyses. In comparison to the analyses proposed by Avgustinov et al. [ATdM07], these analyses do not only analyze and the runtime monitor itself, but they also analyze the complete instrumented program: Clara analyzes the program under test with respect to the stated finite-state property. If Clara can detect that a certain code location can never have an effect on the state of the runtime monitor, or is not on a control-flow path over which one can reach a final state, then Clara removes all instrumentation from this program point. This process results in an optimized program which has a reduced amount of instrumentation and will therefore trigger the runtime monitor less often than a fully instrumented program. In addition, if Clara's analyses detect that no single program point can violate the property, this means that there is no possible execution on which the program could violate the property. This is proof that the program fulfils the stated property.

Figure 1.3 gives an overview of the Clara framework and the software development process that surrounds it. Similar to the traditional runtime-verification approach, a component designer generates a set of AspectJ aspects, either by hand-writing the aspects directly or—more commonly—by defining a set of finite-state specifications which are then translated into AspectJ aspects using a specification compiler. However, the aspects in this setting are special aspects that are annotated with additional information. This information is essential to Clara when it comes to conducting its static analyses. The annotations to the aspects preserve important information about the combinations of events that may lead to a property violation and potentially also the order in which the events have to occur to violate the property. The problem is that the usual generated (or hand-written) AspectJ aspects are Turing-complete and hence it would be very hard if not impossible to reconstruct this information directly from the monitoring aspects. Using the annotations, this information is explicitly preserved and readily available to Clara.

Through its annotation languages, Clara moreover provides researchers with a unified way to specify finite-state properties of Java programs. Up until now, no such format exists, and hence every research prototype uses its own special input format. This makes it hard to compare static analyses that analyze these properties. We believe that Clara's annotation languages are general enough so that they allow researchers to express a wide variety of properties. Moreover, researchers can integrate



(i) means that the node is explained in Chapter **i** of this thesis

Figure 1.3: Overview of hybrid static/runtime verification with Clara

a wide variety of static analyses that analyze programs with respect to the properties that the annotations express.

We extended the abc compiler so that it can parse the annotated aspects. We implemented Clara as an extension to the abc compiler. Therefore, the Clara framework has access to the instrumented program, the monitoring aspects as well as any information encoded in the annotations. Based on this information, Clara conducts a set of different static analyses (details follow), all of which try to determine where, if ever, the instrumented program may violate the stated finite-state property. After its analyses finish, the programmer of the client code obtains an instrumented program, similar to the one that she obtains in the traditional runtime-verification approach. However, this time the program is optimized; it usually contains less instrumentation than a fully instrumented program would contain. In cases where the analyses in Clara have been able to prove the program safe, the resulting program is even free of instrumentation, i.e., it runs without additional runtime overhead.

In certain cases the instrumented program may show a large runtime overhead despite all static analyses. This may happen if some piece of code that does indeed violate the stated property resides in a hot loop, or if the analyses are too imprecise to determine that the code in this loop is actually safe. When this happens, the programmer can opt to have a specially instrumented program generated. This program is suitable for collaborative runtime verification: the programmer can execute the program collaboratively on multiple machines, where each machine will only observe events generated by a subset of instrumentation points. In addition, the programmer can configure these programs such that they will monitor certain instrumentation points only from time to time. This approach is suitable when it comes to testing the program in the field, and when the program is shipped to a large set of customers. Each customer will execute a slightly different program, which monitors a slightly different set of program points. In result, this monitoring approach cannot give any guarantees: due to the partial instrumentation, property violations may be missed. Nevertheless, our approach allows programmers to execute a partially instrumented program that runs with a reduced runtime overhead, but still gives the programmers some opportunity to find property violations in this program. In particular, when programmers ensure that for every single probe at least one user's program copy enables this probe, this yield the same coverage as enabling all probes in a single program copy.

During its static analyses, Clara usually proves a large set of program points safe, i.e., Clara proves that the runtime monitor cannot actually reach a final state through these points. In addition however, the Clara framework allows programmers to inspect the remaining program locations for which the analyses were unable to prove these locations safe. We call such locations potential failure points. To assist the programmer in inspecting these points manually, we devised a specialized ranking and filtering approach. This approach first divides the potential failure points into inter-related potential failure groups. The ranking algorithm then ensures that Clara lists those potential failure points first that are most likely to lead to a property violation at runtime.

1.4 Challenges

In the proposed setting we assume that we want to apply the Clara framework to Java programs of substantial size. Analysing such programs is challenging, first and foremost due to possible aliasing relationships that the programs may expose. For example, consider the finite-state property that the program under test should not write to a file handle that has already been closed. The problem is that the program could reference the same file handle under different names at the same time, e.g. the handle could be referenced simultaneously by a parameter `p` to the currently executing method, a local variable `l` and a field `f`. In order to statically analyse whether or not the program could indeed accidentally write to such a handle after it was closed, an analysis has to track these alias relationships: when the file is closed by calling `f.close()`, then the analyses must be able to infer that it is now unsafe to call `p.write(..)`.

For other properties, even must-alias information is required. Consider the example of a `URLConnection`, which needs to be initialized before it is opened. Assume that this connection is again referenced by variables `p` and `f` and that a call to `f.open(..)` is indeed preceded by a call to `p.init(..)`. In this example, the information that `f` and `p` may reference the same connection is not sufficient to determine that the program fulfils the property. We have to know that `f` and `p` must point to the same connection (on every execution) to determine that the connection indeed has been initialized when it is being opened. Such must-alias information is even more expensive to compute than may-alias information because it requires control-flow information. This poses another challenge to the analyses that Clara implements.

Sometimes, even control flow may matter when determining whether or not a program violates a given property. In the above example, when the program uses the connection before initializing it, this constitutes a property violation; when the program uses the connection after initialization, the property is not violated. Constructing control-flow information for large programs is challenging and specific abstractions are necessary to guarantee that even such large programs can be analyzed efficiently.

Last but not least, the analyses in Clara have to deal with the potential for false positives. All analyses in Clara are designed to be sound. In other words, the analyses will only rule out an instrumentation point, i.e., mark it as safe, if the analysis can prove that the program can never violate the property at this program point. In particular, the optimized instrumented program that Clara emits is behaviourally equivalent to the un-optimized instrumented program. Any sound static analysis of an undecidable property must make conservative assumptions. Our analyses make conservative assumptions about the possible control flow, or possible alias relationships. Some of these assumptions may be *overly* conservative: even if a program location is safe, the analyses in Clara may fail to prove this fact. In a setting where one conducts static verification only, such situations would lead to a potentially long list of false positives—program locations for which the analysis warns that the stated property might be violated at this point, although this is not actually the case. Long lists of potential false positives are quite a burden on the programmer, as she has to go through this list manually to determine the result of the verification outcome.

In our approach, fortunately, the programmer is given the option to defer the property evaluation to the time at which the program actually executes. Because Clara instruments the program under test with an optimized runtime monitor, the programmer can opt to just test-run the program and see whether property violations do actually occur at these potential failure points. For certain programs however, e.g. programs that use dynamic class loading and/or reflection, many conservative assumptions have to be made, and consequently a lot of program instrumentation may remain. It is a challenge to present useful information to the programmer nevertheless, and to produce an instrumented program that can run efficiently even in these cases.

1.5 Contributions

The Clara framework addresses the challenges mentioned above using the following original contributions.

1.5.1 Dependent advice and Dependency state machines

To allow Clara to discover property-specific information from the generated AspectJ aspects we designed two annotation languages called *Dependent advice* and *Dependency state machines*. A dependent advice contains dependency annotations to encode crucial knowledge about the finite-state property. Dependent Advice are meant to capture the essence of the information that is necessary to determine a program's finite-state properties at compile time without taking into account a program's control flow. Dependency state machines subsume dependent advice, but add information about the order in which events need to occur so that the given finite-state property is violated. The designers of software components may opt to annotate aspects by hand when they hand-write monitoring aspects. However, as we show in this thesis, one can easily extend existing specification compilers like abc (for tracematches) and JavaMOP (for PTLTL, FTLTL and ERE) to generate annotated dependent advice or pieces of advice annotated with dependency state machines instead of ordinary AspectJ advice.

We therefore believe that CLARA's annotation languages can be useful to many researchers: researchers can use annotated aspects to express a wide variety of finite-state properties, and in particular, these researchers can implement their static analyses by instantiating the CLARA framework. This enables researchers to compare the precision and performance of their analyses, while eliminating all other accidental variability, such as slight (but nevertheless significant) differences in the property specifications. At the current time, no such standard denotation exists, and this makes it hard to compare static analyses of finite-state properties.

1.5.2 A three-staged static program analysis to evaluate runtime monitors ahead of time

In a second step we extended the abc compiler so that it can parse these annotated aspects. We further extended abc's back-end with a three-staged analysis that evaluates ahead of time, i.e., at compile time, the runtime monitor that the annotated aspect defines (see Figure 1.3). The analysis is staged so that it will run faster. First, CLARA applies a Quick Check that uses syntactic program information only. Often, this check can immediately rule out a lot of potential failure points. When potential failure points remain, the programmer can opt to treat them further, using the flow-insensitive Orphan-shadows Analysis. This analysis uses flow-insensitive context-sensitive points-to information to determine whether a set of events that is necessary to violate the stated property can at all occur (in combination, on the same objects). When potential failure points remain even after this analysis, then the programmer can opt to apply a third analysis stage, the Nop-shadows Analysis. The analysis uses a novel notion of continuation-equivalent states to identify program points that one does not need to monitor. This analysis stage is flow-sensitive, i.e., it considers the order in which events can occur. This stage also uses must-alias information to determine when two pointers must point to the same object. In order to guarantee an efficient analysis, this third stage is intra-procedural, i.e., considers a single method at a time. The analysis models the remaining program with summary information that is based on the flow-insensitive information that the Orphan-shadows Analysis computed. As Figure 1.3 shows, Clara re-iterates the Nop-shadows Analysis and the Orphan-shadows Analysis. This is because the third stage may enable optimization potential for the second stage, and vice versa.

A key contribution of this thesis is a set of novel abstractions that allow each of the three analysis stages to perform its task. While the Quick Check and the Orphan-shadows Analysis contend themselves with flow-insensitive information, the Nop-shadows Analysis requires a more detailed abstraction that essentially models the runtime monitor, a finite-state machine, at compile time. Because the Nop-shadows Analysis is intra-procedural, it has to make conservative assumptions at procedure boundaries, which may cause imprecisions. To regain precision, the abstraction not only models for every object which state(s) this object may be in at every program point, but also which state(s) this object may certainly not be in. Both information is crucial to the approach.

1.5.3 Collaborative runtime verification

In cases where possible failure points remain even after applying all three analysis stages, the programmer still has multiple options. For instance, she can test-run the instrumented program to check whether violations actually occur at runtime. However, if much instrumentation remains even after applying the static analysis, then this may slow down these test runs by large amounts. In these cases, Clara can apply a “spatial partitioning” technique, which effectively partitions the instrumentation points into multiple subsets. One can then execute multiple instances of the instrumented program on different machines, giving each instance another command-line parameter. Depending on this parameter, the instrumented program will enable a different subset of instrumentation points. In result, each program instance on its own will only detect property violations in a specific part of the program, and therefore execute faster than a fully instrumented program would execute. A nice property of spatial partitioning is that, nevertheless, all instances in combination yield complete coverage.

1.5.4 Ranking of potential failure points

While resorting to runtime verification may be a viable option in some settings, it may often be rather desirable to get a complete guarantee about a program already at compile time. If only a few potential failure points remain after Clara's analyses, then the programmer can easily inspect these points manually. However, in cases where the analyses suffers from imprecisions, the list of potential failure points may be long, and therefore the burden on the programmer may be high. In a last stage, we therefore applied machine-learning techniques to Clara, so that the analyses can output a list of potential failure points that is ranked according to some probability value. In result, the potential failure points at which actual matches are most likely to occur are ranked to the top.

1.5.5 Summary of contributions and thesis organization

We organized the remainder of this thesis as follows. In Chapter 2 we describe a case study that reveals finite-state specification patterns that are common to the constraints that ought to be enforced on clients of the Java Runtime Library. As our study will show, one can indeed express many of these constraints with finite-state specifications. In Chapter 3 we then explain how programmers can use the traditional runtime-verification approach to monitor violations of such finite-state properties at runtime. We explain existing specification formalisms and specification compilers, and also give background information about how AspectJ compilers weave the AspectJ aspects that the specification compilers generate into the program under test.

Table 1.1 summarizes the main contributions of this dissertation. The remaining chapters address these contributions. In Chapter 4 we explain how specification compilers can preserve additional information about the given finite-state property within the generated aspects, in the form of a novel AspectJ language extension, called dependent advice. In this chapter we not only state the syntax and semantics of dependent advice but also explain two flow-insensitive optimizations that resolve advice dependencies ahead of time: the Quick Check and the Orphan-shadows Analysis. These analyses make use of the dependency information present in the annotations and a set of novel key abstractions. In Chapter 5 we extend this approach to dependency state machines. A dependency state machine subsumes a set of dependent advice, and adds information about the order in which events must occur to cause a property violation. This enables a flow-sensitive analysis stage that we explain in the same chapter. We explain our code-generation scheme for collaborative runtime verification in Chapter 6, and our ranking scheme to aid the manual inspection of remaining potential failure points in Chapter 7. We will discuss related experiments and their results directly at the end of each chapter. In Chapter 8 we discuss related work, and we conclude in Chapter 9. The appendices contain some lengthy proofs and complexity estimates.

Contribution	Chapter
Dependent advice (language definition and implementation)	4
Dependency state machines (language definition and implementation)	5
Three-staged static program analysis	4, 5
Spatial partitioning	6
Ranking of potential failure points	7

Table 1.1: Main contributions of this dissertation

Chapter 2
Safety properties in large-scale Java programs

As we mentioned in the introduction, the CLARA framework is an approach to verifying interesting properties of Java and AspectJ programs through a combination of compile-time and runtime techniques. In this chapter we discuss a wide range of these properties. We start by commenting on a study conducted by Dwyer et al. [DAC99], which surveyed a large set of finite-state property specifications from the scientific literature and other sources. We then present a study that we conducted ourselves. The goal of this study was to distill a list of rules that programmers need to adhere to when they use the Java Runtime Library. The Java Runtime Library is certainly the most widely used Java application interface and therefore its usage contracts deserve special attention, as they concern many people and programs. This chapter will make the reader familiar with the kinds of properties that may require static or dynamic checking in Java programs by giving a large set of examples, many of which we will use again in later parts of this dissertation.

2.1 Property specifications for finite-state verification

In 1999, Dwyer et al. conducted a study [DAC99] of 555 property specifications for finite-state verification tools. To the best of our knowledge, this study is the most substantial of its kind. The patterns were drawn from 35 different sources, mostly scientific papers, but also from the developers of verification tools and from student projects.

The authors could show that, in their data set, the vast majority of specifications (about 92%) are instances of certain specification *patterns*. Table 2.1 summarizes these patterns. As the authors discovered, the patterns Response, Universality and Absence are by far the most common ones. In their study, about 80% of all the specifications were an instance of one of these three patterns.

The authors also distinguished different *scopes*. A given property (i.e., an instance of a property specification pattern) may be required to hold globally (i.e., on the entire program execution), between two events, or after, until or before another event. As the study revealed, more than 80% of all the pattern instances used a global scope.

These results are encouraging in the sense that, despite the fact that finite-state specifications have the potential for being quite complex, they will usually be relatively simple. Program analyses like ours can exploit this fact. Indeed, as we show later in this work, our analyses make some assumptions that do not always hold but do hold in *most* cases. In these common cases the analysis works well, while in other less frequent cases the analysis may not be as effective.

The study by Dwyer et al. is important because it reveals a set of common specification patterns. However, with respect to the work presented in this dissertation, it was unclear whether the authors' results would also hold up in the Java-based setting that we consider here. After all, the authors had drawn their example specifications mostly from the verification literature, and from people who had produced this literature. Hence it seems legitimate to ask whether or not the same kinds of specifications are actually of concern to programmers of Java applications. In addition, Dwyer et al. drew their specifications from literature that only considered finite-state specifications to begin with. An interesting question, that we will answer, is therefore whether or not programmers may be interested in specifying properties that go beyond what one can express with a finite-state specification formalism.

Absence	A given state/event does not occur within a scope.
Existence	A given state/event must occur within a scope.
Bounded Existence	A given state/event must occur k times within a scope. Variants of this pattern specify at least k occurrences and at most k occurrences of a state/event.
Universality	A given state/event occurs throughout a scope.
Precedence	A state/event P must always be preceded by a state/event Q within a scope.
Response	A state/event P must always be followed by a state/event Q within a scope.
Chain Precedence	A sequence of states/events $P_1, \ldots, P_n$ must always be preceded by a sequence of states/events $Q_1, \ldots, Q_m$. This pattern is a generalization of the Precedence pattern.
Chain Response	A sequence of states/events $P_1, \ldots, P_n$ must always be followed by a sequence of states/events $Q_1, \ldots, Q_m$. This pattern is a generalization of the Response pattern. It can be used to express bounded FIFO relationships.

Table 2.1: Common patterns in finite-state property specifications; from [DAC99]

2.2 Property specifications in Java programs

We therefore decided to conduct an informal survey on our own, with the goal to determine common safety properties in Java programs, be they finite-state or not. Like Dwyer et al., we sought to identify common characteristics of such properties that would indicate some form of possible specification reuse.

2.2.1 Setup of the survey

As mentioned, we focused our study on the Java Runtime Library (JRL). We first searched the library for locations at which certain exceptions may be thrown. Programmers should avoid program executions that can cause these exceptions. For instance, if the JRL throws an `IllegalStateException`, this usually indicates that the client program has violated a usage contract of the JRL. In addition, we used a simple tool to extract JavaDoc API documentation from the JRL's source code. We then browsed through the extracted documentation, searching for key phrases like *call*, *initialize*, *access* or *do not*. Last but not least we uncovered a set of concurrency-related properties from the excellent book Java Concurrency in Practice [PGB$^+$05]. This book contains many descriptions of common pitfalls with respect to concurrency in Java. As it turns out, one can express most of these pitfalls as finite-state properties.

2.2.2 Property specifications in the Java Runtime Library

When we inspected the Java Runtime Library, we found a large set of properties that could benefit from static and/or dynamic verification. Most of the properties, but not all of them, relate to collection classes and I/O streams that the library provides. Because virtually every Java program uses the Java Runtime Library, a verification tool that could check these properties would benefit many software developers. In the following we describe the different specification candidates that we found in the Java Runtime Library.

Asynchronous access to synchronized collections. In Java, a programmer can synchronize access to a collection *c* by wrapping *c* into a wrapper object that delegates to the wrapped collection *c* using synchronized methods only. To do so, the programmer invokes the method `Collections.synchronizedCollection(c)`. Performing such a method call is a clear signal that the programmer intends the access to *c* to be synchronized, so that multiple concurrently executing threads can access *c* without causing race conditions [BH08]. Therefore, after performing this method call, the programmer should not access the original collection *c* directly. (In [PGB$^+$05] this is called the "set and forget" policy.) Using the JRL, programmers can both synchronize collections and maps. In the following we will refer to the property mentioned above by the name "LeakingSync". The LeakingSync property describes that one may not access a collection or map any more that was passed to any of the `Collections.synchronized*` methods.

Even when a programmer has synchronized a collection by wrapping it into a synchronized wrapper object, in certain situations it may still not be thread-safe to access this collection through multiple threads. This is because one cannot only inspect and modify the contents of the collection via the collection's own (and now synchronized) methods, but also via methods of any iterator associated with this collection. Although access to the object returned by `synchronizedCollection` is synchronized, access to the iterators that it produces is not. Hence, comments in the JRL state that the programmer should explicitly synchronize on the synchronized collection (using a synchronized block) while iterating over the collection. Similarly, when the programmer produces a synchronized map, she has to synchronize on the map while iterating over the map or any of its set representations like its key set or value set, as these sets only provide views on the map (more on views below). In the following we will refer to these properties by the names "ASyncIterC" (for collections) and "ASyncIterM" (for maps).

A particularly subtle case is the one where the programmer calls a collection-traversing method like `c.containsAll(d)`. Even when both *c* and *d* are synchronized collections, this access may not be thread safe, for the following reason: the method `containsAll` does synchronize on its wrapper object, i.e., on *c*, however it fails to synchronize on the argument object *d*. Therefore, another thread could modify *d* while the current thread computes the expression `c.containsAll(d)`. (After all, it can obtain *d*'s lock.) While programmers may consider this a bug in the implementation of the JRL, Sun developers argue that the programmer calling this method should be able to guess that the method iterates over the argument collection and hence it is the responsibility of the programmer to wrap the method call into an appropriate synchronized block[1]. If a programmer happens to forget the synchronization, this may lead to a subtle data race [Sen08]. The property "ASyncContainsAll" expresses that a thread must own the lock of the argument collection when invoking `c.containsAll(d)` on synchronized collections `c` and `d`.

Interestingly, a runtime-monitoring approach as we present it can recover from such errors once the errors have been detected. In the case mentioned above, when the monitor detects a method call `c.containsAll(d)`, then the monitor can (1) acquire `d`'s lock, (2) proceed with the original method call, and (3) release the lock again after the original method call completed.

Fail-safe iterators. Even in a single-threaded program, the use of iterators can lead to programming errors. For example, a programmer could accidentally modify a collection *c* while at the same time iterating over *c* using an iterator. By definition of the `Iterator` interface in the JRL, this is forbidden because it may leave the iterator in an undefined state. After all, should the iterator be iterating over the modified or unmodified collection? Iterators provided by the JRL therefore have fast-fail semantics, which means that when such a situation occurs, the next call to the `next()` method of the iterator that follows the modification of the underlying collection should throw a `ConcurrentModificationException`[2]. To throw this exception at the right point in time, every single implementation of the `Iterator` interface in the JRL contains special monitoring code that keeps track of the status of the underlying collection.

[1] see `http://bugs.sun.com/view_bug.do?bug_id=4505651`

[2] The name of this exception is somewhat misleading. The error has nothing to do with concurrent execution; it can well occur in single-threaded programs.

It seems tedious and error-prone to implement a property that way that is actually common to all iterators. As we will show, programmers can express this property, and have the property checked, much more elegantly with a finite-state specification. The older collection classes in the JRL, like `Vector`, produce iterators of type `Enumeration`, which do not have the fail-fast semantics. Indeed, with such enumerations, a property violation will go unnoticed if no runtime monitor is present. In the following, we will refer to the property mentioned above as "FailSafeIter". We also define a related property over maps: a programmer may not modify a map while iterating over the map's key set or values. We refer to this property by the name "FailSafeIterMap". The properties "FailSafeEnum" and "FailSafeEnumHT" are similar, but define conditions on vectors and enumerations, respectively hash tables and enumerations instead of collections, iterators and hash maps.

Removing elements through an iterator. Programmers can easily break the above programming rule when trying to filter a collection according to a given predicate: it seems natural to iterate over the collection, test each element to see if the predicate matches the element, and when it does, remove the element from the collection. Because this idiom is so common, the `Iterator` interface provides a `remove()` method that allows the programmer to remove the current element from the collection safely, i.e., without an exception being thrown. However, the `remove()` method comes with its own usage constraints: it may only be called after the iterator was advanced using `next()` at least once. Also, after `remove()` was called, `remove()` may not be called again until the iterator was advanced again.

Views on collections make the problem even worse. One other aspect that makes the problem of an accidental concurrent modifications of collections in Java even worse is the fact that several methods provided by the JRL return collections that are views on other maps or collections. For instance, the method call `d = Collections.unmodifiableCollection(c)` returns a wrapper object d that cannot be modified (all methods that would normally modify the state of this collection throw an `UnsupportedOperationException` instead). However, a programmer can still modify the argument collection *c* itself, and by doing so could accidentally raise a `ConcurrentModificationException` if the program happens to be iterating over *d* at the same time. The same is true for key sets and value sets of maps: the map may be modified while iterating over one of the sets, and one of the sets may be modified (hence altering the map) while iterating over the map. A particularly interesting case is the method `List.subList(int,int)`[3]. This method returns a view on a sub-list of the receiver of the call. While it is unlikely that a programmer would wrap an unmodifiable collection in another one, it is possible (and probable) that a programmer may call `subList` again on a list that is already a sub-list of some other list. This would lead to a chain of views of arbitrary length, making concurrent modifications yet more likely.

Iterating past the end of a collection. Another possible mistake with respect to iterators is that a programmer could iterate past the end of a collection. Therefore, a programmer should virtually always check whether `i.hasNext()` holds before advancing an iterator *i*. The same holds for enumerations as well. There are notable exceptions to this rule, however. In general, a programmer can determine the number of elements in a collection first, and then make sure by other means than by checking `hasNext()` that the iterator is only advanced as many times as the collection has elements. This is a particularly common idiom in cases where programmers want to access an element contained in a set. The `Set` interface provided by the JRL allows programmers to compare, join and intersect sets, but it provides no access to actually retrieve the elements of a set. Hence, the only way to access an element of a set is to create an iterator for this set and then retrieve the element via a call `i.next()` to the iterator. To check whether this call is valid, programmers frequently check the set for non-emptiness rather than calling `i.hasNext()`. Nevertheless, the property is an interesting one in general. In the following we will refer to the property by the names “HasNext” (for iterators) and “HasNextElem” (for enumerations).

[3]Thanks to Kevin Bierhoff for pointing out this example.

Streams, readers and writers. The Java Runtime Library contains many definitions of streams, readers and writers that programmers can connect in various combinations. For instance, an `InputStreamReader` can read inputs from an `InputStream`. The reader is connected to the stream when it is instantiated: the programmer passes the stream as an argument to the constructor call that initializes the reader. One can also connect readers to other readers in the same way, or streams to other streams. Conversely, one can connect writers to output streams or other writers. In general, in any such chain of connected readers, writers and/or streams, it is an error to close any of the involved objects (by calling the `close()` method) and then call any of the `read(..)`, respectively `write(..)`, methods on the same or any other object in the same chain. This is because no reader can read from a closed reader or stream, etc. Even more subtly, if a reader is closed, then all wrapped readers and input streams are (transitively) closed as well, and hence should not be accessed any more. In consequence, a programmer should be very careful about connecting one stream to multiple readers: if one of the readers is closed, the stream becomes inaccessible and the other reader(s) hence should not be accessed any more either. (The same holds for writers and output streams.) In the following we will refer to these properties by the names "Reader" for readers and "Writer" for writers respectively.

Again, there are subtle exceptions from this rule, some of which benefit the programmer, others of which are rather disconcerting. Closing a `ByteArrayInputStream` or `ByteArrayOutputStream` has no effect at all. In fact, the implementation of the `close()` methods is empty. This is because these classes access an in-memory byte array only and perform no actual input/output operations. There is no semantics to closing an array and hence, the methods perform no operation. So in this case, the programmer does not need to worry about accidentally closing one of these streams. The disconcerting case is the implementation of `PrintWriter`, as this implementation in fact violates a contract inherited from the `Writer` class that `PrintWriter` extends. To cite from the JavaDoc documentation[4] of the `close()` method in the class `Writer`: "Once the stream has been closed, further write() or flush() invocations will cause

[4]see `http://java.sun.com/j2se/1.4.2/docs/api/java/io/Writer.html`

an IOException to be thrown." The class `PrintWriter` violates this contract, as it states in its own JavaDoc documentation: "Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`." To us it is unclear as to why the developers of the JRL opted for this decision, but we assume that the decision was made due to the special buffering semantics of the `PrintWriter`. The same decision was made for the class `PrintStream`. The field `System.out` usually references an object of class `PrintStream`. This causes the problem that in the following program our call for help would go completely unnoticed. This program prints nothing and throws no exception:

```
public static void main(String[] args) {
    System.out.close ();
    System.out.println("Help!");
}
```

URL connections. Another class in the JRL that comes with subtle usage contracts is the class `URLConnection`. Programmers have to go through a four-step initialization process to establish a connection using this class. To cite from the JavaDoc documentation:

1. The connection object is created by invoking the `openConnection` method on a URL.
2. The setup parameters and general request properties are manipulated.
3. The actual connection to the remote object is made, using the `connect` method.
4. The remote object becomes available. The header fields and the contents of the remote object can be accessed.

For each of these four steps, there exist methods that may be called in this step and in this step only. Fortunately, the documentation contains complete lists of these methods. Nevertheless, programmers who are unfamiliar with this class and fail to consult the documentation may easily make programming errors.

2.2.3 Properties that Clara cannot express

While in our survey we did not actually find any properties that would go beyond the capabilities of finite-state specifications, we are well aware of some specification patterns from the literature that one cannot express with finite-state machines, or that go beyond the capabilities of Clara and the runtime-monitoring tools that we consider in this thesis for other reasons.

One limitation of finite-state specification formalisms, as we consider them in this thesis, is that they allow developers only to specify joint properties of a fixed number of objects. In some cases, this may be too restrictive. The runtime verification tool PQL [MLL05] for example uses a recursive query language that allows developers to reason about a combination of an unbounded number of objects. Using such queries, developers can for example specify a "tainted string" property: no string entered into the system by a user or produced through concatenation with such a string may be passed to a certain sensitive method. Because a program can produce an unbounded number of strings from a string that a user enters, there can be no bound on the number of objects that one needs to examine to check this property. This is different from the finite-state properties that we consider in this thesis. The finite-state runtime-verification tools allow developers to reason about a bounded number of objects, for instance a collection `c` that may not be modified while an iterator `i` is used over `c`, but using these tools it is not possible to express specifications like the "tainted string" example.

Nevertheless, one can use modern finite-state runtime-verification tools to specify some specification patterns that go beyond the expressiveness of regular languages. For instance, the PQL paper [MLL05] also mentions an example of matching method entry and exit: one may want to check that a resource that was acquired during the execution of a method `m` is released before leaving `m`. In order to get the right behavior in case `m` calls itself recursively one needs to match the entry into `m` with the correct exit out of `m`. AspectJ-based runtime-verification systems can often express such properties by using special pointcuts. A built-in pointcut `cflowdepth(i)` holds for a number `i` if and only if the depth of the current control flow is `i`. Using such special pointcuts, developers can specify properties that require the matching of method entry and exit despite the usual limitations of regular languages.

A similar problem arises with the `subList` property that we mentioned earlier. Programmers can create an arbitrarily long chain of sub-lists using repeated calls to `subList` on objects that a `subList` call itself returns. Typical finite-state runtime monitors like the ones generated from tracematches or JavaMOP cannot capture constraints over this unbounded number of sub-lists in combination because they provide programmers with an arbitrarily large but nevertheless fixed number of variables. Hence, programmers could express the property for up to n sub-lists for any fixed n, but they cannot write a single monitor specification that captures the property for every such n.

Chapter 3
Runtime Monitoring through history-based aspects

In the last section we discussed a wide range of program properties that may benefit from automated checking in Java programs. As we argued, one can express most of these properties as finite-state properties. Finite-state properties, in turn, can be denoted in a variety of different finite-state specification languages. In the following, we explain the syntax and semantics of four such languages on which we focus our attention in this thesis. The four specification languages are:

1. tracematches (a special form of regular expressions),
2. past-time linear temporal logic (PTLTL),
3. future-time linear temporal logic (FTLTL) and
4. extended regular expressions (ERE).

Tracematches [AAC+05] are provided as an extension `abc.tm` to the AspectBench Compiler [ACH+05a], while the specification compiler JavaMOP [CR07] provides the other three specification formalisms. The reader may ask why we support tracematches at all, because, after all, we already support a regular-expression-based syntax through JavaMOP. One reason is that when we developed Clara we first implemented Clara as an extension to the tracematch implementation and only later on generalized it to support other specification languages as well. The second reason is that although tracematches and the ERE language supported by JavaMOP have a similar syntax, their semantics show important differences. We will discuss these differences later in this chapter.

As we show in Figure 3.1, for each of the four specification languages there exists a specification compiler that generates a runtime monitor in the form of an AspectJ aspect (and some support classes). Because tracematches are implemented as an extension to the AspectBench Compiler, this allows the tracematch implementation to pass the generated aspects and classes that implement the runtime monitor to the matching and weaving back-end in the form of an abstract syntax tree (AST) and some auxiliary information that is also kept in memory.

The runtime-verification tool JavaMOP [CR07] implements the other three specification formalisms that we mentioned. JavaMOP is an open runtime-verification framework that—by design—allows multiple specification languages to co-exist. One design goal of JavaMOP is to make the framework independent of the actual AspectJ compiler that inserts the monitoring aspects into the program under test. Therefore, JavaMOP generates the monitoring aspects and support classes in the form of `.java` text files. In theory, programmers could then weave these files into the program under test using any AspectJ compiler. However, in our setting we focus on the case shown in Figure 3.1, where programmers use the AspectBench Compiler to perform the weaving.

It is worth emphasizing that the aspects that both JavaMOP and the tracematch extension to abc generate are *history-based*. A history-based aspect contains pieces of advice that execute conditionally, based on the observed execution history. The notion of history-based aspects will be important in later chapters of this thesis, as all the analyses and optimizations that Clara supports can be applied to history-based aspects in general, no matter where they originate from.

We structured the remainder of this chapter as follows. In the next section we explain the syntax and semantics of tracematches, and discuss important implementation details. In Section 3.2 we then discuss the syntax, semantics and implementation

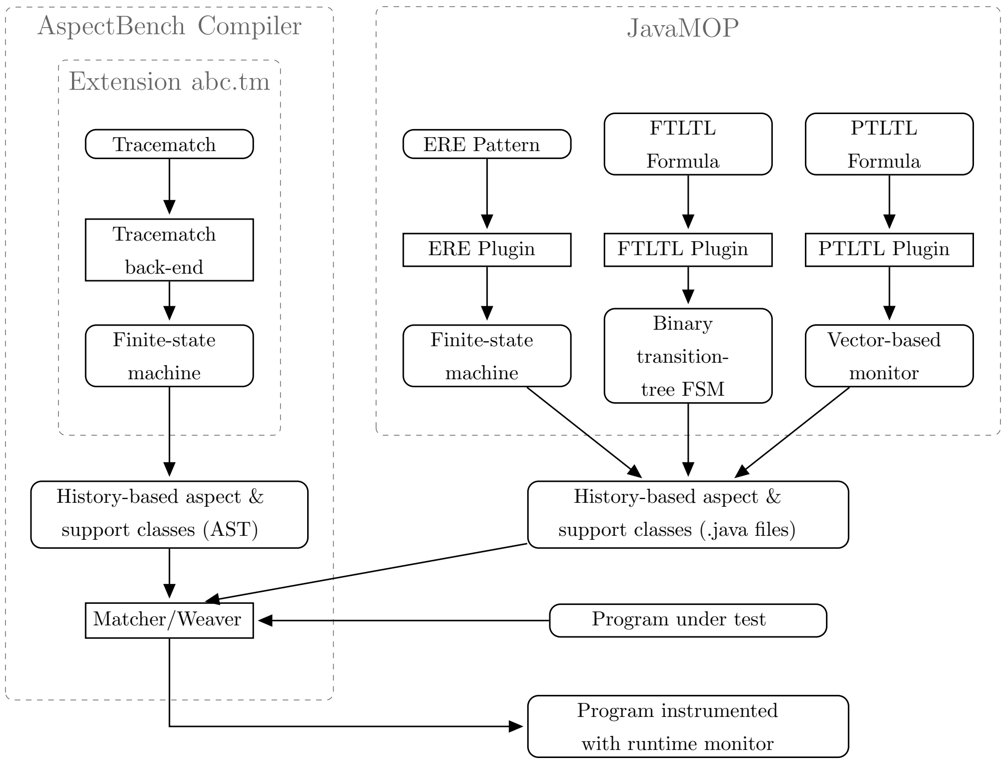


Figure 3.1: Overview of code-generation and weaving

of the three finite-state specification languages supported by JavaMOP. Section 3.3 discusses why developers may prefer to use multiple specification languages at all, opposed to using a single formalism only. Note that this section focuses on the runtime checking of these specifications—we will explain our static analyses in Chapters 4 and 5. We conclude the chapter in Section 3.4, where we motivate the need for Clara's static analyses through experiments.

3.1 Tracematches

Tracematches [AAC+05] allow programmers to define a finite-state specification via a regular expression. The tracematch runtime then matches this regular expression against each suffix of the execution trace. Figure 3.2 presents an example program that we would like to partially verify using tracematches. The program populates a collection, which the program then passes to the method `print` for printing. We explicitly added copy statements at lines 4 and 12 to emphasize the problem of aliasing. While the introduction of aliases through such copy statements presents no problem to runtime monitoring, in later chapters of this thesis it will become clear that aliasing presents a major challenge to static analyses. Like many Java programs, our example uses iterators and collections. As we showed earlier, these objects come with implicit contracts defining how they ought to be used. Tracematches can verify such contracts.

3.1.1 HasNext example tracematch

In Figure 1.2 (page 4) we already showed the `HasNext` verification tracematch, along with the automaton that the AspectBench Compiler uses to evaluate the tracematch at runtime. (Details about this evaluation will follow in Section 3.1.3.) This tracematch identifies suspicious traces where a program calls `i.next()` twice in a row without any intervening call to `i.hasNext()`.

Like all finite-state specifications that we consider in this thesis, a tracematch definition contains (1) an alphabet of *symbols*, (2) a *specification pattern or formula*

```
void main() {
    Collection c1 = new LinkedList();
    c1.add("something");                 //update(c1)
    Collection c2 = c1;
    c2.add("somethingElse");             //update(c2)
    print(c2);
}

void print(Collection c3) {
    Iterator i1 = c3.iterator();         //create(c3,i1)
    while(i1.hasNext()) {                //hasNext(i1)
        Iterator i2 = i1;
        System.out.println(i2.next());   //next(i2)
    }
}
```

Figure 3.2: Example program with shadows

over this alphabet and a (3) *body* of code. In the case of tracematches, the specification pattern has the form of a regular expression.

Symbols associate abstract tracematch events with concrete program events. Developers define symbols using *pointcuts* in the aspect-oriented programming language AspectJ. In this dissertation we use some terminology from the field of aspect-oriented programming. In this terminology, a program's execution consists of a sequence of *joinpoints*, events in the program execution that are exposed to aspects and which aspects can therefore intercept. Aspects intercept joinpoints by describing sets of joinpoints via pointcuts. A pointcut is a predicate that may (1) match or not match a given joinpoint, and when it does match a joinpoint, then it may (2) expose information to the aspect about the execution context in which the joinpoint occurred. In the example from Figure 1.2, the pointcut in line 3 matches on calls to the method `hasNext()` defined on the `Iterator` interface (with any return type, denoted by "`*`"). At the same time, the pointcut exposes the `target` object of the call by binding it to a variable `i` that the programmer declared in line 1. Most examples in this thesis use `call` pointcuts. In principle however, programmers could use any AspectJ pointcut in their tracematch symbols. These other pointcuts can for instance match on field reads and writes, exception handling, object initialization, etcetera. Apart from the `target` object of a call, pointcuts can also expose the currently executing object (`this`) and all argument objects to a call (`args`). There is also a means to expose a value returned from a method call using a special "`after returning`" clause, or an exception thrown by a method call, using an "`after throwing`" clause.

The fact that pointcuts, and therefore tracematch symbols, may bind objects is important to tracematches. In our example, line 1 of the tracematch declares that symbols in the `HasNext` tracematch may bind an `Iterator i`. In general, tracematches may also bind primitive values of type `int`, `float`, etc.. However, most examples considered in this thesis will only reason about objects. Lines 2–5 define symbols `hasNext` and `next`, which capture calls to the `hasNext()` and `next()` methods of `i`. These two symbols establish the alphabet for the tracematch's regular expression "`next next`" at line 7. Any occurrence of the `hasNext` symbol on a iterator `i` discards partial matches for `i`. Line 7 also holds the body of code to be executed every time the regular expression matches. In this work, we focus on tracematches for program verification. Our tracematch bodies report errors, but could instead contain error-recovery code.

In the following we distinguish the concrete program trace, which consists of all AspectJ joinpoints (including method calls, field assignments, etc.), from the abstract event sequence, as seen by tracematches. The abstract sequence consists only of symbol names and, as we will see later, variable bindings. This sequence therefore abstracts from AspectJ's concrete joinpoint model. The tracematch runtime matches the regular expression against each *suffix* of this abstract (symbol-based) execution trace. For instance, symbols map the concrete call sequence

```
hasNext() next() next() next()
```

to an event sequence

```
hasNext next next next,
```

which the regular expression "`next next`" matches twice, executing the body at the second and third `next` events. abc implements these suffix-matching semantics by augmenting the initial state(s) of the tracematch automaton with a Σ-loop, as shown in Figure 1.2, where Σ represents the entire alphabet, i.e., set of symbols, of the tracematch[1].

One feature that tracematches have in common with the specification languages that JavaMOP supports is that they require consistent variable bindings for a pattern to match. Our example therefore would only match if two calls to `next` occur on the same iterator. (Variable bindings for variables of a primitive type are compared via equality.) Hence, with iterators `i1` and `i2`, the call sequence we considered earlier could actually be

```
i1.hasNext() i2.next() i1.next() i2.next(),
```

giving an abstract event sequence of

[1]In the tracematch literature [AAC+05, ATdM07] this loop is often omitted and only implied, however its suits our comparison to other finite-state formalisms better to render this loop explicitly.

`hasNext(i=`o`(i1)) next(i=`o`(i2)) next(i=`o`(i1)) next(i=`o`(i2))`.

Here and in the remainder of this dissertation, the notation o(`x`) denotes the object referenced by program variable `x`. Conceptually, tracematches project the event sequence onto distinct sub-sequences separated by variable bindings. Our example sequence contains two projections: (1) "`hasNext next`" for `i=`o(`i1`), and (2) "`next next`" for `i=`o(`i2`). The regular expression does not match projection (1), but it does match projection (2), and the runtime would therefore execute the tracematch body only once, at the last call to `next()`, with the binding `i=`o(`i2`).

3.1.2 FailSafeIter example tracematch

Tracematches and the specification languages supported by JavaMOP differ from many previous approaches in that they enable developers to bind *multiple* variables[2]. We demonstrate this feature with the `FailSafeIter` tracematch in Figure 3.3: this tracematch uses multiple variables. The tracematch reports cases where the program modifies a `Collection c` while an `Iterator i` is active on `c`. The figure also shows the corresponding automaton. Statically evaluating properties that reason about multiple objects is particularly challenging, and we will return to this particular example in the following sections.

3.1.3 Tracematch implementation

In our approach, the programmer expresses verification properties using tracematches, and then feeds the tracematches and the program under test to the AspectBench Compiler (see Figure 3.1, page 31). In the tracematch implementation provided by Allan et al. [AAC+05], the extension `abc.tm` to abc first creates an automaton from the regular expression of each tracematch. Figure 1.2 (page 4) presents the automaton for `HasNext` below the tracematch definition. Based on this aspect, the tracematch

[2]Moreover, not all symbols need to bind all variables; the only requirement is that for each *complete* pattern match and each tracematch variable v there must be some matched symbol that binds v. This guarantees that when the tracematch body executes, for every declared variable v there will have been some observed event that bound v to a value.

```
pointcut collection_update(Collection c):
( call(* java. util . Collection+.add*(..)) || ... ||
  call(* java. util . Collection+.remove*(..)) ) && target(c);

tracematch(Collection c, Iterator i) {
    sym create after returning(i):
      call(* java. util . Collection+.iterator ()) && target(c);
    sym next before:
      call(* java. util . Iterator+.next()) && target(i);
    sym update after: collection_update(c);

    create next* update+ next { ... } }
```

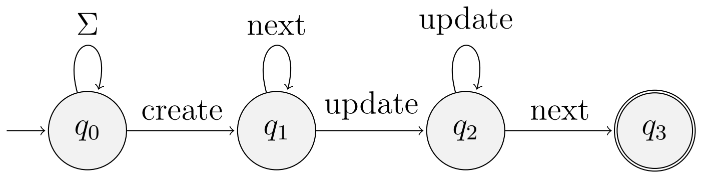


Figure 3.3: FailSafeIter tracematch and automaton: detect updates to a Collection which is being iterated over.

implementation generates (1) a history-based AspectJ aspect that modifies the program under test to issue the events of interest that are mentioned in the tracematch definition, and (2) a set of support classes which implement the actual runtime monitor that consumes these program events. In the next section we explain the internals of the generated AspectJ aspects. We explain important implementation details of the runtime monitoring code in Section 3.1.5.

The abc extension also emits an implementation of this automaton in the form of multiple Java classes. Secondly, the extension generates an AspectJ aspect whose contents are directly based on the symbol definitions contained in the tracematch. For every symbol in the tracematch definition, `abc.tm` generates pieces of advice that trigger appropriate state transitions in the finite automaton when the events that the symbol describes occur at runtime. In a last step, abc then weaves this aspect into the program under test, just as it would weave any other AspectJ aspect: abc identifies instrumentation points, or *shadows* [MKD03], as described by the aspect's pieces of advice (respectively the tracematch's symbols). The compiler then converts every piece of advice to an advice method and, at each joinpoint shadow, adds calls to the appropriate advice methods, which in turn will execute the code that updates the tracematch state in response to program events. The example program in Figure 3.2 includes shadows as comments. For instance, the comment "*//hasNext(i1)*" tells the reader that at this line, the program contains a shadow for the symbol `hasNext` that binds tracematch variable `i` to program variable `i1`. Hence, when the program execution reaches this program line, this shadow will issue a `hasNext` event, which the runtime monitor will consume. The variable binding for this event will consist of a mapping $\{i \mapsto o(\texttt{i1})\}$.

3.1.4 Aspect generated for a tracematch

The general purpose of the AspectJ aspect that the tracematch implementation generates is to modify the program under test in such a way that it will automatically notify the runtime monitor when the events of interest take place at runtime. As Allan et al. explain in their primary paper on the tracematch implementation [AAC+05, Section 4.7], the generated aspects contain several pieces of advice for every tracematch:

- one piece of "sync" advice,
- one piece of "symbol" advice for every tracematch symbol,
- one piece of "some" advice, and
- one piece of "body" advice for every tracematch symbol that may lead to a complete match.

Figure 3.4 shows the pieces of advice that the abc extension `abc.tm` would generate for the HasNext tracematch from Figure 1.2. The piece of advice in lines 2–5 first synchronizes on a lock unique to the tracematch to assure mutual exclusion of what follows. Computing a transition is a two-step operation in the tracematch implementation. First, the two pieces of advice in lines 8–10 and 13–15 register the fact that a `hasNext` event, respectively a `next` event, has occurred. These pieces of advice also store a reference to the object `i` mentioned in the `target` pointcut. Then secondly, the piece of advice in lines 18–21 (Avgustinov et al. [AAC+05] called this the "some advice") commits these events, i.e., updates the automaton state according to the events that the two pieces of symbol advice registered. The reason for this two-staged commit computation of transitions is that this way tracematches can implement a well-defined semantics in cases where symbol definitions overlap, i.e., two symbols refer to an overlapping set of joinpoints. When this happens, the "some" advice will have the information that more than one abstract event occurred at the same joinpoint, and can update the state of the monitor accordingly. Such a two-staged update protocol is used in other monitoring tools like J-LO [Bod05] as well. JavaMOP uses a simpler technique that leads to unexpected behavior when symbol definitions overlap. The last piece of advice, in lines 24–27, iterates through all the complete matches that the tracematch's runtime monitor computed, i.e., it iterates over every iterator `iter` on which two consecutive `next` calls occurred, and executes the tracematch body (line 7 in Figure 1.2) with the tracematch's variable `i` bound to `iter`.

```
//"sync advice"
before() : call(* java.util.Iterator.hasNext()) && target(Iterator) ||
    call(* java.util.Iterator.next()) && target(Iterator) {
    //1) acquire mutex lock
}

//"symbol advice" for "hasNext"
before(Iterator i) : call(* java.util.Iterator.hasNext()) && target(i) {
    //2a) register transition for "hasNext" event, exposing target iterator "i"
}

//"symbol advice" for "next"
before(Iterator i) : call(* java.util.Iterator.next()) && target(i) {
    //2b) register transition for "next" event, exposing target iterator "i"
}

//"some advice"
before() : call(* java.util.Iterator.hasNext()) && target(Iterator) ||
    call(* java.util.Iterator.next()) && target(Iterator) {
    //3) compute and commit transitions for events registered above; release lock
}

//"body advice" for final symbol "next"
before() : call(* java.util.Iterator.next()) && target(Iterator) {
    //4) for every complete match on an iterator "iter":
    //      execute tracematch body with tracematch variable "i" bound to "iter"
}
```

Figure 3.4: Aspect generated for HasNext tracematch (simplified)

All pieces of advice shown here are before advice, because the symbol definitions in the tracematch from Figure 1.2 are also before symbols. The "sync advice" and "some advice" need to execute whenever any of the other pieces of advice can execute, and hence their pointcut is a disjunction of the pointcut definitions of all the symbol definitions. Because these pieces of advice need no access to the bound variables (here the iterator `i`), their pointcuts are "flattened", i.e., they use `this/target` and `args` pointcuts with types as parameters instead of variables. This speeds up the execution of these pieces of advice. The same holds for the "body advice": it also uses a flattened pointcut. This piece of advice executes the original tracematch body, which is only necessary for events that may lead to a complete match. By analyzing the regular expression "`next next`" of the tracematch definition, the tracematch implementation can determine that a match can only be completed using a `next` symbol. Hence, the pointcut for this piece advice only matches on calls to `next`, not `hasNext`.

The listing in Figure 3.5 shows how an AspectJ compiler like abc applies these aspects to a code sequence "it .hasNext(); it .next()" in the program under test. Every code position at which a piece of advice could apply is called a *joinpoint shadow*, or shadow for short, in AspectJ terminology [MKD03]. In the statement sequence mentioned above, there exist two shadows, one for the call to `next`, one for the call to `hasNext`. However, not all of the five generated pieces of advice apply to both shadows. By the way their pointcuts were generated, the pieces of "sync" and "some" advice apply to both shadows, but the pieces of "symbol" advice apply to their respective method calls only. The piece of "body" advice only applies to the `next` call. It is important to note that the order in which these pieces of advice apply and execute at the joinpoint shadow is well-defined: when all pieces of advice are before advice, then it is the same order in which these pieces of advice are mentioned in the aspect[3]. The tracematch implementation takes care to always produce an ordering that makes the pieces of advice execute in the order required to guarantee the correct semantics.

[3]For a complete documentation of AspectJ's rules on advice ordering see `http://www.eclipse.org/aspectj/doc/released/progguide/semantics-advice.html`.

```
//1) acquire mutex lock
//2a) register transition for "hasNext" event, exposing target iterator "it"
//3) compute and commit transitions for events registered above; release lock
it.hasNext();

//1) acquire mutex lock
//2b) register transition for "next" event, exposing target iterator "it"
//3) compute and commit transitions for events registered above; release lock
//4) for every complete match on an iterator "iter":
//      execute tracematch body with tracematch variable "i" bound to "iter"
it.next();
```

This Figure shows the way in which an AspectJ compiler would apply the pieces of advice from Figure 3.4 to the statement sequence "it.hasNext(); it.next()".

Figure 3.5: Joinpoint shadows generated by aspect from Figure 3.4

Apart from these aspects, the tracematch implementation also generates a set of specialized classes for each tracematch, which implement the actual runtime monitor. The generated aspect directly accesses these classes.

3.1.5 Runtime monitor generated for a tracematch

The classes that make up a tracematch's runtime monitor implement constraints over variable bindings [AAC+05]. Constraints are logical formulae which a program can evaluate to determine whether a given variable-to-object binding holds in the given state. The aspect associates one constraint with each state. Every constraint is stored in Disjunctive Normal Form and consists of a set of disjuncts, where each disjunct associates tracematch variables with objects. For instance, a constraint $c = o(\texttt{l}) \wedge i = o(\texttt{it})$ on a state q, for tracematch variables c (e.g. a collection) and i (e.g. an iterator) encodes the information that the combination of the objects that program variables `l` and `it` reference is currently in state q. A negative constraint $i \neq o(\texttt{it})$ on a state q would denote that the object referenced by `it` is not in q.

The runtime system initially associates *true* ($\mathbf{tt}$) with the initial automaton state and *false* ($\mathbf{ff}$) with all other states, since all objects start at the initial automaton state. The constraint at the initial state always remains $\mathbf{tt}$ because tracematches may start a match anytime—as we noted earlier, a tracematch is matched against each suffix of a program's execution trace. Because the `HasNext` automaton (Figure 1.2) has three states (q_0, q_1, q_2), its initial configuration is $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$.

As the program executes, the woven pieces of advice notify the runtime monitor as events occur; the monitor in turn updates the constraints accordingly. Figure 3.6 shows the evaluation of the `HasNext` automaton at the start of the `print` method from Figure 3.2 with initial configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ **$(\mathbf{A}_0)$**. The `hasNext` shadow at line 11 has no effect on this configuration, since q_0 has no `hasNext` transition and all other states contain $\mathbf{ff}$ **$(\mathbf{A}_1)$**, i.e., no object is in any of these other states. The assignment that copies `i1` to `i2` has no effect on the configuration either, since none of the declared symbols match this statement. Hence, we obtain configuration **$(\mathbf{A}_2)$**. At line 13, the `next` shadow binds tracematch variable `i` to the object stored in `i2`, denoted $o(\texttt{i2})$. The `next` transition from q_0 to q_1 in the automaton causes the following update:

$$\begin{aligned} c'(q_1) &\equiv c(q_1) \vee (\ c(q_0) \wedge i = o(\texttt{i2})\ ) \\ &\equiv \mathbf{ff} \vee (\ \mathbf{tt} \wedge i = o(\texttt{i2})\ ) \\ &\equiv i = o(\texttt{i2}). \end{aligned}$$

Here $c(q_i)$ denotes the original constraint at q_i and $c'(q_i)$ the constraint after executing line 13. The update results in the configuration $(\mathbf{tt}, i = o(\texttt{i2}), \mathbf{ff})$ **(B)**. Another call to `next()` on the same iterator $o(\texttt{i2})$ would propagate $i = o(\texttt{i2})$ to the final state q_2, and the runtime would execute the tracematch body. However, the example program contains a loop, so control flow wraps around to line 11, again issuing the first event `hasNext(i=`o`(i1))`. Because q_1 has no `hasNext` self-loop, object $o(\texttt{i1})$ cannot possibly be in q_1 after this event, and the tracematch runtime hence conjoins q_1's constraint with a negative binding $i \neq o(\texttt{i1})$. Since the incoming configuration

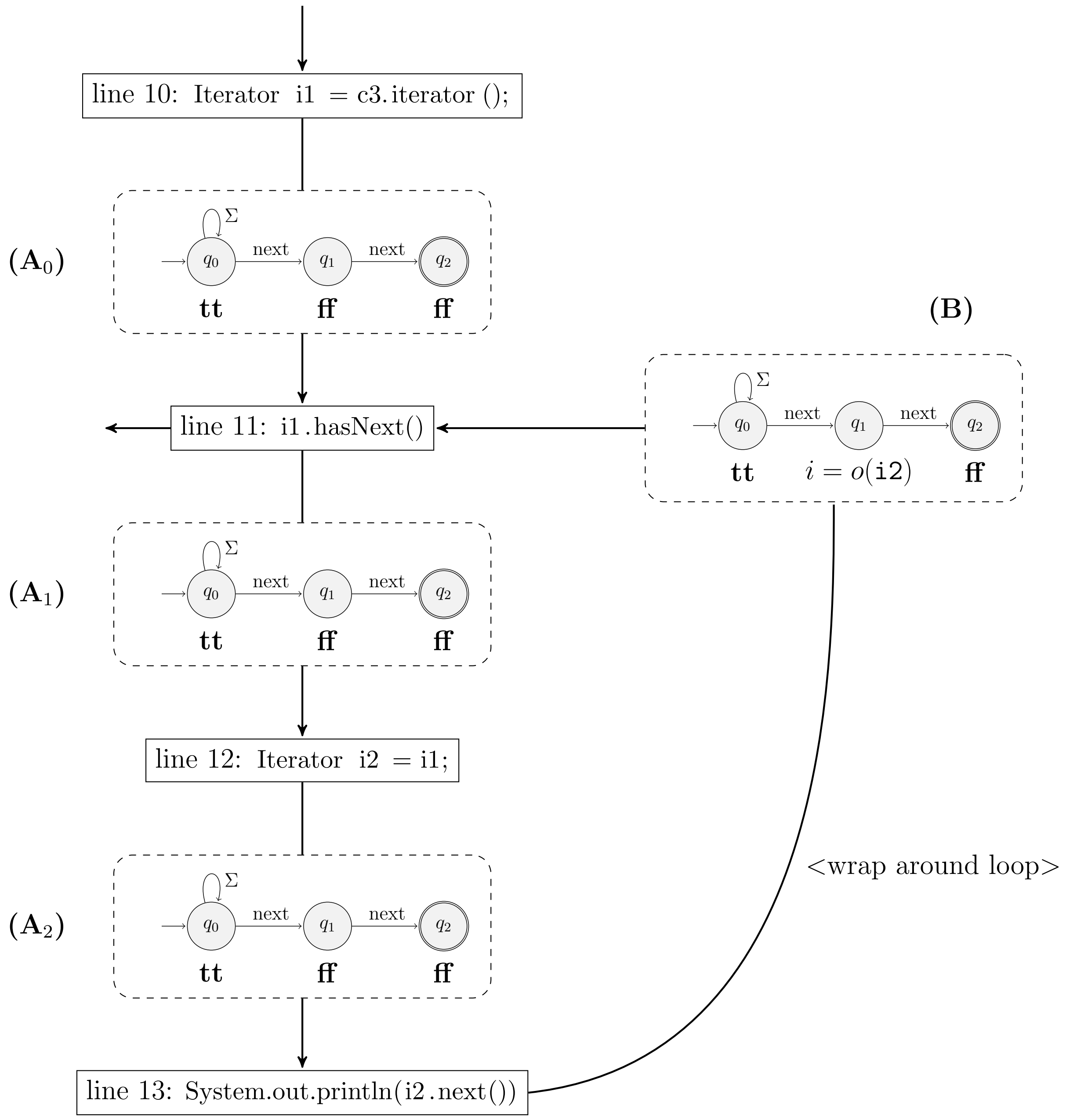

line 10: Iterator i1 = c3.iterator ();
(A0)
Σ
next
next
q0
q1
q2
tt
ff
ff
line 11: i1.hasNext()
(B)
i = o(i2)
(A1)
line 12: Iterator i2 = i1;
(A2)
<wrap around loop>
line 13: System.out.println(i2.next())

Figure 3.6: Effect of `print` (from Figure 3.2, page 33) on HasNext automaton.

is $(\mathtt{tt}, i = o(\mathtt{i2}), \mathbf{ff})$, and because $o(\mathtt{i2}) = o(\mathtt{i1})$, we get:

$$\begin{aligned} c'(q_1) &\equiv c(q_1) \wedge i \neq o(\mathtt{i1}) && (3.1)\\ &\equiv i = o(\mathtt{i2}) \wedge i \neq o(\mathtt{i1}) && (3.2)\\ &\equiv \mathbf{ff}, && (3.3) \end{aligned}$$

which again yields the configuration $(\mathtt{tt}, \mathbf{ff}, \mathbf{ff})$ $(\mathbf{A}_1)$. Observe that this configuration, at line 13, has not changed from the previous iteration.

One effect of the static analyses that we present in Chapters 4 and 5 is that the analyses will allow us to determine situations like the one above, where configurations loop without reaching a final state, and remove the instrumentation from these program points, omitting the unnecessary monitor updates. Note, however, that aliasing information is critical for any static analysis that approximates the runtime configurations: in the above example, any analysis must know that $o(\mathtt{i2}) = o(\mathtt{i1})$ to conclude that the constraint updates inside the loop have no effect (as in Equation (3.2)).

3.2 Finite-state specification languages in JavaMOP

JavaMOP provides an extensible logic framework for specification formalisms [CR07]. Via logic plug-ins, one can easily add new logics into JavaMOP and then use these logics within specifications. As we already showed in Figure 3.1, JavaMOP has several specification formalisms built in, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars. In this dissertation we focus our attention on the finite-state formalisms ERE, PTLTL and FTLTL. While no formal semantics exists for these formalisms, their implementation closely follows the semantics of tracematches, and we therefore restrict our discussion to explaining crucial differences to tracematches.

3.2.1 JavaMOP specifications in general

All JavaMOP specifications follow a general scheme. Figure 3.7 shows a specification for the HasNext pattern (from Figure 1.2), written in JavaMOP's ERE syntax. Lines

1–5 in this specification contain certain modifiers that parametrize JavaMOP's code-generation scheme for this particular specification. While we cannot go into details here, [CR07] gives an overview of these modifiers. Important is however the modifier logic = ERE, which tells JavaMOP that what follows is a specification based on Extended Regular Expression syntax. Like in tracematches, a JavaMOP specification has a name and a vector of parameters that can be bound to objects (line 6). In lines 7–8, the specification also contains a set of symbol definitions not much different from the ones found in tracematches. While the syntax differs slightly from the one chosen for tracematches, the general intent and semantics of these symbols coincide. The actual specification pattern is shown in line 9 in the form of a "formula". Because the programmer chose the ERE plug-in in line 5, JavaMOP will parse this "formula" as an extended regular expression, using the respective logic plug-in.

In lines 11–13 follows a so-called "Validation handler". JavaMOP allows both validation and violation handlers to be present. A validation handler is called whenever the given pattern is matched, respectively the given formula is fulfilled at the current state of the execution. A violation handler, on the other hand, is executed whenever the monitor reaches a violating configuration. JavaMOP generally uses deterministic monitors that will always have a unique "unproductive" configuration from which no accepting configuration can be reached any more. This configuration is, by definition, the violating configuration. In case of the HasNext pattern, the error situation arises when the pattern *matches*, i.e., two consecutive `next` calls were seen, and therefore the programmer uses a validation handler.

3.2.2 General code-generation scheme in JavaMOP

Similar to the tracematch implementation, JavaMOP generates a set of pieces of advice from every property specification. However, unlike the tracematch implementation, JavaMOP generates no "sync" and no "some" advice (see Section 3.1.4). In other words, JavaMOP generates exactly one piece of advice per symbol definition. Whenever the program under test reaches a joinpoint that is matched by one of the generated pieces of advice, that piece of advice triggers a transition in the monitor

```
sync
partial
centralized
scope = global
logic = ERE
HasNext(Iterator i){
  event hasNext<i> : end(call(∗ i.hasNext()));
  event next<i>: begin(call(∗ i.next()));
  formula : (hasNext + next)∗ next next
}
Validation handler{
   System.err.println("called next twice in a row!");
}
```

Figure 3.7: Definition of the HasNext pattern in JavaMOP's ERE syntax

that is associated with the objects that the advice references. The monitor itself can have different implementations, depending on the specification formalism that was used to generate the monitor: even manual monitor implementations [CR07] are possible. Due to the lack of "sync" and "some" advice, the current implementation of JavaMOP cannot decide whether two abstract events occur at the same joinpoints or at different joinpoints (where one precedes the other). This may lead to unexpected results in rare cases in which symbol definitions overlap. Nevertheless, from our experience it seems to be the case that usually symbol definitions within the same property specification describe disjoint sets of joinpoints, and hence we do not see this implementation detail of JavaMOP as problematic.

The tracematch implementation associates automaton states with objects via constraints. This leads to an indexing scheme where a set of objects (or vector of objects, stored in a sequence of maps) is associated with a state. Hence, when the tracematch machinery computes a transition, it retrieves all the bindings for the source state of that transition and then propagates these bindings onto the transition's target state, the indexing scheme is:

$$\text{State} \rightarrow \text{1st bound object} \rightarrow \ldots \rightarrow \text{n-th bound object}$$

The indexing scheme in JavaMOP is inverted. In JavaMOP the runtime implementation first indexes on the objects bound at the current joinpoint shadow and for these objects retrieves a generic monitor instance; the indexing scheme is:

$$\text{1st bound object} \rightarrow \ldots \rightarrow \text{n-th bound object} \rightarrow \text{Monitor}$$

This scheme gives JavaMOP additional flexibility when it comes to implementing the runtime monitor. Because the monitor (and its state) is at the end of the indexing scheme, JavaMOP can easily substitute the implementation of one monitor for another. In particular, JavaMOP uses simple deterministic finite-state machines for the implementation of its ERE language, however uses other automata to implement the other specification languages.

The chosen indexing scheme limits the generality of the current JavaMOP implementation. The current implementation allows programmers only to define property specifications in which the first matched symbol binds all the property's free variables. This is because, if one of the variables were unbound, JavaMOP could not insert the monitor in its indexing tree, because it would have to index on a value that has not yet been bound. Tracematches circumvent this problem through their constraint logic. The limitation matters in cases like ASyncContainsAll: this property first binds one synchronized collection only, and then afterwards the second one. Hence, JavaMOP cannot currently express this and other similar properties. In recent work [CR09] Chen and Roşu introduce a novel indexing algorithm that solves this limitation. However, Chen and Roşu do not discuss how this algorithm would work in combination with weak references. As Avgustinov et al. showed [ATdM07], a combination of both efficient indexing and weak references is necessary to obtain an efficient monitor implementation.

Every JavaMOP monitor is just a Java object. When an event of interest occurs in the program execution, the matching piece of advice first looks up the monitor in a set of generated maps, and then advances the internal configuration of this monitor by calling one of several transition methods that JavaMOP generated on the monitor class. When the monitor reaches an accepting or violating configuration, the monitor class executes the associated validation or violation handler.

In the following we describe the three different finite-state specification languages supported by JavaMOP. Because the general indexing scheme is always the same, we restrict ourselves to explaining the nature of the monitor classes that JavaMOP generates when it comes to explaining the implementation.

3.2.3 Extended regular expressions (ERE)

Regular expressions were first described by Kleene in 1956 [Kle56] who used them to describe event patterns in neural networks. Regular expressions use concatenation, alternation and the Kleene-star operator to describe regular languages in a purely positive way: as a regular expression grows, the language that the expression describes grows as well. Sometimes it can be convenient to express regular languages as the inverse or difference with respect to some other regular language. Extended regular expressions hence add negation and intersection operations. While this makes certain regular expression more concise, it does not add to their expressiveness [Lin01]. We already showed an example of the Extended Regular Expression syntax in Figure 3.7. A general description of the syntax can be found on the JavaMOP website[4].

The monitor that JavaMOP generates from an extended regular expression is a deterministic finite-state machine that is implemented as a simple Java class with a single integer field that holds the number of the current state. The transition methods generated by JavaMOP simply update this integer according to a transition table that is computed from the original regular expression. The monitor executes the validation handler when it switches to an accepting state, and it executes the violation handler when it moves into an unproductive state from which no final state can be reached any more.

Unlike tracematches, the ERE syntax in JavaMOP does not match the regular expression against each suffix of the execution trace. A tracematch would match the

[4]Descr. of ERE syntax: `http://fsl.cs.uiuc.edu/index.php/ERE_Plugin_Input_Syntax`

regular expression “`a a`” twice on an input trace “`a a a`”: once after the second `a` and once after the third. The semantics of the ERE plug-in of JavaMOP are however that the expression is matched against the entire trace. Hence, JavaMOP would execute the validation handler when reading the second `a` of the trace “`a a a`” (because “`a a`” is in the language) and it would execute its violation handler when reading the third `a`, because one cannot extend the trace “`a a a`” so that the extended trace is in the regular language described by the expression “`a a`”. Due to this semantics, the automata that JavaMOP uses to evaluate an extended regular expression have no Σ-self-loop on the initial state (unlike the ones used for tracematches, see Figure 1.2). As Chen and Roşu show [CR07], there are cases where the tracematch semantics can be more convenient, and cases where the ERE semantics chosen by JavaMOP can be more convenient. Clara hence supports both formalisms.

In general, the rule for the execution of validation and violation handlers are as follows. The validation handler is executed after every event that extends the abstract execution trace in such a way that this execution trace is a word in the language that regular expression describes. The violation handler executes exactly after each other monitored event, e.g. after every single event that extends the abstract execution trace in such a way that this execution trace is not a word in the language that the regular expression describes.

3.2.4 Future-time linear temporal logic (FTLTL)

Future-time linear temporal logic [Pnu77] (FTLTL, or often just LTL for short) is a fragment of the tree logic CTL* [EH86] that reasons about a single path in a labeled transition system. Hence, the model over which an LTL formula is evaluated is a single finite or infinite sequence of position where each position is labeled with a (possibly empty) set of atomic propositions. LTL uses temporal operators such as Finally, Globally, Until, and Next, as well as the usual Boolean operators to express formulae that talk about the future (or remainder) of a path, as seen from a “current position”. In JavaMOP, pointcuts are treated as propositions. JavaMOP interprets the execution trace as a path of finite length where each node of this path is a joinpoint that is matched by at least one of the formula's symbols. For each pointcut that matches at such a node, JavaMOP assumes a proposition at this node with the symbol's name. Because, as mentioned earlier, multiple symbols can match the same joinpoint, a node can hold multiple propositions; and because FTLTL is a propositional logic, an FTLTL formula can distinguish all the different situations of which propositions do or do not hold at a given node.

A complete description of the input syntax for JavaMOP's FTLTL plug-in can be found on the JavaMOP website[5]. To give an example, an FTLTL formula for our HasNext pattern (Figure 1.2) looks as follows:

```
next /\ o next
```

Here, `o` is the "next" operator (often called **X**) from LTL. Therefore the formula holds if the "next" symbol occurs next after having already seen another occurrence of the "next" symbol.

JavaMOP's FTLTL plug-in outputs a runtime monitor in the form of a binary transition tree finite-state machine (BTT-FSM) [CR03]. A BTT-FSM is a state machine in which each state holds a Binary Transition Tree, i.e., a Boolean function. The BTT-FSM determines the target state of a transition by computing this Boolean function when an event is received.

FTLTL formulae are traditionally evaluated over a path of infinite length. An execution trace of a terminating execution is always of finite length, however. Some LTL-based runtime-verification tools like J-LO [Bod05] have hence defined a specialized LTL semantics over finite traces. Researchers even developed special four-valued variants of LTL [BLS07] which allow not only for the answers "validated" or "violated" but also "possibly validated in the future" and "possibly violated in the future". The developers of JavaMOP however opted, to stick to the standard LTL semantic for finite traces. To make this work, one has to argue about all possible ways in which the finite execution trace could be extended, had the execution of the program not been interrupted or terminated. JavaMOP consequently assumes a formula as violated only when the execution seen so far (1) violates the formula and (2) cannot

[5]Descr. FTLTL syntax: `http://fsl.cs.uiuc.edu/index.php/FTLTL_Plugin_Input_Syntax`

be extended in any way such that the formula would be fulfilled again. It executes the violation handler when such a violation is detected. Conversely, it executes the validation handler when a formula is (1) validated and (2) no extension of the trace can falsify the formula again. For example, consider the formula "after `a` never `b`". When reading a trace "`a b`", this trace cannot be extended so that the formula is fulfilled again. For other formulae, this can be different. Consider the formula "after every `a` follows a `b`". This formula is violated on the trace "`a`", however it holds over the extended trace "`a b`". In fact, with the FTLTL semantics that JavaMOP uses, this kind of formula can never be be validated nor violated on any execution: For any execution trace ending in the symbol `b` (and hence making the formula hold), this trace can be extended with another `a`, falsifying the formula. Conversely, any trace ending with an unmatched `a` can be extended with an additional `b`, making the formula hold again.

3.2.5 Past-time linear temporal logic (PTLTL)

PTLTL [Pnu77] is similar to FTLTL in the sense that it is a temporal logic that is evaluated over the same paths as FTLTL. However, as the name suggests, PTLTL comes with past-time temporal operators, such as Previously (at the preceding position), Since, "at some point in the past" and "always in the past".

For PTLTL as well, a complete description of the plug-in's input syntax can be found on the JavaMOP website[6]. A possible specification of the HasNext example in PTLTL looks as follows:

```
next -> (*) hasNext
```

Here, `(*)` is the "previously" operator. Hence, the formula states that when "next" occurs on an iterator then "hasNext" must have been the previous operation. This specification would be used in combination with a violation handler: we want to notify the programmer of an error when the formula is violated, i.e., when `next` was called and the previous event was not `hasNext`.

[6]Descr. PTLTL syntax: `http://fsl.cs.uiuc.edu/index.php/PTLTL_Plugin_Input_Syntax`

Unlike the ERE plug-in and the FTLTL plug-in, the PTLTL plug-in in JavaMOP generates a monitor which has a vector of bits as its internal state [CR03].

The semantics of JavaMOP with respect to the execution of validation and violation handlers in the case of PTLTL follows the one from ERE, rather than the FTLTL semantics. The validation handler is executed after every event that extends the abstract execution trace in such a way that this execution trace validates the formula. The violation handler executes exactly after each other monitored event, e.g. after every single event that extends the abstract execution trace in such a way that this execution trace violates the formula.

3.3 Why multiple specification formalisms?

The JavaMOP specification compiler supports multiple finite-state specification formalisms, and likewise in the case of Clara we took special care that the approach would not be restricted to tracematches and regular expressions only, but would also support the static analysis of programs with respect to specifications written in other finite-state specification languages. However, what is the motivation for this generality? After all, one could argue, every finite-state specification is expressible in every finite-state specification language—by definition. While this is true, it holds that certain properties can be expressed more easily and more direct by some formalism while other properties are more easily expressed by another. In the following we give examples of such properties.

3.3.1 Cases that favour PTLTL over ERE & tracematches

The HasNext pattern is a good example for why it may be useful to have multiple specification formalisms. Above we mentioned a possible specification for the pattern in PTLTL, `next -> (*) hasNext`. Note that this specification also captures the case where `next` is called on a newly constructed iterator object. After all, also in this case `next` is called and the previous event was not `hasNext`, because there is no previous event. This semantics benefits the programmer in this case, as the formula correctly captures this additional error situation.

It is possible to capture the same property using the ERE pattern `next \/ (hasNext next)* next next`. As the reader can see, this pattern would be much more awkward to write and harder to read and understand. The problem is that a regular expression can only make positive statements about events, but not negative ones, i.e., a regular expression cannot explicitly require an event not to occur. Hence the regular expression needs to enumerate all the event sequences that may lead to an error state.

3.3.2 Cases that favour FTLTL over ERE & tracematches

In other cases, FTLTL may be the formalism of choice. This is particularly true when one wants to specify the absence of events on the remainder (i.e., in the future) of the execution trace. Consider a specification which defines that a file handle that is opened should also be closed. This kind of specification is a bounded liveness property, i.e., it requires some event to occur eventually (or in other words, the program to be alive). We can specify this rule in FTLTL as `open -> o<>close`, which says that when `open` holds at some execution state, from the next (`o`) state on at some point (`<>`) `close` will hold. In combination with a violation handler, this specification would aid the programmer in detecting erroneous situations where file handles are not properly closed. As above, this form of specification is very direct, and it would be hard to express this property with a tracematch. Again we would have to extend the alphabet, this time with an artificial end symbol `shutdown` that models program shutdown. The regular expression `open shutdown` expresses situations in which a handle is opened and then the program shuts down without the handle being closed in between. Again, this kind of specification is very indirect, plus it is actually non-trivial to instrument a Java program in such a way that it will issue a `shutdown` event before it shuts down. Runtime-verification tools that specialize on LTL, like J-LO [Bod05], use special “shutdown hooks”[7] to notify the programmer

[7]Shutdown hooks: `http://java.sun.com/j2se/1.5.0/docs/guide/lang/hook-design.html`

about formulae that become false at the end of an execution trace.

3.3.3 Cases that favour ERE & tracematches over LTL

A large set of specifications is nevertheless expressible as a regular expression, and in many of these cases, a regular expression is also most concise. For example consider the regular expression “`close write`” which, as we assume here, expresses that it is an error to close a file handle and then later on write to the same handle. This specification is very direct, and in particular no temporal operators are required, as the temporal operator “concatenation” is implicit in regular expressions: simply by placing the word `write` behind the word `close` we can specify that `write` has to follows `close` temporally for a property violation to occur. In FTLTL and PTLTL, temporal operators are required. A possible PTLTL formula would be “`write -> <*>!close`” (when we see a `write` then before we saw no `close`), a FTLTL formula would be “`close -> o[]!write`” (after `close` never `write`). Although both formulae are still quite direct, they do require explicit temporal operators. Especially in rare cases where specification refers to three or more events, a regular expression can be more concise, because nested PTLTL and FTLTL formulae quickly become hard to reason about.

3.3.4 Tracematches vs. ERE in JavaMOP

In their primary paper on JavaMOP, Chen and Roşu provide a detailed discussion of the difference in semantics between tracematches and ERE as they are implemented in JavaMOP—we here give a brief overview. As mentioned, tracematches match their regular expression against each suffix of the execution trace, while JavaMOP always matches the expression against the entire trace. Both can have advantages and disadvantages. As Chen and Roşu mention, using tracematches a programmer may easily profile the number of files that are opened and then closed using the simple expression “`open close`”. In JavaMOP such a pattern would only match the first sequence of opening and closing a particular file. If the same file was opened and closed again, JavaMOP would miss this match. A programmer would have to use the more complex expression "`(open+close)* open close`" to achieve the same semantics as tracematches. Using tracematches, on the other hand, it is more complex to match on the first occurrence of an event on the execution trace. A tracematch with the simple expression "`open`" would match whenever any file is opened, while its JavaMOP equivalent would only match the first such event. A programmer that uses tracematches can avoid this problem by introducing an artificial `start` symbol that matches the program start. Because the symbol `start` is guaranteed to occur only once on the program's execution trace, the expression "`start open`" then matches only at the first call to `open` (on any particular file).

3.3.5 Discussion & Conclusion

As we showed, PTLTL is particularly useful when it comes to specifying the absence of certain events in the past. The past of a single Java object begins when the object is constructed, and programmers can refer to this point of construction *implicitly* in a PTLTL formula, while in FTLTL or ERE/tracematches, the programmer would have to make this object-creation event explicit. FTLTL is more convenient when it comes to specifying "liveness properties" that require that some event should occur before the program terminates. Similar to the construction of an object being implicit in PTLTL, the logic FTLTL provides the end of a program (and hence the definite destruction of a Java object) as an implicit event, which makes such properties easier to express in FTLTL. Last but not least, ERE or tracematches are most convenient when it comes to specifications that refer to many events or that can easily be expressed as positive statements about event sequences. Regular expressions are more convenient and easy to write and read than PTLTL or FTLTL formulae in such cases because the concatenation operation in regular expression implies a temporal ordering without increasing the nesting depth of the formula: while any LTL formula reasoning about three events has to have a nesting depth of at least three, in regular expression the nesting depth does not necessarily increase as events are added. Whether to choose ERE or tracematches in the particular situation depends on the particular finite-state property. If matching on repetitive event sequences, tracematches with its suffix-matching semantics can be more convenient; when seeking to match the very first occurance of an event, the JavaMOP's ERE language is the language of choice.

We therefore conclude that it is beneficial to support a range of finite-state specification formalisms, as they give developers the opportunity to express a property in the formalism that they see most fit. We designed Clara to support any finite-state formalism, hence giving the developer a large degree of freedom.

3.4 Need for static analyses and optimizations

In this section we will motivate the need for the static analyses and optimizations that we will present in the next chapters. Throughout this dissertation we will consider two possibilities of verifying that a program satisfies a finite-state property: statically and through runtime verification. When a programmer weaves a monitoring aspect into her program under test, the AspectJ compiler emits a list of source code locations at which runtime events of interest could occur—the joinpoint shadows. After the compilation has finished, the programmer could in principle inspect all joinpoint shadows manually to see whether the shadows may indeed contribute to a property violation. However, this is only a viable approach if only a handful of shadows exist. The second approach is to not consider the joinpoint shadows at all, but to instead run the woven program and see if it reports a property violation at runtime. This approach requires good test coverage and, equally important, requires the monitoring instrumentation to induce only a low runtime overhead. In our experiments we applied a number of monitoring aspects to a number of programs and then sought to determine whether either approach, static inspection or runtime verification would be possible for any combination of aspect and program.

3.4.1 Experimental setup

For our experiments we first wrote a set of twelve tracematch specifications for different finite-state properties regarding collections and streams in the Java Runtime Library. We already explained all of these properties in Section 2.2. Table 3.1 gives brief descriptions for each of these properties. We selected properties of the Java Runtime Library because this allowed us to find a large set of programs to which these properties are of interest. Although this selection of properties induces a bias to our results, we believe that this bias is minor: we have no reason to believe that the properties of other application interfaces would be in any way more complex than those of the Java Runtime Library, especially in light of the study by Dwyer et al. [DAC99] (see Chapter 2). In fact we did study other application interfaces but this study did not yield any results that we felt worthwhile mentioning in this dissertation.

Although one can apply Clara to any AspectJ-based runtime monitor, we decided to restrict our experiments to monitors generated from tracematch specifications. In particular, we will not consider JavaMOP-based specifications. In previous work [BCR09] we showed that Clara always produces equivalent analysis results for equivalent runtime monitors, independent of the specification formalism that the programmer used to define these monitors. Hence there is little benefit from redoing all the experiments again with equivalent JavaMOP-based specifications. Note that the monitors that abc generates from tracematches are already heavily optimized [ATdM07] to induce a minimal runtime overhead. Therefore our baseline is by no means a naïve baseline. All our tracematch definitions are available for download at `http://www.bodden.de/clara/`.

Note that our tracematches contain no recovery code or even notification code of any kind: the bodies of the tracematches are empty. This is for two reasons. Firstly, the content of the bodies would only impact our static-analysis results if the body called back into the program under test. This is unlikely for a monitoring aspect. Hence, we can just as well assume that the body is empty. The content of the body does however have an impact on the runtime overhead of the runtime monitor. We are interested in measuring the time that the monitor has to consume to update its internal state based on the events that it monitors. By using empty tracematch bodies we can make sure that we measure only this overhead, and not any additional overhead that error-reporting or recovery code would cause.

For our benchmarks we used version 2006-10-MR2 of the DaCapo benchmark

property name	description
ASyncContainsAll	synchronize on `d` when calling `c.containsAll(d))` for synchronized collections `c` and `d`
ASyncIterC	only iterate a synchronized collection `c` when owning a lock on `c`
ASyncIterM	only iterate a synchronized map `m` when owning a lock on `m`
FailSafeEnum	do not update a vector while iterating over it
FailSafeEnumHT	do not update a hash table while iterating over its elements or keys
FailSafeIter	do not update a collection while iterating over it
FailSafeIterMap	do not update a map while iterating over its keys or values
HasNextElem	always call hasMoreElements before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	do not use a Reader after its InputStream was closed
Writer	do not use a Writer after its OutputStream was closed

Table 3.1: Monitored specifications for classes of the Java Runtime Library; available at `http://www.bodden.de/clara/`

suite [BGH+06]. The suite contains eleven different workloads that exercise ten different programs. In Table 3.2 we give brief descriptions of the benchmarks (taken from [BGH+06]) and also state the number of methods that they contain. Note that, on average, a DaCapo benchmark has about four times as many methods and about four times as much code as one of the well-known SPEC benchmarks [Coo01, Coo99]. The benchmarks hsqldb, lusearch and xalan are multi-threaded. The benchmarks luindex and lusearch are two different workloads that produce two different program runs of the same program lucene. All benchmarks use dynamic class loading, and the benchmark jython even generates classes dynamically which it then executes.

We used the AspectBench Compiler [8] to weave any of the twelve aspects separately into each one of the ten DaCapo programs. By default, the compiler weaves only into the application itself, not into the Java Runtime Library. Hence the runtime monitors that we use do, for example, monitor events where the benchmark program uses collections and streams of the Java Runtime Library, but it does not monitor events where these objects are used inside the Java Runtime Library. As a baseline, we also compiled every benchmark program with the AspectBench Compiler but with no aspects present.

3.4.2 Number of shadows after weaving

Table 3.3 shows for every tracematch/benchmark combination the number of shadows that the woven program for this combination contains. Note that the benchmarks luindex and lusearch share the same code base. Therefore, the Quick Check produces identical numbers for these two benchmarks. As the table shows, the compilation process produced some shadows in all but eleven of the 120 combinations. Moreover, the number of shadows is usually quite large. 100 cases result in more than 10 shadows, and 88 cases result in even more than 50 shadows; on average the compilation results in 326 shadows spread over 82 methods. Therefore, in all but a few lucky cases it would be impractical for a programmer to investigate all these program points

[8] Clara is available as part of the AspectBench Compiler (abc). For our experiments we used Subversion revision 7508 of abc, in combination with Subversion revision 3326 of Soot.

benchmark	classes	methods	description
antlr	224	2972	parses one or more grammar files and generates a parser and lexical analyzer for each
bloat	263	3986	performs a number of optimizations and analysis on Java bytecode files
chart	512	7187	uses JFreeChart to plot a number of complex line graphs and renders them as PDF
eclipse	344	3978	executes some of the (non-gui) JDT performance tests for the Eclipse IDE
fop	967	6889	takes an XSL-FO file, parses it and formats it, generating a PDF file
hsqldb	385	5859	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	508	7240	interprets a the pybench Python benchmark
luindex	311	3013	uses lucene to index a set of documents; the works of Shakespeare and the King James Bible
lusearch	311	3013	uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	530	4785	analyzes a set of Java classes for a range of source code problems
xalan	562	6463	transforms XML documents into HTML

Table 3.2: The DaCapo benchmarks, taken from `http://dacapobench.org/`

	antlr	bloat	chart	eclipse	fop
ASyncContainsAll	0	71	6	10	0
ASyncIterC	0	1621	498	214	146
ASyncIterM	0	1684	507	236	176
FailSafeEnumHT	133	102	44	217	205
FailSafeEnum	76	3	1	117	18
FailSafeIter	23	1394	510	391	288
FailSafeIterMap	130	1180	374	548	1374
HasNextElem	117	2	0	89	10
HasNext	0	849	248	109	72
LeakingSync	170	1994	920	1325	2347
Reader	50	7	65	218	102
Writer	171	563	70	1045	429

	hsqldb	jython	luindex lusearch	pmd	xalan
ASyncContainsAll	0	31	18	10	0
ASyncIterC	33	128	149	671	0
ASyncIterM	39	138	152	718	0
FailSafeEnumHT	114	153	37	100	319
FailSafeEnum	120	110	61	21	222
FailSafeIter	112	253	217	546	158
FailSafeIterMap	252	250	136	583	540
HasNextElem	53	64	22	6	63
HasNext	16	63	74	346	0
LeakingSync	528	1082	629	986	1005
Reader	1216	139	226	102	106
Writer	1378	462	146	62	751

Table 3.3: Number of shadows right after weaving

manually to see if these points contribute to a property violation. It may, however, be possible to monitor these programs for violations at runtime.

3.4.3 Runtime overhead through monitoring code

To determine the runtime overhead that the monitoring aspects cause, we first executed the un-instrumented programs to establish a baseline. We used the standard workload size of the DaCapo harness. The DaCapo benchmark suite comes with a `-converge` option, that tries to make the determined runtime values better comparable. When the option is enabled, then DaCapo runs the benchmark in question multiple times until the relative standard deviation of the determined runtimes drops below 3%. DaCapo then assumes that the benchmark has reached a "stable state", e.g. that the virtual machine has loaded and just-in-time compiled all of the benchmark's methods. DaCapo then runs the benchmark one more time and reports the runtime of this last run. In the past we have experienced problems with this approach: if, coincidentally, the last run is extraordinarily better or worse than the previous runs then the runtime that DaCapo reports will deviate from the "normal" runtime. We hence modified the harness so that it would instead proceed as follows.

DaCapo first runs the benchmark once without collecting any timing information—a warm-up run. Then DaCapo re-runs the benchmark multiple times, again until the relative standard deviation of the determined runtimes drops below 3% (but at least 5 times and at most 20 times). Then we report the mean of these runs. This gives us the advantage that the number that we report originates from a sample of runs from which we know that this sample deviated no more than 3%.

It is important to note that we modified the monitor definitions slightly, to cope with the fact that we re-run the benchmark without shutting down the virtual machine between two consecutive runs. Normally, a runtime monitor would keep on monitoring until the virtual machine shuts down, and hence it would accumulate internal state over multiple runs of the test harness, potentially leading to a progressive slowdown, as the monitor needs to store the state for more objects in each run. We hence modified the property definitions so that the monitor would be reset whenever the test harness re-executes. We added a new symbol `newDaCapoRun` which matches at the beginning of the method that triggers a new benchmark run. We did not modify the tracematches' regular expressions. Hence, by the way tracematches are implemented, reading the symbol `newDaCapoRun` will automatically cause the implementation to delete all monitoring state, effectively resetting the monitor.

We executed the benchmarks using the HotSpot Client VM (build 1.4.2_12-b03, mixed mode), with its standard heap size on a machine with an AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 7.10 with kernel version 2.6.22-14 and 4GB RAM. Then we executed, in the very same way, the programs that have had the monitoring aspects woven into them.

Table 3.4 shows for every benchmark the baseline execution time (with no aspect present, in milliseconds), and for every monitoring aspect the relative runtime overhead that this aspect causes, in percent. For instance, a value of 20 means that the presence of the monitoring aspect caused the instrumented program to run 20% longer than the un-instrumented baseline version of the program. We do not show values that are within the 3% error margin. The value ">1h" means that a single benchmark run took longer than one hour. In this case we aborted the run after this first hour.

As the results show, runtime verification is indeed a viable solution for many of the benchmark/property configurations that we consider. The vast majority of test runs do not expose any perceivable overhead. This is true even for benchmark/property combinations that have a high number of shadows. For instance, chart-ASyncIterM has 507 shadows but shows no perceivable runtime overhead. This is because the specific test run does not exercise the instrumented collections and iterators a lot. Nevertheless, there are some cases for which the overhead is high. In about one quarter of our test runs the overhead was at least 10%, and in five cases the overhead was so high that even a single execution of the benchmark took longer than one hour. Such overheads clearly prevent programmers from using runtime verification in these particular cases. In the next sections we will show that our static analyses can significantly lower and sometimes completely eliminate the runtime overhead in the majority of cases.

	antlr	bloat	chart	eclipse	fop
baseline (ms)	4079	9276	14666	44725	2562
ASyncContainsAll		-4		-4	
ASyncIterC		**140**			5
ASyncIterM		**139**			
FailSafeEnumHT	**10**				
FailSafeEnum				-4	
FailSafeIter		**>1h**	8		**14**
FailSafeIterMap		**>1h**			7
HasNextElem					
HasNext		**329**			
LeakingSync	9	**163**	**91**		**209**
Reader	**30218**				
Writer	**37862**	**>1h**			5

	hsqldb	jython	luindex	lusearch	pmd	xalan
baseline (ms)	12235	11105	17144	13940	13052	13424
ASyncContainsAll						
ASyncIterC					**28**	
ASyncIterM					**35**	
FailSafeEnumHT		**>1h**	**32**			
FailSafeEnum			**30**	**18**		
FailSafeIter			5	**20**	**2811**	
FailSafeIterMap		**13**	5		**>1h**	
HasNextElem			**12**			
HasNext					**70**	
LeakingSync		**>1h**	**34**	**365**	**16**	
Reader				**77**		
Writer						

Table 3.4: Runtime overheads with static analyses disabled; baseline in milliseconds, rest in percent, values within the 3% error margin omitted, values of at least 10% in boldface

Chapter 4
Flow-insensitive optimizations through Dependent Advice

As we showed in the last chapter, although runtime verification can be a powerful tool for bug detection, it can slow down programs in many situations. In addition, because runtime verification only observes a single program execution at a time, it can virtually never guarantee that a program fulfils a crucial property on every execution. Hence, in this chapter and the following one, we explain the core principles of Clara, our framework for evaluating runtime monitors ahead of time, i.e. at compile time. Using the analyses in Clara, programmers can often prove the program under test correct with respect to the property in question. When this is not the case, the analyses facilitate optimizations that remove instrumentation from program points at which this instrumentation is unnecessary because the finite-state property cannot be violated through these points. By applying our analyses, programmers can produce a partially instrumented program that is usually much more efficient than a fully instrumented program would have been.

History-based aspects. The analyses and optimizations in Clara exploit the fact that the aspects used for runtime monitoring are almost exclusively *history-based aspects.* A history-based aspect executes its pieces of advice conditionally, based on the observed execution history. Figure 4.1 shows a simplified example of a history-based aspect, the “ConnectionClosed” aspect. This aspect monitors the events of disconnecting and reconnecting a connection `c`, as well as writing data to `c`. Note that almost all the aspect code is concerned with bookkeeping internal state. In the example, the error message at line 17 implements the only functionality that is visible outside the aspect. As we and others showed, this bookkeeping can induce a large runtime overhead [ATdM07, BHL07, CR07, MLL05, GOA05].

Inter-dependent pieces of advice. It is important to note that the example aspect prints the error only if *both* the advice “disconn” and “write” execute on the same connection `c`. In addition, the advice “reconn” only has to execute on connections that the program under test both disconnects and writes to at some point in time. Compilers could use this important information to apply powerful optimizations: For example, one does not have to monitor “disconn(`c`)” if the connection `c` is never written to. Unfortunately a programmer cannot express this crucial domain knowledge in plain AspectJ syntax, and also specification compilers like abc (for tracematches) and JavaMOP have no means of expressing this information in plain AspectJ. To make things worse, it would be very hard for an AspectJ compiler to re-construct this knowledge solely based on the aspect code. After all, specification compilers generate code that is very much specialized to the specified property. Such code can become quite complex and the lack of additional knowledge about the specification impedes crucial optimizations.

Preserving dependency information through annotations. In the remainder of this chapter we explain how we overcome this problem with *dependent advice*. A dependent advice contains dependency annotations to encode crucial domain knowledge: a dependent advice needs to execute only when its dependencies are fulfilled. For the “connection” example from Figure 4.1, a programmer could add the annotation

dependency{ **strong** disconn, write; **weak** reconn; }.

```
aspect ConnectionClosed {
    Set closed = new WeakIdentityHashSet();

    after /*disconn*/ (Connection c) returning:
        call(* Connection.disconnect()) && target(c) {
        closed.add(c);
    }

    after /*reconn*/ (Connection c) returning:
        call(* Connection.reconnect()) && target(c) {
        closed.remove(c);
    }

    after /*write*/ (Connection c) returning:
        call(* Connection.write(..)) && target(c) {
        if(closed.contains(c))
            error("May not write to "+c+", as it is closed!");
    }
}
```

Figure 4.1: ConnectionClosed monitoring aspect

This annotation conveys the information that the execution of the advice "disconn" and "write" both depend on one another, and in addition the execution of "reconn" depends on both "disconn" and "write" executing at some point in time[1].

Programmers can use dependent advice to document design intent or to aid static verification. For instance, dependencies can encode forbidden combinations of events and static whole-program analyses, like ours, can prove that such combinations cannot occur. In addition, however, programmers can use dependent advice to aid an efficient implementation of history-based aspects in general, and runtime monitors in particular: the flow-insensitive whole-program analysis that we present in this chapter removes dispatch code for dependent advice from program locations at which the advice's dependencies cannot be fulfilled. The results of our evaluation show that the use of dependent advice can yield significant speedups at runtime, and it often allows programs to statically prove that their history-based aspect can never reach an error state. That way, dependent advice can facilitate static program verification.

However, writing dependency annotations by hand can be error-prone and time-consuming. Therefore it would be beneficial if tools could generate these annotations automatically. Fortunately, as we discussed earlier in this dissertation, many people do not write history-based aspects by hand either: researchers have proposed several tools [AAC+05, CR07, MH06, KLM06] that generate history-based AspectJ aspects automatically, from formal specifications from runtime verification or model-driven development. As we show in this dissertation, these specifications convey enough domain knowledge to generate dependent advice automatically. We modified abc to generate dependent advice from tracematches. Likewise, Feng Chen modified JavaMOP to generate dependent advice from specifications that express monitoring properties using past-time and future-time linear temporal logic and regular expressions.

Annotating aspects, not programs. It is important to note that the annotations we present here are annotations to the monitoring aspects, not to the (potentially

[1]The names "strong" and "weak" were suggested by Patrick Lam, as an analogy to strongly and weakly connected graphs. This analogy will become clearer when we discuss the annotations' semantics.

much larger) program under test. Moreover, we will show how finite-state monitoring tools can generate these annotations automatically. Hence, the annotations should merely be seen as an intermediate language to communicate specification knowledge to Clara. While programmers could in principle write these annotations manually, for instance to statically analyze programs based on hand-written runtime monitors, we do not see this as the primary application of our annotation language.

Chapter organization. In the next section we explain dependent advice, their syntax and semantics. We present our implementation of dependent advice in Section 4.2, and in Section 4.3 we explain an algorithm to generate dependent advice from any finite-state based monitor specification. In Appendix A, we also prove this algorithm correct and "stable": it generates equivalent dependency annotations for equivalent finite-state specifications, even if these specifications are written in different formalisms. Section 4.4 demonstrates, through experiments, that the use of dependent advice speeds up runtime monitoring and can, in many cases, help prove already at compile time that programs fulfil the given finite-state property.

The annotations presented in this chapter encode flow-insensitive information, and consequently we resolve the dependencies that the annotations encode through a flow-insensitive analysis. In Chapter 5 we will explain how to lift both annotations and analyses to a flow-sensitive level.

4.1 Syntax and semantics of dependent advice

In this section we describe dependent advice. We start by explaining their syntax, first in a short form and then in a more verbose form. Then we explain how to type-check dependent advice and give a matching semantics.

4.1.1 Syntax

Dependent advice are a backwards-compatible AspectJ language extension that comprise the following syntactic changes (Figure 4.2 shows the complete Syntax in EBNF, as a syntactic extension to AspectJ.):

- Pieces of advice can have a `dependent` *modifier*,
- every `dependent` advice is given a *name*, and
- an aspect can hold a set of *dependency declarations.*

A dependency declaration has the following form:

dependency{
    strong $s_1, \ldots, s_n$;
    weak $w_1, \ldots, w_m$;
}

Here s_1 through s_n, and w_1 through w_m, are *advice references*: names of dependent advice declared in the same aspect as the dependency declaration. Figure 4.3 shows how to use dependent advice for ConnectionClosed.

Informally, one can describe the meaning of the dependency annotation **dependency**{ **strong** disconn, write; **weak** reconn; } by the dependency graph that we show in Figure 4.4. The declaration of strong advice references **strong** disconn, write; defines that the two pieces of advice form a strongly connected component (SCC) in this dependency graph; in other words, both pieces of advice depend on each other. The additional declaration of **weak** reconn; defines that `reconn` is weakly connected to this SCC: it depends on both `disconn` and `write`, but these pieces of advice do not depend on `reconn`. The meaning of an edge $a \rightarrow b$ in this graph is that a only needs to execute if itself and b have a chance of executing. Hence, in the ConnectionClosed example, the dependency states that if `disconn` was to execute on a Connection `c` for which it is known that `write` never occurs on `c` then the execution of `disconn` can safely be omitted—and the other way around. In addition, due to the weak dependency, `reconn` only has to execute on Connections `c` for which both `disconn` and `write` execute at some point.

Note that this makes sense when considering the implementation of ConnectionClosed from Figure 4.3. When `disconn` can never execute on a connection object c, *Modifier* **::=** "`public`" | "`synchronized`" | ... | **"`dependent`"**.

AdviceDecl ::= *Modifier** [*RetType*] *BefAftAround* ***AdviceName***
"(" [*ParamList*] ")" [*AftRetThrow*] ":" *Pointcut Block*.

AdviceName ::= ID

AspectMemberDecl ::= *AdviceDecl* | ... | ***DependencyDecl***.

***DependencyDecl* ::=**
"`dependency`" "{" "`strong`" *AdviceNameList* ";"
["`weak`" *AdviceNameList* ";"] "}".

AdviceNameList* ::= *AdviceRef* | *AdviceRef* "," *AdviceNameList.

***AdviceRef* ::= *AdviceName* | *AdviceName* "(" *VarList* ")"**.

VarList* ::= *VarName* | *VarName* "," *VarList.

***VarName* ::=** ID | **"*"**.

Figure 4.2: Syntax of dependent advice, as extension (shown in boldface) to the syntax of AspectJ [ACH$^+$05a]

then *c* will never be added to the set `closed` and hence both other pieces of advice degrade to a no-op when executing on *c*. Similarly, when `write` can never execute for a collection *c*, then the aspect can never issue an error message for *c* either. Hence, there is no need for `disconn` to add *c* to the `closed` set in the first place, and therefore there is also no need for `reconn` to remove *c* from the set. On the other hand, `disconn` and `write` do not depend on `reconn`: it may well happen that a program disconnects a connection *c* and then writes to *c*, which triggers the error message; reconn does not need to execute.

Note that the dependency annotation in Figure 4.3 (line 2) omits the variable name `c` of the Connection. This is because, by default, a dependency annotation infers variable names from the formal parameters of the advice declarations that it references (e.g. line 6). The dependency annotation from Figure 4.3 is a short hand for the more verbose

```
aspect ConnectionClosed {
    dependency{ strong disconn, write; weak reconn; }

    Set closed = new WeakIdentityHashSet();

    dependent after disconn(Connection c) returning:
        call(* Connection.disconnect()) && target(c) {
        closed.add(c);
    }

    dependent after reconn(Connection c) returning:
        call(* Connection.reconnect()) && target(c) {
        closed.remove(c);
    }

    dependent after write(Connection c) returning:
        call(* Connection.write(..)) && target(c) {
        if(closed.contains(c))
            error("May not write to "+c+", as it is closed!");
    }
}
```

Figure 4.3: ConnectionClosed with dependent advice

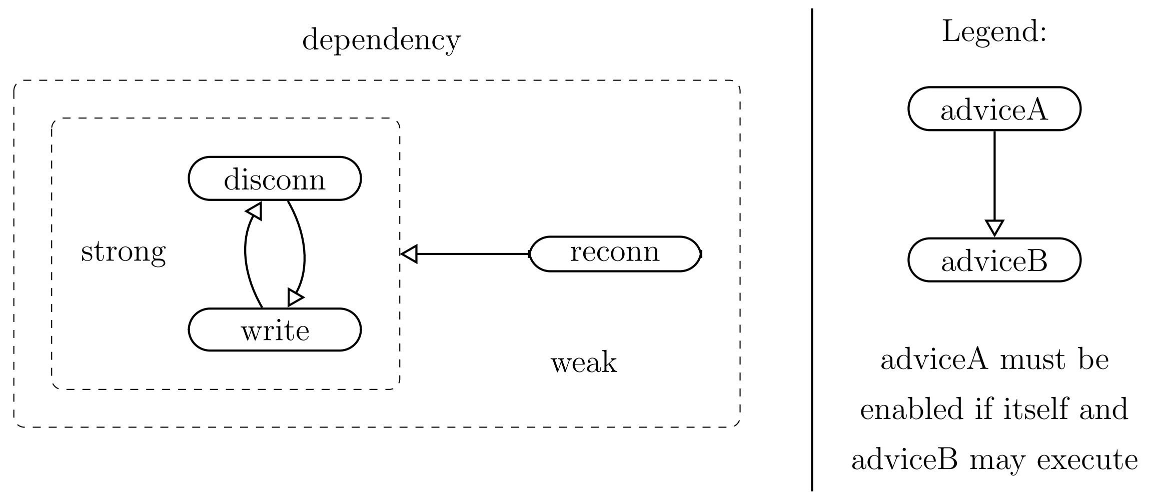


Figure 4.4: Informal dependency graph for the ConnectionClosed example

dependency{ **strong** disconn(c), write(c); **weak** reconn(c); }

The semantics of variables in dependency declarations is similar to unification semantics in logic programming languages like Prolog [CM03]: The same variable at multiple locations in the same dependency refers to the same object. For each advice name, the dependency infers variable names in the order in which the parameters for this advice are given at the site of the advice declaration. Variables for return values from `after returning` and `after throwing` advice are appended to the end. For instance, the following advice declaration would yield the advice reference createIter (c, i):

dependent after createIter(Collection c) **returning**(Iterator i):
  call(∗ Collection . iterator ()) {}

We decided to allow for this kind of automatic inference of variable names because both code-generation tools and programmers frequently seem to follow the convention that equally-named advice parameters are meant to refer to the same objects. That way, programmers or code generators can use the simpler short-form as long as they follow this convention. Nevertheless the verbose form can be useful in rare cases.

Assume the following piece of advice:

dependent before detectLoops(Node n, Node m):
  call(Edge.**new**(..)) && **args**(n,m) {
  if(n==m) { System.out.println("No loops allowed!"); }}

This advice only has an effect when `n` and `m` both refer to the same object. However, due to the semantics of AspectJ, the advice cannot use the same name for both parameters, so the inferred annotation would be detectLoops(n,m). The verbose syntax for dependent advice allows us to state nevertheless that for the advice to have an effect, both parameters actually have to refer to the same object, say `k`:

dependency{ **strong** detectLoops(k,k); }

We next define the subset of syntactically valid dependent advice that we consider well-typed.

4.1.2 Well-typed dependent advice

In the following, whenever we speak of a dependent advice then we mean an advice annotated with the `dependent` modifier. We say that an AspectJ aspect holding dependent advice and dependency annotations is well-typed if all of the following holds:

- Only dependent advice have names and every dependent advice has a name that is unique in the declaring aspect.
- Each advice name mentioned in a dependency declaration refers to an existing dependent advice in the declaring aspect.
- Each dependent advice is referenced by at least one dependency declaration.

In addition, for each dependency declaration it must hold that:

- The list of `strong` advice names is non-empty.
- The `strong` and `weak` lists of advice names are sets, i.e., reference each advice only once, and both sets are disjoint.

- The number of variables for an advice name equals the number of parameters of the unique advice with that name, including the after-returning/throwing variable. (When variable names are inferred, then this is guaranteed to hold. The rule can only be broken when the programmer supplies variable names manually.)

- Advice parameters that are assigned equal names have compatible types: For two advice declarations `a(A x)` and `b(B y)`, with `a(p)` and `b(p)` in the same dependency declaration, both `A` and `B` must be cast-convertible [GJSB05, §5.5].

Further, in the verbose form, each variable should be mentioned at least twice inside a dependency declaration. If a variable v is only mentioned once we give a warning, because in this case the declaration states *no* dependency with respect to v. The warning suggests to use the wildcard "`*`" instead. Semantically, `*` also generates a fresh variable name. However, by stating `*` instead of a variable name, the programmer acknowledges explicitly that the parameter at this position should be ignored when resolving dependencies.

4.1.3 Matching semantics

We define the matching semantics of dependent advice as a semantic extension to ordinary advice matching in AspectJ. In the semantics of AspectJ, pieces of advice are executed at joinpoints, each joinpoint being a period of time during the program's execution. Depending on the kind of advice, the advice will execute before or after the joinpoint:

- a before advice executes before the joinpoint,

- an after-returning advice executes after a regular (i.e. non-exceptional) return from the joinpoint,

- an after-throwing advice executes after an exceptional return from the joinpoint,

- an after advice executes whenever the equivalent after-returning or a after-throwing advice would execute, and

- an around advice executes, like before advice, before the joinpoint, however during its execution can replace the original joinpoint by custom code.

Because an advice executes at an entry or exit of a joinpoint, it is convenient to label these entry and exit points. In the remainder of this dissertation we will call these points events.

4.1.3.1 Joinpoints versus events

A joinpoint effectively describes a period: it has a beginning and an end, and code can execute before or after the joinpoint (i.e., at its beginning or end) or instead of the joinpoint. In particular, joinpoints can be nested. For instance, a field-modification joinpoint can be nested in a method-execution joinpoint. Runtime monitors read a trace of atomic events. Because events are atomic, they cannot be nested. Joinpoints merely induce these events.

Definition 4.1 (Events)**.** Let j be an AspectJ joinpoint. Then j induces two events, j_{before} and j_{after} which occur at the beginning respectively end of j. For any set $\mathcal{J}$ of joinpoints we define the set $\mathcal{E}(\mathcal{J})$ of all events of $\mathcal{J}$ as:

$$\mathcal{E}(\mathcal{J}) := \bigcup_{j \in \mathcal{J}} \{j_{\text{before}}, j_{\text{after}}\}.$$

In the following we will often just write $\mathcal{E}$ instead of $\mathcal{E}(\mathcal{J})$, if $\mathcal{J}$ is clear from the context.

4.1.3.2 Refining the matching function

A program can generally have multiple aspects with dependent advice. However, since the semantics of dependent advice in one aspect is defined independently from other aspects, in the following we assume one fixed aspect A, without loss of generality. (While it would be interesting to consider dependencies between entire aspects, this topic is out of the scope of this dissertation.)

Let $\mathcal{A}$ be the set of A's pieces of advice, $\mathcal{D}$ the set of dependency declarations in A, $\mathcal{V}$ the set of all valid variable names, $\mathcal{O}$ the set of all heap objects allocated on a given program execution and $\mathcal{E}$ the set of all AspectJ events on that execution.

We refer to the set of all pieces of advice that a dependency declaration d references as $\mathit{all}(d)$, and to the sets of advice that d references as strong or weak by $\mathit{strong}(d)$ and $\mathit{weak}(d)$ respectively.

In the following let us assume that variables in d have been fully inferred (see Section 4.1.1) and that any occurring wildcard `*` has been replaced by a fresh variable name. The set $\mathit{vars}(d)$ is the set of variables mentioned in d. Our type checks ensure that d references each advice $a \in \mathit{all}(d)$ only once. Therefore d induces for each advice a a mapping σ_a^d from variables in $\mathit{vars}(d)$ to a's parameters: If d references an advice declaration `adv(T1 p1,...,Tn pn)` using the advice reference `adv(v1,...,vn)` then we obtain the mapping

$$\sigma_{\texttt{adv}}^d = \{\texttt{v1} \mapsto \texttt{p1}, \ldots, \texttt{vn} \mapsto \texttt{pn}\}.$$

Note that σ_a^d is the identity function when variable names were inferred for a in d.

4.1.3.3 Advice matching for normal advice

We model advice matching in AspectJ [HH04] as a function

$$\mathit{match}: \quad \mathcal{A} \times \mathcal{E} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\perp\}.$$

For each pair of advice $a \in \mathcal{A}$ and event $e \in \mathcal{E}$, match returns $\perp$ in case a does not execute at e. If a does execute then match returns a variable binding β, a mapping from a's parameters to objects ($\{\ \}$ for parameter-less advice).

Adapting variable bindings to a dependency's name space. In the following it will be useful to rename variable bindings in such a way that they do not map from advice parameters to objects, but instead from the appropriate variable names defined in the dependency declaration to these objects. Formally, we define an overloaded version of the substitution $\sigma_{\texttt{adv}}^d$, called $\hat{\sigma}_{\texttt{adv}}^d$ and defined as follows:

$$\hat{\sigma}_{\texttt{adv}}^d(\beta) = \lambda v.\ \beta(\sigma_{\texttt{adv}}^d(v))$$

For example, assume two pieces of advice, declared as `a(X x1)` and `b(X x2)`, and both referenced in a dependency declaration d: **dependency**{ **strong** a(k),b(k); }.

This yields the mappings $\sigma^d_{\mathtt{a}} = \{\mathtt{x1} \mapsto \mathtt{k}\}$, and $\sigma^d_{\mathtt{b}} = \{\mathtt{x2} \mapsto \mathtt{k}\}$. Also assume that a matches some event, yielding a variable binding $\beta_a = \{\mathtt{x1} \mapsto o_1\}$ and that b also matches some event, yielding a variable binding $\beta_a = \{\mathtt{x2} \mapsto o_2\}$. Then we can apply $\hat{\sigma}^d_{\mathtt{a}}$ to β_a and $\hat{\sigma}^d_{\mathtt{b}}$ to β_b accordingly, yielding $\hat{\sigma}^d_{\mathtt{a}}(\beta_a) = \{\mathtt{v} \mapsto o_1\}$, respectively $\hat{\sigma}^d_{\mathtt{b}}(\beta_b) = \{\mathtt{v} \mapsto o_2\}$.

This renaming allows us to define a function *renMatch*, which is essentially equivalent to AspectJ's original matching function *match* but instead produces variable bindings in terms of the variables of a particular dependency declaration d:

$$\mathit{renMatch}: \quad \mathcal{A} \times \mathcal{E} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$\mathit{renMatch}(a, e, d) = \begin{cases} \hat{\sigma}^d_a(\mathit{match}(a, e)) & \text{if } \mathit{match}(a, e) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Compatible events. In the remainder of this section we will refer to "compatible events". We say that two events e_a and e_b are *compatible* with respect to a dependency declaration d and two pieces of advice a and b if a executes at e_a with a variable binding β_a, b executes at e_b with a variable binding β_b respectively, and both β_a and β_b assign the same objects to equal variables, with variable names substituted as defined through d. Formally we define a predicate *compatible* as follows:

$$\mathit{compatible}: \quad \mathcal{E} \times \mathcal{A} \times \mathcal{E} \times \mathcal{A} \times \mathcal{D} \quad \rightarrow \quad \mathbb{B}$$

$$\mathit{compatible}(e_a, a, e_b, b, d) =$$

$$\text{let } \beta_a := \mathit{renMatch}(a, e_a), \beta_b := \mathit{renMatch}(b, e_b) \text{ in}$$

$$\beta_a \neq \bot \wedge \beta_b \neq \bot \wedge \ \forall v_a \in \mathit{dom}(\beta_a)\ \forall v_b \in \mathit{dom}(\beta_b): \ v_a = v_b \rightarrow \beta_a(v_a) = \beta_b(v_b)$$

Here by $\mathit{dom}(\beta)$ we denote the domain of the partial function β.

4.1.3.4 Advice matching for dependent advice

Dependent advice differ in their matching semantics from normal AspectJ advice, and we therefore define a function *depMatch* that matches *dependent* advice against events, based on $\mathcal{D}$ and *match*. *depMatch* also has access to a function *activates*. This function is a parameter to *depMatch* (description follows).

$$\mathit{depMatch}: \quad \mathcal{A} \times \mathcal{E} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$\mathit{depMatch}(a,e) = \begin{cases} \mathit{match}(a,e) & \text{if } \mathit{match}(a,e) \neq \bot \wedge \\ & \quad \exists d \in \mathcal{D} \ : \ \mathit{activates}(d,a,e) \\ \bot & \text{otherwise} \end{cases}$$

The function *depMatch* refines the original *match* function provided by AspectJ: It only produces a match if the Boolean predicate *activates* holds for at least one advice dependency. When $\mathit{activates}(d,a,e)$ holds, we say that the dependency d *activates* the dependent advice a at e. The predicate *activates* is a parameter to our matching semantics. A compiler may choose between different implementations of *activates* but we define that any *sound* implementation of dependent advice *must* guarantee:

Condition 4.1 (Soundness condition)**.**

$$\forall d \in \mathcal{D} \ \ \forall a \in \mathcal{A} \ \ \forall e_a \in \mathcal{E} \ :$$
$$\Big(\ a \in \mathit{all}(d) \ \wedge \ \forall b \in \mathit{strong}(d) \ \exists e_b \in \mathcal{E} : \mathit{compatible}(e_a, a, e_b, b, d) \ \Big)$$
$$\longrightarrow \mathit{activates}(d, a, e_a) = \mathbf{true}$$

Informally, Condition 4.1 states that a dependency d *must* activate a at event e_a, if d references a (as strong or weak advice), and for each *strong* advice b in d there is some event e_b (at some time earlier or later in the program execution, or the current event itself) that is compatible with e_a (with respect to d, a and b).

Note that, to guarantee that a dependency be activated, the dependency's *strong* pieces of advice need to execute on the same objects, but its *weak* pieces of advice are not required to execute to activate the dependency. This is why in the ConnectionClosed example from Figure 4.1 (page 69), the dependency declaration **dependency**{ **strong** disconn, write; **weak** reconn; } references `reconn` weakly: the aspect may emit an error message even when it never observes any `reconn` event. Note however, that it would be incorrect to not reference `reconn` at all. Not referencing `reconn` would disable this advice completely (and in fact our type checks would prevent such a program from compiling), and hence a `reconn` event may accidentally be missed. This could in turn lead the monitor to believe that the connection is disconnected although it is not, and yield a false warning. The weak reference guarantees that the event will not be missed.

Note also that there may be multiple dependency declarations referring to the same dependent advice. This is intentional. For a dependent advice to be activated on a vector of objects it just needs to be activated by one dependency that references the advice. When dependency declarations are generated from finite-state specifications, it can easily happen that we have to generate multiple dependencies for a single finite-state monitor. This is the case when the finite-state machine induced by the specification can reach a final state along multiple paths. The reader be also reminded that our type checks ensure that every piece of advice declared as `dependent` must be referenced by at least one dependency declaration. This ensures that it has at least some chance of being activated.

The predicate "activates". As mentioned, the predicate *activates* is a parameter to the semantics above. The most conservative implementation of *activates* would be the constant function **true**. This would effectively treat dependent advice just as ordinary AspectJ advice (*depMatch* degenerates to *match* as our type-checks ensure that $\mathcal{D} \neq \emptyset$).

An optimizing implementation would instead want to return **false** from *activates* whenever possible, but without jeopardizing soundness. A perfect implementation would determine *activates* such that it returns **false** whenever the antecedent (left-hand side) of Condition 4.1 does not hold. That way, the implementation would disable dependent advice whenever possible but still guarantee soundness. Unfortunately, determining *activates* that way is undecidable: at the time where *activates* needs to decide whether or not to activate a dependency at the current event, it may need to know whether a compatible event will occur in the future.

A sensible implementation of dependent advice must therefore find an approximation to *activates*. It must *try* to return **false** on a best-effort basis, but only when the soundness condition permits, i.e., when the antecedent of the soundness condition does not hold. In the following section, we explain an effective implementation based on this principle.

4.2 Implementing dependent advice

We next explain the static abstraction of Condition 4.1 that we use in our implementation of Clara. The abstraction considers all possible program executions, in a flow-insensitive fashion. In Subsection 4.2.2 we prove this abstraction sound. We explain the details of our concrete implementation in the AspectBench Compiler in Subsection 4.2.3.

4.2.1 A static abstraction of Condition 4.1

Our soundness condition, Condition 4.1, defines when $\mathit{activates}(d, a, e_a)$ *must* return **tt**. As noted earlier, an effective implementation of dependent advice should attempt to return **ff** from this function whenever possible, i.e., whenever the antecedent of Condition 4.1 does not hold. This is exactly the case when its negation holds:

Condition 4.2 (Negation of the antecedent of Condition 4.1)**.**
$a \notin \mathit{all}(d) \;\vee\; \exists b \in \mathit{strong}(d)\; \forall e_b \in \mathcal{E} : \neg \mathit{compatible}(e_a, a, e_b, b, d)$

According to Condition 4.2, a dependency d can fail to activate a dependent advice a for two reasons. In the first case d does not reference a at all, i.e., $a \notin \mathit{all}(d)$. This is the trivial case. (Note that our type checks demand that a be referenced by *some* dependency, so there must be another dependency d' which at least gives a a chance of being activated.) The second reason is that there is a strong advice b in d so that there exists no event e_b that is compatible with e_a. This is the condition that our static analysis exploits.

Note that we can fully determine the following parts of Condition 4.2 at compile time. For each dependency d we can determine the sets $\mathit{strong}(d)$ and $\mathit{all}(d)$. For any advice $a \in \mathit{all}(d)$ the variable substitution subst_a^d (used within $\mathit{compatible}$) is also statically determined. Hence, the only parts of Condition 4.2 that our static analysis needs to approximate are:

1. the set $\mathcal{E}$ of all events, and
2. the variable binding $\mathit{match}(a, e)$ that occurs when advice a matches at joinpoint e (also used within *compatible*).

Approximating events through joinpoint shadows. As mentioned earlier, a woven AspectJ program generates the events j_{before} and j_{after} for every joinpoint j by executing a piece of code generated by the AspectJ compiler before, respectively after, a specific program location, j's *joinpoint shadow*, $\mathit{shadow}(j)$. We define the set $\mathcal{S}$ of all shadows as:

$$\mathcal{S} = \bigcup_{j \in \mathcal{J}} \{s \mid s = \mathit{shadow}(j)\}$$

Note that the sets that are joined in this union may not necessarily be disjoint in the first place: multiple joinpoints can share the same shadow if the shadow resides in a loop or is reached multiple times through re-entrant calls.

We can now define our static approximation of Condition 4.2 via joinpoint shadows. Given a dependent advice a, a shadow s_a and a dependency d, we define:

Condition 4.3 (Static approximation of Condition 4.2)**.**
$a \notin \mathit{all}(d) \;\vee\; \exists b \in \mathit{strong}(d)\; \forall s_b \in \mathcal{S} : \neg \mathit{stCompatible}(s_a, a, s_b, b, d)$

The function *stCompatible* is a static approximation of *compatible* that accepts shadows instead of joinpoints. Both functions are very similar. The only difference is that *compatible* uses *match* to compute mappings from variables to runtime objects. At compile time we have no access to runtime objects. *stCompatible* therefore approximates this mapping through a compile-time function *stMatch*, as defined below.

Approximating objects through points-to sets. Because we now deal with joinpoint shadows, we redefine *match* as a function *stMatch* over inputs from $\mathcal{S}$ instead of $\mathcal{E}$. A function call $\mathit{match}(a, e)$ returns $\perp$ when advice a does not execute at e. This is a runtime decision: several AspectJ pointcuts have to be evaluated at runtime. For instance the pointcut `this(A)` only matches if the concrete runtime type of the currently executing object is a subtype of `A`. AspectJ compilers allow the AspectJ runtime to determine a match by weaving a *dynamic residue* [HH04] in place of the joinpoint shadow. In some cases a compiler can statically determine that an advice a can never apply at a given joinpoint shadow $s = \mathit{shadow}(j)$. For instance, in the above example it could be that the currently executing object must be of a final type (i.e., can have no subtypes) that is not a subtype of `A`. In this case `this(A)` cannot hold at s, and the compiler generates a "never" residue that instructs the compiler not to weave any advice code for a at s. In the following we will say that $\mathit{never}(a, s)$ holds in this situation.

The other difference between *match* and *stMatch* is that, because *stMatch* is evaluated at compile time, it cannot return a mapping from advice parameters to runtime objects. Every joinpoint shadow does however give us access to a mapping *local* which maps each advice parameter p to the local program variable l that the compiler inserts to bind p to its runtime value when the advice is executed at this shadow. For a local variable l we can determine its points-to set [LH03] $\mathit{pointsTo}(l)$. A points-to set $\mathit{pointsTo}(l) = \{s_1, \ldots, s_n\}$ is a set of allocation sites[2]. The set models the fact that l is only ever assigned objects that are allocated at the sites $s_1, \ldots, s_n$. We denote the set of all points-to sets by $\mathbb{P}$. This allows us to define *stMatch* as follows.

$$\mathit{stMatch}: \quad \mathcal{A} \times \mathcal{S} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathbb{P}\} \cup \{\bot\}$$

$$\mathit{stMatch}(a, s) = \begin{cases} \bot & \text{if } \mathit{never}(a, s) \\ \lambda p \;.\; \mathit{pointsTo}(\mathit{local}(p)) & \text{otherwise} \end{cases}$$

Just as we defined a renamed version *renMatch* of *match*, we also define a renamed version *renStMatch* of *stMatch*:

$$\mathit{renStMatch}(a, s, d) = \begin{cases} \hat{\sigma}_a^d(\mathit{stMatch}(a, s)) & \text{if } \mathit{stMatch}(a, s) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

This makes us almost ready for defining our static approximation of the function *compatible*. The last insight that we exploit is that two run-time objects referenced

[2]In principle, points-to sets can also be implemented differently; in particular they can hold additional information, not just allocation sites. For this dissertation we only require that it must be possible to check two points-to sets for disjointness and impose no restrictions on the internal representation.

by advice parameters p and q cannot point to the same objects if $\mathit{pointsTo}(\mathit{local}(p)) \cap \mathit{pointsTo}(\mathit{local}(q)) = \emptyset$: In this case p and q are only assigned values from local variables that themselves are definitely not assigned objects from the same allocation site. This yields the following definition of *stCompatible*.

$$\mathit{stCompatible} : \quad \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{A} \times \mathcal{D} \quad \rightarrow \quad \mathbb{B}$$

$$\mathit{stCompatible}(s_a, a, s_b, b, d) =$$

$$\text{let } \beta_a := \mathit{renStMatch}(a, s_a), \beta_b := \mathit{renStMatch}(b, s_b) \text{ in}$$

$$\beta_a \neq \bot \wedge \beta_b \neq \bot \wedge \; \forall v_a \in \mathit{dom}(\beta_a) \, \forall v_b \in \mathit{dom}(\beta_b) : \; v_a = v_b \rightarrow \beta_a(v_a) \cap \beta_b(v_b) \neq \emptyset$$

(Here and in the remainder of this dissertation $\mathit{dom}(\varphi)$ denotes the domain of a function φ.)

4.2.2 Soundness of the approximation

We next define what it means for this abstraction to be sound, and prove soundness based on this definition.

Theorem 4.1 (*stCompatible* soundly approximates *compatible*)**.**
$\forall e_a, e_b \in \mathcal{E} \;\; \forall d \in \mathcal{D} \;\; \forall a, b \in \mathit{all}(d) \;:$

$$\mathit{compatible}(e_a, a, e_b, b, d)$$

$$\longrightarrow \mathit{stCompatible}(\mathit{shadow}(e_a), a, \mathit{shadow}(e_b), b, d)$$

Proof 4.1 (Proof of Theorem 4.1)**.** The proof of Theorem 4.1 is almost immediate if one assumes that points-to sets are computed in a sound way, i.e., if o is an object created at allocation site s and assigned to a program variable l then $s \in \mathit{pointsTo}(l)$—a general assumption that we make in this dissertation. We conduct the proof in inverse order, from the right to the left. If $\mathit{stCompatible}(\mathit{shadow}(e_a), a, \mathit{shadow}(e_b), b, d)$ does not hold then this can have two reasons: (1) we have $\mathit{never}(a, s_a)$ or $\mathit{never}(b, s_b)$, or (2) the two shadows induce variable bindings that assign disjoint points-to sets to the same variable from d (used at different locations). In case (1) $\neg\mathit{compatible}(e_a, a, e_b, b, d)$ holds trivially because $\mathit{never}(a, s_a)$ implies $\mathit{match}(a, e_a) = \bot$, and the same holds for

b, s_b and e_b. Similarly, (2) disjoint points-to sets imply distinct runtime objects (assuming sound points-to sets). □

Theorem 4.1 directly implies the following corollary, therefore proving our approximation sound.

Corollary 4.1 (Condition 4.3 soundly approximates Condition 4.2)**.** For every event $e \in \mathcal{E}$ of a joinpoint $j \in \mathcal{J}$ with $s := \mathit{shadow}(j)$, every dependency d and every dependent advice $a \in \mathit{all}(d)$, it holds that Condition 4.3 implies Condition 4.2.

This concludes the discussion of our static abstraction. In the following we give some additional detail about the actual implementation within the AspectBench Compiler.

4.2.3 Implementation in abc

Figure 4.5 gives an overview of our implementation of dependent advice as an extension "`abc.da`" to the AspectBench Compiler (*abc*). The user provides a Java or AspectJ program as input, plus a set of aspects augmented with dependency annotations. In a first step, our compiler extension parses and type-checks the aspects and annotations. It then splits apart the dependency information from the aspects and the base program[3]. *abc* then matches the resulting plain-AspectJ aspects against the base program, producing a "weaving plan". This plan holds information about which advice applies where in the program. *abc* next weaves the appropriate pieces of advice into the program (based on the weaving plan) and produces a woven program—still un-optimized. At this stage, our extension intercepts the compilation to analyze the woven program based on the previously extracted dependency annotations. For each potential match recorded in the weaving plan, we statically analyze if the dependencies for the matched advice can potentially be fulfilled at the matched program location. If not, then we remove this potential match from the plan. In this chapter

[3] In the terminology of aspect-oriented programming, people frequently refer to the fragment of the program that is free of any aspect functionality as base program.

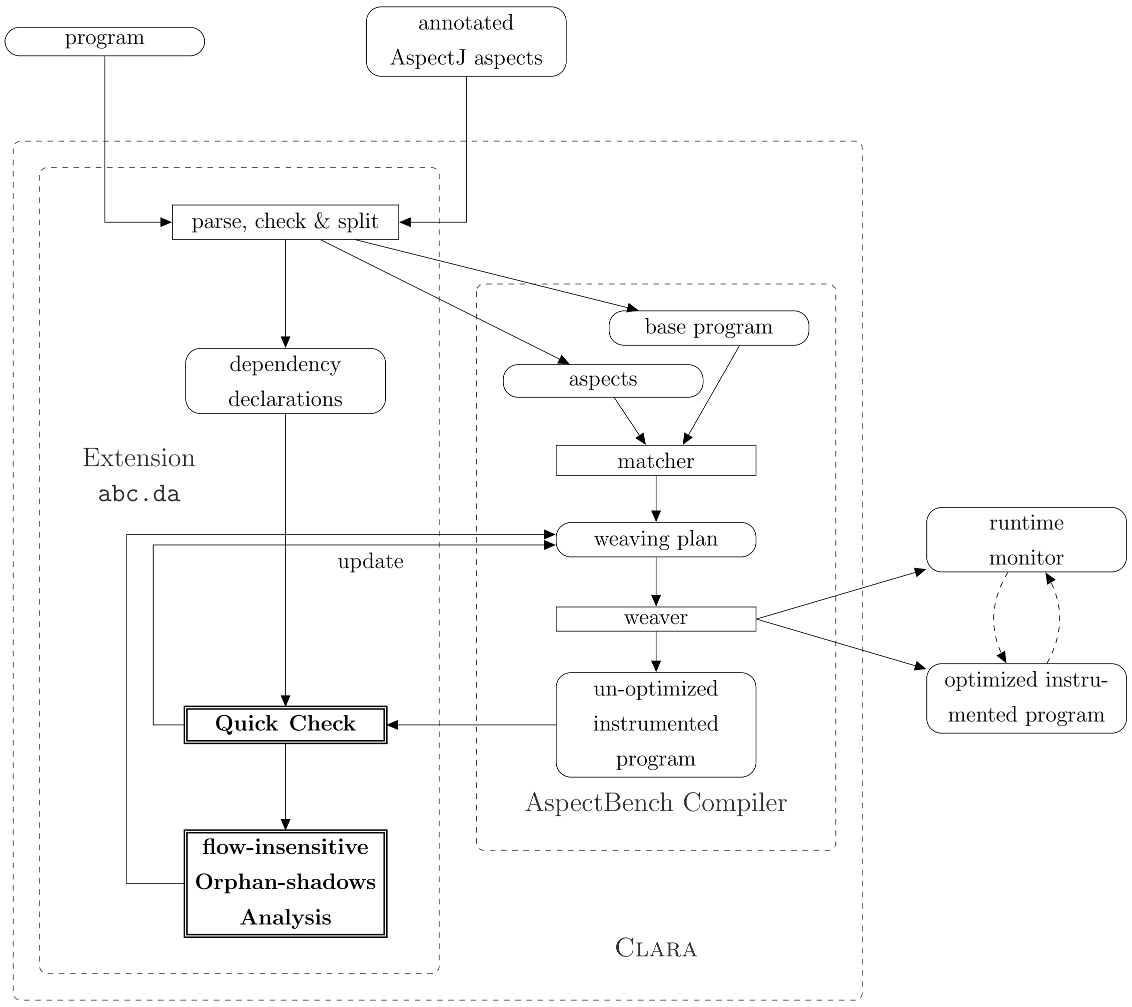


Figure 4.5: Overview of our implementation of dependent advice

we present two such static analyses, the Quick Check and the Orphan-shadows Analysis. After the analyses finish, we re-weave the entire program, i.e., we instruct *abc* to un-do the previous weaving process and weave the base program again, this time with the updated weaving plan. After the program was re-woven, *abc* automatically emits Java bytecode for the woven (and now optimized) program. We next explain the internals of the static analyses, highlighted in Figure 4.5.

As mentioned earlier, our analysis executes right after weaving, analyzing the woven program. It has access to the base program, all aspects, all dependent advice in these aspects, and abc's weaving plan. The weaving plan $\mathcal{W}$ contains a list of tuples (s, a, r) where s is a joinpoint shadow, a is an advice possibly executing at s, and r the dynamic residue that the runtime will evaluate to determine whether a must indeed execute at a concrete event induced by s.

4.2.3.1 Quick Check

Our analysis iterates through the weaving plan, considering each entry separately, first using the "Quick Check" shown in Algorithm 4.1. The Quick Check changes the residue of an entry $(s, a, r) \in \mathcal{W}$ to (s, a, never) if no advice dependency d activates a at s for the trivial reason that at least one strong advice b in d matches *nowhere* in the entire program, as determined by the weaving plan, line 9. Note that the condition in line 9 depends on whether the algorithm already processed weaving-plan entries for b itself. We therefore iterate Algorithm 4.1 until a fixed point is reached. The Quick Check is "quick" because it does not require points-to information. In our benchmarks it therefore always finished in under 3.3 seconds.

If active advice applications remain after the Quick Check, then we apply Sridharan and Bodík's demand-driven refinement-based context-sensitive points-to analysis [SB06] to the woven program. This analysis first produces context-insensitive points-to sets using Spark [LH03]. Then next, when queried for the points-to sets of a local variable l the analysis refines the points-to sets for l with context information. Essentially, this changes the representation of a points-to set from a set $\{s_1, \ldots, s_n\}$

Algorithm 4.1 Quick Check

```
for $(s_a, a, r_a) \in \mathcal{W}$ do // for every entry in the weaving plan
  if $(r_a \neq \text{never}) \wedge (a$ is dependent advice$)$ then
    activated := false
    for $d \in \mathcal{D}$ with $a \in all(d)$ do // for every dependency $d$ referencing $a$
      // see if $d$'s strong advice match
      allStrongAdviceMatch := true
      for $b \in strong(d)$ do
        // if there is no shadow $s_b$ at which $b$ may match, then $b$ never matches
        if $\neg\exists s_b \in \mathcal{S}$ such that $\exists(s_b, b, r_b) \in \mathcal{W}$ with $r_b \neq$ never then
          allStrongAdviceMatch:= false
        end if
      end for
      if allStrongAdviceMatch then
        activated := true
      end if
    end for
    if ¬activated then // no active dependency found; disable $s_a$
      $\mathcal{W} := (\ \mathcal{W} \setminus \{(s_a, a, r_a)\}\ ) \ \cup \ \{(s_a, a, \text{never})\}$
    end if
  end if
end for
```

of allocation sites to a set $\{(c_1, s_1), \ldots, (c_m, s_n)\}$, where the different c_i are static representations of calling contexts. This makes the points-to sets more precise. Context information [SB06] is necessary to optimize pieces of advice that reference objects created inside factory methods, e.g. iterators, which are all produced by a call to the same method `iterator()`. Because we query the analysis only on variables that actually bind values at joinpoint shadows of dependent advice, this demand-driven approach likely executes faster than an analysis that determines context information for every program variable.

4.2.3.2 Flow-insensitive Orphan-shadows analysis

We then apply the flow-insensitive "Orphan-shadows Analysis", as shown in Algorithm 4.2. The algorithm essentially proceeds like the Quick Check (Algorithm 4.2 only shows the differences to Algorithm 4.1), however an advice a only activates a dependency d if every strong advice b of d has a shadow[4] that is compatible with s_a, as determined by *stCompatible*. Again we iterate Algorithm 4.2 until we reach a fixed point. This iteration is not a bottle-neck: in all our experiments we reached the fixed point after two or three iterations. We named the analysis Orphan-shadows analysis because it identifies shadows that are lacking other shadows to activate any dependency.

4.2.3.3 Implementation details

In the following we note some implementation details with respect to the way in which we query the demand-driven refinement-based points-to analysis. This will aid researchers in reproducing our results.

Query parameters. Programmers can parameterize Sridharan and Bodík's demand-driven refinement-based context-sensitive points-to analysis [SB06] with the following values:

[4]Note that at this point, due to the Quick Check, we know that there are shadows for b in general. The Orphan-shadows Analysis determines whether the shadows are compatible with the ones for a.

Algorithm 4.2 Flow-insensitive Orphan-shadows analysis
(only showing differences to Algorithm 4.1)

```
...
 9:        if $\neg\exists s_b \in \mathcal{S}$ such that $\exists(s_b, b, r_b) \in \mathcal{W}$ with
             $r_b \neq$ never $\wedge$ $stCompatible(s_a, a, s_b, b, d)$ then
10:            $allStrongAdviceMatch$ := false
11:        end if
...
```

traversal Make the analysis traverse at most this number of nodes per query. This quota is evenly shared between multiple passes (see next option). (default: 75000)

passes Perform at most this number of refinement iterations. Each iteration traverses at most (traverse / passes) nodes. (default: 10)

For both values, we used their default settings.

Lazy points-to sets. Depending on the size of the analyzed program, the points-to analysis that we use may take several seconds to answer a “query”, i.e., to compute the points-to set of a single variable. Hence, we tried to lower the analysis time by making sure to only query it on variables that matter to us. We added special “lazy” points-to sets to Soot that first content themselves with only context-insensitive information. We only compute context information for lazy points-to sets p_1 and p_2 in the case where we test for non-empty intersection of p_1 and p_2 and the context-insensitive information is not sufficient to prove that $p_1 \cap p_2 = \emptyset$ (either because indeed the variables associated with these sets could point to the same objects at runtime, or because the information is too imprecise). We then compute context-sensitive information. This information may or may not be sufficient to prove that $p_1 \cap p_2 = \emptyset$.

Unfortunately, we found that nevertheless context information was necessary and did have to be computed in the majority of cases. In our benchmark set, we found that context information did have to be computed for 91% of the points-to sets in average, with a maximal number of 1179 points-to sets for bloat-FailSafeIter. For 30 out of the 72 cases to which we had to apply the Orphan-shadows Analysis we needed to compute context information for all the variables for that we compute points-to sets.

Singleton collections and iterators. We found another low-level optimization to be necessary in some cases. Some special methods in the Java Runtime Library return singleton objects. For instance, the runtime library returns a singleton called `emptyEnumerator` when calling `elements()` on an empty `java.util.Hashtable` object. This causes problems for our our analysis in patterns like FailSafeEnumHashtable, where the analysis needs to distinguish different hash table objects to successfully prove that the program under test uses the enumerations over these hash tables correctly. Unfortunately, the runtime library uses the following code (taken from `Hashtable`) to determine whether or not to return the singleton enumeration:

```
private Enumeration getEnumeration(int type) {
    if (count == 0) {
        return (Enumeration)emptyEnumerator;
    } else {
        return new Enumerator(type, false);
    }
}
```

Our pointer analysis completely abstracts from predicates such as `count == 0`. Hence, for any two variables `v1`, `v2` that could point to a `Hashtable` object, the analysis always has to assume the possibility that `v1` and `v2` could indeed point to the same object. A similar problem arises with the objects returned from factory methods such as `Collections.emptySet()`. The return set is the singleton instance of type `java.util.Collections.EmptySet`. However, in this case the problem is less severe: here the analysis will only suffer from this imprecision for variables that can indeed be assigned the return value of a call to `Collections.emptySet()`. In case of the hash table, any hash table could return the singleton enumeration object.

When looking into the problem, it occurred to us that a singleton enumerations or collection is behaviorally equivalent to a set of freshly created enumerations, respectively collections.[5] Hence we decided to add a new option "`-p cg.spark empties-as-allocs`" to Soot, which makes the points-to analysis believe that the singleton objects in these particular cases are indeed creation sites of fresh enumerations and collections. This in turn allows the refinement-based context-sensitive analysis to compute context information for these allocation sites. This optimization proved to be necessary but also sufficient in the cases that indeed do reason about hash maps and enumerations. For instance, when "`empties-as-allocs`" is enabled, for lucene-FailSafeEnumHashtable, the Orphan-shadows Analysis manages to disable all shadows, proving the program sound with respect to this property. With the option disabled the Orphan-shadows Analysis failed to remove a set of nine shadows, which could then not even be ruled out by the Nop-shadows Analysis that we present in the next chapter.

4.3 Generating dependent-advice declarations from finite-state specifications

The above optimizations assumed dependency annotations in the code. Programmers may write dependency annotations by hand, but this can be time consuming and error prone. Fortunately, programmers often opt to have history-based aspects generated automatically, from finite-state monitor specifications. As we showed in Chapter 3, runtime-monitoring tools generate a runtime monitor from such a specification, in the form of a finite-state machine (or an equivalent data structure), along with an aspect that triggers state transitions when monitored events occur. The state machine then executes a user-defined piece of code when those transitions drive it into a final state. As we mentioned, if specifications bind free variables, there exists one state-machine instance per variable binding.

[5]This is not quite true for empty collections. For instance, a program will throw an exception when the programmer tries to add something to a `java.util.Collections.EmptySet`. However, our control-flow graphs cater for such exceptional control-flow.

The remainder of this section is structured as follows. First, in Section 4.3.1, we explain a general algorithm to generate dependency declarations from a finite-state machine. To ease the explanation of this algorithm and also its soundness proof (in Appendix A), we introduce a special automaton model that we call lazy state machines. In Subsection 4.3.2 we explain how we apply the algorithm within JavaMOP to generate dependency declarations from finite-state specifications denoted in the ERE, FTLTL and PTLTL specification languages. In Subsection 4.3.3 we explain the case of tracematches similarly.

4.3.1 Generation from finite-state machines

We next present a general algorithm that exploits domain knowledge in a given finite-state specification to generate sound dependency annotations automatically.

Definition 4.2 (Finite-state machine)**.** A non-deterministic *finite-state machine* $\mathcal{M}$ is a tuple $(Q, \Sigma, q_0, \Delta, Q_F)$, where Q is a set of states, Σ is a set of input symbols, q_0 the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ the transition relation and $Q_F \subseteq Q$ the set of accepting (or final) states. For this dissertation we assume that $q_0 \notin Q_F$. In other words, the finite-state machine will never accept the empty word. This is consistent with the semantics of runtime monitoring in general: a runtime monitor reports a property violation when a final state is reached; it makes no sense to assume a program as erroneous right from the starting state[6].

Definition 4.3 (Words, runs and acceptance)**.** A *word* $w = (a_1, \ldots, a_n)$ is an element of Σ^*. We define a *run* ρ of $\mathcal{M}$ on w to be a sequence $(q_0, q_{i_1}, \ldots, q_{i_n})$ such that $\forall k : (0 \leq k < n) \rightarrow (q_{i_k}, a_{k+1}, q_{i_{k+1}}) \in \Delta$, with $i_0 := 0$. A run ρ is *accepting* if $q_{i_n} \in Q_F$. We say that $\mathcal{M}$ accepts w, $w \in \mathcal{L}(\mathcal{M})$, if there exists an accepting run of $\mathcal{M}$ on w. Since we assume $q_0 \notin Q_F$, both accepted words and accepting runs are non-empty, i.e., $n \geq 1$.

[6]Note that even when monitoring liveness properties like in the LTL formula "Finally p", the monitor would not move into a final, i.e., violating, state before the end of the program is reached.

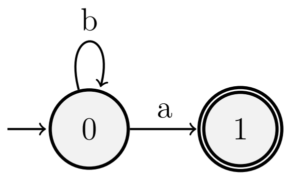
The concepts behind determining strong and weak advice. In this dissertation we assume that finite-state machines are the basis for synthesizing runtime monitors. Our static analyses optimize the evaluation of these runtime monitors by preventing certain events from being passed to the monitor when these events are provably not of interest to the monitor, possibly depending on the monitor's current internal state. In this chapter we focus on dependent advice to encode relationships between input symbols that require monitoring.

As defined above, a finite-state runtime monitor always accepts a word (i.e. identifies an erroneous execution) using an accepting run through a state machine. Statically, this means that the monitor can accept a word using multiple different paths, each ending in an accepting state. To guarantee that optimizations are correct, we need to take two different kinds of dependencies into account along each such path.

Firstly, we need to enable every symbol that makes progress on this path, i.e. brings us closer to the final state. Enabling such "progress symbols" assures soundness: it guarantees that the optimized monitor will not miss any actual property violations. In the dependency declaration for this path we will reference such symbols as strong. After all, one needs to see *all* symbols along the path to reach the final state along this particular path.

Secondly, if all (strong) "progress symbols" match on this path, then the monitor can indeed reach the final state via this path. When this is the case then we also need to activate all symbols that may reject a word along this path. This guarantees that the runtime monitor will not produce any false warnings, i.e., that it will only execute its action handlers at events at which also the un-optimized monitor would have executed it. In our dependency declaration we can reference these symbols as weak: they only need to be active if the monitor can produce a complete match along this path, i.e., if all strong symbols for this path are active as well.

How to determine weak symbols is a non-trivial question. A simple possibility would be to just define that all symbols in the alphabet be weak that are not already strong. However, this would give away optimization potential. Consider the following finite-state machine that monitors for event traces that match the expression b^*a:



Clearly, in this example b is not necessary for a pattern match: it suffices to read an a to reach state 1. Hence, it is sound to disable the monitoring of b altogether. Effectively this would mean that instead of monitoring b^*a over the alphabet $\{a, b\}$ one only would monitor the expression a over the alphabet $\{a\}$. Therefore, in this example we would want to avoid referencing b as a weak symbol in our dependency declaration. The general rule is that along each accepting path, if a symbol, like b, is not a progress symbol (it does not lead towards the final state) and there is a b loop on every state along this path (not taking the final state itself into account), then b does not need to be monitored along this path. Conversely, if there was a state with no b loop then b would have to be monitored: when reading a b in this state, the monitor would have to exit the state to implement the correct semantics.

The issue of loops at final states. There is one notable exception to this rule. Consider now the finite-state machine shown in Figure 4.6 instead. It monitors for traces of the form b^*ab^*. In this example, there exist an additional b-loop at the final state 1. Again, the symbol b makes no actual progress towards reaching the final state, it never changes a state at all. Hence, b is not a strong symbol in this example. In terms of our general rule above, b would not even be weak: there is a b loop on every state along the accepting path. This actually makes sense because in terms of pure language theory it would be sound to not consider b in this example: an event trace is matched by the regular expression b^*ab^* over the alphabet $\{a, b\}$ if and only if it is matched by the regular a over the alphabet $\{a\}$. The problem however, in terms of runtime monitors, it is not true that an event trace is matched by the first regular expression *when* and only *when* it is matched by the second one. The above monitor ought to execute its validation handler two times when reading the sequence "ab". By removing b from the monitor's alphabet the monitor would only see an event trace "a" and hence execute the handler only once. Because we want our static optimizations to preserve the behavior of the instrumented program, we need to take special care in these cases.

The problem with the above case is that the b loop on the final state has an effect (it triggers the handler), *although* it has no effect in terms of whether the state machine accepts the current trace. As we will describe later, the algorithm that we derived for generating dependency annotations hence caters for loops on final states in a special way.

Lazy state machines. A second problem with the traditional model of non-deterministic finite-state machines is the following inverse relationship between loops and a state machine's operations. When a non-deterministic finite-state machine is in a state q and reads a symbol a then (1) it *does* actively move out of q if there is *no* a loop on q; on the other hand, (2) reading a has *no* effect if there *is* an a loop on q. Hence, the transitions in the state machine do not really map one-to-one to operations in the implementation of this state machine in the runtime monitor. If a transition is a loop, then the monitor needs to perform no operation. On the other hand, if there is no such transition on the current state then the monitor needs to actively move out of the state to discard the partial match appropriately.

This inverse relationship between transitions and actions complicates the formulation of our analyses. For instance, consider the non-deterministic finite-state machine shown in Figure 4.7a. Assume that this finite-state machine is used as a runtime monitor, over a program in which the event c can never occur. For such a program it holds that the state machine in Figure 4.7a does not need to monitor b either. This is the case because, when c cannot occur, then the automaton can only accept words of the form b^*ab^*d, and in this case, the occurance of b in a word w does not matter to decide whether or not this automaton accepts w. The problem is that this property is not obvious by just looking at the state machine in Figure 4.7a.

We hence introduce in this thesis a new, equivalent automaton model that we call lazy state machines. Figure 4.7b shows the state machine that is equivalent to the non-deterministic finite-state machine from Figure 4.7a. In this automaton model, state machines have no self-loops. Instead, their states are labeled with a set of

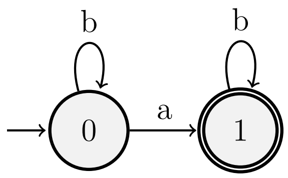
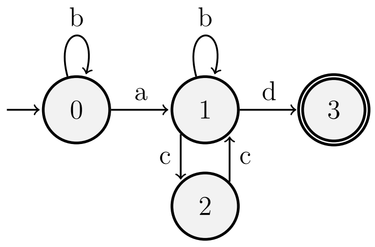
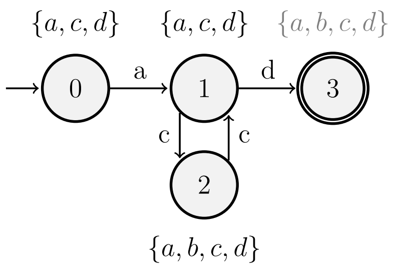


Figure 4.6: Finite-state machine with loop at final state



(a) Non-deterministic finite-state machine



(b) Equivalent lazy finite-state machine

Figure 4.7: Traditional non-deterministic vs. lazy state machines

exit labels that tell when to exit the annotated state. In this model, the property mentioned above is easy to determine. When c cannot occur in a program, then state 2 cannot be reached, and hence no reachable state nor reachable transition references b (except the final state, which does not need to be considered). Hence, in this scenario, b does not need to be monitored either.

Definition 4.4 (Lazy state machine)**.** A lazy state machine (LSM for short) $\mathcal{M}$ is a non-deterministic finite-state machine with a non-standard acceptance conditions and in which states are labeled with an *exit set* of symbol names. Formally, the lazy state machine $\mathcal{M}$ is a tuple $(Q, \Sigma, q_0, \Delta, Q_F, \mathit{exit})$ where Q, Σ, q_0, Δ and Q_F are defined as for traditional non-deterministic finite-state machines, and *exit* is defined as follows. *exit* is a labeling function of type $Q \to \mathcal{P}(\Sigma)$, which assigns a subset of input symbols to each state in Q. For every state $q \in Q$ we call $\mathit{exit}(q) \subseteq \Sigma$ the *exit set* of q. In addition, the transition relation Δ has to adhere to the following "no-self-loop condition": $\forall q \in Q\ \forall a \in \Sigma\ \neg\exists(q, a, q) \in \Delta$.

The crucial difference between lazy and traditional finite-state machines is that lazy state machines are, by default, lazy: they remain in a state q until reading a symbol that is in q's exit set. This is also the reason for why lazy state machines have no self-loops. They are simply not needed because a lazy state machine resides in its current state(s) by default. Note that lazy state machines do allow loops that span multiple transitions.

Definition 4.5 (Words, runs and acceptance for lazy state machines)**.** As earlier, a *word* $w = (a_1, \ldots, a_n)$ is an element of Σ^*. We define a *run* ρ of a lazy state machine $\mathcal{M}$ on w to be a sequence $(q_0, q_{i_1}, \ldots, q_{i_n})$ such that $\forall k \ : \ (0 \leq k < n) \rightarrow \big((q_{i_k}, a_{k+1}, q_{i_{k+1}}) \in \Delta \ \vee \ (q_{i_k} = q_{i_{k+1}} \wedge a_{k+1} \notin \mathit{exit}(q_{i_k})\big)$, with $i_0 := 0$. A run ρ is *accepting* if $q_{i_n} \in Q_F$. We say that $\mathcal{M}$ accepts w, $w \in \mathcal{L}(\mathcal{M})$, if there exists an accepting run of $\mathcal{M}$ on w.

From non-deterministic state machines to lazy state machines and back. For every non-deterministic finite-state machine there exists a lazy state machine that accepts the same language, and the other way around. Moreover the conversion is straightforward. A non-deterministic finite-state machine $\mathcal{M}$ exits a state q when reading a symbol a if it has no a-self-loop (q, a, q). Hence, we can define a lazy state machine, equivalent to $\mathcal{M}$, as follows.

Definition 4.6 (Lazy state machine for a non-deterministic finite-state machine)**.** Let $\mathcal{M} = (Q, \Sigma, q_0, \Delta, Q_F)$ be a non-deterministic finite-state machine. We define the lazy state machine $\mathcal{M}^{\text{lazy}}$ as $\mathcal{M}^{\text{lazy}} := (Q, \Sigma, q_0, \Delta^{\ell}, Q_F, \mathit{exit})$, with Δ^{ℓ} and *exit* defined as:

$$\begin{aligned} \Delta^{\ell} &:= \{(q_s, a, q_t) \in \Delta \mid q_s \neq q_t\} \\ \mathit{exit} &:= \lambda q \ . \ \{a \in \Sigma \mid \neg\exists (q, a, q) \in \Delta\} \end{aligned}$$

The inverse conversion is easy as well:

Definition 4.7 (Non-deterministic finite-state machine for a lazy state machine)**.** Let $\mathcal{M} := (Q, \Sigma, q_0, \Delta^{\ell}, Q_F, \mathit{exit})$ be a lazy state machine. We define the non-deterministic finite state machine $\mathcal{M}^{\text{busy}}$ as $\mathcal{M}^{\text{busy}} = (Q, \Sigma, q_0, \Delta, Q_F)$, where Δ is defined as:

$$\Delta := \Delta^{\ell} \cup \{(q, a, q) \mid q \in Q, a \in \Sigma, a \notin \mathit{exit}(q)\}$$

By construction, these finite-state machines are equivalent:

Theorem 4.2. (Equivalence of non-deterministic and lazy state machines) For every non-deterministic finite-state machine $\mathcal{M}$ it holds that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}^{\text{lazy}})$. For every lazy state machine $\mathcal{M}$ it holds that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}^{\text{busy}})$.

The theorem can be proven correct very easily, by comparing the possible runs of both state machines. We leave the proof to the reader. □

For the remainder of this section we assume that the non-deterministic finite state-machine that we obtain from the finite-state specification has been converted to a lazy state machine as explained above.

Algorithm 4.3 defines the function *generateDependencies*, which generates dependency declarations from a lazy state machine $\mathcal{M}$. The algorithm generates a single dependency declaration d for every accepting path in $\mathcal{M}$, where we denote d by (s, w) if $s = \mathit{strong}(d)$ and $w = \mathit{weak}(d)$. When $\mathcal{M}$ accepts a word using a path p, we need to reference the symbols at transitions on p as strong: events labeled with these symbols all need to occur for $\mathcal{M}$ to accept the word along this path. Conversely, it suffices to reference the exit symbols of states on p as weak: events that are necessary to exit a state on p need to be monitored only if $\mathcal{M}$ can reach a final state using p, i.e., if events for all of p's transitions (all strong symbols) can occur.

The first argument to *generateDependencies* is a current state q. The second argument is an accumulator that holds all states on the current path leading up to q, the second argument is an accumulator that holds all symbols at transitions on this path. The programmer initializes the algorithm by calling *generateDependencies*$(q_0, \epsilon, \epsilon)$, where ϵ is the empty word. Intuitively, *generateDependencies* recursively explores $\mathcal{M}$ in a depth-first manner to find all paths p through $\mathcal{M}$ that satisfy the following two conditions: (1) the path ends in an accepting state (line 2), and (2) the path does not visit a state more than twice (line 1)[7]. When *generateDependencies* finds such a path, it adds a new dependency declaration to the global set $\mathcal{D}$. The dependency references the labels of all edges on p as strong. Further, it references all those symbols as weak that are not already strong but are either exit symbols of one of p's states or symbols that loops on the final state (lines 3–7). The latter (looping) symbols could cause the monitor's handler to re-execute and therefore it would be unsound to omit them from the set *weak*.

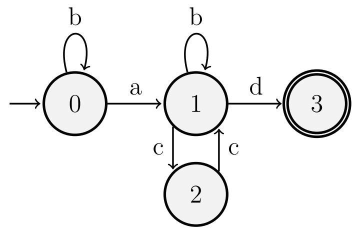
Figure 4.8 shows an example run of Algorithm 4.3. Part 4.8a shows the non-deterministic finite-state machine $\mathcal{M}$ that we saw earlier, and part 4.8b shows again the lazy counterpart $\mathcal{M}^{\text{lazy}}$. 4.8c shows the two paths P1 and P2 that Algorithm 4.3 discovers. Note that the paths exclude the final node 3 (shown dashed). 4.8d shows the two resulting dependency declarations: D1 for P1 and D2 for P2. D1 does not reference b because b occurs neither as a transition label nor within an exit set along P1. D2, however, includes b because b is contained in the exit set of state 2: if $\mathcal{M}$ reads b while in state 2, $\mathcal{M}$ will discard the partial match.

Our static approximation of dependent advice (Section 4.2) can make use of these generated dependencies as follows. Assume a program in which the advice that normally triggers symbol c never matches at any joinpoint shadow. c is necessary to reach the accepting state via P2—that is why c is strong in D2. Because c is strong in D2, and c never matches, D2 is not activated for this program. The symbol b is only referenced by D2; hence, when D2 is inactive, there's no active dependency referencing b. This means that for this program it is safe to not monitor b, i.e., it suffices to monitor a and d only. (The reader be reminded that we said that c does not match in this program and hence c is not monitored anyway, although D1 references c weakly.)

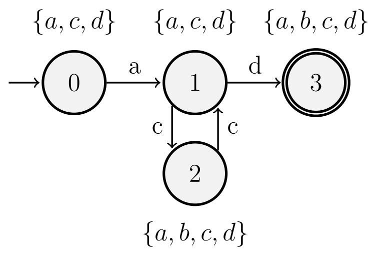
[7]Condition (2) assures termination in the case where the finite-state machine has loops. It assures that on the computation of every path, every edge will be visited only once. In Appendix A we prove that it is indeed correct to consider such paths only, because every longer accepting paths can be broken down into smaller fragments, and the dependencies induced by these fragments are equivalent to those of the original path.

Algorithm 4.3 *generateDependencies*(q, qs, as), with $q \in Q, qs \in Q^*, as \in \Sigma^*$

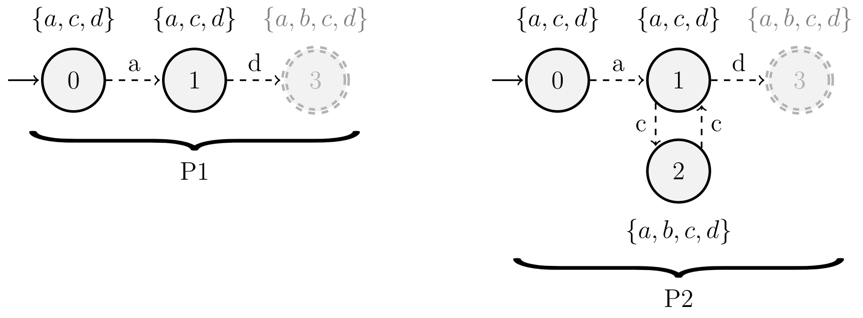
```
// algorithm is initialized by calling $generateDependencies(q_0, \epsilon, \epsilon)$
// dependencies will be added to the global set $\mathcal{D}$, as they are discovered
Global variable: $\mathcal{D} := \emptyset$
1: if $q$ occurs at most twice in $qs$ then // if processed edge at most once before
2:     if $q \in Q_F$ then // if reached final state, add dependency declaration
3:         $strong := \bigcup_{a \in as} \{a\}$ // all symbols $a$ from edges on this path
4:         $finalLoopSyms := \Sigma - exit(q)$ // all symbols that loop on final state $q$
5:         $exitSyms := \bigcup_{q' \in qs} exit(q')$ // all other exit symbols of states on this path
6:         $weak := (exitSyms \cup finalLoopSyms) - strong$
7:         $\mathcal{D} := \mathcal{D} \cup \{(strong, weak)\}$
8:     end if
9:     for $a \in \Sigma, q' \in Q$ such that $(q, a, q') \in \Delta$ do // for every outgoing edge
10:        $qs' := qs \cdot q$ // add current state to path accumulator
11:        $as' := as \cdot a$ // add current symbol to path accumulator
12:        $generateDependencies\ (q', qs', as')$ // recurse
13:    end for
14: end if
```



(a) Example non-det. state machine $\mathcal{M}$



(b) Equivalent lazy state machine $\mathcal{M}^{\text{lazy}}$



(c) Paths that *generateDependencies* determined for $\mathcal{M}^{\text{lazy}}$

for P1 : **dependency**{ **strong** a,d; **weak** c; } *//D1*

for P2 : **dependency**{ **strong** a,c,d; **weak** b; } *//D2*

(d) Dependency declarations generated for $\mathcal{M}$, respectively $\mathcal{M}^{\text{lazy}}$

Figure 4.8: An example run of Algorithm 4.3

Correctness and Complexity of Algorithm 4.3. One can prove that Algorithm 4.3 is correct, meaning that it generates dependency declarations that are both sound and complete, or in other words, the runtime monitor without dependency annotations accepts exactly the same words as the same monitor augmented with dependency annotations. The proof is quite lengthy and hence we present it in Appendix A.

Let us now consider the theoretical worst-case complexity of Algorithm 4.3. We designed the algorithm in such away that it visits every transition in Δ at least once, but at most twice. The initial state q_0 is fixed; there is no choice possible. As the generation of dependency declarations only happens at final states, the complexity of the algorithm is clearly dominated by the number of possible recursive descents. On its first recursive descent, the algorithm has at most $|\Delta|$ transitions to choose from, that lead out of the current state q (line 9). In the worst case, a fully connected graph but without self-loops, $|\Delta|$ could be as large as $|\Sigma| \cdot (|Q| - 1)$. Then, on every recursive descent, the algorithm can repeat this choice of transitions. However, the number of recursive descents on every path is bounded: every state must occur at most twice in qs, and as soon as it is seen a third time, line 1 will abort the recursive descent for that path. Hence, there can be at most $2 \cdot (|Q| - 1)$ recursive descents. Therefore, the number of calls to *generateDependencies* is bounded by the following number:

$$1 + \big(|\Sigma| \cdot (|Q| - 1)\big) \;\cdot\; \big(2 \cdot (|Q| - 1)\big) \quad = \quad 1 + (2 \cdot |\Sigma|) \cdot (|Q| - 1)^2 \quad \in \quad O(|\Sigma| \cdot |Q|^2)$$

Table 4.1 (page 106) shows this upper bound for reasonable values of $|\Sigma| \leq 5$ and $2 \leq |Q| \leq 10$. (The reader be reminded that our automata have at least two states, as $q_0 \notin F$ and we assume that $F \neq \emptyset$). In practice we have found that for usual specifications, Δ will be a lot smaller than its theoretical worst-case size: Algorithm 4.3 never generated more than 48 dependencies for our example specifications. It always terminated within milliseconds. As it turns out, one can still significantly reduce the number of generated dependency declarations.

Formula: $1 + (2 \cdot |\Sigma|) \cdot (|Q| - 1)^2$

	1	2	3	4	5	$\leftarrow |\Sigma|$
2	3	5	7	9	11	
3	9	17	25	33	41	
4	19	37	55	73	91	
5	33	65	97	129	161	
6	51	101	151	201	251	
7	73	145	217	289	361	
8	99	197	295	393	491	
9	129	257	385	513	641	
10	163	325	487	649	811	
$|Q| \uparrow$						

Table 4.1: Theoretical worst-case complexity of Algorithm 4.3 by example

Eliminating redundant advice dependencies

Algorithm 4.3 may generate different but equivalent sets of dependency declarations when given different but equivalent finite-state machines as input. For instance, a minimal state machine for a given finite-state property will likely have fewer paths to a final state than a non-minimal state machine expressing the same property may have. In Appendix A we prove that Algorithm 4.3 will always produce equivalent sets of dependency annotations for such state machines. In this section, we are interested in finding a reduced representation of such a set of dependency declarations.

Assume two dependency declarations d_1 and d_2. We define that d_1 implies d_2 when d_1 is active in all cases in which d_2 is active, and when d_1 activates at least all pieces of advice that d_2 activates:

$$d_1 \rightarrow d_2 :\Longleftrightarrow \mathit{strong}(d_1) \subseteq \mathit{strong}(d_2) \wedge \mathit{all}(d_1) \supseteq \mathit{all}(d_2)$$

Based on this definition, we then define a function *reduce* as follows.

Definition 4.8 (Function *reduce*)**.** Let $D \subseteq \mathcal{D}$ be a set of dependency declarations. Then we define:

$$\mathit{reduce}(D) := \{d \in D \mid \neg\exists d' \in D \text{ such that } d' \neq d \wedge d' \rightarrow d\}$$

Note that the resulting set *reduce*(D) is not necessarily the minimal set of dependencies that is equivalent to D. For instance, {**dependency** { **strong** s; **weak** w1; }, **dependency** { **strong** s; **weak** w2; }} is in reduced form but has the minimal representation **dependency** { **strong** s; **weak** w1, w2; }. In our experiments we found that *reduce* can significantly reduce the number of dependencies that CLARA needs to consider and that in practice there is little room for further reduction. As we will show in Section 4.4.1 (page 112), our reduction even resulted in a minimal number of dependencies for all the properties that we consider in this thesis. Note that, in any case, reducing the size of D only saves analysis time, it does not impact CLARA's analysis precision.

Example 4.1 (Eliminating redundant dependencies)**.** As an example, consider the state machine from Figure 4.9, a non-minimal finite-state machine for the HasNext pattern from Figure 1.2 (page 4). Algorithm 4.3 would generate the following set D of dependencies for this state machine:

dependency { **strong** next; **weak** hasNext; } *//d1*
dependency { **strong** next, hasNext; } *//d2*

Here, d_1 is the dependency for the path $q_0q_1q_2$, while d_2 is the dependency for paths of the form $q_0(q_3q_0)^+q_1q_2$. The reduced version of D contains only d_1, because for d_2 it holds that there exists another dependency $d' = d_1$ with $d_1 \rightarrow d_2$. d_1 is active when only `next` matches, and therefore is active at least in the cases in which d_2 is active. Further, d_1 activates all the pieces of advice from d_2. Hence:

reduce(D) = {**dependency** { **strong** next; **weak** hasNext; }}.

CLARA automatically reduces the set of all declared dependencies before CLARA's static analyses commence. Therefore, the programmer or any code generator that provides dependency annotations does not need to bother about eliminating equivalent dependencies on the client side.

Stability. An interesting property of Algorithm 4.3 is also that it is "stable": for equivalent finite-state machines, it generates two equivalent sets of dependencies.

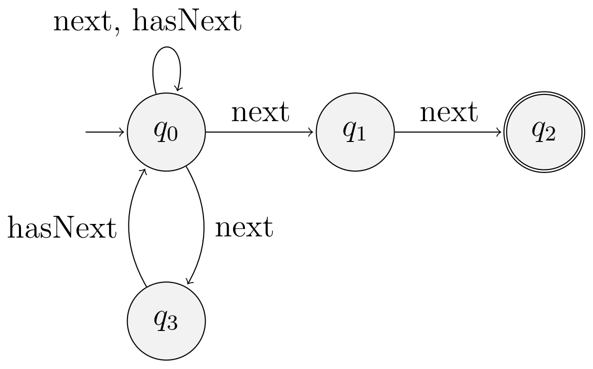


Figure 4.9: Non-minimal finite-state machine for HasNext property

Theorem 4.3 (Stability of Algorithm 4.3)**.** Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two equivalent finite-state machines, i.e. $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the dependencies that Algorithm 4.3 generates for $\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly. Then $\mathcal{D}_1 \equiv \mathcal{D}_2$, i.e., $\mathcal{D}_1$ and $\mathcal{D}_2$ are logically equivalent. We therefore say that Algorithm 4.3 is "stable".

Stability is an interesting property that may be useful beyond the applications within out own work, for instance to reduce finite-state machines for certain purposes. Just as soundness, we prove the stability theorem in Appendix A.

4.3.2 Implementation in JavaMOP

The left-hand side of Figure 4.10 illustrates our implementation in JavaMOP. As noted earlier, we generate dependency declarations from finite-state specifications denoted in the specification formalisms ERE, FTLTL and PTLTL. For this purpose, Feng Chen, one of the developers of JavaMOP, extended the three logic plug-ins for these formalisms such that they convert the domain-specific automaton that they use to generate a runtime monitor into an equivalent standard non-deterministic finite-state machine.

ERE. The monitoring code generated by the ERE plug-in in JavaMOP is already a standard finite-state machine [CR03].

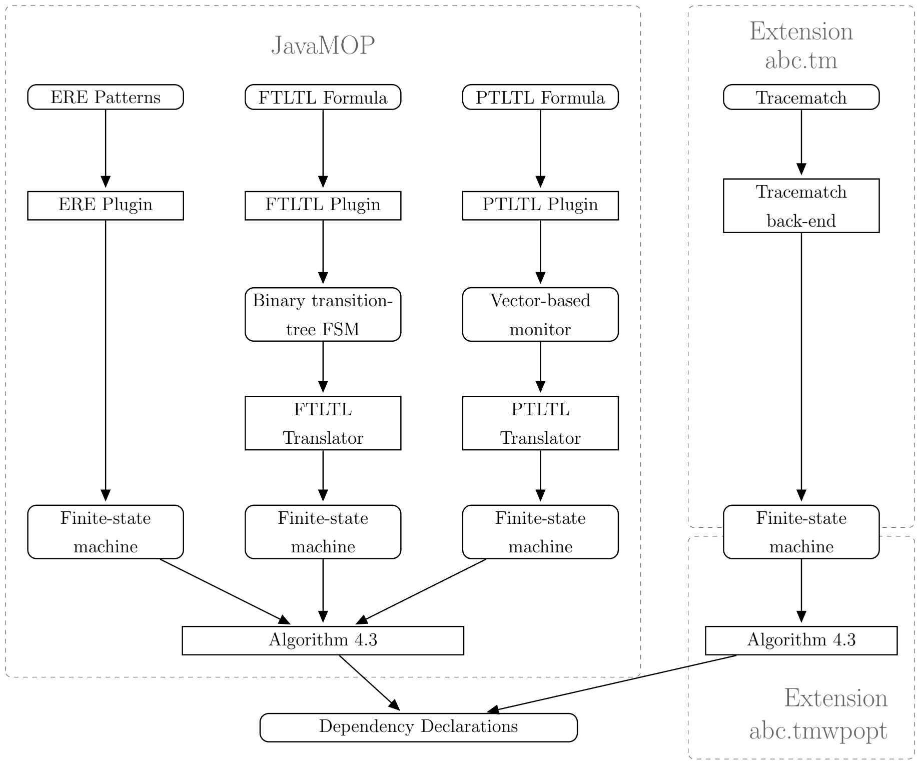


Figure 4.10: Generating dependent advice in JavaMOP and abc

FTLTL. JavaMOP's FTLTL plug-in outputs a binary transition tree finite-state machine (BTT-FSM) [CR03]. A BTT-FSM is a state machine in which each state holds a Binary Transition Tree, i.e., a Boolean function. The BTT-FSM determines the target state of a transition by computing this Boolean function when an event is received. We translate a BTT-FSM into a standard finite-state machine by symbolically computing its BTTs exhaustively in each state. According to Feng Chen, the complexity of this translation is linear in the number of states of the BTT-FSM: the conversion algorithm visits each state of the BTT-FSM only once and converts the binary transition tree of this state into a set of transitions in the finite-state machine in constant time.

PTLTL. Unlike the ERE plug-in and the FTLTL plug-in, the PTLTL plug-in in JavaMOP generates a monitor which has a vector of bits as its internal state [CR03]. We implemented an algorithm to exhaustively explore all possible states of the PTLTL monitor in order to construct an equivalent finite-state machine. Again according to Feng Chen, the conversion algorithm is linear in the number of states of the finite automaton that is used to monitor the PTLTL property. However, in its worst case, it can be exponential in the number of bits occupied by the bit vector. This number of bits is bounded by the number of symbols that are declared in the PTLTL specification.

In practice, the generation of the finite-state machines took never longer than a few milliseconds in our experiments.

JavaMOP next applies the general Algorithm 4.3 to obtain the dependency information from the state machine. As noted earlier, every JavaMOP monitor supports both validation and violation handlers. For a violation handler we instead fix $Q_F := \{q_r\}$, where q_r is the state from which no accepting state can be reached. JavaMOP uses minimized deterministic state machines and therefore q_r is unique, and the property monitored by JavaMOP is violated exactly when q_r is reached. We then emit the appropriate set of dependencies, depending on whether the monitor uses only a validation handler, only a violation handler, or both. (Note that generating dependencies this way is in sync with the semantic definition that we gave for the three finite-state specification formalisms in Section 3.2).

JavaMOP writes AspectJ source code to disk. Our extension to JavaMOP adds dependency declarations to this output and also modifies the output so that each generated piece of advice is given a unique name. The dependency declarations reference those names. In a second step, the programmer can then use the dependent-advice extension `abc.da` to abc (see Figure 4.5, page 88) to read this generated code again from disk and weave monitoring code into a base program of her choice, making full use of the optimizations that we explained in Section 4.2.

4.3.3 Implementation for tracematches

As we explained in Section 3.1, tracematches use yet another data structure to implement their runtime monitors: they use constraints.

JavaMOP's automata are deterministic and minimized, and therefore have a unique reject state (the only state from which no final state can be reached). Tracematches, however, use non-deterministic automata. They actively reject traces using "skip-loops" [AAC+05]. Every state q holds a skip-loop with label a for every a for which q has no "normal" a-self-loop. Note that this means that q holds a skip-loop with label a exactly if $a \in \mathit{exit}(q)$ in the equivalent lazy state machine. Further, because tracematches are matched against each suffix of the execution trace, the tracematch automaton remains always also in the initial state q_0. In tracematches, this is implemented implicitly: q_0 has no Σ-loop, but instead its constraint is constantly set to **true**.

Hence, to prepare the tracematch automaton so that it can be passed as a valid input to Algorithm 4.3, we pre-process the automaton as follows. First, we make the implicit Σ-loop explicit: for each $a \in \Sigma$ we add an a-loop to q_0. We also remove all skip loops from the automaton. Algorithm 4.3 is directly applicable to the resulting state machine.

Another notable difference of our tracematch-based implementation compared to JavaMOP is that for tracematches, we never write AspectJ source code to disk. Tracematches, like dependent advice, are implemented as an extension to abc, and they generate history-based aspects directly in the form of Java bytecode. We therefore enhanced the abc extension "`abc.tm`" for tracematches with another extension "`abc.tmwpopt`" for whole-program optimization (see the right-hand side of Figure 4.10). This extension injects dependency annotations directly into the back-end of our abc extension "`abc.da`" for dependent advice (below the "parse, check & split" in Figure 4.5). Every advice generated from a tracematch already carries a unique name, so we can re-use those names when we generate the dependency declarations.

4.4 Experiments

In this section we report on the effects of both the Quick Check and the Orphan-shadows Analysis. We will see that both analyses can significantly reduce the number of shadows that programmers or later-running analyses need to consider.

4.4.1 Result of eliminating redundant dependencies

Table 4.2 shows the effect of eliminating redundant dependencies. As we explained in Section 4.3.1 (page 106), eliminating redundant dependencies should not affect the precision of the analysis, it should only accelerate it. We therefore checked that our implementation is correct by analyzing all benchmark/tracematch combinations twice, once with our reduction enabled and once with the reduction disabled. We found that indeed the analyses reported exactly the same results in both cases and therefore have no reason to believe that we implemented the reduction incorrectly.

Table 4.2 shows that our reduction has significant effects. For most patterns, only a single dependency remains. This confirms the intuition that we developed in Chapter 2, which tells us that most properties are in fact relatively simple. In particular, most of the properties that we consider here describe one single error situation. Hence it is only natural to describe this situation through a single sequence

	before	after
ASyncContainsAll	6	1
ASyncIterC	14	2
ASyncIterM	28	2
FailSafeEnumHT	4	1
FailSafeEnum	4	1
FailSafeIter	4	1
FailSafeIterMap	10	1
HasNextElem	3	1
HasNext	3	1
LeakingSync	2	1
Reader	48	8
Writer	48	8

Table 4.2: Number of advice dependencies before and after reduction

of events that will lead to an error state. Part of the reason for why the initial number of dependencies was so high is that we added the symbol `newDaCapoRun` to the monitor specifications, as we explained in Section 3.4.3. While this symbol does not introduce any new non-redundant dependencies, the symbol's presence complicates the automaton structure and hence leads to some more dependencies before reduction.

We can observe that larger numbers of dependencies remain for the ASyncIter*, Reader and Writer patterns. The regular expression for both ASyncIter* properties contains a disjunction: it is a bug to create an iterator i for a synchronized collection c without holding c's lock when i is created, but it is also a bug to iterate using i without holding c's lock, even if the program held the lock when it created i. The regular expression for Reader and Writer even concatenates three disjunctions, and hence there are $2^3 = 8$ possible paths to a final state. In result, we can say that, in our benchmark set, the number of dependencies that remain after reduction reflects exactly the number of possible ways in which a program could violate the property at hand. In other words, the reduction that we proposed in Section 4.3.1 is optimal for the finite-state properties that we consider in this thesis.

4.4.2 Effects of applying the Quick Check

We next explain the effects that the Quick Check has on (1) the number of dependencies that need to be considered further, (2) on the number of shadows and (3) on the runtime overhead that the related monitoring code induces.

4.4.2.1 Dependencies eliminated by the Quick Check

The Quick Check can significantly reduce the number of dependencies that need to be considered by further analyses. We quantify this effect in Table 4.3. The second column of the table re-states for every finite-state property the total number of dependencies that remain after reduction (see above) but before applying the Quick Check. The remaining columns then show the total number of dependencies that remain after we applied the Quick Check to the respective program/property combination. As the tables show, the Quick Check is very effective on the ASync* and LeakingSync patterns, eliminating all dependencies for almost all benchmark programs. This is not surprising: because these properties refer to synchronized collections, and many programs, especially single-threaded ones, never produce synchronized collections, the Quick Check suffices to detect that these programs can never violate the properties. The Quick Check is also quite effective on the FailSafeEnum* properties. This is because the `Enumeration` interface was superseded by the `Iterator` interface a long time ago (and vectors by collections) and therefore many programs do not use enumerations any more. antlr and xalan are notable exceptions: as the tables show, they use enumerations, but apparently no iterators. The cases xalan/FailSafeIter* are interesting: xalan does update collections at many places, but at all these places xalan uses collections of type `Vector`. On these vectors, xalan then only uses enumerations, not iterators. This is why the Quick Check is so effective for these two cases. The same holds for hsqldb. For other properties, the Quick Check is only effective occasionally. We require more sophisticated static analyses like the Orphan-shadows Analysis to improve on these cases.

	before	antlr	bloat	chart	eclipse	fop
ASyncContainsAll	1	0	0	0	0	0
ASyncIterC	2	0	0	0	2	0
ASyncIterM	2	0	0	0	2	0
FailSafeEnumHT	1	1	0	0	1	1
FailSafeEnum	1	1	0	0	1	1
FailSafeIter	1	0	1	1	1	1
FailSafeIterMap	1	0	1	1	1	1
HasNextElem	1	1	1	0	1	1
HasNext	1	0	1	1	1	1
LeakingSync	1	0	0	0	1	0
Reader	8	6	0	4	8	2
Writer	8	4	8	0	8	2

	before	hsqldb	jython	luindex lusearch	pmd	xalan
ASyncContainsAll	1	0	0	1	0	0
ASyncIterC	2	0	0	2	0	0
ASyncIterM	2	0	0	0	0	0
FailSafeEnumHT	1	1	1	1	0	1
FailSafeEnum	1	1	1	1	1	1
FailSafeIter	1	1	1	1	1	0
FailSafeIterMap	1	1	1	1	1	0
HasNextElem	1	1	1	1	1	1
HasNext	1	1	1	1	1	0
LeakingSync	1	0	0	1	0	0
Reader	8	8	8	8	5	8
Writer	8	8	4	4	5	8

Table 4.3: Number of enabled dependencies before and after Quick Check

4.4.2.2 Shadows eliminated by Quick Check

If the Quick Check removes all advice dependencies, then of course it also disables all dependent-advice shadows in the program, as there is no active dependency any more that could activate these shadows. In cases where the Quick Check only disables a subset of the dependencies, one may however ask how many of the shadows the Quick Check can disable as a result of disabling the dependencies. As we can see in Table 4.4, and as one would expect, the ratio of disabled shadows does not always correspond directly to the number of disabled dependencies. For instance, for antlr-Reader, the Quick Check removed two out of eight dependencies, but the remaining six dependencies were enough to activate all of the shadows, i.e., the Quick Check could not remove any shadow for antlr-Reader.

4.4.2.3 Runtime overhead after Quick Check

Table 4.5 shows for every benchmark/property combination the runtime overheads (in percent) before and after applying the Quick Check (we omit the values for hsqldb and xalan, as these benchmark show no perceivable overheads). As one would expect, the table shows that after applying the Quick Check there is no perceivable remaining overhead in cases where the Quick Check managed to disable all shadows. In other cases, like antlr-Writer, the Quick Check manages to lower the overhead by large amounts, however not to eliminate it completely.

As our results demonstrate, the Quick Check is essential in showing that a property does not really apply to a given program and that therefore programmers or later-running analyses do not need to consider such a property any further for this program. However, the Quick Check is not powerful enough to prove such programs partially correct (with respect to the given property) that actually do contain shadows for all of the events that the property describes. The Orphan-shadows Analysis can therefore improve on the Quick Check's results by resorting to pointer information.

	antlr		bloat		chart		eclipse		fop	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll	0	0	71	0	6	0	10	0	0	0
ASyncIterC	0	0	1621	0	498	0	214	214	146	0
ASyncIterM	0	0	1684	0	507	0	236	236	176	0
FailSafeEnumHT	133	133	102	0	44	0	217	217	205	205
FailSafeEnum	76	76	3	0	1	0	117	117	18	18
FailSafeIter	23	0	1394	1394	510	510	391	391	288	288
FailSafeIterMap	130	0	1180	1180	374	374	548	548	1374	1374
HasNextElem	117	117	2	2	0	0	89	89	10	10
HasNext	0	0	849	849	248	248	109	109	72	72
LeakingSync	170	0	1994	0	920	0	1325	1325	2347	0
Reader	50	50	7	0	65	33	218	218	102	68
Writer	171	115	563	563	70	0	1045	1045	429	226

	hsqldb		jython		luindex lusearch		pmd		xalan	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll	0	0	31	0	18	18	10	0	0	0
ASyncIterC	33	0	128	0	149	149	671	0	0	0
ASyncIterM	39	0	138	0	152	0	718	0	0	0
FailSafeEnumHT	114	114	153	153	37	37	100	0	319	319
FailSafeEnum	120	120	110	110	61	61	21	21	222	222
FailSafeIter	112	112	253	253	217	217	546	546	158	0
FailSafeIterMap	252	252	250	250	136	136	583	583	540	0
HasNextElem	53	53	64	64	22	22	6	6	63	63
HasNext	16	16	63	63	74	74	346	346	0	0
LeakingSync	528	0	1082	0	629	629	986	0	1005	0
Reader	1216	1216	139	139	226	226	102	102	106	106
Writer	1378	1378	462	227	146	74	62	62	751	751

Table 4.4: Number of enabled shadows before and after Quick Check

	antlr		bloat		chart		eclipse		fop	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll				4			-4			
ASyncIterC			140						5	
ASyncIterM			139							
FailSafeEnumHT	9	10								
FailSafeEnum				5			-4			
FailSafeIter			>1h	>1h	8	9			14	19
FailSafeIterMap			>1h	>1h					7	6
HasNextElem										
HasNext			329	329						
LeakingSync	8		163		91				209	
Reader	30151	29923								
Writer	37779	37727	>1h	>1h					5	5

	jython		luindex		lusearch		pmd	
	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll								
ASyncIterC							28	
ASyncIterM							35	
FailSafeEnumHT	>1h	>1h	32	29				
FailSafeEnum			30	30	18	17		
FailSafeIter			5	5	20	20	2811	2806
FailSafeIterMap	13	12	5	5			>1h	>1h
HasNextElem			12	12				
HasNext							70	71
LeakingSync	>1h		34	35	365	347	16	
Reader					77	72		
Writer								

Table 4.5: Runtime overheads after applying Quick Check, in percent; as before, hsqldb and xalan show no perceivable overheads

4.4.3 Shadows reachable from the program's main class

So far, when we reported the "number of shadows" in a program, then this referred to the number of shadows in both the code that can and cannot be reached when executing the particular benchmark through its main class. Our Orphan-shadows Analysis operates on points-to sets and the points-to analysis that we use computes these points-to sets only over the part of the program that one can reach by executing a user-supplied main class. To establish a baseline for the Orphan-shadows Analysis we will hence only consider the number of reachable shadows in the remainder of our experiments. In Tables 4.6 and 4.7, we therefore report for each property/benchmark combination the number of shadows that is reachable from the benchmark's main class, as determined by the static call graph that Clara computes.

4.4.3.1 Problems due to dynamic class loading

As the reader will notice, we do not include numbers for the benchmark eclipse. This is because we were unfortunately unable to set up Clara in such a way that it would construct a complete and hence sound call graph for eclipse. This is due to problems caused by reflection. Many of DaCapo's benchmarks load classes dynamically, using the method `Class.forName(String)`. (In fact, DaCapo loads the entire benchmarks dynamically.) Without further information, Clara has no way of knowing which particular classes a program could load and execute through such calls. Hence, when analyzing such programs, the programmer must provide Clara with the set of classes that may be loaded through such method calls. It is usually easy to determine these classes, at least for a single program run. We wrote an AspectJ aspect that would print at every call to `forName` and a few other reflective calls the name of the class that this call loads. (We also double-checked with Ondřej Lhoták who compiled such lists of dynamic classes earlier.) In addition, the aspect would also print the location from which the program loads the class, in the form of a URL. Usually this URL will point to a JAR file or directory, and the programmer can then produce a sound call graph by informing Clara about the names of the dynamically loaded classes using Soot's `-dynamic-class` option. In addition, the programmer must add the JAR files

	antlr		bloat		chart		fop		hsqldb	
	all	rch	all	rch	all	rch	all	rch	all	rch
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	133	107					205	170	114	8
FailSafeEnum	76	45					18	12	120	3
FailSafeIter			1394	929	510	160	288	55	112	
FailSafeIterMap			1180	751	374	114	1374	1116	252	8
HasNextElem	117	86	4				12	8	53	6
HasNext			849	565	248	82	72	8	16	
LeakingSync										
Reader	50	41			33		68	3	1216	33
Writer	115	85	563	226			226	70	1378	120

(a) total number of shadows (all) vs. absolute number of reachable shadows (rch)

	antlr	bloat	chart	fop	hsqldb
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	20			17	93
FailSafeEnum	41			33	98
FailSafeIter		33	69	81	100
FailSafeIterMap		36	70	19	97
HasNextElem	26	100		33	89
HasNext		33	67	89	100
LeakingSync					
Reader	18		100	96	97
Writer	26	60		69	91

(b) ratio of unreachable shadows, in percent

Table 4.6: Number of reachable shadows, part 1

	jython		luindex		lusearch		pmd		xalan	
	all	rch	all	rch	all	rch	all	rch	all	rch
ASyncContainsAll			18		18	1				
ASyncIterC			149	24	149	42				
ASyncIterM										
FailSafeEnumHT	153	104	37	21	37	9			319	68
FailSafeEnum	110	71	61	29	61	22	21	13	222	6
FailSafeIter	253	119	217	49	217	62	546	379		
FailSafeIterMap	250	165	136	38	136	48	583	455		
HasNextElem	64	47	22	16	22	6	11	6	63	3
HasNext	63	31	74	12	74	22	346	250		
LeakingSync			629	139	629	260				
Reader	139	50	226	9	226	24	102	77	106	12
Writer	227	13	74	5	74	10	62	21	751	8

(a) total number of shadows (all) vs. absolute number of reachable shadows (rch)

	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		100	94		
ASyncIterC		84	72		
ASyncIterM					
FailSafeEnumHT	32	43	76		79
FailSafeEnum	35	52	64	38	97
FailSafeIter	53	77	71	31	
FailSafeIterMap	34	72	65	22	
HasNextElem	27	27	73	45	95
HasNext	51	84	70	28	
LeakingSync		78	59		
Reader	64	96	89	25	89
Writer	94	93	86	66	99

(b) ratio of unreachable shadows, in percent

Table 4.7: Number of reachable shadows, part 2

and directories that the aspect printed to Clara's classpath. When analyzing our benchmarks we did exactly this. The page `http://www.sable.mcgill.ca/soot/tutorial/usage/` gives an explanation of Soot's command line options. Clara understands all of these command-line options, as does abc.

Unfortunately, eclipse caused problems. Eclipse uses URLs that do not point directly to a JAR file or directory but instead to a location relative to a "OSGi resource bundle". (In a nutshell, eclipse loads classes not from JAR files that are stored in the file system but instead from JAR files within JAR files.) Clara has no notion of such bundles, and hence cannot cope with such URLs. In result, while we could determine the names of the classes that eclipse loads dynamically, we could not determine the exact locations from where eclipse loads these classes. We therefore decided to not report further results on eclipse at all. After all, this case nicely shows the limitations of static whole-program analyses.

4.4.3.2 Problems due to dynamic class generation

Jython exposed a different problem: to interpret a Python program, jython generates code for this program at runtime, which it then executes. This causes problems not only for our static analysis but also even when performing runtime monitoring. After all, jython generates the code at runtime. So how should an AspectJ compiler instrument these classes with runtime-monitoring code ahead of time? Programmers can only solve this problem by resorting to load-time weaving [Tea04]. Unfortunately, abc does not support load-time weaving at this time. Therefore, programmers cannot instrument the runtime-generated code with abc. Because therefore the runtime-generated code will not send any notifications to the runtime monitor anyway, we do analyze jython in our experiments, but restrict ourselves to a generally unsound call graph that does not contain any call edges to the runtime-generated code.

In Section 3.4 we explained that the DaCapo suite has two different workloads, luindex and lusearch, which are both actually two different runs of the same program lucene. Note that luindex and lusearch produce different static call graphs because they use a different set of dynamically loaded classes.

As the data shows, some of the DaCapo benchmarks have unfortunately quite bad code coverage. For instance consider hsqldb in Table 4.6b. As the table shows, almost all the shadows are unreachable for this benchmark.

When Clara determines that a shadow is unreachable from the program's main class then it disables this shadow. Note that this will have no impact on the program's runtime because the program execution will never reach the shadow (assuming that we supplied a correct set of dynamically loaded classes and the call graph is therefore sound).

4.4.4 Effects of applying the Orphan-shadows Analysis

In the following, we discuss the effects of applying the Orphan-shadows Analysis.

4.4.4.1 Shadows eliminated by the Orphan-shadows Analysis

In Tables 4.8 and 4.9 we give, as before for the Quick Check, the number of shadows before and after applying the Orphan-shadows Analysis. Before we conducted these experiments, we already suspected that the Orphan-shadows Analysis would be very effective on the patterns that reason about synchronized collections, namely ASync* and LeakingSync. This is because, for instance in the case of ASyncIterC, even if a program does create a synchronized collection and even iterates over some collection (hence causing the Quick Check to be too imprecise), it is not likely that the program in fact iterates over exactly those collections that it synchronizes. It did however come as a surprise to us that the Orphan-shadows Analysis was so effective on the Reader and Writer patterns, too. We remind the reader that for instance the Reader pattern detects cases in which a program writes to a closed Reader. Indeed we could confirm, through manual inspection, that all of the benchmarks for which the Orphan-shadows Analysis is 100% effective on Reader or Writer (lucene, xalan, fop and jython), indeed fail to close the readers and writers that they use on the runs that the DaCapo benchmark suite produces. The Orphan-shadows Analysis is also very effective on xalan and lucene, in combination with iterators. Those benchmarks iterate over some of the vectors that they use, and they modifies some of their vectors but they do not iterate over vectors that they modify.

As we can see, the Orphan-shadows Analysis is very effective in many cases. However, there are also some cases in which the Orphan-shadows Analysis fails completely. One of these cases is the HasNext property. For this property it is in fact very likely that the Orphan-shadows Analysis will be too imprecise to rule out any shadow in any benchmark. To understand why this is, consider when the Orphan-shadows Analysis could remove a shadow that is induced by the HasNext tracematch. For this to happen, a program would have to use an iterator on which the program only invokes `hasNext` but not `next`. Obviously, this is not likely to happen.

The crucial ingredient that the Orphan-shadows Analysis misses to handle cases like HasNext is flow-sensitivity. Consider the common case in which we have iterators on which both `next` and `hasNext` are invoked. In this case, if our analysis could determine that a given program calls `next` only after a `hasNext` call then our analysis could prove that this program cannot violate the HasNext property. In the next chapter we will present an annotation language that encodes flow-sensitivity and an analysis that makes use of flow-sensitive information. In combination, this third analysis stage will be able to significantly improve on HasNext and other cases.

4.4.4.2 Number of potential failure groups

Instead of just reporting a simple list of remaining shadows, Clara reports a list of groups of inter-connected shadows after finishing the Orphan-shadows Analysis. We call any such group of shadows a "potential failure group" (PFG for short). Every group consists of a "final" shadow that can drive the monitor into a final state, and all (final or non-final shadows) that overlap with this shadow. Shadow groups allow programmers to divide the work of manual inspection into smaller, related units. The number of potential shadow groups can therefore significantly influence the amount of work that the programmer has to perform when inspecting shadows by hand.

Table 4.10 shows the number of potential failure groups for each benchmark/property combination. As one can see the average number of such PFGs is significantly

	antlr		bloat		chart		fop		hsqldb	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	107	30					170		8	3
FailSafeEnum	45	3					12	7	3	
FailSafeIter			929	922	160	160	55			
FailSafeIterMap			751	446	114	49	1116	1114	8	
HasNextElem	86	86					8	8	6	6
HasNext			565	565	82	82	8	8		
LeakingSync										
Reader	41	14					3		33	3
Writer	85	44	226	19			70		120	10

(a) absolute number of shadows

	antlr	bloat	chart	fop	hsqldb
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	72			100	63
FailSafeEnum	93			42	100
FailSafeIter		1	0	100	
FailSafeIterMap		41	57	0	100
HasNextElem	0			0	0
HasNext		0	0	0	
LeakingSync					
Reader	66			100	91
Writer	48	92		100	92

(b) ratio of shadows disabled by Orphan-shadows Analysis, in percent

Table 4.8: Enabled shadows before and after Orphan-shadows Analysis, part 1

	jython		luindex		lusearch		pmd		xalan	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll					1					
ASyncIterC			24		42					
ASyncIterM										
FailSafeEnumHT	104	76	21	15	9	5			68	
FailSafeEnum	71	47	29		22	5	13	10	6	
FailSafeIter	119	116	49	27	62	36	379	302		
FailSafeIterMap	165	151	38		48		455	314		
HasNextElem	47	47	16	16	6	6	6	6	3	3
HasNext	31	31	12	12	22	22	250	250		
LeakingSync			139		260					
Reader	50	4	9		24		77	24	12	
Writer	13		5		10		21	7	8	

(a) absolute number of shadows

	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll			100		
ASyncIterC		100	100		
ASyncIterM					
FailSafeEnumHT	27	29	44		100
FailSafeEnum	34	100	77	23	100
FailSafeIter	3	45	42	20	
FailSafeIterMap	8	100	100	31	
HasNextElem	0	0	0	0	0
HasNext	0	0	0	0	
LeakingSync		100	100		
Reader	92	100	100	69	100
Writer	100	100	100	67	100

(b) ratio of shadows disabled by Orphan-shadows Analysis, in percent

Table 4.9: Enabled shadows before and after Orphan-shadows Analysis, part 2

lower than the average number of shadows that remains after the Orphan-shadows Analysis. For instance, for jython-FailSafeIterMap, 165 shadows remain, but they form only 4 consistent shadow groups. Note that this does not necessarily mean that the programmer would only have to inspect four combinations of maps and iterators in this case, though: the Orphan-shadows Analysis models objects through points-to sets and if these points-to sets are imprecise then this may lead to larger shadow groups that represent multiple such actual combinations.

4.4.4.3 Runtime overhead after Orphan-shadows Analysis

As we show in Table 4.11, the Orphan-shadows Analysis can lower the overhead that runtime monitoring induces, sometimes by a significant amount. In the case of antlr-Writer, for example, the Orphan-shadows Analysis reduced the overhead from more than 377-fold to just about 38%. For fop the analysis eliminated all perceivable overhead in two out of three cases, for luindex in half of the cases and for lusearch in all cases. In case of the HasNext* properties, the Orphan-shadows Analysis failed to reduce the overhead because the analysis failed to disable any shadows in these cases. We can see that in 16 cases, runtime overheads of 10% or more still remain. In the next chapter, we will present a flow-sensitive analysis that will disable yet more shadows and lower the overhead even further.

4.4.4.4 Analysis time

The Quick Check never took longer than one second to execute on any of our benchmarks. For the Orphan-shadows Analysis, it is hard to determine the actual runtime for the actual analysis itself. Because we use lazy points-to sets, context information for these points-to sets may be computed on demand, during the execution of the Orphan-shadows Analysis. Including this computation of context information, the Orphan-shadows Analysis took never longer than 91 seconds on any of our benchmarks. In earlier work [BCR09] we used the Orphan-shadows Analysis on a similar benchmark set, but without lazy points-to sets. Therefore, we could measure the overhead that the Orphan-shadows Analysis causes more precisely in this case, and

	antlr	bloat	chart	fop	hsqldb
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	6				1
FailSafeEnum	1			1	
FailSafeIter		259	38		
FailSafeIterMap		258	38	1	
HasNextElem	41			4	3
HasNext		266	38	3	
LeakingSync					
Reader	4				1
Writer	3	1			1

	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	24	4	2		
FailSafeEnum	2		1	2	
FailSafeIter	4	6	10	90	
FailSafeIterMap	4			32	
HasNextElem	26	8	3	3	2
HasNext	14	6	10	98	
LeakingSync					
Reader	1			5	
Writer				2	
Writer					

Table 4.10: Number of potential failure groups after Orphan-shadows Analysis

	antlr		bloat		chart		fop	
	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll			4					
ASyncIterC								
ASyncIterM								
FailSafeEnumHT	10	4						
FailSafeEnum			5					
FailSafeIter			>1h	>1h	9	8	19	
FailSafeIterMap			>1h	>1h			6	5
HasNextElem								
HasNext			329	329				
LeakingSync								
Reader	29923							
Writer	37727	38	>1h	229			5	

	jython		luindex		lusearch		pmd	
	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll								
ASyncIterC								
ASyncIterM								
FailSafeEnumHT	>1h	>1h	29	28				
FailSafeEnum			30		17			
FailSafeIter			5	4	20		2806	526
FailSafeIterMap	12	14	5				>1h	>1h
HasNextElem			12	12				
HasNext							71	70
LeakingSync			35		347			
Reader					72			
Writer								

Table 4.11: Runtime overheads after applying Orphan-shadows Analysis, in percent; as before, hsqldb and xalan show no perceivable overheads

we determined that the analysis itself never took longer than 17 seconds, with an average of 1.4 seconds.

We first compute context-insensitive points-to sets using Spark [LH03], and then refine these sets with context information on demand. Computing the initial context-insensitive points-to sets can take quite some time: on average, this took two and a half minutes for our benchmark set, and up to almost five minutes in the case of fop-FailSafeIterMap. This may seem like a long time, however, considering the currently available technologies, we believe that any analysis that needs to cope with aliasing on a whole-program level would require a similar points-to analysis. The initialization of Sridharan and Bodík's demand-driven analysis [SB06] may take several seconds, too. Because we initialize the analysis on demand, within our lazy points-to sets, this overhead shows up as part of the computation time for the Orphan-shadows Analysis that we mentioned above. The complete compilation and analysis with both Quick Check and Orphan-shadows Analysis enabled never took longer than nine minutes on any of our benchmarks.

Chapter 5
Flow-sensitive optimizations through Dependency State-machines

In the previous chapter, we showed how static program analyses can exploit dependency information which is preserved in history-based aspects in the form of dependent advice. In particular, dependent advice capture exactly the minimal amount of information that a pointer-based analysis requires to *flow-insensitively* determine whether an aspect that implements a finite-state runtime monitor can reach a final state when applied to a particular program. If the monitor can reach a final state, then the analysis determines the joinpoint shadows that drive the monitor into this state.

While our results from Section 4.4 showed that the use of dependent advice can greatly enhance the performance of the monitoring aspects, the results also showed that the presented analyses are still too coarse grained when it comes to actually proving programs sound at compile time (with respect to the stated finite-state property). For some finite-state specification patterns, like HasNext for example, even the pointer-based Orphan-shadows Analysis was completely unsuccessful in removing shadows, and brought no runtime improvements whatsoever.

Hence, in this chapter, we introduce a third static analysis that is flow-sensitive in nature, i.e., the analysis considers the order in which events occur. To enable this analysis, the annotations added to our history-based monitoring aspects need to be extended: they also need to retain information about the order of events that can drive a finite-state machine into its final state, or in other words its transition relation. The most simple way to encode and preserve the transition relation of a finite-state machine is to simply encode the entire state machine itself, as a list of transitions. Hence, in this chapter, we introduce *Dependency State machines*, our second AspectJ language extension to preserve information about inter-advice dependencies at compile time.

Chapter organization. We organized the remainder of this chapter as follows. In the next section, we describe the syntax and semantics of dependency state machines. Then, in Section 5.2, we explain how CLARA's third static analysis, the Nop-shadows Analysis, can evaluate a history-based aspect using these state machines. The analysis identifies "nop shadows", i.e., shadows that one does not need to monitor, using a novel notion of continuation-equivalent states. In Section 5.3, we expand on additional experiments, which show that this additional analysis is effective in proving many programs sound (with respect to the stated finite-state property) for which the Orphan-shadows Analysis is too imprecise to prove soundness.

5.1 Syntax and semantics of Dependency State machines

In Figure 5.1, we show how the syntactic extension that we provide could be used to denote the state machine for the ConnectionClosed pattern from Figure 4.1 on page 69. On purpose, we kept the syntax similar to the syntax for dependent advice that we saw in the last chapter. What distinguishes the dependency state machine in Figure 5.1 from its dependent-advice counterpart is that (1) the state machine description contains no mention of strong or weak advice, and (2) instead it contains a list of states, along with their transitions, that encode the entire finite-state machine of the finite-state property. Note that this representation is richer than the one for dependent advice; it contains more information. We can easily reconstruct (1) all the information about strong and weak advice by applying Algorithm 4.3 to the encoded state machine, but in addition, we get (2) flow-sensitive information about the order in which events need to occur so that a final automaton state can be reached.

Line 2 contains references to advice names. These can be parameterized with free variables, just as the strong and weak advice names inside the dependency declarations of dependent advice. In this example, the programmer could have written `disconn(c), write(c), reconn(c)` in line 2. As in dependent advice, variables are inferred from the advice declarations when they are not explicitly mentioned. Lines 3–9 enumerate all states in the state machine in question, and for each state a (potentially empty) list of outgoing transitions. An entry "`s1: l -> s2`" reads as "there exists an `l` transition from `s1` to `s2`". In addition, a programmer can mark states as initial or final (in other words, accepting). We give the complete syntax for dependency state machines in Figure 5.2, as a syntactic extension to the dependent-advice language extension to AspectJ that we gave in Figure 4.2 (page 73).

5.1.1 Type-checking dependency state machines

After parsing, we impose the following semantic checks to the dependency state machines supplied in an aspect:

- Every advice must be referenced only by a single declaration of a dependency state machine.
- The state machine must have at least one initial and at least one final state.
- The alphabet may contain every advice name only once.
- The names of states must be unique within the dependency declaration.
- Transitions may only refer to the names of advice that are named in the alphabet of the dependency declaration, and to the names of states that are also declared in the same dependency declaration.
- Every state must be reachable from an initial state.

```
dependency{
    disconn, write, reconn;
    initial s0: disconn -> s0,
                write -> s0,
                reconn -> s0,
                disconn -> s1;
            s1: disconn -> s1,
                write -> s2;
    final   s2;
}
```

Figure 5.1: Finite-state machine for ConnectionClosed pattern (Figure 4.1, page 69), written in the syntax of our syntactic extension

AspectMemberDecl ::= *AdviceDecl* | . . . | *DependencyDecl* | ***DependencySMDecl***.

DependencySMDecl ::=
  “`dependency`” “{” *AdviceNameList* “;” ***StateList*** “;” “}”.

StateList ::= ***State*** | ***State*** ***StateList***.

State ::= ***StateModifier**** *Identifier* [“:” ***TransitionList***] “;”.

StateModifier ::= “`initial`” | “`final`”.

TransitionList ::= ***Transition*** | ***Transition*** “`,`” ***TransitionList***.

Transition ::= *Identifier* “`->`” *Identifier*.

Figure 5.2: Syntax of Dependency State machines, as extension (shown in boldface) to the dependent-advice extension to AspectJ (see Figure 4.2)

Note that these checks are very minimal and allow for a large variety of state machines to be supplied. For instance, we do allow multiple initial states and final states. We also allow the state machine to be non-deterministic. The state machine can have unproductive states from which no final state can be reached, and the state machine even does not have to be connected, i.e., it may consist of multiple components which are not connected by transitions. (In this case, the state machine essentially consists of multiple state machines that share a common alphabet.) To the advice references in the alphabet of the declaration of a dependency state machine we apply the same warnings that we also apply to dependent advice (see page 77).

5.1.2 The semantics of dependency state machines

As we saw in Chapter 3, runtime monitors frequently operate on "parameterized traces", i.e., traces that are parameterized through variable bindings. The HasNext tracematch from Figure 1.2 (page 4), for example, tracks an internal state for every single iterator object on which `next` or `hasNext` is called. The semantics of state machines are usually defined using words over a finite alphabet Σ. In particular, state machines usually have no notion of variable bindings. In the following, we will call traces over Σ, which are given as input to a dependency state machine "ground traces", as opposed to the parameterized trace that the program execution generates. We will therefore define the semantics of dependency state machines over ground traces. We obtain ground traces by projecting parameterized events of the unique parameterized runtime trace onto ground events. This yields a unique ground trace for every variable binding.

For any declaration of a dependency state machine, the set of dependent advice names mentioned in the declaration of the dependency state machine induces an alphabet Σ, where every element of Σ is the name of one of these dependent advice. For instance, the alphabet for the ConnectionClosed dependency state machine from Figure 5.1 would be $\Sigma = \{\mathrm{disconn}, \mathrm{write}, \mathrm{reconn}\}$. When one matches these pieces of advice against a runtime event e, this results in a (possibly empty) set of matches for this event, where each match has a binding attached. We call this set of event matches the parameterized event $\hat{e}$.

Definition 5.1 (Parameterized event)**.** Let $e \in \mathcal{E}$ be an event and Σ be the alphabet of advice references in the declaration of a dependency state machine. We define the parameterized event $\hat{e}$ to be the following set:

$$\hat{e} := \bigcup_{a \in \Sigma} \{(a, \beta) \mid \beta = \mathit{match}(e, a) \wedge \beta \neq \bot\}.$$

We call the set of all parameterized events $\hat{\mathcal{E}}$:

$$\hat{\mathcal{E}} := \bigcup_{e \in \mathcal{E}} \{\hat{e}\}$$

It is necessary to consider sets of matches because multiple pieces of advice can match the same event. While this is not usually the case, we decided to cater for the unusual cases too. As an example, consider the dependency state machine in the UnusualMonitor aspect in Figure 5.3a. The aspect defines a dependency between two pieces of advice `a` and `b`. Note that the pointcut definitions of `a` and `b` overlap, i.e., describe non-disjoint sets of program events. The advice `b` executes before all non-static calls[1] to methods named `foo`. The advice `a` executes before these events too, because by its definition it executes before any non-static method call.

Next assume that we apply this aspect to the little example program in Figure 5.3b. We show the program's execution trace in the first row of Figure 5.3c (to be read from left to right). This execution trace naturally induces the parameterized event trace that we show in the second row of the figure: this trace is obtained by matching at any event every piece of advice against this event.

Next we explain how we use projection to obtain "ground traces", i.e., words of Σ^*, from this parameterized event trace.

Definition 5.2 (Projected event)**.** For every $\hat{e} \in \hat{\mathcal{E}}$ and binding β, we define a projection of $\hat{e}$ with respect to β:

$$\hat{e} \downarrow \beta := \{a \in \Sigma \mid \exists (a, \beta_a) \in \hat{e} \text{ such that } \mathit{compatible}(\beta_a, \beta)\}$$

[1]The calls cannot be static because the advice binds the call target to `x`. For static calls there is no call target, and the `target` pointcut does not match such calls.

```
aspect UnusualMonitor {
    dependency{
        a, b;
        //transitions omitted from example
    }

    dependent before a(Object x): call(* *(..)) && target(x) { ... }

    dependent before b(Object x): call(* foo(..)) && target(x) { ... }
}
```

(a) UnusualMonitor aspect with overlapping pointcuts

```
SomeClass v1 = new SomeClass();
SomeClass v2 = new SomeClass();
v1.foo(); v1.bar(); v2.foo();
```

(b) Example program

execution trace	`v1.foo();`	`v1.bar();`	`v2.foo();`
parameterized trace $\hat{t}$	$\{(a, x = o(\texttt{v1})),$ $(b, x = o(\texttt{v1}))\}$	$\{(a, x = o(\texttt{v1}))\}$	$\{(a, x = o(\texttt{v2})),$ $(b, x = o(\texttt{v2}))\}$
projected traces	a	a	
for $\hat{t} \downarrow x = o(\texttt{v1})$	b	a	
projected traces			a
for $\hat{t} \downarrow x = o(\texttt{v2})$			b

(c) Resulting traces; note that $o(\texttt{v1}) \neq o(\texttt{v2})$

Figure 5.3: UnusualMonitor aspect, example program and resulting traces

The reader be reminded that two bindings are *compatible* as long as they do not bind the same variable to two different objects.

Definition 5.3 (Parameterized and projected event trace)**.** Any finite program run induces a parameterized event trace $\hat{t} = \hat{e}_1 \ldots \hat{e}_n \in \hat{\mathcal{E}}^*$. For any variable binding β we define a set of projected traces $\hat{t} \downarrow \beta \subseteq \Sigma^*$ as follows. $\hat{t} \downarrow \beta$ is the smallest subset of Σ^* for which holds:

$$\forall t = e_1 \ldots e_n \in \Sigma^* : \quad \text{if } \ \forall i \in \mathbb{N} : \ (1 \leq i \leq n) \rightarrow e_i \in \hat{e}_i \downarrow \beta \ \text{ then } t \in \hat{t} \downarrow \beta$$

In the following we will call traces like t, which are elements of Σ^*, "ground" traces, as opposed to parameterized traces, which are elements of $\hat{\mathcal{E}}^*$.

For our example, the third and fourth row of Figure 5.3c show the four traces that result when projecting this parameterized event trace onto the variable bindings $x = o(\texttt{v1})$ and $x = o(\texttt{v2})$. For $x = o(\texttt{v1})$ we obtain the two traces "*aa*" and "*ba*", for $x = o(\texttt{v2})$ we obtain the two traces "*a*" and "*b*". At runtime, there will hence be four monitor instances for this program run, one for every projected event trace.[2]

A dependency state machine will execute its validation handler whenever a prefix of one of the ground traces of any variable binding is in the language described by the state machine. We exclude the empty trace (with no events) because this trace cannot possibly cause the validation handler to execute. This yields the following definition.

Definition 5.4 (Set of non-empty ground traces of a run)**.** Let $\hat{t} \in \hat{\mathcal{E}}^*$ be the parameterized event trace of a program run. Then we define the set *groundTraces*$(\hat{t})$ of non-empty ground traces of $\hat{t}$ as:

$$\mathit{groundTraces}(\hat{t}) := \left(\bigcup_{\beta \in \mathcal{B}} \hat{t} \downarrow \beta \right) \cap \Sigma^+$$

We intersect with Σ^+ to exclude the empty trace.

[2]Note that by the definition of the example program (Figure 5.3b), it holds that $o(\texttt{v1}) \neq o(\texttt{v2})$. If we had had the case $o(\texttt{v1}) = o(\texttt{v2})$, then this would have resulted in a different set of ground traces, namely "*aaa*", "*aab*", "*baa*" and "*bab*".

The semantics of a dependency state machine

We define the semantics of dependency state machines similar to the way in which we defined the semantics of dependent advice. Before, we defined a specialization of the predicate $match(a, e)$, which models the decision of whether or not the advice $a \in \mathcal{A}$ matches at event $e \in \mathcal{E}$, and if so, under which variable binding. For dependent advice, we defined a specialized version of *match*, called *depMatch*. Now we will provide a similar specialization, *stateMatch*.

$$
\begin{aligned}
&\mathit{stateMatch}: \quad \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\} \\
&\mathit{stateMatch}(a, \hat{t}, i) = \\
&\qquad \text{let } \beta = \mathit{depMatch}(a, e) \text{ in} \\
&\qquad\qquad \begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in \mathit{groundTraces}(\hat{t}) \text{ such that } \mathit{necessaryShadow}(a, t, i) \\ \bot & \text{else} \end{cases}
\end{aligned}
$$

As we can see, *stateMatch* takes as arguments not only the piece of advice for which we want to determine whether it should execute at the current event, but also the entire parameterized event trace $\hat{t}$, and the current position i in that event trace. Note that $\hat{t}$ also contains future events, which makes the function *stateMatch* undecidable. Our static optimization, however, will provide a sound approximation of all possible future traces, which makes the approximation decidable again. The function *necessaryShadow* mentioned above is a parameter to the semantics which can be freely chosen, as long as it adheres to a certain soundness condition that we define next.[3]

Soundness condition. The soundness condition will demand that an event needs to be monitored if we would miss a match or obtain a spurious match by not monitoring the event. A dependency state machine $\mathcal{M}$ matches, i.e., executes its validation handler after every prefix of the complete execution trace that is in $\mathcal{L}(\mathcal{M})$.

[3]We wish to emphasize that above we formulated *stateMatch* as a refinement of *depMatch*, not *match*. This reflects the fact that we only apply the flow-sensitive analysis when the flow-insensitive analysis, which determines *depMatch*, fails. We could also have formulated *stateMatch* as a refinement of *match*, but *depMatch* can be determined faster than *stateMatch* and the formulation that we chose here is more practical.

Definition 5.5 (Set of prefixes)**.** Let $w \in \Sigma^*$ be a Σ word. We define the set *pref*(w) as:

$$\mathit{pref}(w) := \{p \in \Sigma^* \mid \exists s \in \Sigma^* \text{ such that } w = ps\}$$

Definition 5.6 (Matching prefixes of a word)**.** Let $w \in \Sigma^*$ be a Σ word and $\mathcal{L} \subseteq \Sigma$ a Σ language. Then we define the matching prefixes of w (with respect to $\mathcal{L}$) to be the set of prefixes of w in $\mathcal{L}$:

$$\mathit{matches}_{\mathcal{L}}(w) := \mathit{pref}(w) \cap \mathcal{L}$$

We will often write *matches*(w) instead of $\mathit{matches}_{\mathcal{L}}(w)$ if $\mathcal{L}$ is clear from the context.

As before, the predicate *necessaryShadow* can be freely chosen, as long as it adheres to the following soundness condition:

Condition 5.1 (Soundness condition for dependency state machines)**.** For any sound implementation of *necessaryShadow*, we demand:

$$\begin{aligned}
&\forall a \in \Sigma \quad \forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+ \quad \forall i \in \mathbb{N}: \\
&\qquad a = t_i \wedge \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n) \\
&\qquad\qquad \longrightarrow \mathit{necessaryShadow}(a, t, i)
\end{aligned}$$

The soundness condition hence states that if we are about to read a symbol a, then we can skip a if the monitor's validation handler would execute on the complete trace t just as often (and at the same points in time) as it would execute on the partial trace where $t_i = a$ is omitted.

5.2 Flow-sensitive Nop-shadows Analysis

In the last section we explained the syntax and semantics of dependency state machines at runtime. As we saw, the function *necessaryShadow* is a free parameter to these semantics, and it can be freely chosen as long as the implementation of *necessaryShadow* adheres to its soundness condition, Condition 5.1. In this section, we will explain a static analysis that approximates the function *necessaryShadow*, while still fulfilling Condition 5.1. We first situate the static analysis in the context of the compilation process for history-based aspects and then explain our static abstraction.

5.2.1 Weaving process

As Figure 5.4 shows, we extended our package `abc.da` so that it generates dependency declarations from dependency state machines present in the source code, using Algorithm 4.3 (*generateDependencies*). The analysis then proceeds as follows. We apply the flow-sensitive Nop-shadows Analysis after first running the Quick Check and the flow-insensitive Orphan-shadows Analysis. Due to this setup, in cases where the Quick Check and the Orphan-shadows Analysis already manage to eliminate all of the shadows, we do not need to run the Nop-shadows Analysis at all. The Nop-shadows Analysis has access to the dependency declarations (through the Orphan-shadows Analysis), but also has direct access to an internal representation of the dependency state machines. This provides the analysis with flow-sensitive dependency information. Just as in the earlier analysis stages, the Nop-shadows Analysis updates the weaving plan after the analysis has completed.

As the figure shows, it is possible to re-iterate the Orphan-shadows Analysis and Nop-shadows Analysis. We found that the flow-sensitive Nop-shadows Analysis may yield further optimization potential for the flow-insensitive Orphan-shadows Analysis, and the other way around. Hence, after the Nop-shadows Analysis completed, if the Nop-shadows Analysis managed to disable any shadows, then we run the Orphan-shadows Analysis and Nop-shadows Analysis again. We iterate this process until a fixed point has been reached, i.e., until no shadows can be disabled any more. All the analyses update the weaving plan. The program is then re-woven using the updated weaving plan after all analyses have finished, yielding an optimized instrumented program (potentially un-instrumented even) and a runtime monitor.

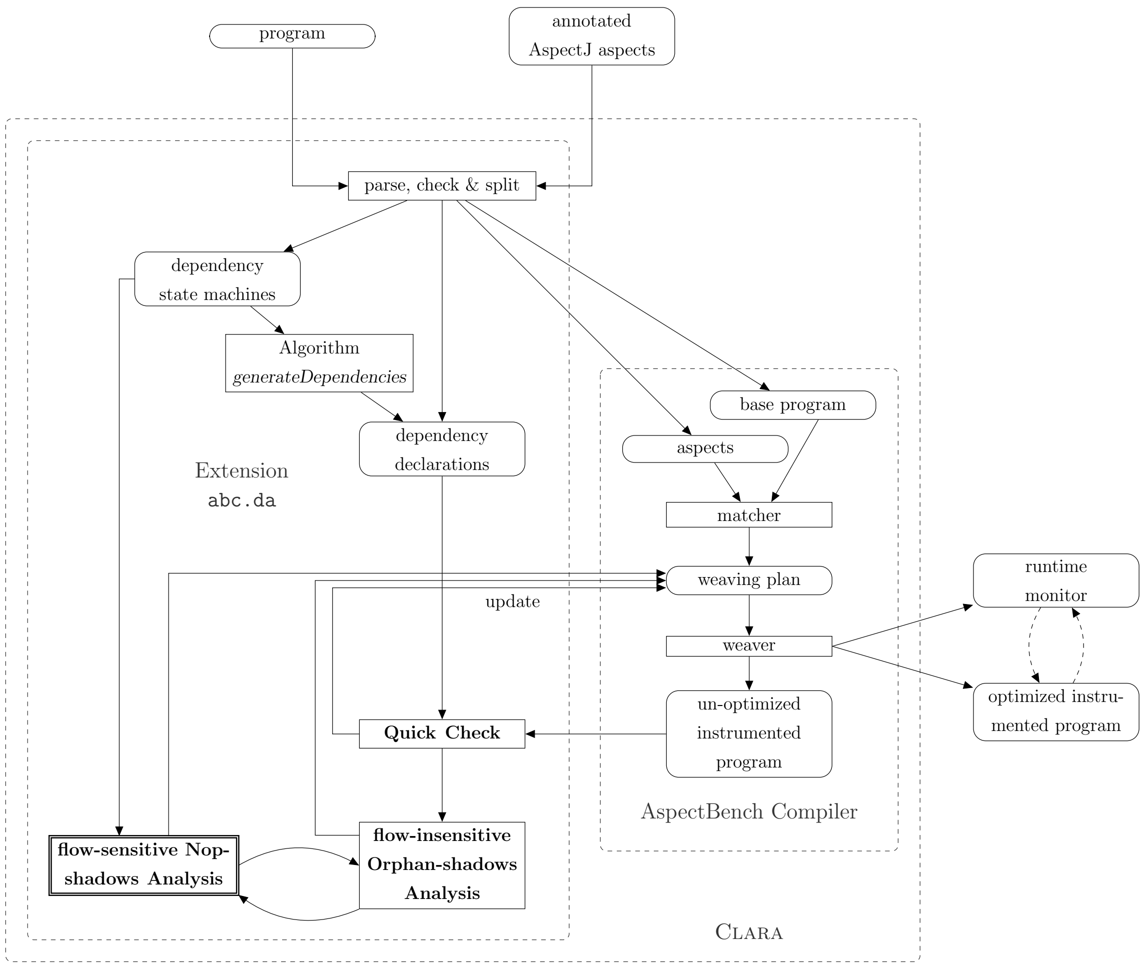

program
annotated AspectJ aspects
parse, check & split
dependency state machines
Algorithm *generateDependencies*
dependency declarations
Extension abc.da
base program
aspects
matcher
weaving plan
update
weaver
runtime monitor
optimized instru-mented program
un-optimized instrumented program
Quick Check
flow-insensitive Orphan-shadows Analysis
flow-sensitive Nop-shadows Analysis
AspectBench Compiler
Clara

Figure 5.4: Overview of our implementation of dependency state machines

5.2.2 General idea of the Nop-shadows Analysis

We wish to motivate the Nop-shadows Analysis by example. In Chapter 4, we already showed the ConnectionClosed aspect, which issues an error message in cases where the program under test writes to a connection that has previously been disconnected and not reconnected since (Figure 4.3, page 74). In Figure 5.5, we show the same code for ConnectionClosed, however, this time annotated with a dependency state machine. This annotation could have been generated automatically, from tools like JavaMOP or abc (for tracematches), or it could have been written by hand. Figure 5.6a visualizes the dependency state machine.

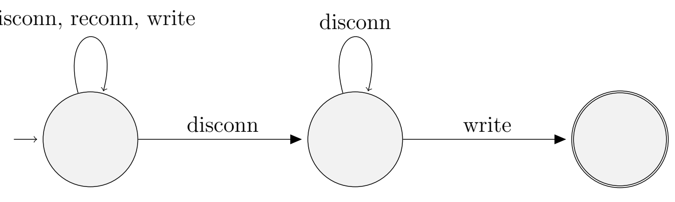
5.2.2.1 Motivating example

In this section, we motivate the analysis principle by a simple example, consisting of a single method with straight-line code, involving only a single connection object. In Section 5.2.3, we will extend the analysis to handle loops, multiple methods, and events on arbitrary combinations of aliased objects. Consider the example in Figure 5.7. The program creates a connection and then executes a few operations on this connection, all of which are monitored by the ConnectionClosed aspect.

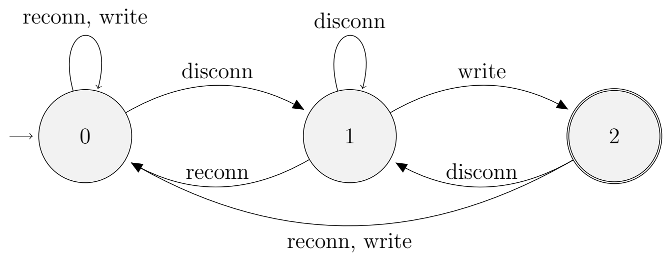
While this example is clearly contrived, it shows the possibilities for optimization by taking control flow into account. (Note that the flow-insensitive Orphan-shadows Analysis is of no use here since both disconn and reconn events occur on the connection.) The only events that actually need to be monitored to trigger the monitor for this example at the right point in time is the write at line 7 and one of the two disconnect events at lines 5 and 6. In particular, the disconnect and reconnect operations at lines 3 and 4 do not need to be monitored because they are on the prefix of a match, and that match can be completed even without monitoring this prefix. Conversely, the operations at lines 8 to 10 do not lead to a pattern violation and hence do not need to be monitored either. Of the the two disconnects at lines 5 and 6, it is sound to omit monitoring one of them, but not both. The static analysis that we present in the following will eliminate the monitoring of exactly those events that we just identified as unnecessary, or "nop shadows" as we will call them. In result,

```
aspect ConnectionClosed {
    dependency{
        disconn, write, reconn;
        initial s0: disconn -> s0,
                    write -> s0,
                    reconn -> s0,
                    disconn -> s1;
                s1: disconn -> s1,
                    write -> s2;
        final   s2;
    }

    Set closed = new WeakIdentityHashSet();

    dependent after disconn(Connection c) returning:
        call(* Connection.disconnect()) && target(c) {
        closed.add(c);
    }

    dependent after reconn(Connection c) returning:
        call(* Connection.reconnect()) && target(c) {
        closed.remove(c);
    }

    dependent after write(Connection c) returning:
        call(* Connection.write(..)) && target(c) {
        if(closed.contains(c))
            error("May not write to "+c+", as it is closed!");
    }
}
```

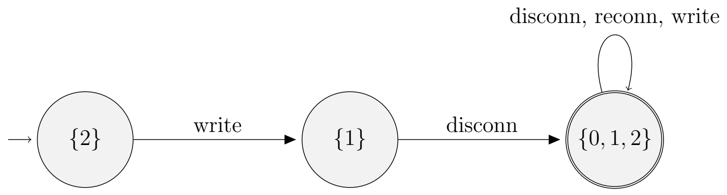
Figure 5.5: ConnectionClosed aspect, annotated with a dependency state machine



(a) Original non-deterministic finite-state machine for $\mathcal{L}$



(b) Deterministic finite-state machine for $\mathcal{L}$



(c) Deterministic finite-state machine for $\overline{\mathcal{L}}$

Figure 5.6: Finite-state machines for Connection example

```
1  public static void main(String args[]) {
2      Connection c1 = new Connection(args[0]);
3      c1.disconnect();      //disconn(c1)
4      c1.reconnect();       //reconn(c1)
5      c1.disconnect();      //disconn(c1)
6      c1.disconnect();      //disconn(c1)
7      c1.write(args [1]);   //write(c1)
8      c1.disconnect();      //disconn(c1)
9      c1.reconnect();       //reconn(c1)
10     c1.write(args [1]);   //write(c1)
11 }
```

Figure 5.7: Simple example program using a single connection object

instrumentation will only remain in lines 5 and 7, or 6 and 7—the minimal set of instrumentation points so that the optimized instrumented program will report an error if and only if the un-optimized program would have reported an error.

5.2.2.2 General rule for identifying nop shadows

The general rule that we use for identifying nop shadows is the following. A shadow s is necessary, i.e., *not* a nop shadow when the following holds.

1. The automaton for some binding compatible with s's own variable binding can be in state q just before s, and
2. the shadow s performs a transition from q to some state set Q', and
3. one of the following:
    (a) $Q' \cap F \neq \emptyset$, i.e., s triggers the validation handler, or
    (b) from just after s, when being in state q, the program can reach a final state using some execution, but not when being in any state of Q', or

(c) from just after s, when being in any state of Q', the program can reach a final state using some execution, but not when being in state q.

We prove this rule correct in Appendix B. In particular, we prove that if a shadow s is a nop shadow by the above rule, then by the soundness condition, Condition 5.1, it is sound to return `false` from *necessaryShadow*(a, t, i) for any event t_i monitored through s.

Part 3b is necessary for soundness: assume that we mistakenly identified some shadow s as nop shadow and disabled it, although one can reach the final state from q but not from any state in Q'. In this case, we would jeopardize soundness: the monitor may fail to trigger during an actual property violation at runtime. Conversely, part 3c is necessary for completeness. Assume we mistakenly disabled a shadow for which we cannot reach the final state from q but we can reach it from Q'. This would mean that the monitor could produce a spurious match, i.e., would notify the programmer about a property violation where there is none—a false positive. Part 3a takes care of the special case where some final state of the monitor contains a self-loop. In this case, when the state machine is in a final state q already, and s leads from q back to q, then both part 3b and 3c do not apply. Nevertheless, we need to keep the s alive, because, according to the semantics of dependency state machines, the repeated execution of s in a final state leads to the repeated execution of the violation handler. (Remember that a validation handler executes after each matched prefix that is in the state machine's language.)

In the following, we will describe a forward and a backward analysis that determine the necessary state sets to decide the above rule. The forward analysis determines the set of states that may reach a shadow (necessary for deciding part 1 or the rule), and the backward analysis computes the sets of states from which a final state can be reached at this shadow (necessary for deciding parts 3b and 3c). We can decide parts 2 and 3a locally, by inspecting a shadow in isolation.

5.2.2.3 Forward analysis

The forward pass determines for each statement the set of automaton states which the automaton could be in when reaching this statement. One way to compute this information is to use a determinized version of the dependency state machine. Clara obtains this automaton using the well-known subset-construction technique:

Definition 5.7 (Deterministic version of a non-deterministic state machine)**.** Let $\mathcal{L} \subseteq \Sigma^*$ be a regular Σ language and let $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$ be a non-deterministic finite-state machine with $\mathcal{L}(\mathcal{M}) = \mathcal{L}$. Then we define the deterministic finite-state machine $det(\mathcal{M})$ as $det(\mathcal{M}) := (\mathcal{P}(Q), \Sigma, Q_0, \delta, \hat{F})$ with:

$$\begin{aligned} \delta &= \lambda Q_s \lambda a \ . \ \{q_t \in Q \mid \exists q_s \in Q_s \text{ such that } \exists (q_s, a, q_t) \in \Delta\} \\ \hat{F} &= \{Q_F \in \mathcal{P}(Q) \mid \exists q \in Q_F \text{ such that } q \in F\} \end{aligned}$$

Figure 5.6b shows the determinized version of the original (non-deterministic) automaton from Figure 5.6a. In the deterministic automaton, we labeled every state with a fresh state number. Figure 5.8 shows the previous example program, but this time we annotated each statement with the state of the deterministic automaton just before and after executing the statement.

In the remainder of this dissertation, we will refer to the deterministic finite-state machine $det(\mathcal{M})$ that the forward analysis uses by the name of $\mathcal{M}_{forward}$.

5.2.2.4 Backward analysis

The backward analysis determines for every statement the set of states from which one could potentially reach a final state using the remainder of the program execution. Similar to the forward analysis, the backward analysis uses a determinized state machine, however this time a determinized state machine for the mirror language $\overline{\mathcal{L}}$.

Definition 5.8 (Mirror word)**.** Let $w = w_1 \dots w_n \in \Sigma^*$ a Σ word. We define the mirror word $\overline{w}$ as $\overline{w} := w_n \dots w_1$.

Definition 5.9 (Mirror language)**.** Let $\mathcal{L} \subseteq \Sigma^*$ a Σ language. Then we define the mirror language $\overline{\mathcal{L}}$ as:

$$\overline{\mathcal{L}} := \{\overline{w} \mid w \in \mathcal{L}\},$$

```
public static void main(String args[]) {
    Connection c1 = new Connection(args[0]);
    ........................  0
    c1.disconnect();
    ........................  1
    c1.reconnect();
    ........................  0
    c1.disconnect();
    ........................  1
    c1.disconnect();
    ........................  1
    c1.write(args[1]);
    ........................  2
    c1.disconnect();
    ........................  1
    c1.reconnect();
    ........................  0
    c1.write(args[1]);
    ........................  0
}
```

Figure 5.8: Example program from Figure 5.7, annotated with states of forward analysis

Given a non-deterministic finite-state machine $\mathcal{M}$ with $\mathcal{L}(\mathcal{M}) = \mathcal{L}$, one can easily obtain a non-deterministic finite-state machine accepting $\overline{\mathcal{L}}$, as follows.

Definition 5.10 (Reversed finite-state machine)**.** Let $\mathcal{M} = (Q, \Sigma, Q_0, \Delta, F)$ be a non-deterministic finite-state machine. Then we define the reversed finite-state machine $\mathit{rev}(\mathcal{M})$ as $\mathit{rev}(\mathcal{M}) := (Q, \Sigma, F, \mathit{rev}(\Delta), Q_0)$ with

$$\mathit{rev}(\Delta) := \{(q_t, a, q_s) \mid (q_s, a, q_t) \in \Delta\}.$$

Note that for any finite-state machine $\mathcal{M}$ it holds that

$$\mathcal{L}(\mathit{rev}(\mathcal{M})) = \overline{\mathcal{L}(\mathcal{M})}.$$

Our backwards analysis operates on the state machine

$$\mathcal{M}_{\mathit{backward}} := \mathit{det}(\mathit{rev}(\mathit{det}(\mathcal{M}))) = \mathit{det}(\mathit{rev}(\mathcal{M}_{\mathit{forward}})).$$

Note that $\mathcal{L}(\mathcal{M}_{backward}) = \overline{\mathcal{L}}$. Figure 5.6c shows the state machine that the backwards analysis uses for the ConnectionClosed example. Note that the states of $\mathcal{M}_{backward}$ are actually subsets of the state set of $\mathcal{M}_{forward}$.

It would be sound to have the backwards analysis operate on *any* state machine recognizing $\overline{\mathcal{L}}$, for instance on a reversed-determinized version of $\mathcal{M}$. However, by using a reversed-determinized version of $det(\mathcal{M})$ instead, we automatically obtain a *minimal* deterministic finite-state machine for $\overline{\mathcal{L}}$. (See [Brz62] for a proof.) Using a minimal deterministic finite-state machine yields additional optimization potential. (In Section 5.2.2.6 we explain why this is.) Figure 5.6c shows the state machine that the backwards analysis uses for the ConnectionClosed example. In this figure, we labeled every state of this state machine with the corresponding state set of $rev(det(\mathcal{M}))$ (Figure 5.6b). Note that, for presentation purposes, we omitted from the figure the reject state, which represents the empty state set.

In Figure 5.9, we show how the states of $\mathcal{M}_{backward}$ evolve during the backwards analysis. Note that, because a dependency state machine executes its validation handler after matching any *prefix* of the execution trace, we have to start the backwards analysis after *every* statement that could lead into a final state. In case of the ConnectionClosed example, the only event that can lead into a final state, and hence cause the validation handler to execute, is a write event. Hence, in this case we have to start one instance of the analysis after every write statement. As a result of this process, the backwards analysis computes for every statement not a single set, but multiple sets of states from which a final state could be reached.

5.2.2.5 Combining forward and backward analysis results

In Figure 5.10 we show the same program, annotated with both the analysis information from the forward analysis (to the left) and from the backward analysis (to the right). For instance, by the forward analysis we know that we will be in state 0 when reaching line 5. When executing the disconn transition in this line, this leads us from state 0 to state 1. According to the backwards-analysis information, every set that contains state 0 also contains state 1, and the other way around. This tells

```
 1  public static void main(String args[]) {
 2      Connection c1 = new Connection(args[0]);
        ........................  ↑ {}  ···  ↑ {0,1,2}
 3      c1.disconnect();
        ........................  | {}  ···  | {0,1,2}
 4      c1.reconnect();
        ........................  | {}  ···  | {0,1,2}
 5      c1.disconnect();
        ........................  | {}  ···  | {0,1,2}
 6      c1.disconnect();
        ........................  | {}  ···  | {1}
 7      c1.write(args [1]);
        ........................  | {}  ···  | {2}
 8      c1.disconnect();
        ........................  | {}
 9      c1.reconnect();
        ........................  | {1}
10      c1.write(args [1]);
        ........................  | {2}
11  }
```

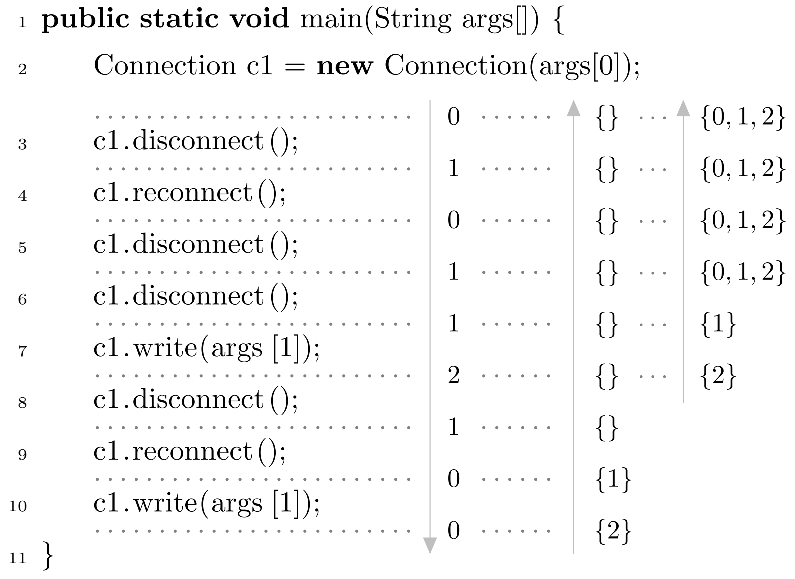
Figure 5.9: Example program from Figure 5.7, annotated with states of backward analysis (note that the analysis is computed bottom-up)

us that we can reach a final state from state 0 in the same way as from state 1. This is how we infer that we can disable the shadow at line 5.

5.2.2.6 Nop shadows transition between continuation-equivalent states

We can now determine and disable nop shadows based on this combined analysis information. Our notion of a nop shadow is strongly connected to a novel notion of "continuation-equivalent states". In the following, for two states q_1 and q_2 we say that q_1 and q_2 are "continuation-equivalent at a shadow s", or "equivalent at s" for short, and write $q_1 \equiv_s q_2$, if for all possible continuations of the control flow after s it holds that the dependency state machine to which s belongs reaches the final state at the same points of the program execution, regardless of whether we are in state q_1 or q_2 when reaching s. We can formally define this equivalence relation as follows.

For every shadow s, let us call the sets of set of states that the backwards analysis determined for the position just after s the *futures* of s. Further, let us call the state



```
1  public static void main(String args[]) {
2      Connection c1 = new Connection(args[0]);
       .........................   0 ......  {} ...  {0,1,2}
3      c1.disconnect();
       .........................   1 ......  {} ...  {0,1,2}
4      c1.reconnect();
       .........................   0 ......  {} ...  {0,1,2}
5      c1.disconnect();
       .........................   1 ......  {} ...  {0,1,2}
6      c1.disconnect();
       .........................   1 ......  {} ...  {1}
7      c1.write(args [1]);
       .........................   2 ......  {} ...  {2}
8      c1.disconnect();
       .........................   1 ......  {}
9      c1.reconnect();
       .........................   0 ......  {1}
10     c1.write(args [1]);
       .........................   0 ......  {2}
11 }
```

Figure 5.10: Example program from Figure 5.7, annotated with combined analysis information

that the forward analysis computed for the position just before s the *source*(s), and the state just after s as *target*(s). For instance, for the disconnect statement at line 5 of Figure 5.10 we have:

$$\begin{aligned} \mathit{source}(\text{line 5}) &= 0 \\ \mathit{target}(\text{line 5}) &= 1 \\ \mathit{futures}(\text{line 5}) &= \{\ \{\}, \{0,1,2\}\ \} \end{aligned}$$

We then define continuation-equivalence as:

$$q_1 \equiv_s q_2 \quad :\Longleftrightarrow \quad \forall Q \in \mathit{futures}(s).\ q_1 \in Q \leftrightarrow q_2 \in Q$$

A shadow is a nop shadow when it transitions between states in the same equivalence class, unless the target state is an accepting state. (Because reaching a final state has the side effect of triggering the monitor, the transition has an effect in this case even though it switches between equivalent states.) Let us denote by F the set of accepting, i.e., violating states of $\mathcal{M}_{forward}$. Then we call a shadow at a statement s a "nop shadow" if:

1. $\mathit{source}(s) \equiv_s \mathit{target}(s)$, and
2. $\mathit{target}(s) \notin F$.

The first case states that the shadow transitions between states that are in the same equivalence class. Hence, monitoring the shadow appears unnecessary. (Note that, as a special case, this condition handles looping: when $\mathit{source}(s) = \mathit{target}(s)$ then Condition 1 holds trivially.) However, there is one exception that we need to consider, and which we handle in Condition 2: When $\mathit{target}(s) \in F$, then the shadow directly triggers the runtime monitor. According to Clara's monitoring semantics, a monitor must signal repeated property violations every time the violation occurs. (This is useful when the monitor executes error-handling code.) For instance, on "`c.close();c.write();c.write()`" the monitor should signal a violation after both "write" events. However, the second "write" event does not change the monitor's state; we have $\mathit{source}(s) = \mathit{target}(s) = 2$. Therefore, Condition 1 holds although the statement is not a nop shadow. Adding Condition 2 handles this corner case.
For example, for the disconnect statement at line 5 of Figure 5.10 it holds that $\mathit{source}(s) = 0$ and $\mathit{target}(s) = 1$. For both sets $Q_f \in \mathit{futures}(s) = \{\ \{\}, \{0, 1, 2\}\ \}$ it holds that $0 \in Q_f \leftrightarrow 1 \in Q_f$. Consequently, we have $0 \equiv_s 1$ and because $1 \notin F$, s is a nop shadow.

For the write statement at line 7, things look differently, though. Here, $\mathit{source}(s) = 1$ and $\mathit{target}(s) = 2$, but there exists a set $Q_f = \{2\} \in \mathit{futures}(s)$ such that $2 \in Q_f$ but $1 \notin Q_f$. Hence, $1 \not\equiv_s 2$, i.e., s is not a nop shadow and may therefore not be disabled.

Note that the condition for determining nop shadows is a generalization of a more restrictive condition: one can disable a shadow if it loops. When a shadow s loops then we always have $\mathit{source}(s) = \mathit{target}(s)$, and hence $\mathit{source}(s) \equiv_s \mathit{target}(s)$ obviously holds.

The above definition of nop shadows also explains why it is beneficial to use a deterministic finite-state machine that is minimal when we perform the backwards analysis: in a minimal state machine, any two equivalent states are collapsed into one. The collapsed state will be labeled with a larger set Q_f of $\mathcal{M}_{forward}$ states than the un-collapsed sets would have been. Hence, through collapsing equivalent states, the chances of obtaining sets Q_f that contain states *source*(s) as well as *target*(s) for any shadow s increase.

Using this procedure, one can easily identify several nop shadows in our example program. For instance, all the shadows at lines 3–6 are nop shadows according to our definition, and indeed it is sound to disable any single (!) one of these shadows. Note however, that we can only remove shadows one-by-one: after a shadow has been disabled, we need to re-compute the analysis information for this method because the transition structure for this method may have been changed. In our example, otherwise we would be disabling both disconn shadows at lines 5 and 6. This is unsound: removing both these shadows leads to the monitor not reaching its final state in line 7.

Hence, our analysis proceeds as follows: it first computes the forward and backward analysis. Then next it tries to determine a single shadow that can be disabled, based on this analysis information. If such a shadow can be found, then the analysis disables the shadow and re-iterates the flow-insensitive Nop-shadows Analysis. Next, it starts over, computing the forward and backward analysis again for the same method. We iterate this process until no shadows can be disabled any more. In our example, this would leave us with one of the shadows at lines 5–6, and with the shadow at line 7—exactly the minimal correct set of shadows in this case.

Note that by following this procedure we disable shadows in a "greedy" manner: by disabling a single nop shadow at a time, without taking other nop shadows into account, we may reach the fixed point in a local optimum, which is not necessarily the global optimum. (The global optimum is the version of the program that carries the minimal number of shadows necessary to allow for a sound runtime monitor, given the approximations of control flow and pointer information that we computed.) In Appendix C, we present a constructed example that demonstrates how one particular order of disabling nop shadows can cause our algorithm to "get stuck" in such a local optimum, while the algorithm could find a global optimum for the same example when disabling nop shadows in a different order. However, as this constructed example shows, such local optima only exist in quite specific situations with quite specific and complex finite-state properties. Because the finite-state properties that programmers are interested in are typically quite simple (see Chapter 2), we have reason to believe that for such properties local optima will not exist in most cases. In these cases, our greedy selection of shadows will discover the global optimum. Our experimental results back this intuition.

The example that we presented here was very simplistic. In particular, it abstracted from the following complications that program analyses face when analyzing real-world Java programs:

1. conditional control flow and loops,
2. methods with virtual dispatch,
3. aliased objects, and
4. more general specification patterns referring to more than one object.

In the following, we explain a general analysis scheme that takes all of the above into account. It is sound for any single-threaded Java program.

5.2.3 Implementing the Nop-shadows Analysis

In this section, we present an effective implementation of the function *necessaryShadow* that is guaranteed to adhere to the soundness condition, Condition 5.1. The function that returns **true** for any input would be a sound implementation of *necessaryShadow*, just as it would be for the predicate *activates* that the Orphan-shadows Analysis uses. However, this function would not be effective, as it would not yield any optimization effect. An effective implementation of *necessaryShadow* instead tries to disable as many shadows as possible. We call our implementation the *Nop-shadows Analysis* .

We decided to implement the Nop-shadows Analysis as an intra-procedural analysis. In other words, the analysis considers only a single method at a time. The reasons for this design choice are twofold. Firstly, an intra-procedural analysis can be made more efficient than an inter-procedural one. Secondly, however, Patrick Lam manually investigated the instrumentation points that remained active after the flow-insensitive Orphan-shadows Analysis had already been applied to our benchmarks. He found that, in most of the cases, intra-procedural analysis information was sufficient to rule out unnecessary instrumentation points, when paired with coarse-grained inter-procedural summary information that had already been computed by the Orphan-shadows Analysis. The result presented in this dissertation confirm these findings.

The Nop-shadows Analysis rules out shadows on a per-method, per-state-machine basis, first computing additional, flow-sensitive alias information for this method, then detecting and removing all nop shadows, using the forward and backward analysis that we already saw in the ConnectionClosed example.

We defined the semantics of a dependency state machine over ground traces that are projections of one single trace of parameterized events. To be effective, our static analysis also needs to separate traces that refer to different variable bindings.

5.2.3.1 Abstracting from objects with object representatives

At runtime, a runtime monitor associates automaton states with variable bindings, i.e., mappings from free variables declared in the dependency state machine to concrete runtime objects. For instance, the variable binding $x = o(\texttt{v1}) \wedge y = o(\texttt{v2})$ denotes the fact that x points to the object currently referenced by `v1`, and y points to the object currently referenced by `v2`. At compile time, we have no access to these runtime objects, and hence need to resort to a static abstraction. For our purposes, we developed a specific static abstraction of runtime objects that we called object representatives [BLH08b]. At compile time, we model a runtime binding $x = o(\texttt{v1}) \wedge y = o(\texttt{v2})$ using a binding $x = r(\texttt{v1}) \wedge y = r(\texttt{v2})$, where $r(\texttt{vi})$ is the object representative of $o(\texttt{vi})$. Explaining exactly how object representatives are computed would go beyond the scope of this thesis. (Details can be found in [BLH08b].) For our purposes, it suffices to know the following properties of an object representative.

An object representative represents the value of a (reference-typed) program variable at a specific program statement. Programmers can compare two object representatives r_1 and r_2 through a must-not-alias query: when r_1 and r_2 cannot represent the same runtime object, then this query returns **true**, which we denote by $r_1 \neq r_2$. We determine must-not-aliasing through a combination of (1) the flow-insensitive context-sensitive whole-program points-to analysis that we computed for the Orphan-shadows Analysis, and (2) an intra-procedural flow-sensitive must-not-alias analysis.

Similarly, a programmer can compare r_1 and r_2 with a must-alias query. However, we resolve must-alias queries on an intra-procedural level only. When r_1 and r_2 represent pointer values at statements from different methods then the query will always return **false**. When r_1 and r_2 represent values at statements within the same method m then we use an intra-procedural flow-sensitive must-alias analysis, specific to m, to determine whether r_1 and r_2 must-alias. When they do, we denote this fact by $r_1 = r_2$. We chose this notation because in our implementation, we in fact defined the `equals` method of object representatives in such a way that r_1`.equals(`r_2`)` if and only if r_1 and r_2 must-alias. This helps us keep our abstraction concise. In particular, when a programmer attempts to add r_2 to a set that already contains r_1, then r_2 will not be added.

In cases where r_1 and r_2 neither must-alias nor must-not-alias, we say that they may-alias, and denote this fact by $r_1 \approx r_2$. Table 5.1 gives an overview of our notation.

We call the set of all object representatives $\tilde{\mathcal{O}}$. For any subset $R \subseteq \tilde{\mathcal{O}}$ of object representatives, we define two sets *mustAliases*(R) and *mustNotAliases*(R) as follows:

$$\begin{aligned} \mathit{mustAliases}(R) &:= \{r' \in \tilde{\mathcal{O}} \mid \exists r \in R \text{ such that } r = r'\} \\ \mathit{mustNotAliases}(R) &:= \{r' \in \tilde{\mathcal{O}} \mid \exists r \in R \text{ such that } r \neq r'\} \end{aligned}$$

5.2.3.2 Abstracting from bindings with binding representatives

At runtime, a variable binding maps from variables to runtime objects. In Chapter 3, we denoted a mapping $\{x \mapsto o(\texttt{v1}), y \mapsto o(\texttt{v1})\}$ by a conjunction of equations, $x = o(\texttt{v1}) \wedge y = o(\texttt{v2})$.

$r_1 \approx r_2$	Object representatives may-alias
$r_1 = r_2$	Object representatives must-alias
$r_1 \neq r_2$	Object representatives must-not-alias

Table 5.1: Aliasing relations between object representatives

At compile time, we model variable bindings through “binding representatives”. A binding representative contains both positive and negative information. The positive information tells the analysis which objects a variable could possibly be bound to. The negative information, on the other hand, tells the analysis which objects a variable cannot be bound to.

We define a binding representative b as a pair (β^+, β^-) of two partial binding functions, a positive binding β^+ and a negative binding β^-. Both binding functions map free variables defined in the dependency state machine to sets of object representatives. For any free variable v, the positive binding $\beta^+(v)$ encodes information about the objects that v may be bound to. The negative binding $\beta^-(v)$, on the other hand, encodes information about the objects that v can certainly not be bound to. We naturally extend both partial binding functions to total functions so that they map a variable v to $\emptyset$ when no other mapping for v is defined.

For example, assume a binding representative

$$(\{x \mapsto \{r_1, r_2\}, y \mapsto \{r_3\}\}, \{x \mapsto \{r_4\}\}).$$

This binding representative expresses that x can only be bound to objects represented by both r_1 and r_2, and can certainly not be bound to objects represented by r_4. Further, y can only be bound to objects represented by r_3. Similar to our presentation of bindings at runtime, it will sometimes be useful to write binding representatives in the form of a conjunction of equations. For instance, we can write the above binding representative as:

$$x = r_1 \wedge x = r_2 \wedge y = r_3 \wedge x \neq r_4$$

Such equations allow us to perform basic Boolean arithmetic on bindings. For instance, if we know that $r_1 \neq r_2$, i.e., r_1 and r_2 must-not-alias, then it holds that:

$$x = r_1 \land x = r_2 \equiv \mathbf{ff}.$$

Certain implications that hold in Boolean logic do not however hold in the three-valued logic that we consider here. For instance, $\neg(r_1 = r_2)$ does not imply $r_1 \neq r_2$. This is because, if two objects representatives do not must-alias, then this does not necessarily mean that they must-not-alias: they could also may-alias. Table 5.2 gives an overview about the possible simplifications that can be safely applied to binding representatives of the form $(x = / \neq r_1) \land (x = / \neq r_2)$. Note that, if we know that $r_1 = r_2$ or $r_1 \neq r_2$, then we can simplify the resulting binding representative in all cases except for $(x \neq r_1) \land (x \neq r_2)$ (shown in gray), in which case we need to store the full binding representative. If r_1 and r_2 may-alias (Table 5.2c) then we must always store the full binding representative. This is one reason for why aliasing information is so essential to our approach: it allows us to keep our abstraction concise. The other reason is that the updates in tables 5.2b and 5.2a allow us to recognize impossible bindings by reducing a binding representative to **ff**. As we will see in the following, this yields precision, as it avoids false positives.

For any variable binding $\beta : \mathcal{V} \to \tilde{\mathcal{O}}$, mapping variables to object representatives, and any binding representative (β^+, β^-), we can determine whether β is compatible with (β^+, β^-):

$$\begin{aligned} \mathit{compatible}(\beta, (\beta^+, \beta^-)) \;\; := \;\; & \neg\exists v \text{ such that } \beta(v) \in \mathit{mustNotAliases}(\beta^+(v)) \\ & \qquad \lor\ \beta(v) \in \mathit{mustAliases}(\beta^-(v)) \end{aligned}$$

Informally, this definition says that a binding is incompatible with a binding representative if it binds some variable v to an object representative that must-not-alias some object representative in v's positive binding, or if this representative must-alias some object representative in v's negative binding. Note that when β is empty, i.e., binds no variables at all, β will be compatible to any binding representative.

We also introduce the notion of a shadow being compatible with a binding representative. Every shadow s induces a variable binding $\beta_s = \mathit{shadowBinding}(s)$ of

$\mathbf{r_1 \neq r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	**ff**	$x = r_2$
$x \neq r_2$	$x = r_1$	$x \neq r_1 \wedge x \neq r_2$

(a) Resulting binding representative when r_1 and r_2 must-not-alias

$\mathbf{r_1 = r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \equiv x = r_2$	**ff**
$x \neq r_2$	**ff**	$x \neq r_1 \equiv x \neq r_2$

(b) Resulting binding representative when r_1 and r_2 must-alias

$\mathbf{r_1 \approx r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \wedge x = r_2$	$x \neq r_1 \wedge x = r_2$
$x \neq r_2$	$x = r_1 \wedge x \neq r_2$	$x \neq r_1 \wedge x \neq r_2$

(c) Resulting binding representative when r_1 and r_2 may-alias

Table 5.2: Possible simplifications of binding representatives using alias information

type $\mathcal{V} \rightarrow \tilde{\mathcal{O}}$. Hence, in the following we will often write *compatible*(s, b) in place of *compatible*(β_s, b).

In the following, we will denote the set of all binding representatives by $\tilde{\mathcal{B}}$. Using the notion of compatibility, we can further define an inclusion relation on binding representatives. Let b_1 and b_2 be two binding representatives. We say that b_2 is "at least as permissive" as b_1, or $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$, if the following holds:

$$b_1 \subseteq_{\tilde{\mathcal{B}}} b_2 \quad :\Longleftrightarrow \quad (\ \forall \beta.\ \mathit{compatible}(\beta, b_1) \rightarrow \mathit{compatible}(\beta, b_2)\ )$$

Informally, $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$ means that for every variable v, every object o that can be bound to v according to the binding representative b_1 can also be bound to o according to the binding representative b_2. We will write $b_1 \subset_{\tilde{\mathcal{B}}} b_2$ if $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$ but b_1 and b_2 are different.

In the following, we will denote the empty binding representative, in which both binding functions β^+ and β^- are undefined for all variables, by $\top$. Note that by the above definition it holds that $\top$ is the most permissive binding:

$$\forall b \in \tilde{\mathcal{B}} : \; b \subseteq_{\tilde{\mathcal{B}}} \top$$

5.2.3.3 The worklist algorithm

Our forward and backward analysis both compute for every statement a set of possible configurations before and after executing the statement. A configuration (Q_c, b_c) is an element of $\mathcal{P}(Q) \times \tilde{\mathcal{B}}$, i.e., a configuration combines a set $Q_c \subseteq Q$ of automaton states with a binding representative b_c. The underlying state set Q is the state set of $\mathcal{M}_{\mathit{forward}}$, regardless of whether we perform a forward or backward analysis: both the forward and backward analysis operate on the same state set, they just use different transition functions where one is the inverse of the other. The Nop-shadows Analysis propagates configurations for both the forward and backward analysis using a general worklist algorithm, Algorithm 5.1.

The algorithm first initializes a worklist wl, which we actually implement as a "work map". We wl as a mapping from statements to sets of configurations: for every statement, wl contains a set of configurations that were determined to reach the statement and for which successor configurations need to be computed. We say that the worklist wl is empty if it maps every statement to the empty set. Also, we call each entry $(\mathit{stmt}, \mathit{cs})$ in wl, where stmt is a statement and cs is a set of configurations, a job.

The algorithm further initializes two mappings *before* and *after* that store the configurations that have been computed so far before, respectively after each statement. These sets allow us to perform a terminating fixed point iteration. Lines 6–7 implement an important optimization. In the last section we defined the relation $\subseteq_{\tilde{\mathcal{B}}}$ in such a way that $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$ if b_2 is as least as permissive as b_1. In lines 6–7, the algorithm first computes the union cs_{temp} of the old *before* set and the configurations that need to be computed at the current statement, according to the job taken from the worklist. Because we implemented the worklist as a mapping from statements to jobs, we know that the current job is the only job for this statement, and hence

Algorithm 5.1 $\mathit{worklist}(\mathit{initial}, \mathit{succ}_{\mathit{cfg}}, \mathit{succ}_{\mathit{ext}}, \delta)$

The syntax $f[x \mapsto y]$ denotes the function that is equal to f on all values v, except for x, in which case it returns y:

$$f[x \mapsto y] := \lambda v \begin{cases} y & \text{if } v = x \\ f(v) & \text{otherwise} \end{cases}$$

```
1:  $wl := initial$
2:  $before := after := \lambda stmt.\ \emptyset$ // associate $\emptyset$ with every statement
3:  while $wl$ non-empty do
4:     pop job $(stmt, cs)$ from $wl$
5:     // reduce configurations so that only most permissive ones remain
6:     $cs_{temp} := cs \cup before(stmt)$
7:     $cs_{new} := \{(Q_c, b_c) \in cs_{temp} \mid \neg\exists(Q_c, b'_c) \in cs_{temp} \text{ with } b_c \subset_{\tilde{\mathcal{B}}} b'_c\} - before(stmt)$
8:     $cs' := \begin{cases} cs_{new} & \text{if } shadows(stmt) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$
9:     for $c \in cs_{new},\ \ s \in shadows(stmt)$ do // compute transition
10:       $cs' := cs' \cup \underline{transition}(c, s, \delta)$
11:    end for
12:    $before := before[stmt \mapsto before(stmt) \cup cs]$ // update before-flow
13:    $cs'_{new} := cs' - after(stmt)$ // filter out configurations already computed
14:    if $cs'_{new}$ non-empty then
15:       $after := after[stmt \mapsto after(stmt) \cup cs'_{new}]$ // update after-flow
16:       // add jobs for $m$-successor statements to worklist
17:       for $stmt' \in succ_{cfg}(stmt)$ do
18:          $wl := wl[stmt' \mapsto wl(stmt') \cup cs'_{new}]$
19:       end for
20:       // add jobs for interprocedural successor statements to worklist
21:       for $stmt' \in succ_{ext}(stmt)$ do
22:          $wl := wl[stmt' \mapsto wl(stmt') \cup \underline{reachingStar}\,(cs'_{new}, \underline{relevantShadows}\,(stmt))]$
23:       end for
24:    end if
25: end while
```

the set cs_{temp} holds the complete information that the algorithm computed so far for the position just before the current statement. Then next, in line 7, the algorithm removes from this set first all these configurations (Q_c, b_c) for which there is a configuration (Q_c, b'_c) with b'_c being more permissive than b_c. This is sound because any shadow that is compatible with b_c will also be compatible with b'_c. Hence, if (Q_c, b_c) causes a shadow to be identified as a "necessary" shadow, then so will (Q_c, b'_c). In our experiments, this optimization had a significant effect, often reducing the number of configurations computed for a method by two to three orders of magnitude. Lastly, in the same line 7, the algorithm removes again from the resulting set all the configurations contained in the *before* set. This is sound because in a previous iteration the algorithm already computed successor configurations for these configurations (using lines 9–11) and there is no need to compute the same information again.

Next the algorithm computes, for every new configuration $c \in cs_{new}$ and any shadow s at a statement, successor configurations using a function *transition*. We will explain this function in detail below. The algorithm then updates the statement's *before* set and checks (line 14) if any configurations were computed that had not been computed at this statement before. If this is the case, then the algorithm propagates these configurations to two different kinds of successor statements. First, in lines 16–19, the algorithm adds new jobs containing the successor configurations cs'_{new} for any statement that is a successor of *stmt* in m's control-flow graph (as determined by $succ_{cfg}$). Lines 20–23 handle inter-procedural control flow. After all, the automaton state of objects which shadows in m refer to may not only be changed by these shadows within m, but also by other compatible shadows. We will explain this part of Algorithm 5.1 in detail in Section 5.2.3.3. First we explain how we implemented the *transition* function.

The transition function. We describe the implementation of our transition function in Algorithm 5.2. Our implementation directly mirrors Avgustinov et al.'s implementation of the tracematch runtime [AAC+05]. For a given configuration and shadow, the algorithm computes a set cs of successor configurations. In line 1, the algorithm first initializes the result set cs. Then, in line 2, the algorithm extracts the shadow's

Algorithm 5.2 *transition*$((Q_c, b_c), s, \delta)$

```
1: $cs := \emptyset$ // initialize result set
2: $l := label(s), \beta_s := shadowBinding(s)$ // extract label and bindings from $s$
3: // compute target states
4: $Q_t := \delta(Q_c, l)$
5: // compute configurations for objects moving to $Q_t$
6: $\beta^+ := \underline{and}(b_c, \beta_s)$
7: if $\beta^+ \neq \bot$ then
8:     $cs := cs \cup \{(Q_t, \beta^+)\}$
9: end if
10: // compute configurations for objects staying in $Q_c$
11: $B^- := \bigcup_{v \in dom(\beta_s)} \{\ \underline{andNot}(b_c, \beta_s, v)\ \} - \{\bot\}$
12: $cs := cs \cup \{(Q_c, \beta^-) \mid \beta^- \in B^-\}$
13: return cs
```

label and binding. As described above, the binding β_s is a mapping from variable names to object representatives. In line 4, the algorithm then computes the set Q_t of target states, according to the shadow's label l. The transition function δ here depends on whether we perform a forward or backward analysis. For the forward analysis, δ will be the transition function of $\mathcal{M}_{forward}$, and for the backward analysis it will be the transition function of $\mathcal{M}_{backward}$ respectively. The remaining part of Algorithm 5.2 deals with variable bindings. Note that, at runtime, the event induced by shadow s moves the internal state of all those runtime monitors from Q_c to Q_t whose variable bindings are compatible with β_s. For those monitors that moved, the resulting variable binding can be described by "$b_c \wedge \beta_s$". The monitors for all other variable bindings (those that are incompatible with β_s) remain in Q_c. Hence, the variable bindings that remain in Q_c can be described by "$b_c \wedge \neg\beta_s$". Lines 6–9 compute successor configurations for all the variable bindings that move to Q_t, using the function *and*. In lines 11–12, the algorithm creates configurations for all these variable bindings that remain in Q_c, using the function *andNot*. We explain both functions *and* and *andNot* further below.

The algorithm applies *andNot* for each bound variable v separately. We adopted this implementation detail from Avgustinov et al.'s implementation of the tracematch runtime [AAC+05]. It can be explained by the following example. Assume that the shadow s induces a variable binding β_s that binds multiple variables, e.g. $x = r(\texttt{v1}) \wedge y = r(\texttt{v2})$. Then we get:

$$\begin{aligned}
\beta^- &\equiv b_c \wedge \neg\beta_s \\
&\equiv b_c \wedge \neg(x = r(\texttt{v1}) \wedge y = r(\texttt{v2})) \\
&\equiv b_c \wedge (\neg x = r(\texttt{v1}) \vee \neg y = r(\texttt{v2})) \\
&\equiv (b_c \wedge \neg x = r(\texttt{v1})) \vee (b_c \wedge \neg x = r(\texttt{v2}))
\end{aligned}$$

Because our abstraction stores all information in Disjunctive Normal Form, we have to return multiple configurations in this case, one for every disjunct.

We present our implementation of *and* in Algorithm 5.3. Both algorithm *and* and *andNot* use the simplification rules from Table 5.2 to achieve two different goals: (1) return $\bot$ whenever the abstraction allows us to conclude that b_c and β_s are incompatible, and (2) minimize the number of bound object representatives in the resulting binding representative as much as possible, without losing precision. Returning $\bot$ means that the current configuration will not be propagated any further (see Algorithm 5.2, lines 7 and 11). This is one essential contribution to the precision of our analysis. Minimizing the number of bound object representatives leads to a smaller abstraction and to a smaller number of possible configurations, thus enabling an earlier termination of the worklist algorithm, Algorithm 5.1.

In line 1, Algorithm 5.3 first creates copies of the original positive and negative binding functions. Then next, for every variable v that the shadow binds, the algorithm first compares the existing positive and negative bindings for v with the object representative $\beta_s(v)$ to which the shadow's variable binding maps v. By the semantics of β^+ and β^-, the bindings are incompatible if $\beta_s(v)$ must-not-alias some object representative in $\beta^+(v)$, or if it must-alias any object representative in $\beta^-(v)$.

Algorithm 5.3 $and((\beta^+, \beta^-), \beta_s)$

```
1: $\beta^+_{new} := \beta^+, \beta^-_{new} := \beta^-$
2: for $v \in dom(\beta_s)$ do
3:     // if bindings incompatible, return $\bot$
4:     if $\beta_s(v) \in mustNotAliases(\beta^+(v)) \vee \beta_s(v) \in mustAliases(\beta^-(v))$ then
5:         return $\bot$
6:     end if
7:     // add new positive binding
8:     $\beta^+_{new} := \beta^+_{new}[v \mapsto \beta^+_{new}(v) \cup \{\beta_s(v)\}]$
9:     // prune superfluous negative bindings
10:    $\beta^-_{new} := \beta^-_{new}[v \mapsto \beta^-_{new}(v) - mustNotAliases(\beta_s(v))]$
11: end for
12: return $(\beta^+_{new}, \beta^-_{new})$
```

In this case the algorithm returns $\bot$. Next, in line 8, the algorithm performs the actual refinement of the positive binding by adding $\beta_s(v)$ to $\beta^+_{new}(v)$. This operation can have two possible outcomes. When $\beta_s(v) \in mustAliases(\beta^+_{new}(v))$ already, then it will not be added to the set again. (The reader be reminded that object representatives implement an `equals` method that ensures this automatically.) The case $\beta_s(v) \in mustNotAliases(\beta^+(v))$ was treated earlier. Hence, $\beta_s(v)$ will only actually be added to the set if it may-aliases all object representatives currently associated with v. In line 10, the algorithm then prunes superfluous negative bindings. Assume that $\beta_s = x \mapsto r(\mathtt{v})$ was just added to $\beta^+_{new}(v)$. Then $x = r(\mathtt{v})$ implies that $x \neq r^-$ for all r^- with $r^- \neq r(\mathtt{v})$. Hence we can remove such object representatives r^- from $\beta^-_{new}(v)$—they encode redundant information. (This is the same case as the one shown in the top right and bottom left cell of Table 5.2a.)

Algorithm 5.4 shows how we compute the function *andNot* with respect to some variable $v \in dom(\beta_s)$. First, in line 2, the algorithm checks if the new binding is compatible with the existing negative binding. If $\beta_s(v)$ must-alias an object representative from v's positive binding, then the bindings are incompatible, and the

Algorithm 5.4 *andNot*$((\beta^+, \beta^-), \beta_s, v)$

```
1: // if bindings incompatible, return $\bot$
2: if $\beta_s(v) \in mustAliases(\beta^+(v))$ then
3:     return $\bot$
4: end if
5: // no need to store negative binding if it must-not-alias some positive binding
6: if $\beta_s(v) \in mustNotAliases(\beta^+(v))$ then
7:     return $(\beta^+, \beta^-)$
8: else // return updated binding
9:     return $(\beta^+, \beta^-[v \mapsto \beta^-(v) \cup \{\beta_s(v)\}])$
10: end if
```

algorithm returns $\bot$. Otherwise, the algorithm adds $\beta_s(v)$ to the negative bindings and returns the updated binding representative. However, there is no need to add $\beta_s(v)$ when it must-not-alias a positive binding. Omitting the update in these cases has the same effect as the pruning of superfluous negative bindings in Algorithm 5.3.

This concludes the description of our transition function. We next continue our explanation of the remainder of the worklist algorithm, Algorithm 5.1.

The external-successor function $\mathsf{succ}_{\mathsf{ext}}$. Lines 20–23 of Algorithm 5.1 handle inter-procedural control flow. Figure 5.11 visualizes this process. The figure shows the current method m as a box. The dashed arrows denote the successor function $succ_{cfg}$, which is given by m's control-flow graph. In addition, the solid arrows show a second, inter-procedural successor function, $succ_{ext}$. During the course of the execution of m, invoke expressions within m may cause methods to be called. These calls either can or cannot transitively perform a recursive call back into m. When the call is recursive, then (1) configurations that we computed for this call site can reach m's entry statement. Conversely, for configurations that we computed for any of m's exit statements, we need to propagate (2) these configurations back to any recursive call site within m. At compile time, we can only determine that a method call may be recursive, not that it must be. Hence, we (3a) also need to propagate configurations from the call site to after itself. For calls that are certainly not recursive (as determined by our call graph), we (3b) only propagate configurations past the call site but not to m's entry statement. In addition, we need to take into account the case in which method m returns (either by throwing an exception or by returning normally—both cases are equally handled through m's control-flow graph), and is then invoked again: to model this case we (4) propagate configurations from m's exit statement(s) to its entry statement.

In line with Figure 5.11 we define the function $\mathit{succ}_{\mathit{ext}}$ as follows. Let $\mathit{heads}(m)$ be the set of entry statements of m, and $\mathit{tails}(m)$ the set of exit statements of m[4]. Further, let $\mathit{recCall}(m)$ be the set of statements of m that contain an invoke expression through which m can potentially call itself recursively. Conversely, $\mathit{nonRecCall}(m)$ contains all statements that contain an invoke expression through which m can certainly not be called. Then:

$$\mathit{succ}_{\mathit{ext}} := \lambda \mathit{stmt}. \begin{cases} \mathit{heads}(m) \cup \mathit{succ}_{\mathit{cfg}}(\mathit{stmt}) & \text{if } \mathit{stmt} \in \mathit{recCall}(m) \\ \mathit{succ}_{\mathit{cfg}}(\mathit{recCall}(m)) \cup \mathit{heads}(m) & \text{if } \mathit{stmt} \in \mathit{tails}(m) \\ \mathit{succ}_{\mathit{cfg}}(\mathit{stmt}) & \text{if } \mathit{stmt} \in \mathit{nonRecCall}(m) \\ \emptyset & \text{otherwise} \end{cases}$$

One may be interested in knowing how many potentially-recursive call sites an average method has. In Figure 5.12 we show how many methods have how many potentially-recursive call sites in our benchmark set. As the graph shows, the largest number of potentially-recursive call sites is 19, however the vast majority of methods have no such sites at all. Only 244 out of 1361 methods have potentially-recursive call sites at all.

[4]Usually, $\mathit{heads}(m)$ will be a singleton set because all methods have a unique first statement. Note however, that our backwards analysis will operate on a reversed control-flow graph. In this graph heads become tails and tails become heads. Therefore, $\mathit{heads}(m)$ can contain more than one element in this setting.

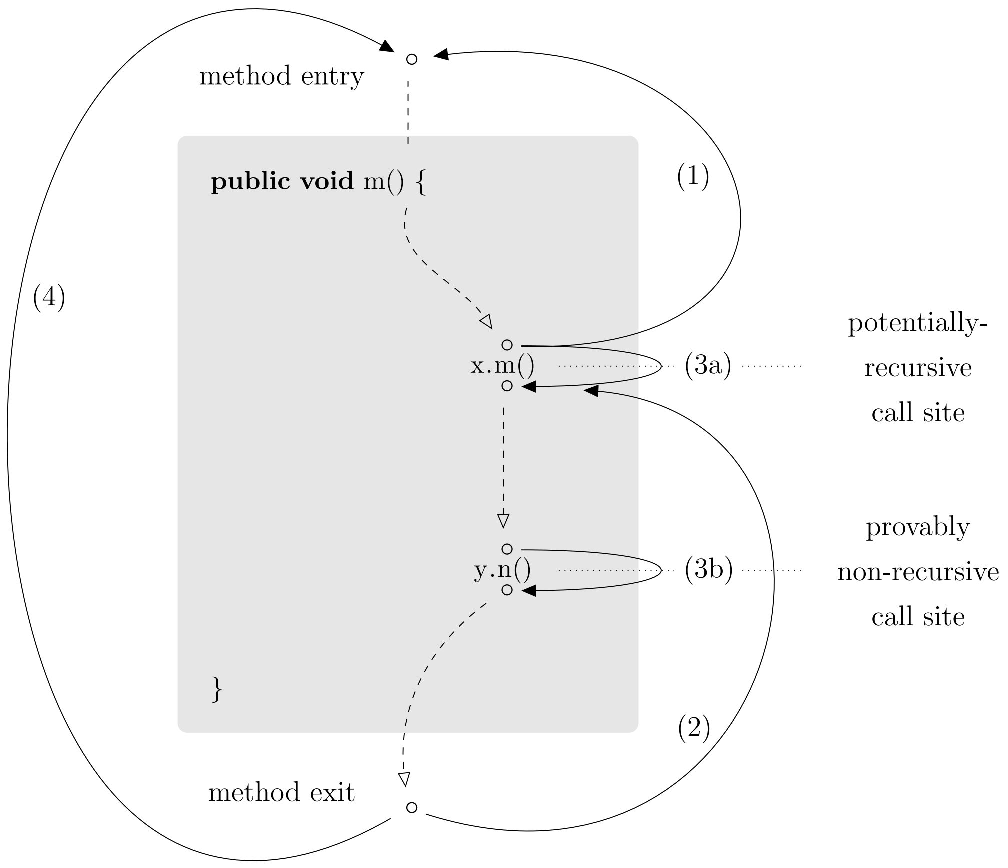


Figure 5.11: Inter-procedural control-flow regarding the current method m

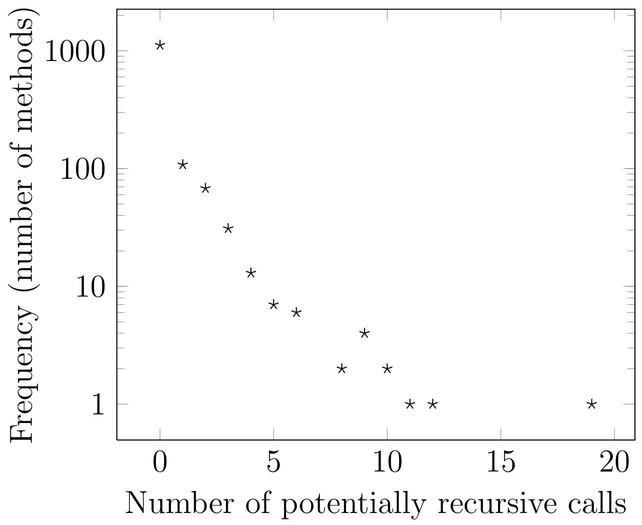


Figure 5.12: Number of potentially-recursive calls (log scale)

When propagating configurations along an edge defined through $succ_{ext}$, it is not enough to copy the configurations from the edge's source statement to it's target statement. Note that between any two executions of m other methods may execute and cause state transitions in the monitoring state machine. To soundly model these potential state transitions through "other methods", Algorithm 5.1 associates in line 22 with any inter-procedural successor not just the set of configurations cs'_{new} but instead the set $\mathit{reachingStar}(cs'_{new}, \mathit{relevantShadows}(\mathit{stmt}))$. *reachingStar* computes successor configurations using the flow-insensitive analysis information obtained through the Orphan-shadows Analysis.

The functions relevantShadows, reachingPlus and reachingStar. Given a statement *stmt* and a set *cs* of configurations just before *stmt*, the function *reachingPlus* computes for any set of configurations the set of configurations that one obtains by executing at least one shadow of a set of shadows "relevant to *stmt*", denoted by $\mathit{relevantShadows}(\mathit{stmt})$. We define this set as follows. If *stmt* contains an invoke expression (potentially recursive or not), then $\mathit{relevantShadows}(\mathit{stmt})$ contains all shadows in all the methods transitively reachable through the method invocation, except for the ones in m itself. After all, these are all the shadows that could potentially be reached before again reaching m's entry statement. Otherwise, i.e., if *stmt* is a head or tail statement of m, then *relevantShadows*(*stmt*) contains all shadows in the entire program, except for the ones in m. Unfortunately, even in the presence of a call graph, we do not know which methods may or may not execute before or after m, and hence we have no way of reducing this set any further without jeopardizing soundness.

For any configuration $c = (Q_c, b_c)$ that reaches an exit point or the point of a recursive call during the analysis of method m, all the shadows that are relevant to this program point may perform transitions on this configuration until the configuration reaches the method's entry point again. However, by definition of the function *transition*, only shadows that are compatible with b_c may actually cause the state set Q_c to change. Hence, for every binding representative $b \in \tilde{\mathcal{B}}$ and shadow set $ss \subseteq \mathcal{S}$, we define the set *compLabels*(b, ss) as:

$$\mathit{compLabels}(b, ss) := \{a \in \Sigma \mid \exists s \in ss \text{ such that } \mathit{compatible}(s, b) \wedge \mathit{label}(s) = a\}$$

This set contains the labels of all those shadows in ss that are compatible with b.

We then define *reachingPlus*(*cs*, *stmt*) as the set of configurations that are reachable from *cs* by applying at least one compatible shadow that is relevant to *stmt*. Formally, we define *reachingPlus*(*cs*, *stmt*) as the smallest set of configurations satisfying the following conditions.

- for every configuration $(Q_c, b_c) \in cs$ and $l \in \mathit{compLabels}(b_c, ss)$: $(\delta(Q_c, l), b_c) \in \mathit{reachingPlus}(cs, \mathit{stmt})$, and
- for every $(Q_c, b_c) \in \mathit{reachingPlus}(cs, s)$ and $l \in \mathit{compLabels}(b_c, ss)$: $(\delta(Q_c, l), b_c) \in \mathit{reachingPlus}(cs, \mathit{stmt})$.

We further define *reachingStar* as the reflexive closure of *reachingPlus*:

$$\mathit{reachingStar}(cs, \mathit{stmt}) := \mathit{reachingPlus}(cs, \mathit{stmt}) \cup cs$$

Hence, *reachingStar* computes the set of configurations that one can reach from *cs* by executing 0 or more shadows that are relevant at *stmt*.

5.2.3.4 Initializing the worklist algorithm

We next explain how we initialize Algorithm 5.1. The initialization depends on whether we perform a forward or backwards analysis. In forward-analysis mode, $succ_{cfg}$ is simply the successor function of m's control-flow graph, and $succ_{ext}$ is the inter-procedural successor function as defined above; the function δ is the transition function of $\mathcal{M}_{forward}$. For the backwards analysis we simply invert the two successor functions, and likewise δ is the transition function of $\mathcal{M}_{backward}$.

Determining the set *initial*, which Algorithm 5.1 uses to initialize its worklist in line 1, is more involved. We show the initialization and subsequent fixed-point iteration for the forward analysis in Figure 5.13. First let us bring to the reader's attention that, because Algorithm 5.1 soundly treats subsequent executions of m using the loop in line 20–23, for the initialization it is sound to assume that m has not executed yet. When m has not yet executed, not even in part, then we can enter m only through its first statement (its head) and not, for instance, by returning from a recursive call site. The configurations that can reach m's head are the ones that are reachable from the initial state set Q_0, by executing any shadows outside of m, in any particular order, for any variable binding (i.e., for $\top$). Hence, for the forward analysis we define:

$$initial := \{\ (h, reachingStar(\{(Q_0, \top)\}, relevantShadows(h))) \mid h \in heads(m)\ \}$$

For the backward analysis, the initialization is more complicated. We visualize the appropriate initialization and iteration in Figure 5.14. Conversely to the forward analysis, we now assume for the initialization that m will not be executed any more *after* the current execution of m. We now have to create a job associating a statement *stmt* with any configuration c from which the remainder of the execution (including the execution of *stmt* itself) could lead into a final state. This is obviously the case if there is a "final" shadow in m itself—a shadow s labeled with a label $l = label(s)$ such that there exists an l-transition into a final state $q_F \in F$.

However, we also need to consider the case where m executes completely, returns, and the remainder of the execution drives the configuration into a final state

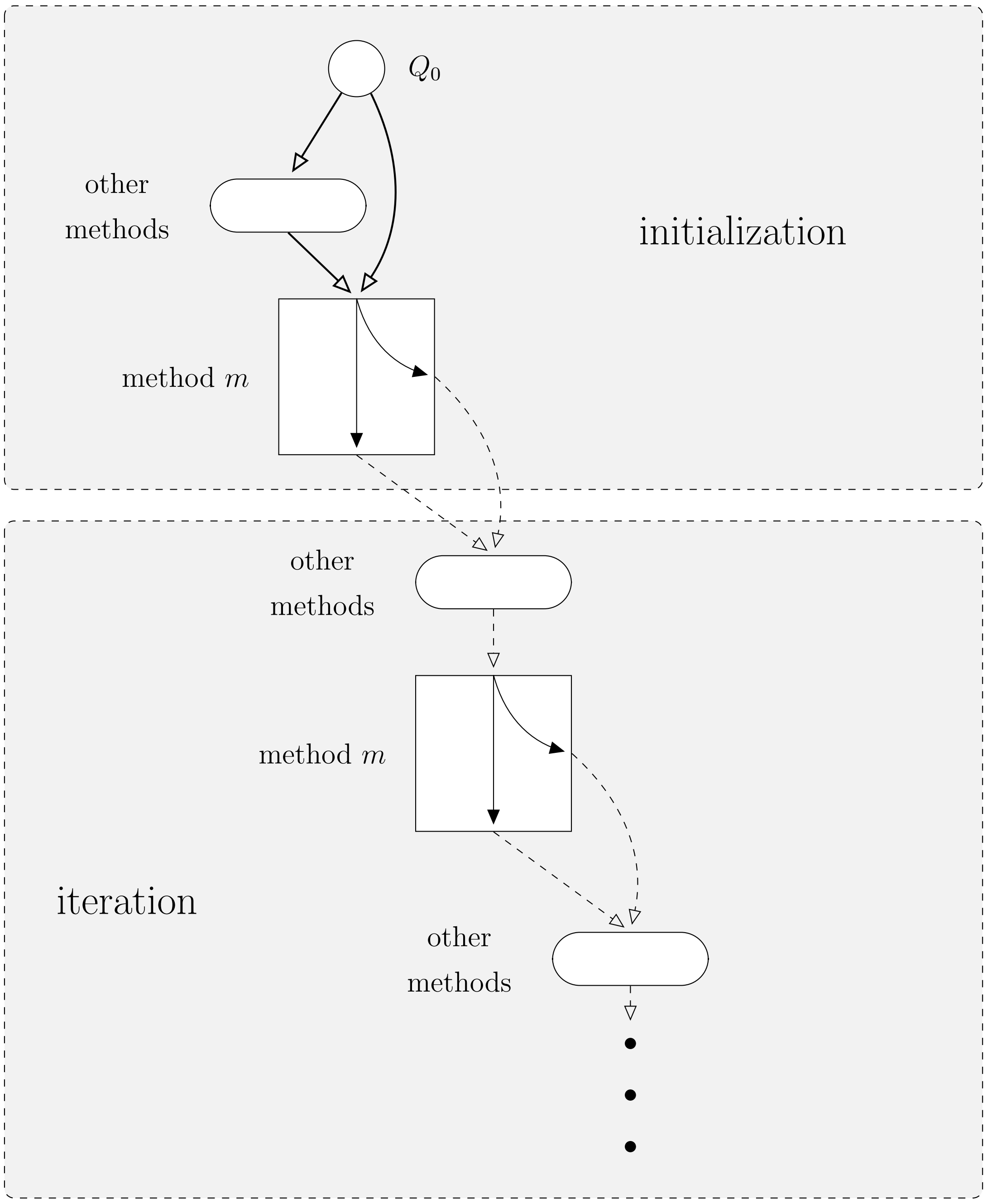


Figure 5.13: Possible configurations reaching a method m's first statement

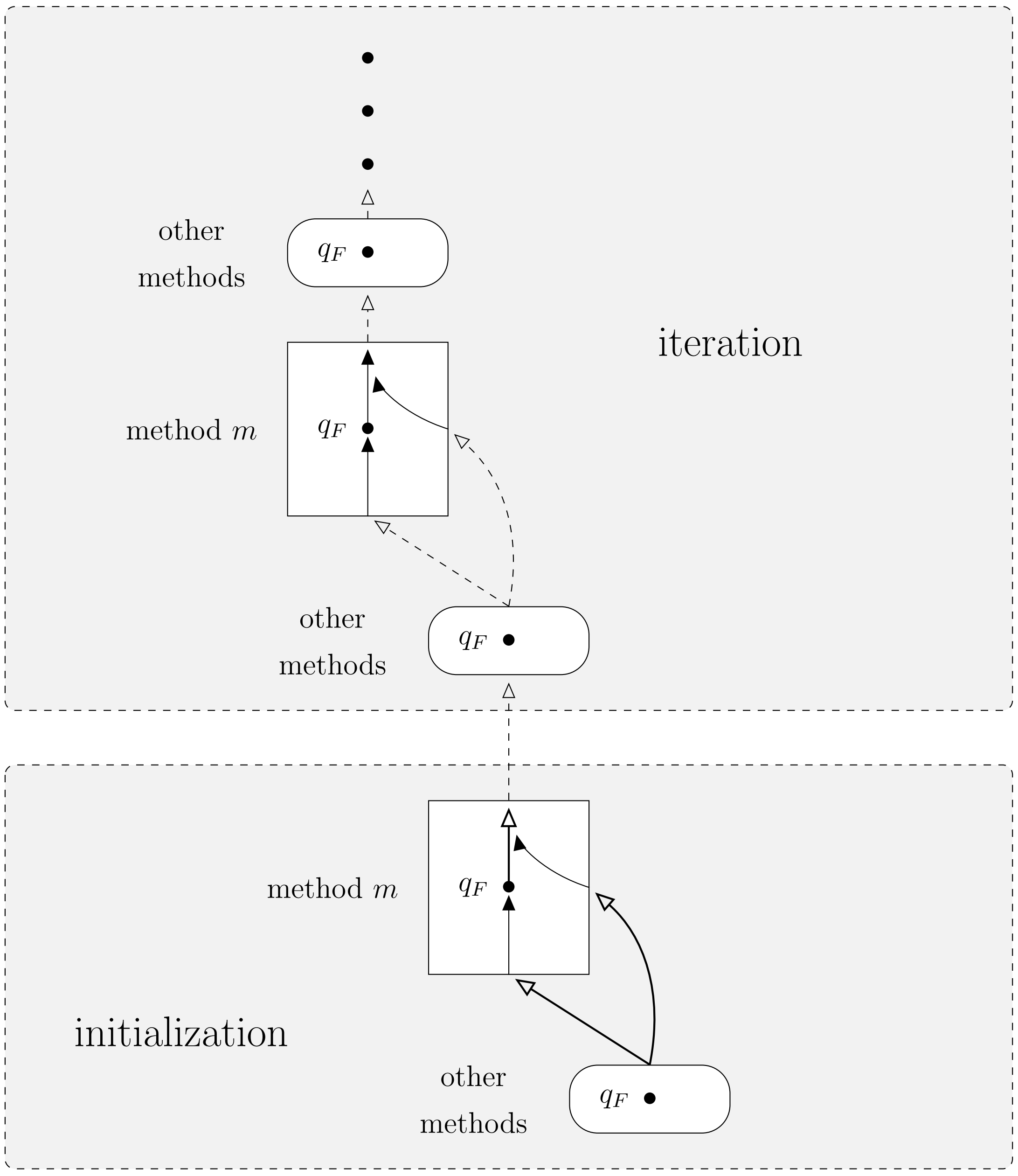


Figure 5.14: Possible control-flow leading to a final state q_F from a statement in m

using shadows in methods other than m. To cater for these cases, we create jobs that associate any tail statement *stmt* of m with any configuration c that is in $\mathit{reachingPlus}(\{F\}, \mathit{relevantShadows}(\mathit{stmt}))$. Note that this is almost the exact opposite of the initialization for the forward analysis. The only difference is that here we now use *reachingPlus*, not *reachingStar*. The reason is, that, by definition, $\mathit{reachingStar}(\{F\}, \mathit{ss})$ is always a super set of F, even if *ss* is empty. However, we cannot actively *reach* a final state from F (hence executing the validation handler) from any of m's tail statements if there are no shadows in any other methods at all. *reachingPlus* computes exactly those configurations that reach in at least one step, and therefore in this case the configurations that lead *into* a final state.

The third case that we need to consider is the case where a partial execution of m leads to a configuration c at a statement *stmt* that contains an invoke statement. Similarly to above, we associate *stmt* with $\mathit{reachingPlus}(\{F\}, \mathit{relevantShadows}(\mathit{stmt}))$. The only difference is that here $\mathit{relevantShadows}(\mathit{stmt})$ will contain all shadows reachable through evaluating *stmt*'s invoke expression, while in the above case, it contains all shadows in methods other than m.

We hence initialize the backwards analysis with the union of two sets. One set holds all configurations that lead into a final state within m, while the other set holds all those that could into a final state outside of m:

$$\begin{aligned}
&\mathit{initial} := \text{let } \mathit{reachingConfigs} = \mathit{reachingPlus}(\{(F, \top)\}, \mathit{relevantShadows}(\mathit{stmt})) \text{ in:}\\
&\qquad \{\ (\mathit{stmt}, \{(F, \top)\}) \mid \exists s \in \mathit{shadowsOf}(\mathit{stmt}) :\ \delta(F, \mathit{label}(s)) \neq \emptyset)\}\\
&\qquad\quad \cup \{\ (\mathit{tail}, \mathit{reachingConfigs}) \mid \mathit{tail} \in \mathit{tails}(m) \cup \mathit{recCall}(m) \cup \mathit{nonRecCall}(m)\ \}
\end{aligned}$$

In the above definition, note that δ is the transition function of $\mathcal{M}_{\mathit{backward}}$.

5.2.3.5 Removing nop shadows

After applying the forward and the backward analysis pass, our analysis tries to identify a single nop shadow, according to the rule that we mentioned in Section 5.2.2.2. If the analysis fails to identify a nop shadow then we proceed with the next shadow-bearing method. Else, if the analysis does identify a nop shadow then we (1) remove the shadow, (2) re-compute the flow-insensitive Orphan-shadows Analysis on all the shadows of the current method m and then (3) re-iterate the Nop-shadows Analysis on m. When the analysis fails to identify any further nop shadows within m then we re-apply the flow-insensitive Orphan-shadows Analysis on the entire program (thereby potentially disabling further shadows) and proceed with the next shadow-bearing method. The current method m may be re-visited if shadows are removed from any other methods in the future, because this yields further optimizations for m. This process terminates when no further nop shadows can be identified any more in any method. In Figure 5.15 we show how often we re-iterate within a single method. As the figure shows, we iterate only a few times for the vast majority of cases (after all, this number is bounded by the number of still-enabled shadows in the method), and there are only twelve few cases in which we have to iterate more than ten times. There was one single method for which we had to re-iterate 78 times: `fillArray` in class `CompactArrayInitializer` of the bloat benchmark in combination with the FailSafeIter tracematch. This method contains a large number of statements that modify a collection, adding instructions to an instruction stream.

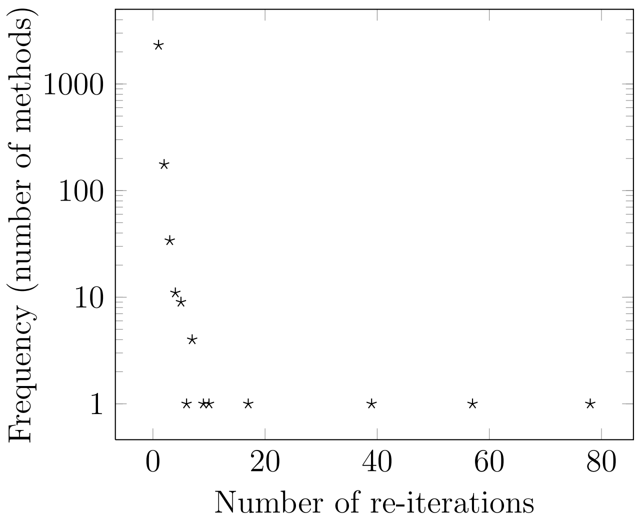


Figure 5.15: Number of re-iterations per method (log scale)

5.2.3.6 Optimizations for increased performance at analysis time

In Section 5.2.3.3 we already discussed an important optimization that eliminated configurations that were less permissive than other configurations at the same statement. In our experiments, this optimization had a significant effect, often reducing the number of configurations computed for a method by two to three orders of magnitude. We perform a set of other optimizations to decrease the analysis time.

Abstracted call graph. The call graph that we use to identify all shadows in the transitive closure of an outgoing method call is an abstracted version of the call graph that the points-to analysis in Spark [LH03] computes: in the abstracted call graph we omit paths that never reach a shadow-bearing method. This accelerates lookups that the analysis performs on the graph. In particular, if a method invocation cannot transitively call any shadow-bearing methods at all then the call graph will not have any call edge for the invocation and the analysis can identify the call as "harmless" in constant time. In our benchmark set, abstracting the call graph was highly effective. On average, the abstracted call graph had only about 4.3% of the edges of the complete call graph, with 12.9% in the worst case (12984 remaining edges in bloat-FailSafeIter), and 0.02% (26 edges in fop-HasNext) in the best case.

Caching. In addition, we use extensive amounts of caching, wherever this makes sense. For instance we cache points-to sets, the results of the must-alias and must-not-alias analysis for every method and the set of methods transitively reachable through a method call. We also cache the set of shadows that are currently enabled in such methods. Note that it would be imprecise to cache this set while analyzing multiple methods: when the analysis disables a shadow it must be removed from the set, and this removal must be visible when analyzing other methods later on. To avoid having to re-compute the set of still-enabled shadows entirely, we store "sets of enabled shadows" using a special class `EnabledShadowSet`. When a shadow s is added to such a set, the set automatically registers itself with the shadow as a listener. When the analysis disables s later on, s notifies all registered sets, which then update themselves by removing s. In addition, if a programmer attempts to add a shadow s to an `EnabledShadowSet` at a time where s is already disabled, the shadow is not actually added in the first place (nor is the set registered with the shadow). This mechanism assures us that an `EnabledShadowSet` is always kept up-to-date in constant time.

Aborting overly long analysis runs. Despite these optimizations we found a small number of methods for which the Nop-shadows Analysis would still take a long time to reach its fixed point. Figure 5.16 shows a much simplified view of the method `peelLoops(int)` in the class `EDU.purdue.cs.bloat.cfg.FlowGraph` of the benchmark bloat (the original method is 456 lines long!). For this particular benchmark, the context-sensitive points-to analysis that we use fails to compute context information for the iterators and collections. Hence, when analyzing this method with respect to the FailSafeIter monitor, the Nop-shadows Analysis gets the imprecise information $r(\texttt{i1}) \approx r(\texttt{i2}) \approx r(\texttt{i3})$, i.e., it has to assume that `i1`, `i2` and `i3` could all point to the same iterator. This, in turn, leads to a large number of possible configurations. Assume that when have a configuration with a binding representative $b \equiv c = r(\texttt{c1}) \wedge i = r(\texttt{i1})$, and we want to compute $b \wedge i = r(\texttt{i2})$. If we did have precise points-to information then this would tell us that $r(\texttt{i1}) \neq r(\texttt{i2})$ (because we know for a fact that both iterators cannot be the same) and hence we would get:

$$\begin{aligned} & b \wedge i = r(\texttt{i2}) \\ \equiv\quad & c = r(\texttt{c1}) \wedge i = r(\texttt{i1}) \wedge i = r(\texttt{i2}) \\ \equiv\quad & c = r(\texttt{c1}) \wedge \mathbf{ff} \\ \equiv\quad & \mathbf{ff} \end{aligned}$$

However because we only know that $r(\texttt{i1}) \approx r(\texttt{i2})$, the analysis fails to reduce "$c = r(\texttt{c1}) \wedge i = r(\texttt{i1}) \wedge i = r(\texttt{i2})$" any further. With the many consecutive loops in `peelLoops(int)` this drastically increases the size and number of configurations that need to be computed before the fixed point is eventually reached. Even worse: because of the imprecise pointer information the analysis fails to identify any single nop-shadow.

```
void foo(Collection c1, Collection c2, Collection c3)
    Iterator i1 = c1.iterator ();
    while(i1.hasNext()) {
        i1.next();
        c2.add (..);
    }
    Iterator i2 = c2.iterator ();
    while(i2.hasNext()) {
        i2.next();
        c3.add (..);
    }
    Iterator i3 = c3.iterator ();
    while(i3.hasNext()) {
        i3.next();
    }
}
```

Figure 5.16: Worst-case example for complexity of Nop-shadows Analysis (in the case of imprecise points-to sets)

To guard the analysis from degenerating in such cases where imprecision is caused by the points-to analysis, we decided to determine the maximal number of configurations that is computed on a successful analysis run, i.e., a run that succeeds in identifying a nop shadow. This run was in method `visitBlock(Block)` of the class `EDU.purdue.cs.bloat.cfg.VerifyCFG`, not quite coincidentally in the same benchmark. The analysis computed 8828 configurations before it removed a shadow from this method; this is the maximal number that we observed on all benchmark runs. Then next we modified the Nop-shadows Analysis so that it would abort the analysis of a single method (continuing with the next one) whenever it computed a number of configurations that exceeded a given fixed quota. We defined this quota to be 15000, i.e., a number that still accommodates a significant number of additional configurations over the 8828 that we observed. We believe that this value is high enough to yield excellent precision in cases where pointer information is precise, but our experiments also showed that it is low enough to significantly decrease the overall analysis time in the benchmarks bloat-FailSafeIter, bloat-FailSafeIterMap and pmd-FailSafeIterMap. (In all other cases, the quota never exceeded 15000 and Clara aborted no analysis runs.)

The benchmarks bloat, chart and pmd all use reflection in connection with collections. For instance, Figure 5.17 shows a simplified version of a `clone` method in chart. The problem with Java is that, despite the fact that its `Cloneable` interface signifies that any object of a class tagged with this interface should support a `clone` operation, the interface does not actually demand that the class implements a publicly accessible `clone` method. The chart developers hence work around this shortcoming by calling the `clone` method reflectively when it exists. Such reflection greatly confuses the points-to analysis that we use, as the analysis has no idea which class's clone method will be called (reflection is not modeled precisely enough in Spark). As a result, there are many possible implementations to consider and the demand-driven analysis fails to compute context in its given quota.

```
public static Object clone(final Object object) throws CloneNotSupportedException {
    if (object == null) {
        throw new IllegalArgumentException("Null 'object' argument.");
    }
    if (object instanceof PublicCloneable) {
        final PublicCloneable pc = (PublicCloneable) object;
        return pc.clone();
    } else {
        final Method method = object.getClass().getMethod("clone",(Class[]) null);
        if (Modifier.isPublic(method.getModifiers())) {
            return method.invoke(object, (Object[]) null);
        }
    }
    throw new CloneNotSupportedException("Failed to clone.");
}
```

Figure 5.17: Clone method in chart using reflection

5.3 Experiments

As before, we applied the Nop-shadows Analysis to our benchmark set. Note that, because the Nop-shadows Analysis is flow-sensitive, this is not sound to apply the analysis to multi-threaded programs. When a program is multi-threaded then this means that the program's threads could execute shadows in an order that the Nop-shadows Analysis did not anticipate. After all, the Nop-shadows Analysis assumes that its intra-procedural control-flow graphs soundly model all possible control flow. In future work we plan to make our analysis thread safe by using a may-happen-in-parallel analysis [Bar05] to determine which methods could potentially execute in parallel. The benchmarks hsqldb, lusearch and xalan are multi-threaded. For now, we just analyzed these three benchmarks like all other benchmarks, i.e.,we made the un-safe assumption that these programs do not execute dependent-advice shadows in parallel.

5.3.1 Shadows eliminated by the Nop-shadows Analysis

Tables 5.3 and 5.4 show the number of shadows that remain before and after applying the Nop-shadows Analysis. As we can see, the Nop-shadows Analysis is very successful on the HasNext* patterns. This is hardly surprising: after all, programs mostly use iterators within a single method. Therefore, the Orphan-shadows Analysis yields its full potential in this case. There are, however, cases where the Nop-shadows Analysis fails even for these patterns. When this happens, this is due to issues with the points-to analysis. As we pointed out earlier, the analysis fails to compute context information for some of the iterators. When this happens, then the Nop-shadows Analysis has to take into account $succ_{ext}$ edges that are not actually realizable at runtime, making the analysis less precise. The Nop-shadows Analysis is also very effective on the Reader and Writer properties. These properties do not usually require context information because programs hardly ever create readers or writers inside factory methods. Hence, the Nop-shadows Analysis succeeds in these cases even when the points-to analysis fails to compute context. In case of fop-FailSafeIterMap, the analysis ran out of memory, despite the fact that we used 3 GB of maximal heap space. The call graph and pointer-assignment graph for the points-to analysis already take up a lot of space for this benchmark, and remaining memory is insufficient to compute the required information about the benchmark's 1114 shadows.

For luindex, lusearch, antlr and fop the Nop-shadows Analysis is very effective, often removing all instrumentation. The analysis manages to prove luindex sound with respect to all the properties. These benchmarks are relatively "well behaved", i.e., make little use of reflection and dynamic class loading, have relatively short-lived objects and therefore have a relatively small number of shadows per object (representative). bloat has very long-lived objects which are used in many different methods. This makes it hard for us to analyze the representatives of these objects and also causes problems for the points-to analysis. Just as chart does, pmd and jython also cause problems due to dynamic class loading. In many other cases in which the analysis cannot reduce the number of remaining shadows to zero, the analysis nevertheless manages to reduce the number of remaining shadows by large amounts.

Many of the few shadows that remain in lusearch actually do lead to a property violation at runtime. As we will discuss later, the benchmarks antlr, bloat, jython and lusearch all do violate some of the properties, which explains why shadows remain in these programs.

5.3.2 Number of potential failure groups

As Table 5.5 shows, as for shadows, the Nop-shadows Analysis was very effective on reducing the number of potential failure groups for benchmark/property combinations that involved the HasNext property. At the same time, the number of potential failure groups was not reduced a lot for the other properties. The number of remaining potential failure groups will be important for the next two chapters. In Chapter 6 we explain how to perform collaborative runtime verification using Clara. In collaborative runtime verification, every single user gets a partially instrumented program. The program is instrumented with a "probe", which is quite similar to a potential

	antlr		bloat		chart		fop		hsqldb	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	30	26							3	3
FailSafeEnum	3						7	6		
FailSafeIter			922	830	160	149				
FailSafeIterMap			446	444	49	49	1114	MEM		
HasNextElem	86						8		6	
HasNext			565	452	82	48	8			
LeakingSync										
Reader	14								3	3
Writer	44	35	19	15					10	10

(a) absolute number of shadows

	antlr	bloat	chart	fop	hsqldb
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	13				0
FailSafeEnum	100			14	
FailSafeIter		10	7		
FailSafeIterMap		0	0	MEM	
HasNextElem	100			100	100
HasNext		20	41	100	
LeakingSync					
Reader	100				0
Writer	20	21			0

(b) ratio of shadows disabled by Nop-shadows Analysis, in percent

MEM=OutOfMemoryException at compile time

Table 5.3: Enabled shadows before and after Nop-shadows Analysis, part 1

	jython		luindex		lusearch		pmd		xalan	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	76	61	15		5					
FailSafeEnum	47	44			5		10			
FailSafeIter	116	112	27		36	16	302	287		
FailSafeIterMap	151	133					314	204		
HasNextElem	47	34	16		6		6		3	1
HasNext	31	24	12		22		250	184		
LeakingSync										
Reader	4	4					24			
Writer							7			

(a) absolute number of shadows

	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll					
ASyncIterC					
ASyncIterM					
FailSafeEnumHT	20	100	100		
FailSafeEnum	6		100	100	
FailSafeIter	3	100	56	5	
FailSafeIterMap	12			35	
HasNextElem	28	100	100	100	67
HasNext	23	100	100	26	
LeakingSync					
Reader	0			100	
Writer				100	

(b) ratio of shadows disabled by Nop-shadows Analysis, in percent

Table 5.4: Enabled shadows before and after Nop-shadows Analysis, part 2

failure group. In Chapter 7 we explain how Clara ranks potential failure groups before it reports these groups to the user.

5.3.3 Runtime overhead after Nop-shadows Analysis

In Table 5.6 we show the reduction in runtime overhead that the Nop-shadows Analysis causes. We do not show lusearch because for this benchmark already the Orphan-shadows Analysis removed all of the monitoring overhead. As our results show, the analysis was able to remove all overhead from luindex, which is not surprising because the Nop-shadows Analysis removed all shadows from this benchmark for all properties. The analysis was equally effective in eliminating the overhead for antlr-HasNextElem. For bloat-FailSafeIterMap, the analysis reduced the overhead by large amounts. However, the remaining overhead is still very large, and large overheads also remain in many other cases. In the next chapter we will present an approach that can lower the runtime overhead further by performing partial instrumentation only.

5.3.4 Analysis time

As mentioned, the Nop-shadows Analysis runs after the Orphan-shadows Analysis and Quick Check have already been applied. Hence, the combined analysis time will always be at least as long as the time that we reported in the last chapter. However, the reader will find that the Nop-shadows Analysis often comes at a rather moderate cost. In all but two cases, the total compilation and analysis time including all three analysis stages was under ten minutes (in the last Chapter we reported nine minutes with only the first two analysis stages enabled). The combination bloat-FailSafeIterMap took almost 18 minutes in total, and bloat-FailSafeIter took just about 25 minutes in total. The bloat benchmark has some very large methods. Many of these methods use iterators an collections. This explains these extraordinarily high analysis times.

The Nop-shadows Analysis itself took under 50 seconds on average. This time includes all re-iterations of the Orphan-shadows Analysis and Nop-shadows Analysis

	antlr		bloat		chart		fop		hsqldb	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	6	6							1	1
FailSafeEnum	1						1	1		
FailSafeIter			259	259	38	38				
FailSafeIterMap			258	258	38	38	1	MEM		
HasNextElem	41						4		3	
HasNext			266	163	38	4	3			
LeakingSync										
Reader	4								1	1
Writer	3	1	1	1					1	1

	jython		luindex		lusearch		pmd		xalan	
	bef	aft	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll										
ASyncIterC										
ASyncIterM										
FailSafeEnumHT	24	9	4		2					
FailSafeEnum	2	2			1		2			
FailSafeIter	4	4	6		10	5	90	90		
FailSafeIterMap	4	4					32	32		
HasNextElem	26	14	8		3		3		2	1
HasNext	14	9	6		10		98	51		
LeakingSync										
Reader	1	1					5			
Writer							2			

MEM=OutOfMemoryException at compile time

Table 5.5: Potential failure groups before and after Nop-shadows Analysis

	antlr		bloat		chart		fop	
	bef	aft	bef	aft	bef	aft	bef	aft
ASyncContainsAll								
ASyncIterC								
ASyncIterM								
FailSafeEnumHT	4	4						
FailSafeEnum								
FailSafeIter			>1h	>1h	8	8		
FailSafeIterMap			>1h	22027			5	MEM
HasNextElem	6							
HasNext			329	258				
LeakingSync								
Reader								
Writer	38	36	229	228				

	jython		luindex		pmd	
	bef	aft	bef	aft	bef	aft
ASyncContainsAll						
ASyncIterC						
ASyncIterM						
FailSafeEnumHT	>1h	>1h	28			
FailSafeEnum						
FailSafeIter			4		526	524
FailSafeIterMap	14	13			>1h	>1h
HasNextElem			12			
HasNext					70	64
LeakingSync						
Reader						
Writer						

MEM=OutOfMemoryException at compile time

Table 5.6: Runtime overheads after applying Nop-shadows Analysis, in percent; hsqldb, lusearch and xalan show no perceivable overheads

that Clara performs. In 90% of the cases, the analysis finished in under one minute. The worst case is also here bloat-FailSafeIter with a little more than 19 minutes of analysis time for this stage. The average analysis time for a single shadow-bearing method was about half a second. The analysis of the method `toString` in the class `EDU.purdue.cs.bloat.context.CachingBloatContext` of the benchmark bloat in combination with FailSafeIter and FailSafeIterMap took, with around five minutes, by far the longest. The method is hard to analyze and the Nop-shadows Analysis reached its quota of 15000 configurations (see Section 5.2.3.6) before it aborted the analysis of this method. The method uses nine consecutive loops which use nine different iterators but for which the demand-driven context-sensitive points-to analysis that we use fails to compute context information. Therefore, our Nop-shadows Analysis has to consider the possibility that all of these iterators may actually be the same iterator. This, in turn, drastically increases the size of the configurations that the Nop-shadows Analysis computes. If we set this quota to a lower value then this would decrease the analysis time for the `toString` method. But, if lowered to a too small value, this may decrease the analysis precision for other methods.

Chapter 6
Collaborative runtime verification

In the last two chapters we have presented a set of static program analyses and optimizations that evaluate runtime monitors ahead of time. These analyses can often reduce and sometimes completely eliminate the performance penalties that runtime monitors induce. However, even the most sophisticated static analysis techniques will fail in some cases: as we saw, for some benchmarks much instrumentation remains even after applying all our static-analysis stages. This may be acceptable in a pre-deployment setting where developers can produce a large number of slow test runs on dedicated machines. But even then the runtime overhead that the instrumentation induces may be too large.

The situation is even worse when considering a setting in which instrumentation-carrying programs are deployed. In the verification community it is now widely accepted that, especially for large programs, verification is often incomplete, and hence bugs still arise in deployed code on the machines of end users. If deployed code carried instrumentation for runtime verification, developers could track down the causes of observed failures more easily.

In such a setting it is mandatory that the monitoring code only induces a low runtime overhead. According to researchers in industry [fMR07], companies would likely be willing to accept runtime verification in deployed code if the verification overhead was below 5%. Hence in this chapter, we show how to reduce the runtime verification-induced overhead further, using methods from remote sampling [LAZJ03].

Because companies that produce large pieces of software (which are usually hard to analyze) often have access to a large user base, one can leverage the size of the user base to deploy different partial instrumentation ("probes") for each user. A centralized server can then combine runtime-verification results from runs with different probes. Although sampling-based approaches have many different applications, we are most interested in using sampling to reduce instrumentation overhead for individual end users. We have developed an approach to partition this overhead, called *spatial partitioning*.

Spatial partitioning works by partitioning the set of instrumentation points into subsets. We call each subset of instrumentation points a *probe*. Each user is given a program instrumented with only a few probes. This works very well in many cases.

We explored the feasibility of our approach by implementing it as part of the Clara framework, and by applying this implementation to those benchmarks from our benchmark set whose overheads persisted after applying the static analyses from the previous chapters. We found that some benchmarks were very suited to spatial partitioning. In these cases, each probe produced lower overhead than the complete instrumentation, and many probes carried less than 10% overhead. Spatial partitioning only works if shadows remain that are not compatible to each other. In one of our benchmarks, all remaining shadows were compatible to each other and hence spatial partitioning failed for this benchmark. In other cases, it may happen that a single probe contains a very hot—that is, expensive—instrumentation point. In those cases, the unlucky user who gets the hot probe will experience most of the overhead.

In previous work [BHL+08], we explained how runtime-verification tools can cope even with such cases by applying a second partitioning approach that we called *temporal partitioning*. Temporal partitioning works by turning the instrumentation on and off periodically, limiting the total overhead. This method works even if there are very hot probes, because even those probes are only enabled some of the time. However, since probes are disabled some of the time, any violations of monitored runtime verification properties may go unnoticed while the probes are disabled. When trying to avoid false positives (a property that we consider crucial) then temporal partitioning requires specific knowledge of the monitor implementation at hand. In our previous work, for example, we implemented temporal partitioning for tracematches, and the implementation exploited special implementation details of tracematches. Unfortunately, Clara's dependency annotations currently do not provide Clara with enough information to implement temporal partitioning in a general way that would apply to every AspectJ-based runtime-monitoring tool. In future work we plan to determine a minimal extension to Clara's annotation languages that would then provide Clara with the appropriate information. In this dissertation, on the other hand, we focus on what one can achieve with dependency annotations alone, and therefore only consider spatial partitioning.

We structured the remainder of this chapter as follows. We explain the concepts of spatial partitioning in Section 6.1, and an implementation based on these concepts in Section 6.2. We discuss our experiments in Section 6.3.

6.1 Spatial partitioning

Spatial partitioning reduces the overhead of runtime verification by only leaving a subset of a program's shadows enabled. However, choosing an arbitrary subset of shadows is more than unsatisfactory; in particular, arbitrarily disabling shadows for weakly referenced symbols may lead to false positives. Consider the example from Figure 6.1, in combination with the HasNext pattern (page 4). The example program was taken from an early version of our own implementation of Clara. The program uses two different iterators, "`entryIter`" and "`iterator`", to print the contents of a map to a StringBuffer. We have underlined the shadows at which events occur that are of interest to the HasNext pattern. In this example, one safe spatial partitioning would be to disable all shadows in the program except for those referring to `entryIter` (lines 3, 4 and 17). However, many partitionings are unsafe; for instance, disabling the `hasNext` shadow on line 3 (an exit shadow, i.e., a shadow that may prevent a match) but enabling the `next` shadow on line 4 (a progress shadow) on a map with two or more entries gives a false positive, since the monitor "sees" two calls to `next()` and not the call to `hasNext()` that would prevent the match.

```
private void mapToString(Map<String, List<String>> map, StringBuffer sb) {
    for (Iterator<Map.Entry<String, List<String>>> entryIter =
            map.entrySet().iterator(); entryIter.hasNext();) {
        Map.Entry<String, List<String>> entry = entryIter.next();
        sb.append(entry.getKey());
        List<String> args = entry.getValue();
        if (!args.isEmpty()) {
            sb.append(”(”);
            for (Iterator<String> iterator = args.iterator(); iterator.hasNext();) {
                String varName = iterator.next();
                sb.append(varName);
            if(iterator.hasNext())
                    sb.append(”,”);
            }
            sb.append(”)”);
        }
        if(entryIter.hasNext()) {
            sb.append(”,”);
        }
        sb.append(” ”);
    }
    sb.append(”\n”);
}
```

Figure 6.1: Example program using two iterators

Enabling arbitrary subsets of shadows can also lead to wasted work. Disabling the `next` shadow in the above example and keeping the `hasNext` shadow enabled would, of course, lead to overhead from the `hasNext` shadow. But, on their own, `hasNext` shadows can never lead to a complete match without any `next` shadows: in the HasNext pattern, `hasNext` shadows can only force the runtime monitor to exit a state; `hasNext` shadows are no progress shadows. We therefore need a more principled way of determining sensible groups of shadows to enable or disable.

In Chapter 4 we have already seen a means to describe sets of symbols (and shadows) that, in combination, can trigger a property violation: dependency annotations. For HasNext, Algorithm *generateDependencies* would generate the dependency declaration **dependency**{ **strong** next; **weak** hasNext; }, which conveys the information that we need to see `next` to see a pattern violation and, given that `next` transitions may occur with respect to some iterator, we also need to keep alive `hasNext` shadows for this iterator.

In spatial partitioning, we use dependency annotations to compute a set of different "probes". Each probe is a set of shadows that is both consistent and complete. The probe is consistent because we require that all shadows that are part of the probe must be compatible with each other (with respect to the dependency declaration that defines the probe). The probe also has to be complete: no shadow that is compatible may be left out.

Let $\mathcal{S}$ be the set of all still-enabled shadows in the program. As before, for $s \in \mathcal{S}$ we write $\mathit{label}(s)$ to denote the symbol for which s triggers an automaton transition.

Definition 6.1 (Strong and weak shadows)**.** Let $d \in \mathcal{D}$ be a dependency declaration. For d, we define two sets $\mathit{strongShadows}(d)$ and $\mathit{weakShadows}(d)$ as follows:

$$\begin{aligned}\mathit{strongShadows}(d) &:= \{s \in \mathcal{S} \mid \mathit{label}(s) \in \mathit{strong}(d)\}\\ \mathit{weakShadows}(d) &:= \{s \in \mathcal{S} \mid \mathit{label}(s) \in \mathit{weak}(d)\}\end{aligned}$$

We would like a probe to be sufficiently large so that it can lead to a match, however otherwise minimal in the sense that it contains the least number of shadows to reach this match. In the following we therefore define "minimal activation sets".

Every minimal activation set contains exactly the minimal set of shadows that is necessary to reach a final state, for a given dependency d.

Definition 6.2 (Minimal activation set)**.** We define *minActivationSets*(d) as a set of sets of shadows:

$$\mathit{minActivationSets}(d) := \\ \{ss \subseteq \mathit{strongShadows}(d) \mid \forall l \in \mathit{strong}(d) \ \overset{1}{\exists}\ s \in ss\ .\ \mathit{label}(s) = l\}$$

Here, $\overset{1}{\exists}$ denotes "there exists exactly one". Hence, *minActivationSets*(d) contains all subsets ss of strong shadows of d such that there is exactly one shadow s in ss for every strong label in d. In other words, all the sets of shadows in *minActivationSets*(d) are minimal in the sense that they contain the minimal set of shadows necessary to activate a dependency declaration.

Definition 6.3 (Probe)**.** For any set of shadows $ss \subseteq \mathcal{S}$ and dependency declaration d we define *compShadows*(ss, d) as the set of all shadows that are compatible (see page 86) to a shadow of ss, with respect to d:

$$\mathit{compShadows}(ss, d) := \bigcup_{s \in ss} \{s' \in \mathcal{S} \mid \mathit{stCompatible}(s, \mathit{label}(s), s', \mathit{label}(s'), d)\}$$

Here, *stCompatible* is the same function as the one defined on page 86. We then define *probes*$(d) \subseteq \mathcal{P}(\mathcal{S})$, the set of d's probes as:

$$\mathit{probes}(d) := \{\ ss \cup ws \mid ss \in \mathit{minActivationSets}(d) \land ws = \mathit{compShadows}(ss, d)\ \}$$

Informally, a probe hence consists of a fixed number of strong shadows that make up a consistent (all shadows are compatible) minimal activation set, and of all shadows that are compatible to those strong shadows.

Probes by example. Consider again the example program from Figure 6.1 in combination with the HasNext property. As mentioned, the dependency declaration d := **dependency**{ **strong** next; **weak** hasNext; } is the only one generated for the

HasNext property. We denote a shadow s at line number n by $(\mathit{label}(s), n)$, yielding:

$$\begin{aligned} \mathit{strongShadows}(d) &:= \{(\texttt{hasNext}, 3), (\texttt{hasNext}, 17), (\texttt{hasNext}, 9), (\texttt{hasNext}, 12)\} \\ \mathit{weakShadows}(d) &:= \{(\texttt{next}, 4), (\texttt{next}, 10)\} \\ \mathit{minActivationSets}(d) &:= \{\{(\texttt{next}, 4)\}, \{(\texttt{next}, 10)\}\} \end{aligned}$$

In the program, `entryIter` and `iterator` cannot point to the same object. Hence, their shadows are not compatible among each other and we get two probes:

$$\begin{aligned} \mathit{probes}(d) := \{ \quad & \{(\texttt{next}, 4), (\texttt{hasNext}, 3), (\texttt{hasNext}, 17)\}, \\ & \{(\texttt{next}, 10), (\texttt{hasNext}, 9), (\texttt{hasNext}, 12)\} \quad \} \end{aligned}$$

The first probe contains all shadows related to `entryIter`, the second all shadows related to `iterator`.

Probes may overlap. Note that this example may mislead the reader into thinking that probes always have to be disjoint. As it turns out, probes are only disjoint for properties like HasNext, that only use one free variable; in this case, the partitioning into probes is a true partitioning that leads to disjoint classes of shadows. In cases where a property refers to multiple variables, however, probes may overlap. For instance, consider the simplified code example in Figure 6.2 in combination with the FailSafeIter tracematch. Again, we have underlined the relevant joinpoint shadows.

In this example, Algorithm *generateDependencies* would yield the single dependency declaration **dependency**{ **strong** create, next, update; }. Because the collection is modified at line 6, we have an `update` shadow at this line, and get:

$$\begin{aligned} \mathit{probes}(d) := \{ \quad & \{(\texttt{create}, 2), (\texttt{next}, 4), (\texttt{update}, 6)\}, \\ & \{(\texttt{create}, 3), (\texttt{next}, 5), (\texttt{update}, 6)\} \quad \} \end{aligned}$$

In this setting, both probes contain the `update` shadow at line 6. This is because clearing the collection `c` may lead to a property violation on either iterator when the respective iterator is progressed after `c` has been modified. Using our approach to collaborative runtime-verification, the workload for verifying this program may

```
void printAndClear(Collection c) {
    Iterator i1 = c.iterator();
    Iterator i2 = c.iterator();
    i1.next();
    i2.next();
    c.clear();
}
```

Figure 6.2: Example program violating the FailSafeIter property for two iterators

be distributed among two users: one user would monitor the first probe, detecting property violations with respect to the iterator referenced by `i1`, while the other user would monitor the other probe for the second iterator.

Spatial partitioning can guarantee complete coverage. Note that, by its definition, spatial partitioning can guarantee complete coverage in the following sense. Assume a program run r that leads to a property violation in the fully instrumented program. Further assume that we obtain k different probes through spatial partitioning, and we distribute differently instrumented versions of the program to a number of users, such that every single one of the k probes is enabled in at least one user's program version. When all these users now re-execute the same program run r then there will be at least one user for which r causes the runtime monitor to notify this user about the property violation.

6.2 Implementation

There are generally two ways to implement spatial partitioning. For n different probes, one could in principle generate up to $2^n - 1$ different program copies where each copy enables a different subset of probes. We instead opted for a more flexible approach where we generate only one program copy. This copy then carries special instrumentation that can select a subset of probes to enable when the program is started. In the following, we present our algorithm for spatial partitioning. We first explain how we compute all probes (based on the flow-insensitive Orphan-shadows Analysis from Section 4.2.3.2). Then, in Section 6.2.2, we explain how we generate supportive data structures for selecting probes at program start-up. We generate bytecode for two arrays: a "map" array mapping from probes to shadows and a "flag" array with one entry per shadow. When emitting code for shadows, we guard each shadow's execution with appropriate array look-ups.

6.2.1 Computing the probes

We compute probes using two algorithms. First, the algorithm *minActivationSets* (Algorithm 6.1) computes all minimal activation sets, i.e., the "*ss*" sets from Definition 6.3. The shadows of each minimal activation set potentially suffice to drive the runtime monitor into a final state when the program under test executes. However, if we just instrumented the program under test with a minimal activation set, then this program could report false positives because we have not yet taken any weak shadows into account. False positives put a burden on the programmer. Therefore Algorithm 6.2 (page 201), called *probes*, adds to every minimal activation set all the strong and weak shadows that are compatible to the probe's minimal activation set. This combined set of shadows forms the probe. When we instrument a program partially, with a given probe, then this program cannot report any false positives because, for any shadow (from the minimal activation set) that can drive the monitor into a final state, we also enabled all compatible shadows that could potentially prevent the monitor from reaching the final state.

Because Algorithm *minActivationSets* is non-trivial, we discuss it in more detail. The algorithm accepts as parameters a dependency declaration d, for which it should generate probes, and a mapping *shadowsOf* from dependent advice to a list of all shadows that are labeled with this piece of advice. It is important that the mapping maps to lists, not sets: the algorithm needs to traverse the shadows in a fixed ordering (the ordering is arbitrary, but it must be fixed for the execution of the algorithm).

Algorithm 6.1 *minActivationSets* $(d, \mathit{shadowsOf})$ with $d \in \mathcal{D}$, $\mathit{shadowsOf} : \mathcal{A} \to \mathcal{S}^*$

```
1: $res = \emptyset$
2: let $\mathit{strongShadows}$ be an ordered list containing all shadows of $\mathit{strong}(d)$
3: // for all shadows labelled with first strong label
4: for all shadows $s_0$ labeled with first advice in $\mathit{strongShadows}$ do
5:   push $s_0$ on $\mathit{stack}$ // try to find a minimal activation set containing $s_0$
6:   $lastShadow := null$ // no last shadow for next strong label yet
7:   repeat
8:     let $s$ be the shadow at the top of $\mathit{stack}$
9:     // search for shadow $s'$ to complete the current minimal activation set
10:    if there is a successor symbol $l'$ of $\mathit{label}(s)$ in $\mathit{strongShadows}$ then
11:      if there is a successor shadow $s'$ to $lastShadow$ in $\mathit{shadowsOf}(l')$ that is
         compatible to all shadows on $\mathit{stack}$ then
12:        push $s'$ on $\mathit{stack}$
13:      end if
14:    end if
15:    if $|\mathit{stack}| = |\mathit{strong}(d)|$ then // if completed the minimal activation set
16:      $res = res \cup \left\{ \{s \mid s \text{ on } \mathit{stack}\} \right\}$ // commit minimal activation set
17:      pop $lastShadow$ off the $\mathit{stack}$ // search for next set
18:    else
19:      if pushed shadow on the $\mathit{stack}$ (executed line 12) then
20:        $lastShadow := null$ // successfully added shadow, move to next symbol
21:      else
22:        pop $lastShadow$ off the $\mathit{stack}$ // try another shadow, after $lastShadow$
23:      end if
24:    end if
25:  until $\mathit{stack}$ is empty // no more choices
26: end for
27: return res
```

Algorithm 6.2 *probes* $(d, \mathit{shadowsOf})$ with $d \in \mathcal{D}$, $\mathit{shadowsOf}\colon \mathcal{A} \rightarrow \mathcal{S}^*$

```
1:  $\mathit{probes} := \emptyset$
2:  for set $ss \in \mathit{minActivationSets}(d, \mathit{shadowsOf})$ do
3:      $ws := \emptyset$
4:      for $s \in \mathcal{S}$ with $\mathit{label}(s) \in \mathit{all}(d)$ do
5:          if $s$ is compatible to all shadows in $ss$ then
6:              $ws := ws \cup \{s\}$
7:          end if
8:      end for
9:      $\mathit{probes} = \mathit{probes} \cup \{ss \cup ws\}$
10: end for
11: return $\mathit{probes}$
```

The algorithm computes exhaustively all minimal activation sets. Starting with the first label in *strong*(d), it attempts to add exactly one compatible (line 11) shadow s' to the current probe for each label (line 10) in *strong*(d). To allow for backtracking, the algorithm holds the partial minimal activation sets in a stack. Once the minimal activation set becomes complete (the size of the stack equals the size of *strong*(d); line 15), it is committed to the set res. Note that in line 11, where the algorithm looks for a successor shadow to $lastShadow$, in case that $lastShadow == null$ the algorithm will just pick any shadow of *shadowsOf*(l') that is compatible to all shadows on $stack$.

The complexity of the algorithm mentioned above is bounded by the number of minimal activation sets. Estimating this number is non-trivial. In Appendix D we compute a reasonable upper bound for this number. In our implementation, Algorithm *minActivationSets* operates on a two-dimensional boolean matrix in which entry (i, j) is true if shadows i and j are compatible. This makes our implementation very efficient.

The more of the remaining shadows are compatible, the more minimal activation sets we will get. However, on the other hand, when Algorithm 6.2 adds compatible shadows to the minimal activation sets then this will result in fewer possible probes, the more of the remaining shadows are compatible. In our experiments, the maximal number of probes that we obtained was 3343. This means that the programmer has the option to spread the instrumentation over as many as 3343 users. However, there is no need to have that many users: one can choose to have a smaller set of users and then simply enable more than one probe at a time for every user.

6.2.2 Generating the arrays that encode the probes

The arrays, along with some glue code in the AspectJ runtime, allow us to dynamically enable and disable probes as desired (using array look-ups). In the context of spatial partitioning, we choose one probe to enable at the start of each execution; however, our infrastructure permits experimentation with more sophisticated partitioning schemes.

In our running example, the "map" array, which maps from probes to shadows, would look like this:

map	3	4	9	10	12	17	← line number
0	**tt**	**tt**	ff	ff	ff	**tt**	
1	ff	ff	**tt**	**tt**	**tt**	ff	

For presentation purposes we show for every shadow the line number of the shadow. Our implementation does not store these line numbers. The array encodes the information that the shadows at lines 3, 4 and 17 all belong to probe number 0, and the shadows at lines 9, 10 and 12 belong to probe number 1. To index into the array, we assign each probe and each shadow a unique number, starting at 0. Note that, when different probes may overlap, there may be multiple rows with a **tt** entry in the same column.

The "flag" array, with one entry per shadow, is initialized to all **ff**:

flag	3	4	9	10	12	17
	ff	ff	ff	ff	ff	ff

If we then assume that the user chooses to enable probe number p, the CLARA runtime sets the Boolean flags in the "flag" array for all shadows of this probe (as determined by the "map" array) to **tt**.

$$\forall i \in \{0, \ldots, |\mathcal{S}| - 1\} \quad : \quad \text{flag}[i] := \text{flag}[i] \vee \text{map}[p, i]$$

For our example, with $p = 1$, this would result in:

flag	3	4	9	10	12	17
	ff	ff	**tt**	**tt**	**tt**	ff

6.2.3 Generating the runtime checks

We also add a dynamic residue to every advice application of every dependent advice. The residue checks the flag for the shadow at which this advice application applies. The advice will only execute at this shadow when the flag is set to **true**.

6.3 Experiments

To validate the feasibility of spatial partitioning, we selected five representative benchmark/property combinations that expose a high runtime overhead even after we had applied all three of CLARA's static analysis stages. We then applied the spatial-partitioning algorithm from Section 6.1 to these five combinations. In CLARA, programmers can easily enable spatial partitioning through a simple command-line flag. CLARA then automatically runs the spatial-partitioning algorithm just after all static analyses have been completed. Because spatial partitioning requires points-to information, CLARA requires that at least the Quick Check and Orphan-shadows Analysis are computed before the spatial-partitioning algorithm is run.

Table 6.1 shows the runtime overheads with full instrumentation and also presents the number of probes generated for each benchmark. For informational purposes we also show the number of minimal activation sets that these probes comprise. Recall that the number of probes depends heavily on the precision of the underlying points-to analysis, as well as intrinsic properties of the benchmark, for instance the lifetimes of bound objects: If objects are longer-lived then they tend to be referenced at more program places than rather short-lived objects. Therefore in these cases many variables share the same points-to sets, yielding larger probes. If the points-to analysis is imprecise (e.g. because it soundly over-approximates dynamic class loading) this may lead to larger points-to sets that overlap a lot among another, which in turn causes probes to be merged that would otherwise remain separate had the points-to analysis been more precise. Consequently, the number of probes varies a lot from case to case depending on these properties. antlr-Writer is an extreme case: here all of the 36 shadows that remain after applying the Nop-shadows Analysis overlap, and hence form only a single probe. One could therefore say that spatial partitioning fails in this case. When this happens, one has to resort to monitor-specific optimizations like temporal partitioning [BHL+08].

Under the spatial-partitioning approach, Clara emits instrumented benchmarks which can enable or disable each single probe at program start-up. We tested the effect of each probe individually by executing each benchmark with one probe enabled at a time. (Note that it is generally possible to enable multiple probes at the same time without jeopardizing soundness.) Because the five different benchmark/property combinations together contain 5428 different probes, this would give us the opportunity to perform up to 5428 different test runs. So many test runs would consume very much time. Hence we randomly picked 100 probes for each combination, except for antlr-Writer, because this combination only contains one probe. We then executed the respective instrumented program with one of these probes enabled and all other probes disabled. To reduce the overall benchmarking time further, we parameterized the DaCapo harness so that it would report the mean runtime of a minimum of 3 runs and not 5 as before. In the end we obtained $1 + 4 \times 100 = 401$ different benchmark runs. Figure 6.3 shows runtime overheads for the probes in our benchmarks. Dots indicate overheads for individual probes. For some benchmarks, many probes were almost identical, sharing the same hot shadows. These probes therefore also had almost identical overheads. We grouped these probes into batches and present them as bars. We labelled each bar with the number of probes that the bar represents.

benchmark	overhead (%)	min. activation sets	probes
antlr-Writer	36	289	1
bloat-HasNext	258	172	163
chart-FailSafeIter	8	9061	1766
jython-FailSafeIterMap	13	5040	155
pmd-FailSafeIter	524	54215	3343

Table 6.1: Full overheads and number of minimal activation sets and probes

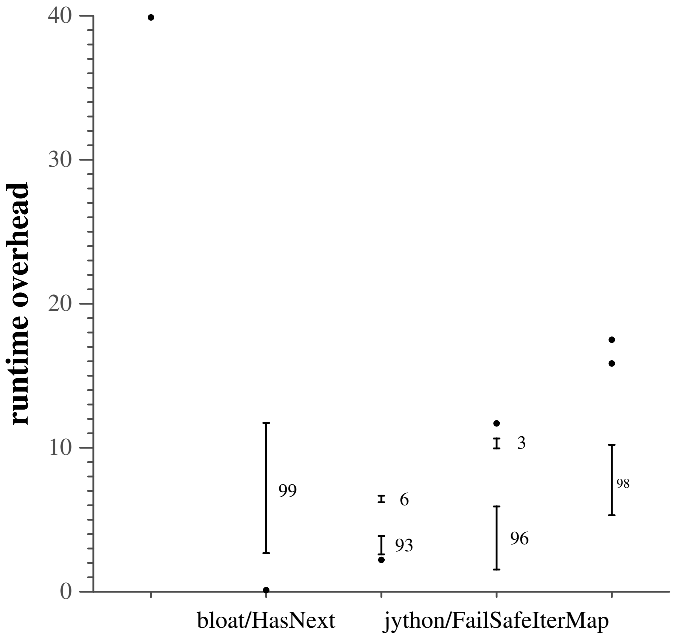


Figure 6.3: Runtime overheads per probe in spatial partitioning (in percent; bars indicate batches of probes, labelled by size of clump)

Our results demonstrate that, in some cases, the different probes manage to spatially distribute the overhead quite well. For the combinations chart-FailSafeIter and jython-FailSafeIterMap the gains are moderate. This is because these benchmark only caused 8%, respectively 13% overhead in the first place. Nevertheless, even for these two benchmarks there are many probes that produce a significantly smaller overhead. For bloat-HasNext and pmd-FailSafeIter, spatial partitioning manages to bring down the overhead from 2.5 fold, respectively 5.2 fold to less than 10% in most cases. However, spatial partitioning does not always suffice. For antlr-Writer there is only one probe and so it is not a surprise that we observe a runtime overhead similar to the one that we would obtain without spatial partitioning.

We conclude that spatial partitioning can often be effective in spreading the overhead among different probes. In some cases, like antlr-Writer, spatial partitioning may however fail.

Chapter 7
Ranking and filtering

Although our static analysis removes many false positives, factors like dynamic class loading and inter-procedural data flow can still lead to imprecise analysis results; for some of our benchmarks we still report up to 259 potential failure groups. In such a case, the large number of potentially interesting program points makes it hard for the programmer to investigate all these points manually. Moreover, it may also be problematic to test-run a program that is instrumented at so many program points: possibly, the runtime overhead will be high. Hence, manually inspecting the program may be the only option to find out whether or not the program under test actually does violate the property in question. If we cannot avoid a manual inspection then at least we want to ease the programmer's inspection task as much as possible. In this chapter we present an approach that helps the developer focus on the most important points first, by ranking, i.e., sorting, the list of potential failure groups that we report. But how can we determine which of the potential failure groups will most likely be relevant?

7.1 Approach

As we describe in previous work [BLH08a], we manually investigated many of the potential failure groups that remained even after applying all our static analyses. As we found out, we can categorize any remaining shadow as follows:

nop-shadows The shadow is actually a nop shadow but the analyses fail to identify this fact, due to some imprecision.

conservative property definition The shadow remains active because the program actually violates the stated property at runtime. However, it turns out that the property definition was wrong; it was too conservative: the program safeguards itself from failing by means that the finite-state property does not capture. For example, a program may cause the HasNext pattern to match by calling `next` twice on an iterator. However, if the collection to which this iterator belongs must have multiple elements, then these calls may still be correct.

program error The shadow remains active because the program actually violates the stated property at runtime, and the property indicates an actual programming error.

Penalizing failure groups that carry imprecisions. It is impossible for our static analyses to tell whether a shadow belongs into the second or third category. After all, Clara only determines whether the given program may violate the given finite-state property. If the property does not accurately describe the error situation that the programmer actually wishes to express, then there is nothing that Clara could do to change this fact. Clara therefore leaves it to the programmer to specify the properties of interest as accurately as possible. In Section 8.7 we will discuss some approaches that try to infer accurate specifications from existing program code. These approaches can ease the programmer's task.

Nevertheless, Clara can actively help to determine whether a shadow falls into the first category, i.e., whether it is likely that the shadow is a nop shadow, although some imprecision caused the analysis to fail to identify this fact. The idea is that in certain cases we can determine that we do have imprecise analysis information for a shadow s. In this case, we will give any potential failure group that contains s a "penalty" value. The higher the penalty of a potential failure group, the further to the bottom of the ranked list of potential failure groups the group will appear.

Ranking criteria. We decided to assign a potential failure group a penalty value if for one of the group's shadows one or more of the following criteria hold.

`ABORTED`: CLARA aborted the Nop-shadows Analysis because the analysis exceeded its quota (see page 178)

`DELEGATE`: the shadow is at a delegating statement (see below).

`NO_CONTEXT`: the shadow is associated with points-to sets that contain no context information—that is, the demand-driven analysis failed to find context information in its time budget.

For each of these three criteria we assign an equal penalty value. We assign a penalty to shadows at delegating calls for the following reason: Assume, for instance, an (unchecked) call `inner.next()` within `wrapper.next()`. According to the Nop-shadows Analysis, such a call could violate the HasNext property, because the `inner.next()` call is not preceded by an `inner.hasNext()` call. However, such potential violations are uninteresting for error detection because clients use `inner` correctly whenever they use `wrapper` correctly. The penalty value will cause such less interesting cases to show up further towards the bottom of the ranked result list.

In our view, a ranking approach is useful to the programmer if it ranks those potential failure groups at the top that (1) describe actual property violations, or (2) for which a programmer can easily tell whether or not the potential failure group describes an actual property violation. The penalty value that we obtain from the three criteria mentioned above helps to determine case (1). To facilitate (2), when CLARA decides whether to rank a potential failure group g_1 before or after a group g_2, if g_1 and g_2 have the same penalty value, then CLARA will show g_1 first if g_1 has fewer shadows than g_2. This is because the larger a potential failure group is, the harder it will be for a programmer to determine whether or not this group describes an actual property violation.

7.2 Experiments

To determine whether our criteria really do rank actual property violations to the top, we inspected all the program points at which shadows remained active after applying the Nop-shadows Analysis. For these points, we marked actual property violations with a flag `ACTUAL`, giving us training data. We also ran the respective benchmarks with monitoring instrumentation to validate the results gained by manual inspection. Interestingly, the benchmark harness did not actually exercise many of the matches that we had identified. This is one of the reasons for why static analysis is useful: because it considers all paths, it provides complete coverage. We then used the Weka machine-learning toolkit [WF05] to create decision trees which would distinguish false positives from true positives. To generate input for Weka, we interpreted the three penalty criteria as features of a feature vector. Weka then determined how the three values in these vectors correlate with the presence of the `ACTUAL` flag.

For our data set, Weka computes the decision tree shown in Figure 7.1. This tree tells us that, as we anticipated, if none of `NO_CONTEXT`, `ABORTED` or `DELEGATE` apply to a potential failure group then one should best decide that this group is a true positive, otherwise as a false positive.

```
NO_CONTEXT = 0
|   ABORTED = 0
|   |   DELEGATE = 0: TRUE_POSITIVE (5.0)
|   |   DELEGATE = 1: FALSE_POSITIVE (7.0)
|   ABORTED = 1: FALSE_POSITIVE (33.0)
NO_CONTEXT = 1: FALSE_POSITIVE (949.0/2.0)
```

Figure 7.1: Decision tree computed by Weka

Weka evaluates its classifiers with 10-fold stratified cross-validation to ensure that the classifiers do not over-fit the data. Cross-validation is a statistical technique that allows the estimation of error without distinct training and testing sets. Cross-validation estimates the error by using a subset of a data set as training data and then applying the trained model to the remainder of the data set. Ten-fold stratified cross-validation is a cross-validation technique that has been proven statistically stable [Koh95].

As Figure 7.1 shows, under cross-validation the decision tree correctly identifies five out of seven actual failure groups, but it incorrectly classifies two actual failure groups as false positives (both due to `NO_CONTEXT`). The tree further correctly identifies all 989 false positives as false positives. This means that these false positives would not show up if we filtered the list of potential points of failure according to this decision tree. However, the filtered list would also not contain two of the seven actual failure groups that we identified through manual inspection. In these cases, our feature vectors do not contain enough information to deduce the correct answer. As the decision tree shows, `NO_CONTEXT` is the criterion that is most useful to us: when we have no context information for a shadow s then it is very likely that the Nop-shadows Analysis will fail to identify s as a nop-shadow, even when it is one. Hence, `NO_CONTEXT` is a large source of false positives in the un-filtered list of potential failure groups.

Result of filtering and ranking. In Table 7.1, we show for every benchmark that has shadows remaining after applying the Nop-shadows Analysis the number of potential failure groups (PFGs), both before and after filtering. Where applicable, we show the number of false negatives in brackets. We also show the number of actual failure groups (AFGs) identified through manual inspection. In the last column, we show the ranks of these actual failure groups in the ranked but un-filtered list of potential failure groups. Note that through filtering we can pinpoint the actual failure group in bloat-HasNext. Also note that we filter out all potential failure groups for benchmarks that have no actual failure. Hence the false-positive rate is zero for our benchmark set. However, as the table shows, we also miss two actual violations because the filtering is too eager in this case. In the case of jython-Reader, the filter would filter out the only actual failure group (due to `NO_CONTEXT`). However, since the benchmark only contains this one potential failure group to begin with, the programmer would likely not have enabled the filter anyway. For pmd-HasNext, the filter also incorrectly classifies one potential actual failure group as a false positive (also due to `NO_CONTEXT`). In the un-filtered ranked list of potential failure groups, this group shows up at position 7.

Suspicious code and defects. Using our filtered results, we identified the following defects and pieces of suspicious code in our benchmarks by manually inspecting the program code. In pmd-HasNext, a method passes an iterator i to another method. The callee method then extracts i's first element without further checks. While this is not an actual bug, the undocumented precondition (that i has a next element) on the callee might cause problems for long-term software maintenance. Interestingly, pmd's developers fixed the method in a later version of pmd by using Java5's for-each loops, which avoid the explicit use of iterators. Our filter misclassifies this actual failure point as a false positive (producing a false negative) because the potential failure group is marked with `NO_CONTEXT`.

The pmd benchmark also calls `next()` on iterators without calling `hasNext()` before. This is not an error because pmd makes sure that the iterator can be advanced by checking the size of the underlying collection. Although not an error, these cases still constitute a violation of the stated property.

The actual point of failure in jython-Reader indicates an actual defect. The code may close `Reader` objects and then read from them. The developers "cured" this defect by returning `null` from the method that reads from the Reader, in case of an `IOException`. Unfortunately, as Table 7.1 shows, Clara would filter out this actual error when filtering is enabled, also due to `NO_CONTEXT`.

The bloat benchmark extracts two elements from a collection without any checks, leading to an actual match in the bloat-HasNext benchmark.

Conclusion. This concludes our description of Clara. We have described how programmers can use Clara to analyze runtime monitors ahead of time. We also showed how these analyses enable the programmers to find programming errors earlier in the development process or faster at runtime. We next discuss related work.

benchmark-property	PFGs before filtering	PFGs after filtering	AFGs	AFG positions in ranked list
antlr-FailSafeEnumHT	6	1	1	1
antlr-Writer	1			
bloat-FailSafeIter	259			
bloat-FailSafeIterMap	258			
bloat-HasNext	163	1	1	1
bloat-Writer	2			
chart-FailSafeIter	38			
chart-FailSafeIterMap	38			
chart-HasNext	4			
fop-FailSafeEnum	1			
hsqldb-FailSafeEnumHT	1			
hsqldb-Reader	1			
hsqldb-Writer	1			
hsqldb-Writer	1			
jython-FailSafeEnumHT	9			
jython-FailSafeEnum	2			
jython-FailSafeIter	4			
jython-FailSafeIterMap	4			
jython-HasNextElem	14			
jython-HasNext	9			
jython-Reader	1	0 (-1)	1	1
lusearch-FailSafeIter	5			
pmd-FailSafeIter	90			
pmd-FailSafeIterMap	32			
pmd-HasNext	51	3 (-1)	4	1,2,3,7
xalan-HasNextElem	1			

Table 7.1: Number of potential failure groups (PFGs) before and after filtering, number of actual failure groups (AFGs), and positions of AFPs in ranked, un-filtered list of PFGs

Chapter 8
Related work

We first discuss, in Section 8.1, how our analysis relates to the general subject of typestate analysis. In Section 8.2 we then discuss three very related approaches that analyze tracematches statically, one by Naeem and Lhoták and two from our own previous work. Section 8.3 presents other approaches to runtime monitoring and explains which of these approaches programmers could use in combination with Clara. In Section 8.4 we point out why we decided to use Sridharan and Bodík's points-to analysis, what properties this analysis has and how exactly we use the analysis. We present related work on aspect-oriented programming in Section 8.5, while Section 8.6 shows how Clara relates to more lightweight static checking tools. Lastly, in Section 8.7, we present a number of tools that can infer specifications from a program's code or execution traces. In the future, programmers may use such automatically inferred specifications in combination with Clara.

8.1 Typestate analysis

The scientific literature has frequently referred to the finite-state verification problem that we describe in this dissertation under the name of typestate analysis.

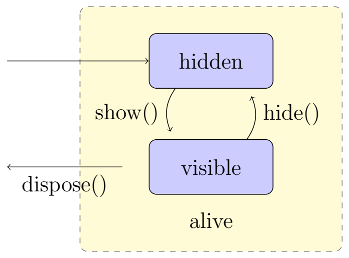
8.1.1 Strom and Yemini

In their original paper on typestate [SY86], Strom and Yemini first describe the idea of having a value's type depend on an internal state, the typestate, associated with that value. Certain operations can change a value's type by transitioning from one typestate to another. Strom and Yemini used state charts [Har87] to describe the possible state transitions for a class of objects. Statecharts allow programmers to capture behavioural sub-typing [LW94], an important property of object-oriented languages. Figure 8.1 shows three example statecharts. The first state chart (8.1a) models a visual Frame as it exists in the Swing framework. When created, the frame is hidden and therefore the frame starts off in state *hidden*, a sub-state of *alive*. The frame can switch to state *visible* by calling the `show()` method. When the program disposes a frame, the monitor leaves the state *alive* (and all its sub-states).

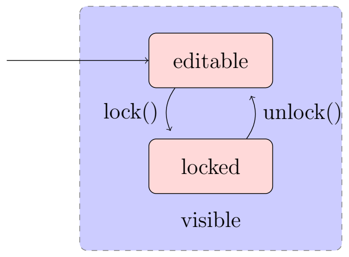
A subtype of Frame can then refine Frame's protocol as shown in Figure 8.1b. This subtype implements a lockable frame whose widgets can be locked from modification, but only if the frame is currently visible. Hence, the statechart refines the state *visible* with two new states, *editable* and *locked*. The semantics of statecharts can be defined through "flattening" as shown in Figure 8.1c where the refined state appears inlined in the statechart for Frame.

Clara currently does not allow programmers to enter dependency state machines in the form of statecharts, however we will consider such a feature for a future release of Clara. The implementation should be straightforward: one can always flatten state charts so that the flattened state chart is effectively a non-deterministic finite-state machine.

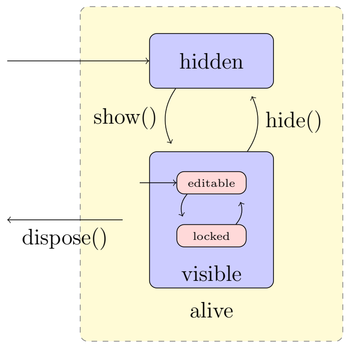
In the description by Strom and Yemini, typestate properties are restricted to describing the state of single objects. For example, their model does not allow the state of an iterator i to change when the iterator's collection c is modified. This is because the author's model has no means of associating i with c. Recently, typestate properties have been enjoying renewed interest, and many current analyses, including ours, do support the analyses of such "generalized" typestate properties.



(a) Statechart modelling behaviour of a frame



(b) Statechart refining state *visible* for a lockable frame



(c) Statechart, combined through "flattening"

Figure 8.1: Example statecharts

8.1.2 Fink et al.

Fink et al. present a static analysis of typestate properties [FYD+06]. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. The authors' analyses allow only for specifications that reason about a single object at a time. This prevents programmers from expressing multi-object properties such as FailSafeIter. Like us, Fink et al. aim to verify properties fully statically. However, our approach nevertheless provides specialized instrumentation and recovery code, while their approach only emits a compile-time warning. Also, CLARA supports a range of input languages so that developers can conveniently specify the properties to be verified, while Fink et al. do not say how developers might specify their properties.

8.1.3 Bierhoff and Aldrich

Bierhoff and Aldrich [BA07] recently presented an intra-procedural approach that enables the checking of typestate properties in the presence of aliasing. The author's approach aims at being modular, and therefore abstains from potentially expensive whole-program analyses like the ones that CLARA uses. To be able to reason about aliases nevertheless, Bierhoff and Aldrich associate references with special access permissions. Their abstraction is based on linear logic, and using access permissions it can relate the states of one object (e.g. an iterator) with the state of another object (e.g. a collection). These permissions classify how many other references to the same object may exist, and which operations the type system allows on these references. The authors use reference counters to reclaim permissions to help their type system to accept more valid programs. In their approach, they assume that every method is annotated with information about how access permissions and typestates change when this method is executed. Of course this does not necessarily imply that it has to be the programmer who adds these annotations. Many approaches exist that can infer typestate properties, and we discuss some of these approaches in Section 8.7.

Bierhoff and Aldrich's approach has the advantage of being modular: given appropriate annotations it can analyze any method, class or package on its own. CLARA on the other hand needs the whole program to be present, and in particular expects a complete but nevertheless sufficiently precise call graph. When the whole program is available, and can be analyzed, then Clara gives programmers the advantage that it does not require any program annotations. Clara only requires annotations that describe error situations, not the program, and then automatically analyzes the program to see whether such error situations can occur. We have found that worst-case assumptions coupled with coarse-grained side-effect information are surprisingly effective.

Bierhoff and Aldrich define typestate properties via a textual representation of statecharts. Hence, programmers can conveniently model behavioral sub-typing, as in the original typestate-checking methodology that Strom and Yemini proposed.

Because Bierhoff and Aldrich's work defines a type system and not a static checker like Clara, the workflow that a programmer has to follow in Bierhoff and Aldrich's approach is slightly different than it is in the case of using Clara. Clara allows the programmer to define a program that may violate the given safety property. Clara then tries to verify that the program is correct, and when this verification fails it delays further checks until runtime. Bierhoff and Aldrich's approach defines a type checker, and hence the idea is that the programmer is prevented from compiling a potentially property-violating program in the first place. This gives the advantage of strong static guarantees. After all, if the program does compile then the programmer knows that the program must fulfill the stated property. On the other hand, the type checker may reject useful programs that appear to violate the stated property but will not actually violate the property at runtime.

8.1.4 DeLine and Fähndrich

DeLine and Fähndrich's approach [DF04] is similar in flavor to Bierhoff and Aldrich's. The authors implemented their approach in the Fugue tool for specifying and checking typestates in .NET-based programs. Fugue checks typestate specifications statically, in the presence of aliasing. The authors present a programming model of typestates for objects with a sound modular checking algorithm. The programming model handles typical features of object-oriented programs such as down-casting, virtual dispatch, direct calls, and sub-classing. The model also permits subclasses to extend the interpretation of typestates and to introduce additional typestates, similar to the statecharts-based approach by Strom and Yemini. As in Bierhoff and Aldrich's approach, DeLine and Fähndrich assume that a programmer (or tool) has annotated the program under test with information about how calls to a method change the typestate of the objects that the method references. One fundamental difference between the two approaches is the treatment of aliasing. While Bierhoff and Aldrich used access permissions to reason about aliases, Fugue's type system tracks objects merely as "not aliased" or "maybe aliased". Objects typically remain "not aliased" as long as they are only referenced by the stack. The respective objects can change state only during this period. Once they become "maybe aliased", Fugue forbids any state-changing operations on these objects. This makes Fugue's type system less permissive than the system that Bierhoff and Aldrich describe: in the latter type system objects can change states even when they are aliased.

8.1.5 Dwyer and Purandare

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [DP07]. Their work identifies "safe regions" in the code using a simple static typestate analysis similar to [DCCN04]. Safe regions can be methods, single statements or compound statements (e.g. loops). A region is safe if its deterministic transition function does not drive the typestate automaton into a final state. A special case of a safe region would be a region that does not change the automaton's state at all. The authors call such a region an identity region. For regions that are safe but no identity regions, the authors summarize the effect of this region and change the program under test to update the typestate with the region's effects all at once when the region is entered, instead of at the individual shadows that the region contains. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. One possible disadvantage of such summary transitions may be that one loses the connection between the places in the code that perform a state transition and the places that actually cause these transitions. This makes it harder for programmers to investigate these program places manually to decide for themselves whether this part of the program could or could not violate the property at hand. Our static analysis does not attempt to determine regions; we instead decide if each single shadow is a nop-shadow. Dwyer and Purandare's analysis should be easily implementable in Clara and we encourage such an implementation.

8.2 Static analyses for tracematches

Clara is the result of a large body of research that we conducted over a period of three years. During this research, we made several attempts at finding analysis implementations that were correct, precise and efficient. In Section 8.2.1, we comment on some of our earlier attempts and their weaknesses. We also describe how Clara improves on these earlier algorithm versions, yielding an implementation that is indeed correct, precise and efficient. Earlier versions were bound to tracematches only and therefore programmers could not apply these analyses to monitoring aspects that do not originate from tracematches. Clara instead provides these implementations as instantiations of a common framework that supports a large variety of AspectJ-based runtime-monitoring tools.

Motivated by our work, Naeem and Lhoták also developed a static analysis to analyze tracematches at compile time. Their analysis differs from our analyses mostly in the chosen abstraction. In Section 8.2.2 we discuss in detail which impact the choice of abstraction has.

8.2.1 Earlier versions of Clara's analyses

During the development of this dissertation, we published two papers on evaluating tracematches ahead of time. Clara integrates this early work with small but significant improvements, and it generalizes the previous work to apply to a large variety of runtime-monitoring tools, as opposed to just tracematches. In the first paper [BHL07] we presented a three-staged analysis consisting of a Quick Check, a flow-insensitive

Consistent-shadows analysis and a flow-sensitive Active-shadows analysis. The Quick Check is essentially the same as the one that we present in this thesis, except for one minor difference: the quick check in our earlier work would consider an entire state machine, and simply disable checking of the whole state machine, if one cannot reach any final state along any path. The Quick Check that we present here acts on every path separately: when the state machine cannot reach a final state any more along a path p then the check disables monitoring of the events on p, even if one can reach the final state along other paths. As we showed in Section 4.4.2.2 (page 116), this can be beneficial in case of properties that yield state machines with multiple accepting paths, e.g. for Reader: in this example there is one path that reports a violation when writing to an InputStream whose Reader was closed, and the other path reports a violation when writing to a Reader whose InputStream was closed.

The Consistent-shadows analysis is also similar to the Orphan-shadows Analysis that we present here. In fact, both analyses yield the exact same result. However, the Orphan-shadows Analysis improves over the Consistent-shadows analysis in terms of analysis time and memory requirements. The Consistent-shadows analysis can be regarded as a generate-and-test algorithm: to determine whether two shadows s_1 and s_2 are compatible, the analysis first creates all sets of compatible shadows and then tests whether there exists a set that contains both s_1 and s_2. The number of sets of compatible shadows is bound only by the size of $\mathcal{P}(\mathcal{S})$ and the amount of aliasing present in the program. In particular, this algorithm is exponential in the number of shadows. We found that for most cases the Consistent-shadows analysis was efficient enough nevertheless, but that the complexity could lead to long analysis times and large memory consumption in cases like bloat-FailSafeIter. The execution time of the Orphan-shadows Analysis is polynomial in the number of shadows, and the Orphan-shadows Analysis uses only $|\mathcal{S}|^2$ bits of memory to cache its results.

The third stage from our earlier work, the Active-shadows analysis, was a first attempt at a flow-sensitive analysis of tracematches. While CLARA's third analysis stage, the Nop-shadows Analysis, is flow-sensitive only on an intra-procedural level, the Active-shadows analysis from our earlier work was a flow-sensitive, context-insensitive analysis of the entire program. Unfortunately, the abstraction that we chose for this Active-shadows analysis only allowed for weak updates because it did not encode must-alias information. Further, we chose a flow-insensitive pointer abstraction, and the computation of typestate information was context-insensitive too. In combination, the analysis was so imprecise that it was unable to identify a single nop-shadow in a benchmark set quite similar to the benchmark set that we consider here. These results, in combination with the work that we present in this thesis, show that choosing the right abstractions is key to obtaining good precision. Clara uses precise flow-sensitive pointer information and context-sensitivity on an intra-procedural level. As we showed in this thesis, this information can yield much optimization potential and therefore significantly improves over the earlier Active-shadows analysis.

In a second paper [BLH08a] we presented an improved version of the the Active-shadows analysis. The analysis presented there is similar to the Nop-shadows Analysis that we present in this dissertation, except for the following points. Firstly, the analysis in [BLH08a] recognizes "necessary shadows" through a shadow history. Unfortunately, this is unsound (explanation follows below). By design, this analysis is also optimistic: it assumes that a shadow s is unnecessary and can be removed, unless proven otherwise, by driving a shadow history containing s into a final state. Optimistic analyses can be more prone to error because their optimistic default answer is unsound. The Nop-shadows Analysis that we present here instead detects "unnecessary shadows", i.e., nop shadows, using a backwards analysis. This analysis is rather pessimistic: it assumes that a shadow is necessary until we prove that it is in fact a nop shadow. The only optimistic part of our computation is the initialization of our worklist algorithm, Algorithm 5.1. We initialize the worklist algorithm for a method m by assuming that m did not execute before the current execution of m that we are analyzing. This is an unsafe, optimistic assumption, and we re-gain soundness only later-on by propagating configurations along edges from $succ_{ext}$.

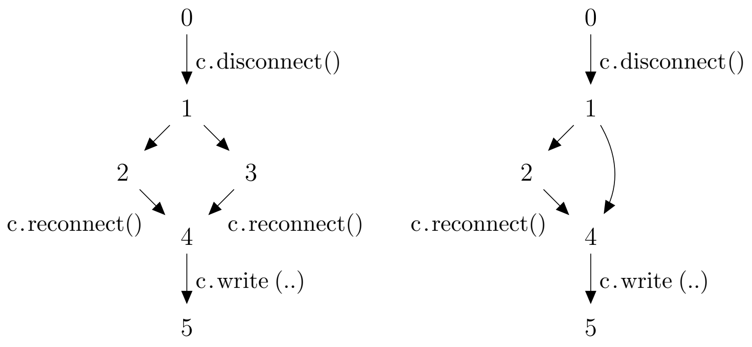
A second difference between the two analyses is the treatment of inter-procedural control-flow. In [BLH08a], we assumed that any state machine instance could be in any state when entering the current method m. This is sound but also the most imprecise assumption one could think of. In the Nop-shadows Analysis that we present here, we use the function *reachingStar* to compute a better approximation. Further, in [BLH08a] we did not use the inter-procedural successor function $succ_{ext}$. Instead, whenever we recognized an outgoing method call that could potentially and transitively call a shadow-bearing method, then we simply "tainted" successor configurations. Our optimization phase, which normally disabled nop shadows, would then not take tainted configurations into account, therefore maintaining soundness. The solution that we present in this paper is not only more elegant, it is also more precise. Tainting makes a worst-case assumption about the outgoing method call, while our current implementation considers such method calls more precisely.

Another difference is that all of the above analyses were designed and implemented to work for tracematches only. We believe that all of them could be modified so that they work on dependency state machines in general, but at the current time, the approach that we present here is, to the best of our knowledge, the only one in which programmers can apply static typestate analyses to optimize any runtime monitor, if this monitor is implemented as an AspectJ aspect and contains appropriate dependency annotations.

Unsoundness. As we briefly mentioned above, the Active-shadows analysis that we presented in earlier work is unsound. The problem is that the Active-shadows analysis uses a forward-analysis phase only, and uses no backward analysis. As we saw in Section 5.2, the backward analysis is essential to determine the set of states from which one can reach a final state using the remainder of the program execution. Our earlier analysis assumed that one could circumvent computing the backward analysis by instead propagating forward a "shadow history": when detecting that a shadow s changes states from some possible source state $source(s)$ to some target state $target(s) \neq source(s)$, then the analysis would not yet decide whether or not s is a nop-shadow, but would instead add a reference to s to the configuration for $target(s)$. When this configuration later-on reaches a final state, then the analysis knows that s (1) changes states and (2) is on a path that reaches a final state, and therefore s is not a nop shadow. In all other cases, the analysis would declare s a nop shadow and disable s. This is unsound, and in Figure 8.2 we give an example for why this is so.

```
1 void foo(Connection c) {
2     c.disconnect();
3     if(?) {
4         c.reconnect();
5     } else {
6         //c.reconnect();
7     }
8     c.write (..);
9 }
```

(a)



(b) (c)

Figure 8.2: Unsoundness in earlier tracematch-based static analyses

First consider the program code that we show in Figure 8.2a, in combination with the ConnectionClosed example property (Figure 5.5, page 144). Figure 8.2b shows the control-flow graph for this method for the case where line 6 is not commented out. In this case, the program cannot violate the ConnectionClosed property: although the program disconnects the connection in line 2, the connection is re-connected *on all paths* that lead up to the write in line 8. In this case, the analysis would determine correctly that all the shadows are nop shadows. At the disconnect in line 2, the analysis would add the disconn shadow to the configuration's shadow history, and then propagate the configuration to both branches. When reaching either of the reconnect statements, the analysis would then conclude that the successor configuration is **ff**. When this happens, the analysis discards the configuration, and the shadow history with it. Therefore, no shadow in any shadow history would ever reach a final state, and the analysis would remove all shadows in the example program.

Now consider Figure 8.2c, which represents the control-flow graph for the version of the code where line 6 is commented out. In this setting it is not sound to remove any of the shadows! The disconnect and read shadow have to be enabled so that they can report an error at runtime, in case the program executes the “false” branch, thus writing to a disconnected connection. The reconnect shadow on the other hand also needs to be enabled so that the runtime monitor will not report a property violation when the program executes the “true” branch instead. However, the Active-shadows analysis that we presented in earlier work would *erroneously disable* the reconnect shadow and may therefore cause a false positive when the instrumented program is executed. Along the “true” branch, the analysis would, as before, conclude that the successor configuration is **ff**, and therefore not propagate this configuration any further. The configuration that the algorithm propagates along the “false” branch does reach a final state when processing the write at line 8. However, the shadow history contains at this point only the disconnect and the write shadow, but not the reconnect shadow, because the configuration in question was only propagated along the “false” branch, and this configuration therefore never visited the reconnect shadow at all.

The Nop-shadows Analysis that we present in this dissertation soundly handles this and all other cases using a backwards analysis instead. In the example from Figure 8.2b, the analysis would first conclude that the write shadow at line 8 is a nop-shadow. This is because after the reconnect event the monitor will be in its initial state, and a write event will loop in this state. Once the write shadow is disabled, even the flow-insensitive Orphan-shadows Analysis suffices to determine that the remaining shadows cannot violate the ConnectionClosed property any more. In the program that we show in Figure 8.2c, the analysis would not, however, be able to disable the write shadow: there is a path (along the “else” branch) that leads into a state from which the write shadow leads into a final state, and hence the write shadow cannot be a nop shadow. The reconnect shadow is also no nop-shadow in this example, because it discards a partial match, and the disconnect shadow is no nop shadow because it propagates a match that the write shadow may complete. Hence the Nop-shadows Analysis will correctly not disable any shadow in this program.

We discovered this unsoundness by running the analysis on a test case that looks much like the one from Figure 8.2a. It took us several weeks to change the analysis so that it would be sound, by basing it on a backwards analysis. Interestingly, this work does not really seem to pay off very much: we did not discover any shadows in our benchmarks that the earlier algorithm would have mistakenly disabled. We presume that this is due to the nature of the cases in which this unsoundness can arise. The example from Figure 8.2a (with line 6 commented out, as shown) shows a program that is conditionally correct: the program does or does not violate the ConnectionClosed property, depending on whether or not the "false" branch can be taken at runtime. In other words, the figure shows a very unusual program. It seems not very likely that programmers make such programming mistakes. Nevertheless, it is good to know that the Nop-shadows Analysis solves the unsoundness problem, so that one can get static guarantees in cases where the Nop-shadows Analysis does disable all shadows.

8.2.2 Naeem and Lhoták's analysis for tracematches

Naeem and Lhoták present a context-sensitive flow-sensitive inter-procedural whole-program analysis [NL08] to analyze typestate-like properties of multiple interacting objects at compile time. Unlike our approach, Naeem and Lhoták restrict themselves to handling specifications that are given in the form of a tracematch. Nevertheless, this is presumably the case because Clara did not exist at the time at which Naeem and Lhoták's work was conducted. We believe that one could integrate the authors' analysis easily into Clara. The analysis that Naeem and Lhoták present can be seen as a generalized version of our Nop-shadows Analysis. Our Nop-shadows Analysis is mostly intra-procedural and uses only flow-insensitive information to model inter-procedural control flow. Naeem and Lhoták's analysis, on the other hand, propagates configurations along call edges and then further through the bodies of called methods. This can potentially lead to enhanced precision in cases where multiple methods use combinations of objects that are relevant to a given specification.

Different pointer abstractions. Another difference to our Nop-shadows Analysis is that Naeem and Lhoták use a different pointer abstraction. Our pointer abstraction, object representatives, is flow-sensitive only on an intra-procedural level, and at outgoing method calls we have to resort to context-sensitive but flow-insensitive pointer information. Naeem and Lhoták instead use a special "binding lattice" that models each object by the variables that may or must point to the object. This representation encodes must-aliasing and must-not-aliasing at the same time. The authors' analysis updates this object representation at the same time as the state information. This is different from our approach: we compute the inter-procedural points-to information only once, before executing the Orphan-shadows Analysis. Further, we also compute the local flow-sensitive must-alias analysis and must-not-alias analysis for any method m only once, before applying the worklist algorithm (Algorithm 5.1) to m for the first time.

```
s0 : x := new ();
s1 : y := new ();
z := y;
if (?) {
    y.close ();
    z := x;
}
s2 : z.read();
```

(a)

```
s3 : f := new ();
while (?) {
    s4 : f.read();
    if (?) {
        f.close ();
        s5 : f := new ();
    }
}
```

(b)

Figure 8.3: Program fragments illustrating the effect of aliasing on typestate verification

In theory, one can gain additional precision by computing typestate information at the same time as the pointer information like Naeem and Lhoták do. Consider the two example programs in Figure 8.3, taken from Field et al. [FGRY05]. In the following, we quote Field et al.:

"The principal difficulty in doing precise verification arises from determining how aliasing interacts with operations on objects. Some prior work on typestate verification has employed a two-step approach to the problem, in which an initial phase performs a conservative heap analysis of the program, and a subsequent phase uses the information from the heap analysis to do typestate analysis. However, we can see from the program fragments in [Figure 8.3] that such an approach can sometimes lead to imprecise results. One can easily verify that in both [Figure 8.3a and 8.3b], all sequences of file operations on a given object are prefixes of `read ; close`; i.e., that no read ever follows a close.

However, consider a two-phase analysis in which the heap analysis is separate from the typestate analysis. In [Figure 8.3a], a precise (and correct) heap analysis will determine that program variable z at program point s2 may point to the object created at s0 or the object created at s1. Furthermore, a precise typestate analysis will determine that the object created at s1 could be in a closed state at s2. A two-phase analysis must therefore erroneously conclude that the read could be performed on a closed file. Similarly, in [Figure 8.3b], any conservative heap analysis would determine that objects created at program points s3 and s5 could reach the read statement at s4. In addition, a typestate analysis would also determine that the objects created at program points s3 and s5 could be in a closed state at s4. The analysis would, however, not be able to discover that f can never point to a closed object at s4, and would incorrectly indicate a possible error."

The example by Field et al. is therefore a good example for an intra-procedural case in which Naeem and Lhoták's abstraction is superior to ours. Interestingly, in our benchmark set we could not find any instances where we failed to identify a nop-shadow for the reasons that Field et al. put forward. Hence we conclude that although we pre-compute our aliasing information, this may not actually cause much of an imprecision in practice. As Naeem and Lhoták note [NL08, Section 5], there are however also examples in which our analyses succeed in identifying a nop shadow, but Naeem and Lhoták's fails on the same shadow. This is due to the context-sensitive points-to analysis that we use. Generally, this analysis is very precise, due to the large amount of context information that it uses. When this context information matters, the pointer abstraction that Naeem and Lhoták chose may be less precise.

Naeem and Lhoták are currently re-implementing their analysis to increase precision and performance. Their current implementation is written in Scala, while the rewritten version will be written in Java, and will be using customized, more efficient data structures. We therefore did not compare our analysis to Naeem and Lhoták directly at this stage. In the future we plan to create a joint comparative study in which we consistently use the very same tracematch specifications and analyze the same benchmark versions with the same runtime library, taking into account the exact same set of potentially dynamically loaded classes.

Context-sensitivity. It is worthwhile noting that the way in which we treat outgoing method calls in our Nop-shadows Analysis can also be considered context-sensitive: when analyzing method m, then we analyze an outgoing call to another method m' only with respect to the configurations that reach the entry of m' through m, and not through other call sites that may call m' too. Therefore, the Nop-shadows Analysis analyzes m' in the context of m. The analysis is flow-insensitive in that case because the analysis propagates configurations along an edge of $succ_{ext}$, but it is context-sensitive.

Naeem and Lhoták's analysis builds on our Active-shadows analysis from earlier work, and therefore the analysis suffers from the same unsoundness problem that we described above: Naeem and Lhoták's analysis also uses shadow histories. It should be possible to fix Naeem and Lhoták's analysis by making it use an analysis scheme that is similar to Clara's, where one conducts both a forward and a backward analysis, possibly maintaining the pointer abstractions that the authors are using now. One problem that one would face then, would be that Clara re-computes the analysis information every time after Clara has disabled any shadow, because the program's transition structure may change when a shadow is disabled. As we showed, this can be done efficiently on an intra-procedural level. However, it would be unrealistic to re-iterate an inter-procedural analysis every time a shadow gets disabled: every single inter-procedural iteration typically takes already several minutes to compute. One would hence have to augment shadows and analysis information with dependency information to determine which parts of the analysis information need to be updated once a shadow is disabled. We plan to consider such an approach in future work.

8.3 Other approaches to runtime monitoring

In the following we discuss a number of monitoring tools that influenced the design and implementation of Clara. We also discuss whether programmers could use these tools in combination with Clara.

8.3.1 Stolz and Huch

Our work was originally motivated by Stolz and Huch's work [SH05] on runtime-verifying concurrent Haskell programs. The authors specify program properties using linear-temporal-logic formulae. Such formulae are generally evaluated over a propositional event trace: a formula refers to a finite set of named propositions and any of the propositions can either hold or not hold at a given event. Stolz and Huch implemented a runtime library that would generate a propositional event trace at runtime and update a linear-temporal-logic formula according to the monitored propositional values. The library reports a property violation when the formula reduces to **ff**. The formulas that Stolz and Huch allow for can be parameterized by different values, similar to the object-to-variable bindings that Clara supports.

8.3.2 J-LO

We ourselves developed J-LO, the Java Logical Observer [Bod05], a tool for runtime-checking temporal assertions in Java programs. J-LO follows Stolz and Huch's approach in large parts, however the propositions in J-LO's temporal-logic formulae carry AspectJ pointcuts as propositions. The J-LO tool accepts linear-temporal-logic formulae with AspectJ pointcuts as input, and generates plain AspectJ code by modifying an abstract syntax tree. J-LO extends the AspectBench Compiler, which allows it to then subsequently weave the generated aspects into a program under test. Pointcuts in J-LO specifications can be parameterized by variable-to-object bindings. While the implementation of J-LO is effective in finding seeded errors in small example programs, it causes a runtime overhead that is too high to allow programmers to use J-LO on larger programs. Nevertheless, one could annotate the J-LO-generated aspects with dependency information and then use CLARA's static analyses to remove some of this overhead.

8.3.3 Tracematches

Allan et al. [AAC+05] are the creators of tracematches. Tracematches share with J-LO the idea of generating a low-level AspectJ-based runtime monitor from a high-level specification that uses AspectJ pointcuts to denote events of interest. Nevertheless, the tracematch implementation generates runtime monitors that are far superior to those that J-LO generates. Avgustinov et al. [ATdM07] perform sophisticated static analyses of the tracematch-induced state machine to determine an optimal monitor implementation that satisfies three main goals:

1. The monitor implementation should be correct.
2. The monitor should allow parts of its internal state to be garbage-collected whenever possible without jeopardizing correctness.
3. The monitor should implement an indexing scheme that allows the monitor, at any event that binds a variable v to an object o, to quickly look up all state-machine instances for the binding $v = o$.

As Avgustinov et al. show, reclaiming memory (2) and indexing of partial matches (3) are both necessary to achieve a low runtime overhead in the general case. In all the experiments that we conducted with tracematches in our work, these optimizations were already enabled. Hence our experiments show that, while these optimizations are necessary, they may not always be sufficient on their own. However, in combination with Clara's analysis, the runtime overhead will be low in most cases. Another difference between Allan et al.'s analyses and ours is that Allan et al. only analyze the state machine, while we analyze both the state machine and the program. This allows us to disable shadows at program points where this is sound, hence making it easier to check the program for potential property violations already at compile time. Allan et al.'s analyses do not analyze or modify shadows.

8.3.4 Tracecuts

Walker and Viggers developed tracecuts [WV04], an approach that monitors programs with respect to a specification given as a context-free grammar over AspectJ pointcuts. Context-free grammars are strictly more expressive than the finite-state patterns that we consider in Clara: in Chapter 2 we showed that some properties exist that finite-state formalisms cannot express but that could be expressed as a context-free language. However, we also showed that most interesting program properties are in fact finite-state properties. It is unclear how much runtime overhead tracecuts induce. In previous work [ATB+06], we tried to compare the relative efficiency of J-LO, tracematches, tracecuts and another tool called PQL (see below). As we reported there, there is an implementation of tracecuts, but it is immature, and while its authors kindly gave us private access to their executables, they did not feel it was appropriate for us to use their prototype for our experiments.

8.3.5 JavaMOP

JavaMOP provides an extensible logic framework for specification formalisms [CR07]. Via logic plug-ins, one can easily add new logics into JavaMOP and then use these logics within specifications. As we already showed in this thesis, JavaMOP has several specification formalisms built-in, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars. JavaMOP translates specifications into AspectJ aspects using the rewriting logic Maude [CELM96]. JavaMOP aims to be a generic framework that should support multiple specification languages. Therefore, the designers of JavaMOP are careful when it comes to making assumptions about the specifications used with their framework. While this helps keep the framework general, this generality makes it hard, if not impossible, to analyze the specification in the way that Avgustinov et al. have done [ATdM07] for tracematches. This can cause problems.

In [CR07], Chen and Roşu report that their implementation outperforms the tracematch implementation in terms of runtime overhead while also being memory-safe. However, this is only true in cases where the validation or violation handler does not actually reference any of the objects that are involved in the match. As soon as the programmer does reference such objects, JavaMOP has to resort to strong references that keep the internal state of the objects' monitors alive indefinitely, thereby causing high memory consumption and runtime overhead. Another shortcoming of JavaMOP is that it currently only supports specifications that bind all the objects of a complete match already at the first transition (see Section 3.2.2). This is fine in cases like FailSafeIter (see page 37): here the create symbol that leads out of the initial state binds both c and i. For such patterns, JavaMOP can easily store the state for this variable binding in a cascaded hash map that maps from c to a map that in turn maps from i to the state in question. For other patterns, things are not so simple. ASyncContainsAll describes the situation in which one calls c_1`.containsAll(`c_2`)` on two synchronized collections c_1 and c_2. Here the first symbol binds only c_1, when this collection c_1 is returned from a call to `synchronizedCollection`. The second symbol binds c_2 and only the final symbol binds both values together. In recent work [CR09] Chen and Roşu present an algorithm that circumvents this problem by using a different indexing structure. However, the authors do not discuss if they allow a program to reclaim monitoring state by using weak references, and more importantly why their algorithm would still be sound if they did. The analysis by Avgustinov et al. combines both weak references and efficient indexing in a provably sound way, but it requires an analysis of the specification pattern.

For the work presented in this thesis, Feng Chen extended [BCR09] the JavaMOP implementation so that it would perform some limited analysis of the specification, so that JavaMOP could annotate the generated monitors with dependency information that CLARA can use to partially evaluate these monitors at compile time. Interestingly, as Chen writes in his most recent work [CR09], this dependency information can also be useful to reduce he number of indexing trees that JavaMOP uses to associate states with object combinations at runtime. We believe that the same information should also be useful to implement memory-saving techniques in JavaMOP like the ones that Avgustinov et al. proposed for tracematches.

8.3.6 PQL

The Program Query Language [MLL05] by Martin at al. resembles tracematches in that it enables developers to specify properties of Java programs, where each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches: it uses stack automata rather than finite state machines, which yields a language slightly more expressive than context-free grammars. Martin et al. propose a flow-insensitive static-analysis approach to reduce the runtime overhead of monitoring programs with PQL. This approach inspired us to implement our Orphan-shadows Analysis. As the authors show and as we confirm in our work, such an analysis can be very effective in ruling out impossible matches. However, we also showed that a flow-sensitive analysis can yield additional optimization potential. PQL instruments the program under test manually, using the BCEL [Dah] bytecode engineering toolkit. If PQL used AspectJ instead, then is should be possible to optimize the generated monitor with CLARA, similar to tracecuts. PQL was published as an open-source project, available for download at `http://pql.sourceforge.net/`. However, it appears that the project is no longer maintained.

8.3.7 PTQL

Goldsmith et al. [GOA05] proposed PTQL, the Program Trace Query Language, which provides an SQL-like language for querying properties of program traces at runtime. The authors also provide "partiqle", a compiler for this language. The compiler instruments the program that is to be queried so that the program notifies monitoring code about the appropriate events at runtime. The monitor itself uses indexing trees to associate the monitor's internal state with the appropriate objects. It may be possible to evaluate parts of a program query at compile time, for instance when comparing a method name to a constant string. Partiqle resolves such parts of a query already during compilation. This is the same as the partial evaluation of pointcuts that happens in standard AspectJ compilers: these compilers also insert runtime checks only for parts of a pointcut that the compilers cannot determine at compile time. Partiqle resorts to a table-based approach to evaluate the remainder of the query at runtime. Because PTQL uses its own compiler, and is not based on AspectJ, one cannot currently use Clara to evaluate PTQL queries ahead of time. Even if PTQL did generate aspects for its monitoring needs, one would have to take into account that the PTQL language is very expressive and probably Turing complete. Hence it remains unclear whether one could effectively determine dependencies within a query at compile time, so that Clara could exploit these dependencies to optimize PTQL monitors.

8.3.8 Sub-alphabet sampling

Dwyer, Diep and Elbaum propose a novel mechanism to guaranteeing low runtime overhead even in the presence of multiple monitoring properties and in cases where programs need to update the internal state of monitors for these properties very frequently [DDE08]. The authors first propose to combine multiple properties over objects of the same class into one large "integrated" property. As the work shows, monitoring of this integrated property can be more efficient than monitoring of the individual original properties. Then second, the authors propose to project the monitor for this integrated property onto multiple sub-alphabet monitors, where each monitor monitors exactly one subset of the original alphabet Σ of events. These sub-alphabet monitors form a lattice that is isomorphic to the power-set lattice of Σ. By the way in which Dwyer et al. define their state-machine semantics, each individual monitoring automaton in this lattice is sound, i.e., cannot report any false positives. The authors show that programmers can gain fine-grained control over the perceived monitoring overhead by selecting a subset of monitors from the lattice. Further, the authors present several heuristics that attempt to select reasonable subsets automatically. As the results show, the sub-alphabet lattice allows for a flexible selection of monitors that gives programmers fine-grained control over their overhead. We therefore believe that the author's technique is a valuable addition to our own efforts of reducing the runtime-monitoring overhead, such as the spatial partitioning that we presented in Chapter 6. It remains unclear, however, whether the author's technique can easily be extended to handle properties that refer to a shared state of multiple interacting objects. Such properties would likely impose certain constraints on the sub-alphabet monitors, rendering some of these automata illegal.

8.3.9 QVM

Arnold, Vechev and Yahav present QVM, the "Quality Virtual Machine", an extension of IBM's J9 Java Virtual Machine that implements a set of techniques that aim at aiding programmers to debug their programs [AVY08]. QVM comes equipped with support for virtual-machine-level monitoring of single-object typestate properties. Programmers can use a simple syntax to define typestate properties for any given Java class. QVM then instruments instances of such classes to track the instances' typestate at runtime. Once QVM detects and report that a typestate property was violated, it starts sampling method calls that the program issues on objects that are allocated at the same allocation site as the object for which the violation occurred. Naturally, the calling sequences for both objects are not necessarily the same. Yet, the authors argue that in most cases these sequences will be similar enough such that the sampled trace will help the programmers pinpoint the actual problem on the violating sequence and hence fix the bug in their program code. QVM's techniques are complementary to all of the static techniques that Clara provides and it would be interesting to integrate both tools into a common solution.

8.4 Demand-driven pointer analysis

Clara makes heavy use of Sridharan and Bodík's demand-driven refinement-based flow-insensitive context-sensitive points-to analysis [SB06]. Context information is important for our work because Clara's analyses need to be able to distinguish multiple objects that are created at the same allocation site. The author's analysis is both demand-driven and refinement-based. Clients can query the points-to analysis on single variables. The analysis answers a query by returning either a context-sensitive points-to set, modeling all possible objects that the program under test could assign to this variable, or, in case the analysis fails, it returns a context-insensitive points-to set (computed by Spark [LH03]) instead. The analysis can "fail" because it is demand-driven: a client analysis provides the points-to analysis with a given quota that the points-to analysis can then use to refine points-to sets with context information. If the analysis exceeds the quota then it aborts and returns context-insensitive information. In our analyses we used the default quota settings.

Sridharan and Bodík's analysis works by trying to prove that certain entries into methods, denoted by "$(_m$", cannot be matched with appropriate exits "$)_m$", or that certain assignments to a field f, denoted by "$[_f$" cannot be matched with appropriate reads "$]_f$" from this field. The analysis first starts with a very coarse-grained pointer-assignment graph that skips many possible method entries and exits and field reads and writes using so-called "match edges". A match-edge can be seen as an artificially introduced epsilon transition in the pointer-assignment graph. Match edges speed up the analysis because they allow the analysis to skip analyzing everything between the match edge's source and target nodes. The analysis is refinement-based: when the analysis fails to prove that a certain allocation site cannot be assigned to a given variable then it successively deletes match edges, thereby analyzing larger and larger parts of the program, until ultimately the whole program will be analyzed or the analysis exceeds its quota.

We chose this particular analysis because our client analyses have two important properties: (1) they require context information and (2) they require only points-to information for some few program variables, namely those variables that bind objects to dependent advice. In Sridharan and Bodík's analysis, every single points-to query can be relatively expensive. However, because we query the analysis only on a small fraction of the total number of variables, the overall points-to-analysis time is still lower than if we had used another context-sensitive points-to analysis that determines context-sensitive points-to information for every program variable.

8.5 Aspect-oriented programming

We next discuss related work from the field of aspect-oriented programming. This work comprises other static analyses, expressive pointcuts and code generators that generate aspects from high-level specifications.

8.5.1 Static optimizations for cflow pointcuts

The cflow pointcut in AspectJ allows programmers to execute a piece of advice only at joinpoints that occur in the control-flow of a certain other joinpoint. This is often useful to define an aspect's scope: for instance one may want to monitor all calls to `println` in a given base program. To avoid the aspect from monitoring `println` calls that the aspect itself triggers, the programmer can use a cflow pointcut to exclude joinpoints that are in the control flow of the aspect's advice execution. Avgustinov et al. [ACH$^+$05b] implemented an abc extension that tries to determine statically whether a given joinpoint shadow may not or must occur in the control flow of another shadow. In case of a "may not" or "must" answer, the analysis can eliminate the appropriate runtime checks for the cflow pointcuts.

The analyses that Clara supports can be seen as a generalization of this analysis approach to support the static approximation of typestates instead of control-flow relationships. Both the cflow analysis and Clara are implemented as extensions to abc. In principle this allows programmers to first reduce the number of joinpoint shadows by applying the cflow analysis and then by applying Clara, or even the other way around. We did not, however, enable the cflow analysis in our experiments. Because none of our specifications uses a cflow pointcut, enabling the cflow analysis would only have increased the analysis time but would not have impacted the precision of our approach.

8.5.2 Association aspects and relational aspects.

Sakurai et al. [SMU$^+$04] proposed association aspects, an AspectJ language extension that allows programmers to restrict advice execution to joinpoints involving objects that the programmer explicitly associated with an aspect. A programmer associates an object `o` with an aspect `A` by calling `A.associate(o)`, and releases the association via `A.release(o)`. In earlier work [BSH08], we showed that one can implement relational aspects, a variant of association aspects, via a syntactic transformation into tracematches. abc implements relational aspects that way, and the implementation automatically integrates into Clara: The optimizations proposed in this dissertation remove advice-dispatch code from locations where the objects involved are known not to be associated with `A`. Further, for objects for which no advice in the relational aspect can ever execute, the optimization will remove the call to the code that associates the object with the aspect in the first place.

8.5.3 Data-flow pointcuts

Masuhara and Kawauchi proposed a pointcut `dflow` [MK03]. Programmers can write pointcuts of the form

$$p \texttt{ \&\& dflow[}s\texttt{,}t\texttt{](}q\texttt{)},$$

which matches if data flows from s to t, where p is a pointcut binding s, and q is an inner pointcut binding t. `dflow` is evaluated at runtime, i.e., it only matches if data-flow does indeed exist. The authors suggest however, to devise a static analysis that would optimize data-flow pointcuts at compile time. Unfortunately, neither dependent advice nor dependency state machines are expressive enough for this purpose: both are defined using tests of pointer equality, and our alias analyses therefore only regard pointer assignments. In general, data-flow can however also comprise the flow of primitive values and flow arising from String concatenation.

8.5.4 maybeShared pointcut

Bodden and Havelund proposed [BH08] a pointcut `maybeShared()` that matches accesses to fields that can potentially be shared among threads. This approach is similar to dependent advice and dependency state machines in that the semantics of `maybeShared()` are also parameterized. Like in dependent advice, a trivial default implementation may return **`tt`** in every case, but Bodden and Havelund use a static thread-local objects analysis [HPV07] to approximate `maybeShared()` in a more effective way. Deciding thread-locality is very different from deciding aliasing and typestates, and therefore CLARA cannot benefit the implementation of `maybeShared()`.

8.5.5 SCoPE

SCoPE [AM07] is an abc extension by Aotani and Masuhara. The extension provides programmers with a means to define highly expressive pointcuts that can reason about the static structure of the program. For instance, consider the following pointcut:

```
pointcut executeLowercaseMethod(): execution(* *(..)) &&
  if(thisJoinPoint.getSignature().getName().matches("^[a-z]+$"));
```

This pointcut will match the execution of any lowercase method. Programmers could write the same pointcut in plain AspectJ, too. However, such an implementation would be very costly: any standard AspectJ compiler would produce a shadow for this pointcut at the execution of *every* method and then attach to the shadow a dynamic residue that in turn dispatches to a method that evaluates the if-check. SCoPE will instead evaluate this pointcut at compile time. SCoPE first inspects the expression within the if-pointcut. If this pointcut only depends on static properties of the `thisJoinPoint` object, then SCoPE evaluates the expression with respect to the joinpoint's shadow at compile time instead of evaluating it with respect to the joinpoint at runtime. In result, SCoPE will weave calls to a piece of advice that is attached to the above pointcut only into methods that actually do have a lowercase method name.

SCoPE and Clara have in common that they attempt to evaluate parts of a runtime check (a "dynamic residue") at compile time. However, the kinds of runtime checks that both approaches can handle are very different. Clara supports stateful typestate checks and resolves aliasing, while SCoPE was designed to filter out joinpoints based on the static context in which they occur. Consequently, both approaches are orthogonal. Clara could benefit from SCoPE by applying SCoPE prior to running Clara's analyses: applying SCoPE would potentially disable some shadows upfront, which the later analysis stages could then skip. On the other hand we do not see a situation in which Clara could benefit the implementation of SCoPE.

8.5.6 S2A and M2Aspects

Maoz and Harel proposed S2A, a tool [MH06] to generate executable AspectJ code from Live Sequence Charts [DH99] (LSCs). An LSC and its generated aspects can either implement functional aspects of a system, or they can be used for runtime monitoring, reporting error messages when they match. Some of the aspects that S2A generates are history-based, and in fact even implement a finite-state machine. We confirmed with Maoz that S2A could, in principle, generate dependency annotations for these aspects and that they could lead to optimization potential similar to what we observed in our experiments, at least when LSCs are used to specify forbidden scenarios, implemented as runtime monitors. M2Aspects [KLM06] generates AspectJ aspects from scenario-based software specifications, denoted as Message Sequence Charts (MSCs). MSCs are less expressive than LSCs. Hence we believe that one could also modify M2Aspects to generate dependent advice.

8.5.7 Alpha

Alpha [OMB05] is an aspect-oriented programming language that allows programmers to define pointcuts via Prolog queries over a program's static structure and dynamic behavior. Alpha is an aspect-oriented extension of a simple Java-like object-oriented core language. Alpha supports classes, single inheritance, and has a static type system. In Alpha, the program runtime evaluates pointcuts over a Prolog facts database.

Because it would be very expensive to evaluate every query at every joinpoint, the authors propose a static analysis that relates queries to only those joinpoint shadows that can impact the result of the query. The Alpha language is very expressive and allows programmers to define many properties similar to the ones that SCoPE supports. Of special interest to us is that fact that Alpha allows for queries over the current execution trace, which means that one can use Alpha to implement runtime monitors. Because the pointcuts that Alpha uses are formulated as Prolog queries, and not in AspectJ syntax, Clara cannot currently parse these pointcuts and therefore it would be difficult to have Alpha generate input that Clara could use to optimize the monitored program. Similarly, Alpha programs are not woven using a standard AspectJ compiler, and therefore, to obtain an optimized program, one would have to derive a new mechanism to feed back Clara's analysis information to Alpha's own matching mechanism.

8.5.8 LogicAJ

LogicAJ [KRH04] is an aspect-oriented programming language that extends AspectJ's pointcut mechanism with logic variables. Logic variables have unification semantics like the variables in dependent advice and dependency state machines: variables with the same name denote the same (meta) objects. The scopes differ, however. In dependent advice the scope spans a dependency declaration (which can reference multiple pieces of advice). In LogicAJ, however, each pointcut has its own scope. Programmers can use logic variables inside pointcuts, in place of the names of packages, types, fields, methods and AspectJ's pointcut variables. In the last case, one uses a logic variable v in the form `this(`v`)`, `target(`v`)` or `args(`v`)` binding v to the receiver, target, respectively argument object of a call. Such a pointcut p only matches when there is a consistent variable binding for all uses of v in p. One could use dependent advice to optimize cases where v is used more than once within *the same* pointcut. However, because in LogicAJ each pointcut has its own scope, one cannot infer (and therefore not exploit) inter-dependencies between *multiple* pieces of advice or their pointcuts in LogicAJ.

8.6 Static checkers

There are a number of static checkers that aim to find potential programming errors. Rutar et al. did an excellent comparative study [RAF04] on some of the best-known tools in the field. They found that all tools succeed in finding programming problems, but interestingly, every tool identified slightly different classes of problems. We give an overview of the tools that are most relevant to our research.

8.6.1 FindBugs

FindBugs [HP04] is a static rule checker developed and maintained by the University of Maryland. It is one of the few static checking tools that has been reportedly used in a production environment of several large companies. FindBugs is popular because it comes pre-equipped with a large set of checkers that identify programming problems with respect to common libraries like the Java Runtime Library. Moreover, the checks that FindBugs implements are very efficient—either because they are mostly modular, i.e., only check a single method at a time, or because they only access very limited global information like the type hierarchy. FindBugs analyzes Java bytecode. Programmers can integrate FindBugs into the Eclipse IDE as a plug-in. Alternatively, they can use FindBugs as an Ant task or run it from a specialized graphical user interface. Companies can extend FindBugs with their own company-specific rules. However, FindBugs offers no special-purpose syntax to support this task; programmers write rules by writing Java code. The rules in FindBugs are usually designed to favor false negatives over false positives. In other words, FindBugs will often miss programming errors, but when it does give a warning then this may give a good indication of an actual programming problem. This is in contrast to the sound analyses in Clara, which we designed to never miss a potential violation, and rather report a false positive if necessary. Of course, due to the limited amount of information that FindBugs checkers use, the checkers often report a fair number of false positives despite the efforts to reduce this number.

8.6.2 PMD

PMD [Cop05] is (apart from being one of our benchmarks) another static checker that is targeted more towards finding violations of "best practices" or programming styles, rather than finding actual programming errors. An example is a rule that says "A class that has private constructors and does not have any static methods or fields cannot be used.". Like FindBugs, PMD has a large user base. PMD operates on the abstract syntax tree of a Java program, and therefore operates on its source code. Programmers can therefore not use PMD on parts of a program for which no source code is available. However, this is usually no problem in practice because most of PMD's analyses are intra-procedural anyway, and usually one has control over the source code for a method that one is interested in checking. Unlike FindBugs, PMD has no support for data-flow analyses. Hence it cannot, for instance, easily determine whether a pointer may be null or whether a variable will be initialized before it is used. Like FindBugs, PMD integrates with Eclipse and Ant, but in addition it also integrates with many other IDEs and even text editors[1].

8.6.3 ESC/Java

ESC/Java [FLL$^+$02] is an "extended static checker" for Java source code, developed by a group of researchers from Compaq. The authors describe the idea of extended static checking as the efficient static checking of undecidable properties (as opposed to type checkers, which should only consider decidable properties). Because the properties are undecidable, any such checker would have to be either unsound or imprecise, and ESC/Java is both. ESC/Java mainly checks pre- and post-conditions, as well as invariants. By default, ESC/Java checks for a set of pre-defined properties like whether the program de-references a null pointer. Programmers can easily add further conditions through an expressive annotation language. Like similar static checkers, ESC/Java supports "ghost fields", which are virtual fields that are only known to the checker, and which therefore the program itself cannot access. Although we do not know of any research that has attempted this, we believe that it should be possible

[1]See `http://pmd.sourceforge.net/integrations.html` for a complete list.

to encode an object's typestate through such a field, which should essentially allow programmers to use ESC/Java for basic typestate checking. Programmers can also use special `assume` annotations to suppress false warnings: if the programmer has knowledge about the program that the analyses in ESC/Java cannot derive, then she can add an `assume` annotation to make the analysis aware of this knowledge. Houdini (see below) is a tool that can generate annotations for ESC/Java automatically.

8.6.4 JML

JML, the Java Modeling language [LC06], is a general specification language that supports in-code specifications via annotations. JML supports pre- and post-conditions and invariants, but also unique features such as model programs [SLN07]. JML is the joint effort of a number of institutions and various people[2] have developed static and dynamic checkers for JML. While these tools vary greatly in their focus and abilities, we are not aware of any JML-based tool that would support the checking of typestate properties, neither at runtime nor at compile time.

8.7 Inferring properties

Clara assumes a set of given property specifications, in the form of runtime monitors that are annotated with dependency information. Generally we assume that programmers define these specifications by hand. However, researchers have developed tools that can assist programmers in inferring specifications directly from their program code. All these approaches share the common assumption that programs will produce mostly correct runs most of the time. Therefore, one should be able to infer possibly useful specifications from the "usual" behavior of a program.

[2]See `http://www.cs.ucf.edu/~leavens/JML/` for a complete listing.

8.7.1 Dynamic approaches

Dynamic approaches generate proposals for possible specifications by mining a set of traces that are either collected on-the-fly or ahead of time, using a number of test runs.

8.7.1.1 Daikon

Daikon [ECGN01] by Ernst et al. infers program invariants at runtime. The invariants are mostly numerical invariants that specify that the numeric value of a variable cannot be outside a certain range. Therefore, the invariants that Daikon infers are not really compatible with CLARA's notion of a specification. We will see that there are other tools that suite CLARA's specification needs better.

8.7.1.2 DIDUCE

DIDUCE [HL02] by Hangal and Lam instruments Java programs to mine and check program invariants at runtime. The goal is that these invariants provide the programmer with information about the root cause of an observed error: when an error occurs in a program that violated an established invariant just before the error occurred then the invariant will likely be helpful in finding the root cause of the error. The approach instruments every expression in the program to keep track of a predefined set of possible changes to the return value of each such expression. These expressions can be field stores and loads and method calls. The implementation tracks every program expression with a bit set in which each bit encodes information about whether or not the invariant holds. DIDUCE induces a runtime overhead of one to two orders of magnitude. DIDUCE can help identifying the root cause of a problem by inspecting (1) invariants that were invalidated just before an error or (2) invariants that are invalidated after a very long initial runtime. The invariants that DIDUCE produces do not describe invariants with respect to the typestate of an object. Therefore they too are not suitable inputs to CLARA. In fact, DIDUCE's approach is complementary to our approach with CLARA: while CLARA aims at identifying property violations, the goal of DIDUCE is to identify the root cause of such violations.

8.7.1.3 Java Anomaly Detector (JADET)

Wasylkowski, Zeller and Lindig propose the Java Anomaly Detector (JADET) for detecting object usage anomalies in programs [WZL07]. JADET first generates common usage patterns from the program, considering control flow, one object at a time. The tool lists uncommon usages contradicting the pattern, ranked by confidence. The authors show that seven uncommon usages in their benchmarks are actual defects. While useful, JADET's analysis may both miss actual defects and yield false warnings.

JADET's mining phase makes the tool fully automated. Clara requires property specifications, but can validate more complicated usage patterns, namely patterns involving multiple objects, like FailSafeIter. Furthermore, finite-state properties are more expressive than JADET's pattern language, which only allows relationships of the form "event e may precede event f". Extending JADET to finite-state properties would be an interesting project.

8.7.1.4 Specification mining

Ammons et al. put forward the idea of specification mining [ABL02]. In this approach, the authors developed an algorithm and tool to mine state-based specifications from dynamic execution traces collected at runtime. The traces consist of basic events, where each event can bind certain values, similar to the traces that we consider in the case of Clara. From these traces, the authors then create "scenarios", where each scenario is a set of events that occurs on a common set of objects. This is similar to our flow-insensitive Orphan-shadows Analysis, except that scenarios are defined in terms of runtime data. The authors then assume that the user provides a seed in the form of a single event of interest. The author's approach identifies the scenarios that contain this seed, normalize the scenarios and then feed them to a probabilistic automaton learner. The learner generates a probabilistic finite-state automaton (PFSA). The PFSA usually contains a hot core (high probability) and some other cold transitions. A "corer" extracts the hot core. Interestingly, when extracting the hot core, the corer has to take care not to remove transitions that are necessary to reach a final state. Note how this is similar to our our definitions of probes in Chapter 6: probes always contain a minimal activation set, which in turn contains enough shadows to reach a final state. The result of this coring process is a non-deterministic finite-state machine that is ready for human inspection.

The approach by Ammons et al. fits CLARA's notion of a specification very well. Firstly, the author's approach generates a finite-state specification in the form of a finite-state machine, which is exactly the input format that CLARA expects. Even better, state transitions in the specifications that Ammons et al.'s approach generates can bind free variables to objects. This allows for expressive specifications that reason about the internal state of multiple related objects. As we learned, transitions in CLARA's specifications for dependency state machines can bind free variables as well. Hence it almost appears as if the output of the author's tool could be easily converted to an input to CLARA. However, there is one important issue, which we plan to address in future work. The specifications that Ammons et al.'s tool generates are "positive" specifications: the finite-state machine describes allowed typestate transitions; a property violation occurs when the program triggers a transition labelled l while being in a state q that has no outgoing l-transition. This is inverse to the semantics of dependency state machines, in which the state machine describes unwanted behavior. It should be simple to define an alternative input format to CLARA, in which programmers or tools could then define dependencies in a "positive" way. CLARA could then automatically convert the supplied state machine to obtain an automaton that describes forbidden event sequences.

Runtime monitoring for violations of such "positive" specifications does, however, pose some new questions. The main problem is that a program can often violate a positive specification in multiple ways. It is important to report to the user in which way the program violated the property. For instance, consider a specification that defines that "files are always closed after they have been opened, and the program writes to files only while they are open". Now consider a monitor that reports that this property has been violated. Just notifying the programmer about "a violation" is of little use: the monitor would have to provide the information if the program violated the property by not closing an open file, or by writing to a closed file. Providing such information requires the monitor to issue a state-dependent error message. The monitor's error message depends on the state in which the monitor was last, just before the last transition lead to a violation. CLARA's dependency state machines circumvent this problem by using specifications of forbidden behavior instead. In the above example, programmers would supply two runtime monitors, one that monitors for files not being closed and one for monitoring writes to closed files.

8.7.1.5 Javert

Javert [GS08] is a novel approach to mining finite-state specifications by Gabel and Su. In their approach, the authors first use efficient techniques to find small patterns in dynamic traces. Then next Javert uses a rule engine to construct larger patterns from these smaller pattern instances. As the authors show, this process can result in impressively comprehensive specifications involving non-trivial combinations of objects. Unfortunately, the authors do not discuss whether the specifications that Javert generates contain any specific information about which objects instances are bound at which transitions of the specification. Such information would be necessary to make the specifications really useful for reasoning about groups of related objects. Javert is currently being turned into a commercial tool, and therefore not publicly available. This prevented us from obtaining more detailed information about the exact specification format at this point in time.

8.7.1.6 Lo and Maoz

Lo and Maoz present an approach [LM08] to infer property specifications of live sequence charts [DH99] from dynamic program runs. Live sequence charts explicitly model the lifetime of multiple objects and the events that these objects interchange. As we mentioned above, Maoz and Harel previously developed S2A, a program that generates monitoring aspects from live sequence charts. One can use S2A to generate monitoring aspects from the specifications that Lo and Maoz's inference engine mines. One could then use CLARA to evaluate these runtime monitors ahead of time.

8.7.2 Static approaches

There are also static approaches to inferring specifications. These approaches do not analyze runtime traces but instead inspect the static program structure to identify method calls and other operations that often happen in combination.

8.7.2.1 PR-Miner

PR-Miner [LZ05] uses frequent-itemset mining to determine "complex patterns": sets of identifiers that are often used in combination within a single method of a program. PR-Miner first parses the program, then hashes names to numbers and determines frequent itemsets. The algorithm makes sure to only generate "closed itemsets", as these closed itemsets subsume others. This speeds up computation. The algorithm filters out frequent itemsets that have low support (i.e., are not that frequent after all) and only retains the ones that have support over a given threshold t. The analyzed program does not violate properties described by sets that have a support of 100%. Sets that have a support of $t < s < 100\%$ describe properties that the program potentially violates. Given the analysis information, one can then easily determine the methods that can cause the violation.

Similar to our Nop-shadows Analysis, PR-Miner inspects the program one method at a time. This may cause a problem where the current method m does not contain some crucial method calls that other similar methods do contain, but where m in fact does call these methods indirectly. When this happens, the implementation of m is actually safe, but PR-Miner's analysis would usually report a false warning in this case. To mitigate the problem, the authors use a simple inter-procedural analysis to prune these false positives. The authors test PR-Miner over C programs, where this works well. It is unclear how well this approach would work in a Java-like language supporting virtual dispatch.

The rules that PR-Miner reports tell a programmer which methods the program under test usually calls on certain objects in combination, but to the best of our knowledge the rules do not tell the programmer in which order the program invokes these methods. Therefore any runtime monitor that would want to check these rules would have to be very basic: it could only report "missing" method calls. One could then however annotate the monitor with dependency annotations based on the programming rule. Because both the rule and the dependency annotation are flow-insensitive, this seems like a natural fit. It is interesting that PR-Miner does not take aliasing into account. PR-Miner instead assumes that different variables represent different objects. This is not so problematic because PR-Miner's analysis is both unsound and incomplete anyway. However, it may lead to patterns being misclassified in cases where objects are aliased.

8.7.2.2 Houdini

Houdini [FL01] is an annotation assistant for ESC/Java (see above). Houdini implements a generate-and-test approach. The tool first generates a set of candidate specifications for a given set of classes and then uses ESC/Java to verify or refute these specifications with respect to the given program. Houdini uses a set of simple heuristics to "guess" the initial set of candidate specifications. For instance, it is generally useful to specify that reference-typed arguments to a method call should not be null. Specifications that pass the verification with ESC/Java can be considered good candidates for an actual program specification. As mentioned above, the specifications that ESC/Java deals with are very different from those that Clara uses. In particular, ESC/Java focuses on data values, while Clara can only reason about typestates. Unfortunately, to the best of our knowledge, Houdini only supports ESC/Java version 1, which in turn only supports Java 1.3 source code.

Summary. As we showed, Clara provides a framework for both specifying a wide variety of expressive typestate-like properties and verifying whether programs adhere to these properties. As opposed to many other approaches that require program annotations, Clara is fully automated. This reduces the burden on the programmer, but on the other hand requires more expensive whole-program analyses where in other approaches efficient, modular, intra-procedural analyses suffice. We also showed that

Clara integrates a large body of own previous work into a common framework, yielding a set of analyses that are correct, precise and nevertheless efficient. We explained that Clara can be used with a wide variety of runtime-monitoring tools and how it relates to static analyses that support the efficient implementation of aspect-oriented language constructs. In relation to lightweight extended static checkers, we found that Clara's analyses are often orthogonal to these checkers: while many checkers try to find a generic set of programming errors, programmers would use Clara to find different classes of programming errors that occur by using methods or fields of an application interface in incorrect combinations or in an incorrect order. Last but not least, we gave an overview of dynamic and static tools for property inference. Researchers could integrate such tools into Clara to generate likely property specifications automatically from existing program code.

Chapter 9
Conclusion and future work

Finite-state specifications can describe important properties of programs, often called typestate properties in the literature. Such properties include descriptions of how programmers must or must not access given application interfaces. Up until now, there exists no standardized way to specify finite-state properties. This makes many analysis approaches incomparable, because these approaches often reason about properties that may seem equivalent at first sight but do reveal significant differences on closer inspection.

Further it would be desirable to verify whether a program fulfils a given set of finite-state properties, as this allows programmers to reason about the correctness of their programs. However, verifying whether a given program fulfils a given finite-state property is a hard problem, which requires precise abstractions of both pointer and control-flow relationships. Moreover, any practical solution to the finite-state verification problem requires efficient algorithms for manipulating these abstractions.

9.1 Static analysis of finite-state properties

We have presented Clara, a framework for specifying and verifying finite-state properties of large-scale object-oriented programs at compile time. Through a syntactic extension of the aspect-oriented programming language AspectJ, Clara allows programmers to specify finite-state properties in a unified way, independently of any finite-state specification formalism such as linear temporal logic or regular expressions. Further, Clara contains a set of static analyses that are both efficient and precise. As we showed, we gain precision by using a set of specially designed abstractions that store, for any abstract representative of a combination of runtime objects, not only information about whether these objects can be in a given automaton state, but also whether they cannot be in a given state. This combination of positive and negative information allows us to obtain precise flow-sensitive analysis information on an intra-procedural level, while at the same time allowing us to model the remainder of the program (all other methods) in a very efficient, flow-insensitive way. We make Clara's analysis even more efficient by executing different analysis parts in stages: simple and fast stages run early and often succeed in proving large parts of the program correct already. Later-running stages then have an easier job of proving the finite-state property for the remainder of the program.

It is also worthwhile noting that all analyses in Clara are fully automated; they require no user input and no program annotations. This is in contrast to other static approaches that use type systems to assure that programs fulfil given typestate properties [BA07,DF04]. Such approaches typically assume program annotations that give information about how a given statement or method may change an object's state. Clara infers such information automatically from the program and from an AspectJ-based specification of the property in question. Of course, this functionality comes at a cost: while approaches based on type systems are typically modular and therefore very efficient, Clara performs a whole-program analysis. A whole-program analysis requires the whole program to be present and analyzable, and it typically takes a longer time to compute. As we showed, the success of Clara's analyses crucially depends on the effectiveness of the pointer analyses that it uses. This can become a problem when programs use dynamic class loading and reflection, both of which cause the pointer analyses to be quite imprecise. Type-system based approaches often avoid this problem by modeling pointer relationships through explicit aliasing annotations. In the near future, we plan to experiment with automated approaches that are more modular, and therefore more efficient, but do not require program annotations.

9.2 Reducing the runtime-monitoring overhead

We designed Clara's analyses in a special way, so that they would not only allow us to report whether or not the given program under test adheres to the given finite-state property. If the program may violate the property then Clara produces a program that is instrumented with runtime checks, but only at exactly these positions at which Clara could not prove the program safe in the first place. As we showed, a program that is instrumented with such a partial set of instrumentation points often runs much more efficiently than a fully instrumented program that attempts to check the finite-state property only at runtime. Further, we showed that in cases where large overheads still remain after applying our static analyses, one can lower these overheads further by partitioning the instrumentation points into different probes and then distributing differently instrumented programs to multiple users. We determine probes in such a way that a partially instrumented program cannot report any false warnings.

9.3 Dependencies between events

Clara's analyses determine whether certain events may violate a given property by resolving dependencies between these events. Events are inter-dependent when they may refer to the same runtime objects and when they are on the same path to a final automaton state. The dependency information allows Clara to disregard an event e when no events exist that depend on e. However, we found that our notion of dependencies may be useful even for purposes that go far beyond static typestate analysis. For instance, Chen and Roşu found the same dependencies to be useful [CR09] to reduce the number of indexing trees in their runtime monitors. In the future, we plan to give this notion of dependencies a more theoretical treatment to see whether there is a more fundamental relationship between dependencies and finite-state machines.

9.4 Clara as a framework

We designed Clara in such a way that it accepts, as input, any finite-state runtime monitor that is implemented as a history-based AspectJ aspect—an aspect that changes its internal state based on the observed execution history. A large number of modern runtime-verification tools are already implemented as specification compilers that generate history-based AspectJ aspects. It is therefore easy to adapt these tools so that they are compatible with Clara. In this dissertation, we presented two forms of aspect annotations: dependent advice and dependency state machines. To make a runtime-verification tool compatible with Clara, the programmer of the tool simply modifies the tool so that it attaches appropriate annotations to the history-based aspects that the tool generates anyway. We have presented a generic algorithm that can generate correct annotations from any finite-state machine. We have implemented the approach for four different finite-state specification languages, on top of two different runtime-verification tools. To conclude, through its architecture, Clara allows a wide range of AspectJ-based finite-state runtime-monitoring tools to make use of state-of-the-art static optimization and verification almost for free.

In addition, we see Clara as an open framework for implementing and experimenting with static typestate analyses. Clara uses aspects to specify typestate properties. Because AspectJ is a very high-level declarative language, a programmer can easily denote typestate properties as aspects. The Clara framework supports convenient abstractions for the program points that matter to a given typestate property, and gives information about which of these program points could refer to the same runtime objects. Further, Clara completely abstracts from the actual specification formalism at hand; it only operates on finite-state machines. This makes Clara very flexible. In this dissertation we presented three static analyses that make direct use of this information. We propose Clara as an open framework that allows researchers to implement their own typestate analyses easily and effectively, but more importantly allows them to compare their analyses directly to each other.

Clara is freely available for download, along with all our benchmarks and benchmarking results, at `http://www.bodden.de/clara/`.

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, October 2005.

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*, pages 4–16. ACM Press, January 2002.

[ACH+05a] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 87–98. ACM Press, March 2005.

[ACH+05b] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, June 2005.

[AM07] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 161–172. ACM Press, March 2007.

[asp03] The AspectJ home page, 2003.
URL: <`http://eclipse.org/aspectj/`>.

[ATB$^+$06] Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondřej Lhoták, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, March 2006.
URL: <`http://www.aspectbench.org/`>.

[ATdM07] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 589–608. ACM Press, October 2007.

[AVY08] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 143–162. ACM Press, 2008.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, October 2007.

[Bar05] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 4339 of *Lecture Notes in Computer Science (LNCS)*, pages 152–169. Springer, October 2005.

[BCR09] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 3–14. ACM Press, March 2009.

[BGH$^+$06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM Press, October 2006.

[BH08] Eric Bodden and Klaus Havelund. Racer: Effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165. ACM Press, July 2008.

[BHL07] Eric Bodden, Laurie J. Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science (LNCS)*, pages 525–549. Springer, 2007.

[BHL$^+$08] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. *Journal of Logics and Computation*, November 2008. doi:10.1093/logcom/exn077.

[BLH08a] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 36–47. ACM Press, November 2008.

[BLH08b] Eric Bodden, Patrick Lam, and Laurie Hendren. Object representatives: a uniform abstraction for pointer information. In *Visions of Computer Science - BCS International Academic Conference*. British Computing Society, September 2008.
URL: <`http://www.bcs.org/server.php?show=ConWebDoc.22982`>.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science (LNCS)*, pages 126–138. Springer, March 2007.

[Bod05] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, November 2005.
URL: <`http://www.bodden.de/publications`>.

[Brz62] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Symposium on Mathematical Theory of Automata*, pages 529–561. Polytechnic Institute of Brooklyn, 1962.

[BSH08] Eric Bodden, Reehan Shaikh, and Laurie Hendren. Relational aspects as tracematches. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 84–95. ACM Press, March 2008.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 4, 1996.

[CM03] W. F. Clocksin and Chris Mellish. *Programming in Prolog*. Springer, 5th edition, 2003.

[Coo99] Standard Performance Evaluation Coorperation. SPECjvm98 Documentation, March 1999. Release 1.03 edition.

[Coo01] Standard Performance Evaluation Coorperation. SPECjbb2000 (Java Business Benchmark) Documentation, 2001. Release 1.01 edition.

[Cop05] Tom Copeland. *PMD Applied.* Centennial Books, November 2005.

[CR03] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *International Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 108–127, July 2003.

[CR07] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588. ACM Press, October 2007.

[CR09] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science (LNCS)*, pages 246–261. Springer, March 2009.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, pages 411–420. ACM Press, May 1999.

[Dah] M. Dahm. BCEL.
URL: <`http://jakarta.apache.org/bcel`>.

[DCCN04] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions of Software Engineering and Methodolology (TOSEM)*, 13(4):359–430, October 2004.

[DDE08] Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *International Conference on Automated Software Engineering (ASE)*, pages 228–237, Washington, DC, USA, 2008. IEEE Computer Society.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science (LNCS)*, pages 465–490. Springer, June 2004.

[DH99] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 1999.

[DP07] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 124–133. ACM Press, May 2007.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, February 2001.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[FGRY05] John Field, Deepak Goyal, Ganesan Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. *Science of Computer Programming*, 58(1-2):57–82, 2005.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe (FME)*, volume 2021 of *Lecture Notes in Computer Science (LNCS)*, pages 500–517. Springer, March 2001.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java.

In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, June 2002.

[fMR07] Wolfgang Grieskamp from Microsoft Research. Personal communication, January 2007.

[FYD$^+$06] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–144. ACM Press, July 2006.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification.* Addison-Wesley Professional, 3rd edition, 2005.

[GMF06] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 364–375. ACM Press, June 2006.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 385–402. ACM Press, October 2005.

[GS08] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 339–349. ACM Press, November 2008.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, August 2004.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 26–35. ACM Press, March 2004.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering (ICSE)*, pages 291–301. ACM Press, May 2002.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 132–136. ACM Press, October 2004.

[HPV07] Richard Halpert, Chris Pickett, and Clark Verbrugge. Component-based lock allocation. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. IEEE Computer Society, September 2007.

[Kle56] Stephen C. Kleene. *Automata Studies*, chapter Representation of Events in Nerve Nets and Finite Automata, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

[KLM06] Ingolf H. Krüger, Gunny Lee, and Michael Meisinger. Automating software architecture exploration with M2Aspects. In *Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM)*, pages 51–58. ACM Press, May 2006.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1042 of *Lecture Notes in Computer Science (LNCS)*, pages 1137–1143. Springer, August 1995.

[KRH04] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2004.
URL: <`http://www.disi.unige.it/person/CazzolaW/RAM-SE04.html`>.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154. ACM Press, June 2003.

[LC06] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML, 2006.
URL: <`http://www.jmlspecs.org/`>.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science (LNCS)*, pages 153–169. Springer, April 2003.

[Lin01] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, 3rd edition, 2001.

[LM08] David Lo and Shahar Maoz. Specification mining of symbolic scenario-based models. In *Workshop on Program analysis for software tools and engineering (PASTE)*, pages 29–35. ACM Press, November 2008.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[LZ05] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 306–315. ACM Press, September 2005.

[MH06] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 219–230. ACM Press, November 2006.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science (LNCS)*, pages 105–121. Springer, November 2003.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science (LNCS)*, pages 46–60. Springer, April 2003.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383. ACM Press, October 2005.

[NL08] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 347–366. ACM Press, October 2008.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science (LNCS)*, pages 214–240. Springer, July 2005.

[PGB$^+$05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, October 1977.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256. IEEE Computer Society, November 2004.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400. ACM Press, June 2006.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21. ACM Press, June 2008.

[SH05] Volker Stolz and Frank Huch. Runtime verification of concurrent haskell programs. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 113:201–216, January 2005.

[SHK98] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality. *Commununications of the ACM*, 41(8):67–73, 1998.

[SLN07] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 351–368. ACM Press, October 2007.

[SMU$^+$04] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 16–25. ACM Press, March 2004.

[SY86] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, January 1986.

[Tea04] The AspectJ Team. The AspectJ 5 Development Kit Developer's Notebook, 2004.
URL: <`http://eclipse.org/aspectj/doc/next/adk15notebook/`>.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann, June 2005.

[WV04] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 159–169. ACM Press, October 2004.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 35–44. ACM Press, September 2007.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, October 2005.

Appendix A
Proof of correctness and stability of dependent-advice generation

A.1 Correctness of Algorithm 4.3

In this section we prove the correctness Algorithm 4.3. The proof is conducted in multiple steps. First, we explain the exact relationship between automaton symbols and the dependent pieces of advice that recognize these symbols. Next, we define a set of *valid* dependencies for a formal language $\mathcal{L}$.

In Theorem A.1 we show that it is correct to not monitor events that are not referenced by any active, valid dependency declaration. This theorem holds for formal languages in general and for regular languages in particular.

Theorem A.3 then shows that all dependency declarations generated by Algorithm 4.3 are indeed valid.

Symbols and advice. In the following, we assume that every automaton symbol is associated with one or more pieces of advice that *recognize* this symbol: Whenever the event represented by the symbol occurs on a program execution, one of the associated pieces of advice sends a notification to the state machine, triggering state transitions for this symbol.

Let us now define some additional notation that will be used in the remainder of the proof.

Definition A.1 (Symbols of a word)**.** Let Σ be an alphabet. We define the function *sym* as:

$$sym(w) := \{w_1, \ldots, w_n \mid w = w_1 \ldots w_n\}.$$

Definition A.2 (Shuffle)**.** Let Σ be an alphabet and $w = w_1 \ldots w_n \in \Sigma^*$. We define a shuffle operator $\|$ over words and symbols as follows.

$$w \parallel a := \bigcup_{1<i\leq n} \{w_1 \ldots w_{i-1} a w_i \ldots w_n\}$$

Preventers. The definition of the shuffle operator allows us to easily define the notion of a *preventer*. Informally a preventer is an event that would prevent an execution trace from leading to a complete automaton match when it happened. For instance, assume that an automaton recognizes when a programmer calls `next()` twice on an iterator without calling `hasNext()` in between. Then the automaton would match the string "`next next`". The symbol "`hasNext`" would be a preventer for this match because the automaton would not match "`next hasNext next`".

Definition A.3 (Preventers)**.** Let Σ be an alphabet and $w = w_1 \ldots w_n \in \Sigma^*$. Also assume a fixed Σ-language $\mathcal{L}$ and $w \in \mathcal{L}$. We say that a symbol $a \in \Sigma$ is a *preventer* for w, $w \notin_a \mathcal{L}$ for short, if $(w \parallel a) \nsubseteq \mathcal{L}$.

Definition A.4 (Closed under shuffling)**.** For any Σ-language $\mathcal{L}$ we say that $\mathcal{L}$ is *closed under shuffling with a* if a is not a preventer for any word in $\mathcal{L}$:

$$\neg\exists w \in \mathcal{L} \;.\; w \notin_a \mathcal{L}$$

.

Example A.1 (Preventers and shuffle closure)**.** Let $\Sigma = \{a, b\}$ and $\mathcal{L}$ defined through the regular expression b^+. Then a is a preventer for the word $bb \in \mathcal{L}$ because $bab \notin \mathcal{L}$. On the other hand, b is not a preventer for bb because $bb \parallel b = \{bbb\} \subseteq \mathcal{L}$, i.e., $\mathcal{L}$ is closed under shuffling with b.

Preventers are important because we must not forget to monitor them, although they do not lead us closer towards the final state. If we accidentally disabled a preventer then the resulting automaton could recognize too many words, in terms of runtime verification leading to false positives.

Valid advice dependencies. We now come to advice dependencies. In the following we assume that a dependency consists of strong and weak *symbols* (opposed to advice). We can easily do so because, as noted above, advice are associated with symbols.

Definition A.5 (Dependency)**.** A dependency $D = (S, W)$ is an element of $\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, a combination of strong and weak symbols, with $S \cap W = \emptyset$.

A word w *activates* an advice dependency D if it contains all the symbols that are *strong* in D.

Definition A.6 (Activation)**.** We say that a word w *activates* a dependency $D = (S, W)$, if $\mathit{sym}(w) \supseteq S$.

On the other hand, a dependency D *assures recognition* of w if (1) w activates D (i.e., D is relevant to w) and (2) D contains all of w's preventers. Condition (1) means that D contains enough symbols to make sure that symbols of w can drive the automaton into a final state ("completeness"). Condition (2) on the other hand assures that every intervening event that could prevent w from matching in the original automaton will also be recognized by the optimized automaton ("soundness").

Definition A.7 (Assuring recognition)**.** We say that a dependency $D = (S, W)$ *assures recognition* of a word Σ-word w in a Σ-language $\mathcal{L}$ if w activates D and

$$\forall a \in \Sigma \;.\; w \notin_a \mathcal{L} \;\rightarrow\; a \in S \cup W$$

In general one could define any dependency declarations for any formal language. However, we demand that dependency declarations be valid: we say that a set $\mathcal{D}$ of dependency declarations is *valid* for a language $\mathcal{L}$ if (1) there are "enough" dependencies such that every word in $\mathcal{L}$ will indeed be recognized by the optimized automaton and (2) for every word, D assures recognition, i.e., activates the right preventers (for soundness).

Definition A.8 (Valid dependencies)**.** We say that a set $\mathcal{D}$ of dependencies is *valid* for a language $\mathcal{L}$ if for all $w \in \mathcal{L}$ it holds that:

1. $\exists D \in \mathcal{D}$ such that w activates D, and
2. for every such D, D assures recognition of w in $\mathcal{L}$.

Correctness of sparse monitoring.

Theorem A.1 (Validity for smaller languages)**.** Let $\mathcal{L}$ be a language and $\mathcal{L}^- \subset \mathcal{L}$. Let $\mathcal{D}$ be a set of dependencies that is valid for $\mathcal{L}$. Then $\mathcal{D}$ is also valid for $\mathcal{L}^-$. The proof is trivial because for every $w \in \mathcal{L}^-$ it holds that $w \in \mathcal{L}$.

Theorem A.2 (Sparse monitoring is correct)**.** Let Σ be an alphabet and $\mathcal{L}$ a Σ-language. Assume that $\mathcal{D}$ is a set of valid dependencies for $\mathcal{L}$. Assume now that we monitor a program in which certain events can be proven not to occur. This yields a reduced input alphabet $\Sigma^- \subset \Sigma$. Assume that one of the events that do not occur in the program is the event a: $a \in (\Sigma \backslash \Sigma^-)$. Then the language that the monitor can actually recognize over this program is: $\mathcal{L}_{\not{a}} := \{w | w \in \mathcal{L}, a \notin \mathit{sym}(w)\}$. Because $\mathcal{L}_{\not{a}} \subseteq \mathcal{L}$, it holds that $\mathcal{D}$ is also valid for $\mathcal{L}_{\not{a}}$ (Theorem A.1). This tells us that the dependencies are "rich enough" to assure recognition also over this reduced alphabet. However, some of these dependencies may have become superfluous, because they cannot be activated any more over this reduced input alphabet. Let $\mathcal{D}^- \subseteq \mathcal{D}$ be the dependencies which are activated by words in $\mathcal{L}_{\not{a}}$. We can define a reduced monitoring alphabet Σ^M that only contains the symbols that occur in dependency declarations activated through words over the program's reduced input alphabet. Let Σ^M be defined as:

$$\Sigma^M := \{b \in \Sigma^- \mid \exists D = (S, W) \in \mathcal{D}^- \ . \ b \in S \cup W\}$$

The resulting language is $\mathcal{L}^M$, defined as:

$$\mathcal{L}^M := \{w \mid w \in \mathcal{L}, \mathit{sym}(w) \subseteq \Sigma^M\}$$

The beauty of the dependency declarations is now that we know that any symbol b which we fail to monitor because it is not contained in any active dependency this symbol b could not have been a preventer in the first place, and therefore it is sound to not monitor b because $\mathcal{L}_{\not{d}}$ is closed under shuffling with b. It holds that:

$$\forall w \in \mathcal{L}^M \; \forall b \in (\Sigma^- \backslash \Sigma^M) \; . \; \neg(w \notin_b \mathcal{L})$$

Proof A.1. We know that $\mathcal{D}$ is valid for $\mathcal{L}$. We also know that w activates some $D \in \mathcal{D}^-$. Assume that $D = (S, W)$ and that there exists an $b \in S \cup W$. Then, because $D \in \mathcal{D}$ and $\mathcal{D}$ is valid for $\mathcal{L}$: $\neg(w \notin_b \mathcal{L})$. □

Example A.2 (Example for sparse monitoring)**.** Let us again consider the example automaton $\mathcal{M}$ from Figure 4.8. This automaton recognizes the language denoted by the regular expression $b^*ab^*(ccb^*)^*d$. Let us again assume a program in which c does not occur, i.e., $\Sigma^- = \{a, b, d\}$. Then $\mathcal{L}_{\not{d}}$ can be described by the regular expression b^*ab^*d, which is exactly the language that $\mathcal{M}$ accepts via the path P1. (Note that this language is closed under shuffling with b.) Consequently, words over Σ^- activate D1, but fail to activate D2 (because c is strong in D2). Hence, $D^- = \{(\{a, d\}, \{c\})\}$, which yields the monitoring alphabet $\Sigma^M = \{a, c, d\}$. Because the program will in fact never trigger c, it does not matter whether or not $c \in \Sigma^M$; we could even define: $\Sigma^M := \{a, d\}$. Now we see that, due to the definition of our dependencies, the regular expression b^*ab^*d over the original alphabet $\{a, b, d\}$ is naturally equivalent to the same expression over the alphabet $\{a, d\}$: b does not need to be monitored to reach a final state (otherwise it would have been *strong* in D1), and it is not a preventer for any words in $\mathcal{L}(b^*ab^*d)$ either (otherwise it would have been *weak* in D1).

Correctness of Algorithm 4.3. We start off with a trivial lemma that we require in the actual proof.

Lemma A.1 (Preventers and loops)**.** Let $\mathcal{M}$ be a state machine that has an a-loop on every non-initial, non-final state. Then it holds that $\mathcal{L}(\mathcal{M})$ is closed under shuffling with a.

This lemma holds trivially and remains without proof.

Note that the opposite direction does not hold: There can be automata whose language is closed under shuffling with a but which do not have an a-loop on every non-initial, non-final state. For instance, Figure A.1 shows two automata that both recognize the the language $\mathcal{L}(ba^*b)$, which is closed under shuffling with a. The left automaton does have an a-loop on every non-initial, non-final state, however the right automaton has not. In the latter case our Algorithm 4.3 will generate less precise dependency declarations (a will be strong in all declarations, although it would suffice to have a weak) but these declarations will still be valid. In our implementation, we determinize automata. In this case, the opposite direction does hold, yielding full optimization potential.

Theorem A.3 (Correctness of Algorithm 4.3)**.** Let $\mathcal{L}$ be a regular language. Then Algorithm 4.3 generates a set of dependency declarations that is valid for $\mathcal{L}$.

Proof A.2 (Proof of Theorem A.3)**.** Let $\mathcal{L}$ be a regular Σ-language with $w \in \mathcal{L}$. Let $\mathcal{M}$ be a finite-state machine with $\mathcal{L}(\mathcal{M}) = \mathcal{L}$. Because $w \in \mathcal{L}(\mathcal{M})$, we know that $\mathcal{M}$ recognizes w with a run ρ ending in a final state. Trivially, this run has a fragment that visits every edge only once, and this fragment visits the same edges as ρ itself. Therefore Algorithm 4.3 must have generated a dependency declaration $D = (S, W)$ for this fragment of ρ and w activates D. Assume now that $a \in \Sigma$ is a preventer of w for $\mathcal{L}$. In this case, ρ must have visited a state q that has no a self-loop (Lemma A.1). Therefore, $a \in S \cup W$. □

A.2 Stability of Algorithm 4.3

In this section we prove that *generateDependencies* is *stable*, i.e., that it computes equivalent sets of dependency declarations for equivalent finite-state machines.

Theorem A.4 (Stability of Algorithm 4.3)**.** Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two equivalent finite-state machines, i.e. $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the dependencies that Algorithm 4.3 generates for $\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly. Then $\mathcal{D}_1 \equiv \mathcal{D}_2$, i.e., $\mathcal{D}_1$ and $\mathcal{D}_2$ are logically equivalent. We therefore say that Algorithm 4.3 is "stable".

Proof A.3 (Proof of Theorem A.4)**.** By Theorem A.3 we know that both $\mathcal{D}_1$ and $\mathcal{D}_2$ are valid for $\mathcal{L}$. Assume now that $\mathcal{D}_1 \not\equiv \mathcal{D}_2$. Then there would be a word $w \in \mathcal{L}$ such

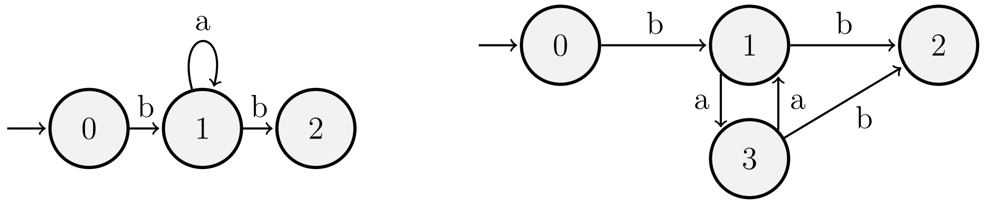


Figure A.1: Two automata recognizing the language $\mathcal{L}(ba^*b)$, which is closed under shuffling with a.

that $\mathcal{D}_1$ (without loss of generality) either (1) has no $D \in \mathcal{D}_1$ such that w activates D or (2) no such D assures recognition of w. This means that $\mathcal{D}_1$ is not valid for $\mathcal{L}$, which is a contradiction. $\square$

Appendix B
Proof of correctness of Nop-shadows Analysis

In Section 5.1.2, we defined the semantics of dependency state machines in terms of a predicate *necessaryShadow* that researchers can be choose freely, as long as it adheres to a given soundness condition, Condition 5.1. In this appendix we will show that if the Nop-shadows Analysis disables a shadow, then the soundness condition will hold for all events that this shadow could notify the runtime monitor about when the program under test is executed. To restate the soundness condition, for any sound implementation of *necessaryShadow* we demand:

$$\begin{aligned}&\forall a \in \Sigma \quad \forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ &\qquad a = t_i \wedge \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n) \\ &\qquad\qquad \longrightarrow \mathit{necessaryShadow}(a, t, i)\end{aligned}$$

Helper definitions. In the following, we define for any transition function δ the function $\hat{\delta}$ as the transitive closure of δ. Also, for any deterministic finite-state machine $\mathcal{M} = (Q, \Sigma, q_0, \delta, F)$, we define for any $q \in Q$ a state machine $\mathcal{M}_q$ as $\mathcal{M}_q := (Q, \Sigma, q, \delta, F)$.

Correctness of condition for shadow removal. Assume that the Nop-shadows Analysis disables a shadow s with $\mathit{label}(s) = t_i$. In the following, we will prove that in this case

$$\mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) = \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n)$$

must hold. When this holds, then we can safely define $\mathit{necessaryShadow}(\mathit{label}(s), t, i) :=$ **false** for all traces t and positions i without violating the soundness condition, because the soundness condition only applies to cases where the two *matches* sets are different.

Assume a (projected and therefore ground) runtime trace $t = t_1 \dots t_i \dots t_n \in \Sigma^+$ and a shadow s with $t_i = \mathit{label}(s)$. For convenience, let us define $w_1 := t_1 \dots t_{i-1}$, $a := t_i$ and $w_2 := t_{i+1} \dots t_n$, i.e., we have that $t_1 \dots t_n = w_1\ a\ w_2$. Let $\mathit{source} := \hat{\delta}(q_0, w_1)$ and $\mathit{target} := \hat{\delta}(q_0, w_1\ a) = \delta(\mathit{source}, a)$. Further assume that the following holds for s:

$$\mathit{target} \notin F \wedge \forall Q_f \in \mathit{futures}(s) : \mathit{source} \in Q_f \longleftrightarrow \mathit{target} \in Q_f$$

From the definition of *matches* we know that:

$$\forall w \in \mathit{pref}(w_1) : w \in \mathit{matches}(w_1\ w_2) \longleftrightarrow w \in \mathit{matches}(w_1\ a\ w_2)$$

Hence, in the following, we only regard words w with $w \notin \mathit{pref}(w_1)$.

Further, because $\mathit{target} \notin F$ we know that $w_1\ a \notin \mathit{matches}(w_1\ a\ w_2)$ and hence it follows that:

$$\forall w \in \mathit{pref}(w_1\ a) : w \in \mathit{matches}(w_1\ w_2) \longleftrightarrow w \in \mathit{matches}(w_1\ a\ w_2)$$

Hence, in the following, we only regard words w with $w \notin \mathit{pref}(w_1\ a)$.

Assume $\exists w = w_1\ a\ w' \in (\mathit{pref}(w_1\ a\ w_2) - \mathit{pref}(w_1\ a))$ with

$$\neg(w \in \mathit{matches}(w_1\ w_2) \longleftrightarrow w \in \mathit{matches}(w_1\ a\ w_2)). \tag{B.1}$$

But this cannot hold: because it holds that

$$\forall Q_f \in \mathit{futures}(s) : \mathit{source} = \hat{\delta}(q_0, w_1) \in Q_f \longleftrightarrow \mathit{target} = \hat{\delta}(q_0, w_1\ a) \in Q_f$$

we know that $\mathcal{L}(\mathcal{M}_{\mathit{source}}) = \mathcal{L}(\mathcal{M}_{\mathit{target}})$. This contradicts the existence of a witness w as described by B.1. □

Therefore, we now only need to show that the set of states that we approximate in our forward and backward analysis is correct. In particular, our implementation has to assure that for every set $Q_f \in \mathit{futures}(s)$ it holds that all states $q \in Q_f$ are indeed continuation-equivalent, i.e., that for all ground traces t that the continuation of the program execution after reading s could produce it holds that

$$t \in \mathcal{L}(\mathcal{M}_{\mathit{source}}(s)) \leftrightarrow t \in \mathcal{L}(\mathcal{M}_{\mathit{target}}(s)).$$

This implies that our implementation must never merge states sets, as merging state sets may cause the analysis to assume invalid equivalencies. Indeed our worklist algorithm, Algorithm 5.1 (page 162), assures that state sets are never merged: while the algorithm does over-approximate pointer information in various ways, it never merges configurations that have differing state sets Q_c. Every configuration $c = (Q_c, b_c)$ represents one element of the set *futures*. Algorithm 5.1 simply propagates these configurations but never merges them. In particular, note that the algorithm has no special treatment for control-flow merge points: when two different configurations reach the same statement along different paths then the algorithm will simply propagate both configurations, without attempting to merge these configurations in any way.

Further, the algorithm takes into account *all* possible continuations because it propagates configurations along all possible intra-procedural and inter-procedural paths. When propagating intra-procedurally, the algorithm refines the binding representative of any configuration using the simplification rules from Table 5.2. These rules hold trivially. When propagating configurations inter-procedurally, along $succ_{\not{rh}}$, the algorithm does not refine the binding representatives. Because the un-refined binding representatives are a least as permissive as refined representatives would be, this is a sound over-approximation.

The correctness of the forward and backward analysis then follows from the fact that we (1) initialize both analyses with configurations for all places at which an initial (or final) state could be reached, and (2) propagate this information along all possible control-flow paths (or abstractions of these), taking into account all "relevant" shadows (as determined by the function *relevantShadow*), and (3) never merges any configurations with differing state sets. $\square$

Minimality of reversed state machine. We mentioned that the reversed state machine that we use is minimal, because we obtain it by reversing an already-determinized finite-state machine. Brzozowski proved already in 1962 the minimality of state machines that are obtained that way:

To quote Brzozowski [Brz62, Theorem 13]:

> "Given any (not necessarily reduced) state diagram G (with starting state q_λ and all other states accessible from q_λ), the graph G^- of predecessors of π_λ is always reduced if only one state is introduced for each predecessor of π_λ. In other words, two states of G^- are equivalent if and only if their predecessor sets are equivalent."

Translated into our terminology, this reads as:

> Given any (not necessarily minimal) non-deterministic finite-state machine $\mathcal{M} = (Q, \Sigma, q_0, \Delta, F)$ (with starting state q_0 and all other states reachable from q_0), the state machine $\mathit{rev}(\mathcal{M})$ is always minimal if for any state q of $\mathcal{M}$ and $a \in \Sigma$ it holds that $|\{q' \in Q \mid \exists (q', a, q) \in \Delta\}| = 1$.

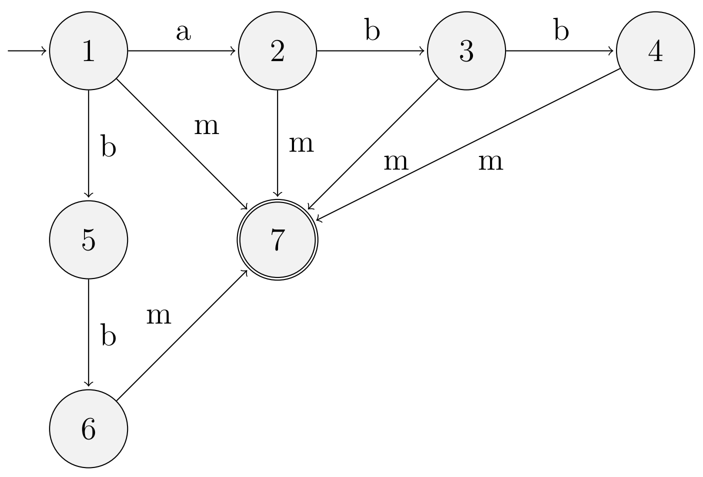
For the deterministic state machine $\mathit{det}(\mathcal{M})$, the condition $|\{q' \in Q \mid \exists (q', a, q) \in \Delta\}| = 1$ holds by definition, and hence $\mathit{det}(\mathit{rev}(\mathit{det}(\mathcal{M})))$ must be a minimal deterministic finite-state machine for $\overline{\mathcal{L}(\mathcal{M})}$.

Appendix C
Non-optimality of greedy shadow deactivation

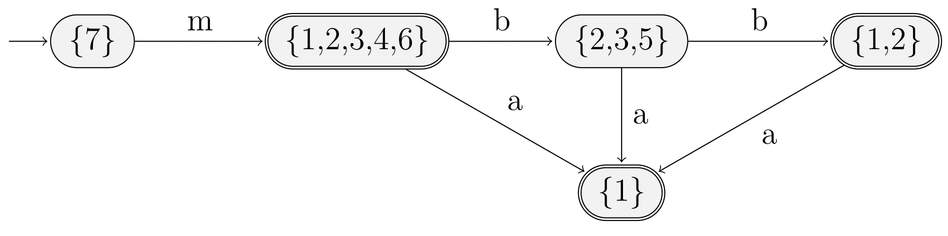
The Nop-shadows Analysis that we presented in Section 5.2 disables in every iteration just a single shadow. This is because the analysis can only determine whether a shadow s is a nop shadow in its current context. Disabling a nop shadow modifies the context of other shadows and may then render other shadows necessary that were formerly nop shadows too. For instance, in Figure 5.10 (page 152) both the shadows at line 5 and 6 are nop shadows. This means that it is safe to disable either one of them. However, disabling the shadow at line 5 will render the shadow at line 6 to be necessary, i.e., not a nop shadow, and the other way around. In this example, it does not matter, though, which shadow we disable: no matter whether the shadow at line 5 or the one at line 6 remains enabled eventually, both partially instrumented programs will result in the same execution traces.

But this need not necessarily be the case. We can construct an example in which disabling nop shadows in a certain order can lead to an optimal result but disabling nop shadows in another order leads to a result that is not optimal: because the shadows are disabled in a "greedy" fashion, we can steer into a local optimum. This prevents the algorithm from finding the global optimum. Consider a property that induces the dependency state machine $\mathcal{M}_{forward}$ in Figure C.1a. This state machine accepts the regular language "$[a[b[b]]]m \mid bbm$". Figure C.1b shows the appropriate backward state machine $\mathcal{M}_{backward}$. Now consider a program with an execution sequence "$a\ b\ b\ m$". In Figure C.2a we show this sequence at the top, along with the analysis information that the Nop-shadows Analysis computes. When executing this program, the state machine matches once, when reading the m. When looking at the underlined state numbers in the figure, we can see that in this example all shadows except for the m-shadows are nop shadows: we can safely drop any single one of these shadows and the state machine will still match once, when reading the m.

Now let us assume that we first drop one of the b shadows, yielding the sequence "$a\ b\ m$". We show this case in the second row of Figure C.2a. Then the other b-shadow is still a nop shadow. When we disable this b-shadow this yields the trace "$a\ m$". Now a is a nop shadow again and disabling it yields the optimal trace "m".

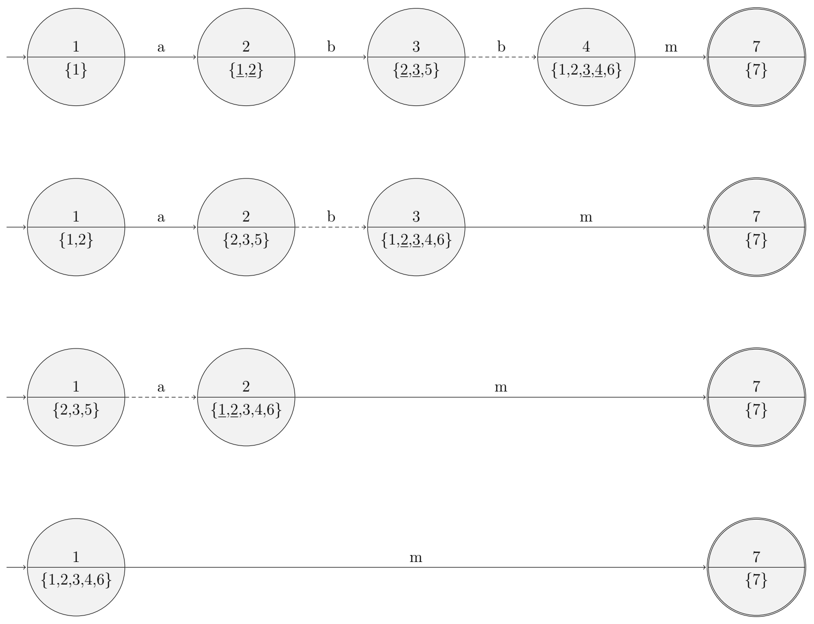


(a) Deterministic finite-state machine $\mathcal{M}_{forward}$

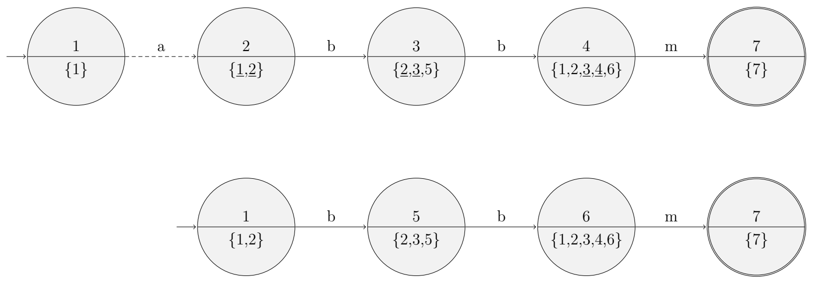


(b) Deterministic inverted finite-state machine $\mathcal{M}_{backward}$

Figure C.1: Example state machines $\mathcal{M}_{forward}$ and $\mathcal{M}_{backward}$


1 {1} a 2 {1,2} b 3 {2,3,5} b 4 {1,2,3,4,6} m 7 {7}
1 {1,2} a 2 {2,3,5} b 3 {1,2,3,4,6} m 7 {7}
1 {2,3,5} a 2 {1,2,3,4,6} m 7 {7}
1 {1,2,3,4,6} m 7 {7}

(a) When disabling shadows in order b, again b, and then a, all shadows except for the last one can be disabled.


1 {1} a 2 {1,2} b 3 {2,3,5} b 4 {1,2,3,4,6} m 7 {7}
1 {1,2} b 5 {2,3,5} b 6 {1,2,3,4,6} m 7 {7}

(b) When disabling the a-shadow first, the analysis fails to identify any further nop shadows.

Figure C.2: Example showing that the order in which shadows are disabled matters. Top half of state: information from forward pass. Bottom half of state: information from backwards pass. Dashed edge: nop shadow that is removed in next step. Underlined state number(s) identify equivalencies that cause a nop shadow to be identified.

Let us now roll back to the original program "a b b m" and let us assume that we disable the nop shadows in a different order, disabling a first. Figure C.2b depicts this situation. When we disable a, this yields the sequence "b b m". In this sequence, no shadow is a nop shadow: on the trace "b b m" the state machine matches after reading the m, but if we disable any single of the shadows, the state machine will not match any more on the resulting trace.

The "problem" with the state machine in this example is that if an a is read initially, then it does not matter whether we see zero, one or two b events before reading the m event, however if the initial a is missing then the number of b events does matter: "b b m" is in the state machine's language and so is "m", but "b m" is not an element of the language.

This example shows that disabling shadows in a random, greedy manner may not be optimal. For the example from Figure C.2b, a more clever analysis could be able to infer that we may disable both b-shadows in combination, yielding the same optimal trace "m" that we obtained in Figure C.2a. Such an analysis could infer see that this would be sound because both 1 and 6 are elements of the set $\{1, 2, 3, 4, 6\}$ that we obtain after the second b-shadow. However, such an analysis would have to compute dominance relationships to prove that the analysis information holds for all possible paths between the locations just before the first b-shadow and the one just after the second b-shadow. When considering single shadows in isolation, such as our Nop-shadows Analysis, this dominance relationship is obvious because every shadow dominates itself.

It is important to note that the example from Figure C.1 was constructed to demonstrate a theoretical problem. We believe that in practice state machines will have a simple enough form and programs a simple enough structure such that the greedy algorithm that we propose will actually yield the globally optimal result in many cases. (By "optimal" we here mean optimal with respect to the alias information and state sets that the Nop-shadows Analysis computed, i.e., the algorithm will yield an optimal result when assuming a perfect approximation of control flow and aliasing.)

Appendix D
Number of minimal activation sets

The complexity of Algorithm 6.1 (page 200) is bounded by the number of possible minimal activation sets.

Number of minimal activation sets. Assume that we have k strong symbols and $|\mathcal{S}| =: n$ shadows. Each minimal activation set contains exactly k of these n shadows. Hence, the number of minimal activation sets can be computed as follows. Let $p \in \mathcal{P}(\mathcal{P}(\mathcal{S}))$ be a k-partitioning of shadows, i.e., $|p| = k$, with $p = \{p_1, \ldots, p_k\}$. Then the number of minimal activation sets is bounded by the maximal number of subsets of $\mathcal{S}$ which contain exactly one element of each p_i. In other words, if $\forall i : |p_i| =: n_i$, then we are looking for $max(\prod_{1 \leq i < n} n_i)$ such that $\sum_{1 \leq i < n} n_i = n$. However, this is the same as looking for the maximal volume of a k-dimensional hyper-rectangle with a fixed perimeter. For any number of dimensions, it holds that for any given perimeter, the hyper-rectangle with this perimeter and a maximal volume is a hyper-cube. In our particular case, we may not be able to form a perfect hyper-cube, though, because the length of every edge needs to be a whole number. We can hence determine the volume of the hyper-rectangle that comes closest to the hype-cube as follows:

$$\left\lfloor \frac{n}{k} \right\rfloor^{k-(n \bmod k)} \cdot \left\lceil \frac{n}{k} \right\rceil^{n \bmod k}$$

Here, $\left\lfloor \frac{n}{k} \right\rfloor$ is the length of the "short sides" of the hyper-rectangle, and $\left\lceil \frac{n}{k} \right\rceil$ is the length of the "long sides". The value $k - (n \bmod k)$ is the number of "short sides" and $n \bmod k$ is the number of "long sides".

Example. Assume that we have $n = 5$ shadows and $|\mathcal{S}| = k = 2$. In this case, the combined length of the two sides that span up the rectangle must be $n = 5$. According to the above rule, the two lengths of sides that come closest to forming a square, then, are the lengths $\left\lfloor \frac{5}{2} \right\rfloor = 2$ and $\left\lceil \frac{5}{2} \right\rceil = 3$. In this case, there are $2 - (5 \bmod 2) = 1$ short sides (of length 2) and $n \bmod k$ long sides (of length 3). For $n = 6$ and $k = 2$ there would be 4 equal sides (all "short") of length 3 instead.

Impact of aliasing. The number of minimal activation sets that we gave above is the maximally possible number, which is obtained in the case where every shadow is compatible with every other shadow. This will usually not be the case, because quite often shadows will bind variables to must-not-aliased pointers. To model the impact of the "amount of aliasing" on the number of minimal activation sets, we introduce a "compatibility ratio" c, which is the average ratio of mutually compatible shadows:

$$\left\lfloor \frac{n}{k} \cdot c \right\rfloor^{k-(n \bmod k)} \cdot \left\lceil \frac{n}{k} \cdot c \right\rceil^{n \bmod k}$$

Table D.1 gives an impression of the growth rate of this function for reasonable values of $0 \leq k \leq 5$, $0 \leq n \leq 1000$. For the value $n = 1000$ we show both the numbers for the worst case with $c = 1$ and the numbers for the more likely case of $c = 0.1$. In our benchmark set, there is only one property with an n-value as large as 4: FailSafeIterMap.

n	1	2	3	4	5	100	1000	1000
c	1	1	1	1	1	1	1	0.1
1	1	2	3	4	5	100	1000	100
2	0	1	2	4	6	2500	250000	2500
3	0	0	1	2	4	37026	37036926	37026
4	0	0	0	1	2	390625	$3.9 \cdot 10^9$	390625
5	0	0	0	0	1	3200000	$3.2 \cdot 10^{11}$	3200000
k								

Table D.1: Maximal number of minimal activation sets for up to 5 strong symbols and up to 1000 shadows