Continuation Equivalence: a Correctness Critetion for Static Optimizations of Dynamic Analyses

Eric Bodden
Software Technology Group, Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt (CASED)
bodden@acm.org

ABSTRACT

Dynamic analyses reason about a program's concrete heap and control flow and hence can report on actual program behavior with high or even perfect accuracy. But many dynamic analyses require extensive program instrumentation, often slowing down the analyzed program considerably.

In the past, researchers have hence developed specialized static optimizations that can prove instrumentation for a special analysis unnecessary at many program locations: the analysis can safely omit monitoring these locations, as their monitoring would not change the analysis results. Arguing about the correctness of such optimizations is hard, however, and ad-hoc approaches have lead to mistakes in the past.

In this paper we present a correctness criterion called Continuation Equivalence, which allows researchers to prove static optimizations of dynamic analyses correct more easily. The criterion demands that an optimization may alter instrumentation at a program site only if the altered instrumentation produces a dynamic analysis configuration equivalent to the configuration of the un-altered program with respect to all possible continuations of the control flow.

In previous work, we have used a notion of continuation-equivalent states to prove the correctness of static optimization for finite-state runtime monitors. With this work, we propose to generalize the idea to general dynamic analyses.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms

Performance, Verification, Theory

Keywords

Static optimization, Correctness, Dynamic analysis, False positives, False negatives, Runtime verification

WODA '11, July 18, 2011, Toronto, ON, Canada

1. INTRODUCTION

Dynamic analyses reason about a program's concrete heap and control flow and hence can report on actual program behavior with high or even perfect accuracy, and in terms of an actual program execution. This is in stark contrast to static analyses that need to abstract from program inputs and the environment and hence must make conservative approximations on many levels, often leading to false positives that distract from the actually interesting analysis results.

Many dynamic analyses, however, share the problem of requiring extensive program instrumentation, which can often slow down the analyzed program by up to several orders of magnitude [1]. While such slowdowns are always undesired, they can sometimes even prohibit practical use of a particular dynamic-analysis technique in practice.

But what if the expensive instrumentation could be omitted for at least some of the program's code location? In the past, researchers have developed specialized static optimizations that exploit properties of the program's code to prove program instrumentation unnecessary at certain program locations [4–9, 14]. Focusing on one special dynamic analysis problem, such static optimizations try to find program locations for which a static approximation can prove that monitoring those locations can safely be omitted. In general, omitting to monitor such a location is safe when the analysis reports the same result no matter whether the location is monitored or not.

It is non-trivial, however, to argue about the soundness of such a static optimization, and many papers on this topic argue about soundness in a rather ad-hoc way. In publications by this author [8] and others [11], such ad-hoc proofs have lead to mistakes within the proofs and eventual unsoundness of the design and implementation of the proposed optimization. A general method or framework for proving the correctness of such static optimizations would aid researchers in conducting those proofs more easily and reliably.

We address this very problem by presenting a general correctness criterion for static optimizations of dynamic analysis, called Continuation Equivalence. The criterion demands that an optimization alters the instrumentation at a program site only in such a way that, when visiting the site at runtime, the altered instrumentation will produce a dynamic analysis configuration that is equivalent to the configuration of the un-altered program with respect to all possible continuations of the control flow that can follow after that site. The particular notion of equivalence depends on the chosen dynamic analysis and is hence a parameter to our approach.

In previous work we have used a notion of continuation-

equivalent states to prove correct a static optimization for finite-state runtime monitors [3,4]. In this work, we propose to generalize this notion to the setting of general dynamic analyses, by lifting the notion of Continuation-equivalent automaton states to Continuation-equivalent analysis configurations.

To summarize, this paper contains the following original contributions:

- examples of dynamic analyses and possible static optimizations,
- a discussion of the criteria that static optimizations of dynamic analyses need to fulfil, in order to be correct, and
- the idea of a general framework for proving the correctness of static optimizations of dynamic analyses using a notion of continuation-equivalent configurations.

2. EXAMPLES

In this section we briefly discuss examples of dynamic analyses and static optimizations that could go along with them. We will aim to identify factors that could impact the soundness of the respective optimizations.

2.1 Optimizing Finite-state Runtime Monitors

One large class of dynamic analyses is concerned with runtime verification [12]. In runtime verification, programmers use specialized tools to instrument a program under test with runtime checks. Those checks often resemble a finite-state machine: program events move the machine from one state to another, and the instrumentation will issue an error message if the program ever causes the state machine to reach a declared error state. Figure 1, for example, shows a finite-state runtime monitor for the "Connection" property: a connection is initially connected but can be disconnected and re-connected; it is an error to write to a disconnected connection.

Runtime-verifying such finite-state properties has been shown to be very costly, as the dynamic analyses must associate different states with different (e.g. Connection) objects. Program slow-downs of several orders of magnitude have been observed [1,2]. Static optimizations can lower the monitoring overhead by removing program instrumentation (and hence the number of events dispatched to the runtime monitor) at program locations at which it is correct to do so. We found that in about 80% of all our benchmarks, such static optimizations are so precise that they can prove *all* instrumentation unnecessary, rendering runtime monitoring of the program under test entirely obsolete [4].

In earlier work, Fink et al. have developed an approach completely based on static analysis [10] that attempts to solve a very similar problem. For any object of any type in the program whose object's needs to follow a given finite-state protocol, the authors use a forward analysis to track the possible states of each object at any given program location. If the analysis encounters an error state at a statement s, the analysis signals a possible property violation at s.

In earlier work, we and others [8,11] independently developed different approaches based on Fink et al.'s idea, but with the goal to use the analysis results to optimize finite-state runtime monitors. As an example, consider the code in Figure 2a. First consider a version of this code where line 6 is

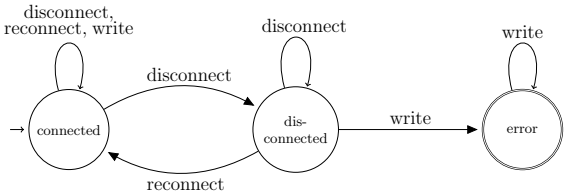

Figure 1: Finite-state machine for "Connection" property

not commented out. This code would yield the control-flow graph in Figure 2b. For this program fragment, monitoring of the "Connection" property from Figure 1 is entirely unnecessary: When the program writes to the connection in line 8, the connection is guaranteed to be connected, as it is re-connected along both branches. Hence, the runtime monitor can never reach its error state on this program, and runtime monitoring of this program is obsolete altogether. In particular, note that the `reconnect` call at line 4 requires no monitoring—we will return to this statement in a minute.

In the example just discussed, a simple forward analysis works correctly. The following example will show, however, that a pure forward analysis is generally insufficient when it comes to optimizing a dynamic analysis: it lacks knowledge about the possible "future", or more formally the continuation, of the execution. Hence, let us now consider the case in which line 6 is commented out, as shown in Figure 2a. This code yields the control-flow graph in Figure 2c. Opposed to the earlier version, this program version can cause the runtime monitor to trigger an error message: when taking the else-branch, the program will write to a closed connection. Hence, it should be clear that one may not omit runtime monitoring of the disconnect and write statements at lines 2 and 8: removing the instrumentation at those statements would cause the runtime monitor to falsely not report the violation of the Connection property—a false negative.

More interestingly, however, also the reconnect statement at line 4 requires monitoring. To see why this is the case, assume for a moment that we removed the instrumentation at line 4, thereby disregarding reconnect events during static analysis. Now further assume that the program executes its then-branch, i.e., line 4. On this execution, the runtime monitor should *not* issue an error message, as the connection was correctly reconnected before being written to, however because the optimized program now monitors a `close` followed by a `write`, the monitor will report a (false) error message for the optimized program—a false positive.

The two analyses presented in earlier work were both incorrect, as they handled this latter case incorrectly. The problem is that, when focusing on the reconnect statement at line 4, a forward analysis is not sufficient to tell apart the case in Figure 2b from the one in Figure 2c: in both cases, the reconnect is preceded by a close, and hence, a forward analysis will yield the same information in both cases, although monitoring of the reconnect may be omitted in one case but not the other.

The essence of the problem can be described by an apparent lack of analysis information at line 4. The decision of whether or not it is correct to optimize away the instrumentation of this statement depends on the nature of the possible continuations of the control flow after line 4. In the case of Figure 2b, no error state can be reached at any statement on this continuation, in Figure 2c, an error state

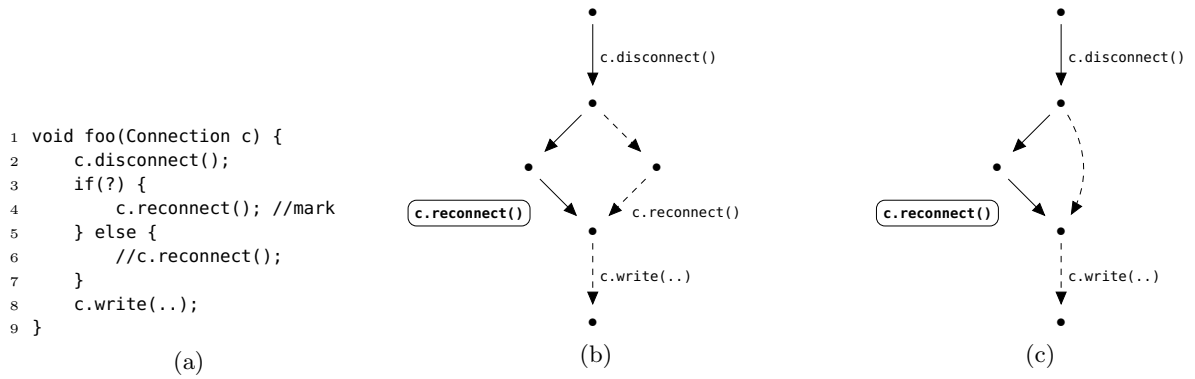

Figure 2: Example exposing unsoundness in earlier static optimization. The marked `reconnect` statement requires no monitoring in (b) but does in (c). Yet, a forward analysis alone cannot possibly distinguish the situations (b) and (c) from one another, as it propagates information to this statement only along the solid edges, which are equal in both situations.

can be reached at line 8. The applicability (and hence correctness) of the optimization hence depends on knowledge about the future, which can most easily be obtained through a backwards program analysis.

Hence, in a more recent paper [4] we solved the problem by using a novel notion of continuation-equivalent states. At a statement s, two states q_1 and q_2 are said to be continuation-equivalent if and only if along all possible continuations of the control flow after s it does not matter whether the runtime monitor is in q_1 or q_2: along each continuation, the monitor will or won't reach an error state either way. We compute continuation-equivalent states using a backwards analysis. This analysis provides additional analysis information that is sufficient to tell both cases apart when inspecting the reconnect statement at line 4. In the case of Figure 2b, the `reconnect` statement would transition only between states that are provably continuation equivalent, while in Figure 2c those states would not be equivalent, and the optimization at this statement would hence correctly be suppressed.

Note that, to be correct, such a backward analysis needs to compute its information along all possible continuations of the control flow, which may involve complicated control flows such as loops, method calls, method re-executions, exception throws and recursion. Fortunately, the construction of continuations can be reused for different static analyses that are based on the notion of continuation equivalence. We hence propose to develop a conceptual framework along with an application framework that allows the construction of and reasoning about the relevant abstractions.

2.2 Execution Profiling

To demonstrate that the idea of Continuation Equivalence is not restricted to the field of Runtime Verification, we next consider an execution profiler. In this example, we assume an execution profiler that constructs calling-context trees, such as the one recently proposed by Sarimbekov et al. [13]. A calling-context tree contains a node for each method call in each context, but it contains no duplicates in case the the same method is called in the same context multiple times. Hence the code in Figure 3a, where the profiling code is shown as a comment, could be statically optimized as shown in Figure 3b, where only the last call to `accept` is instrumented. Also in this example, the removal of the instrumentation of the `accept`-call at line 2 is only valid (and can only be proven correct) because it is known that another call to `accept` will follow, and because this call remains instrumented. (Note that for ease of presentation we abstract from the problem of possible exceptional control flow. In other words, we assume that the second `accept`-call post-dominates the first one.)

Conclusions from Examples

Both examples show that dynamic analyses can benefit from static optimization. The example of finite-state runtime monitoring shows in particular, that static optimizations can benefit from information about the program's past execution: instrumentation for a finite-state runtime monitor can only safely be disabled if the current possible states at the optimized statement are known. Information about the past execution can be computed using a forward analysis.

The same example also showed, however, that information drawn from a forward analysis alone may not be enough. This is because omitting instrumentation of a given statement, and therefore omitting the processing of events otherwise generated by that instrumentation, is only sound if this omission is (1) either irrelevant to the final analysis information that the dynamic analysis computes, or (2) is guaranteed to be made-up for by other instrumentation on the continuation of the control flow. This second case is demonstrated by our second example, profiling for calling-context trees. After executing the first call to `accept`, the dynamic analysis is first in an incorrect configuration in which the tree-node for `accept` is missing. However, this omission is guaranteed to be made-up for by the remainder of the execution, i.e., all its possible continuations.

3. SHOWING CORRECTNESS BY PROOVING CONTINUATION EQUIVALENCE

This section consists of two parts. First, we present a dynamic view, defining under which circumstances a code transformation is correct with respect to the dynamic analysis of a single dynamic trace. In Section 3.2 we then show how to use the notion of Continuation Equivalence to prove correctness with respect to all traces.

```
1 void visitNode(Visitor v) {
2     left.accept(v);
3     //insertEdge("Node.accept(Visitor)");
4     right.accept(v);
5     //insertEdge("Node.accept(Visitor)");
6 }
```

(a) Naïvely instrumented code

```
1 void visitNode(Visitor v) {
2     left.accept(v);
3     right.accept(v);
4     //insertEdge("Node.accept(Visitor)");
5 }
```

(b) Code with optimized instrumentation

Figure 3: Code with instrumentation (shown as comments) for recording calling-context trees

3.1 Correctness of Program Transformations

Let E be a (likely to be infinite) set of possible program events. Executing program p on input i will yield an event trace $\mathit{trace}(p,i) = e_1 \dots e_n \in E^*$. For any trace $t = e_1 \dots e_n$ we define $\mathit{head}(t) := e_1$ and $\mathit{tail}(t) := e_2 \dots e_n$.

A dynamic program analysis d is, in the most general sense, a function that computes, in a step-wise manner, for a trace t a final "result" configuration C_r, starting from an initial analysis configuration C_0, such that:

$$\begin{aligned} d(t, C_0) &= \\ d(\mathit{tail}(t), d(\mathit{head}(t), C_0)) &= C_r \end{aligned}$$

The dynamic event trace is assumed to be generated by the execution of an instrumented program. For any event $e \in E$ we define $\mathit{loc}(e)$ as the event's instrumentation point, i.e., as the static program location causing event e to happen.[1] We define the program path

$$\begin{aligned} \mathit{path}(p,i) &:= \\ \mathit{path}(\mathit{trace}(p,i)) &:= \mathit{loc}(e_1) \dots \mathit{loc}(e_n) \end{aligned}$$

to be the statically projected program path of the dynamic event trace $\mathit{trace}(p,i)$.

Next assume a program transformation *transf* on a sequence of statements s. Then Figure 4 depicts the circumstances under which this transformation is correct. First, we can divide any program path π into three sub-paths a, b, c such that $\pi = a \cdot b \cdot c$ and neither a nor c contain any statements in s. Because a contains no statements in s, we know that $\mathit{transf}(a) = a$, and hence also $d(\mathit{transf}(a), C_0) = d(a, C_0) =: C_a$. In other words, *transf* cannot affect the computation of the dynamic analysis on the prefix a.

The transformed section of the program now leads to potentially diverging configurations before and after transformation. Let $d(a \cdot b, C_0) = d(b, C_a) =: C_{ab}$. Further, let $\mathit{transf}(b) = b'$. Then for the transformed program we have $d(\mathit{transf}(a \cdot b), C_0) = d(\mathit{transf}(b), C_a) = d(b', C_a) =: C_{ab'}$.

Definition (Correctness of *transf*)
Crucially now, we say that *transf* is correct with respect to $\pi = a \cdot b \cdot c$, if and only if $d(\mathit{transf}(c), C_{ab}) = d(\mathit{transf}(c), C_{ab'})$. In other words, even though the transformed statement sequence b' may temporarily compute an analysis configuration different from the one that b would have computed, this difference does not matter. Either way we will end up in the final configuration C_{abc}, under the assumption that c will still be processed. We say that a transformation *transf* is correct in general if it is correct with respect to all paths π.

[1] For certain events, such as timeouts, no such statement might exist. Indeed, proofs based on Continuation Equivalence are not applicable if such events are allowed. Hence we abstract from such events in the remainder this paper.

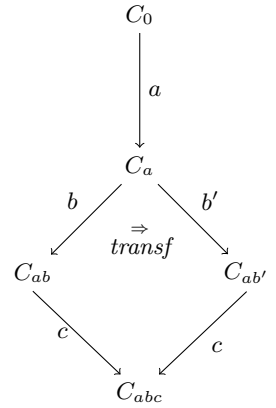

Figure 4: A correctly optimized dynamic analysis: even through the transformed sequence b' may yield a different intermediate configuration, this difference will be made up for by the remainder of the execution.

While this correctness criterion only gives a definition of the circumstances under which the static optimization of a dynamic analysis is correct, the criterion itself is not constructive. In the following, we will show a general way to prove correctness in the above sense, using the notion of Continuation Equivalence.

3.2 Proving Correctness by Showing Continuation Equivalence

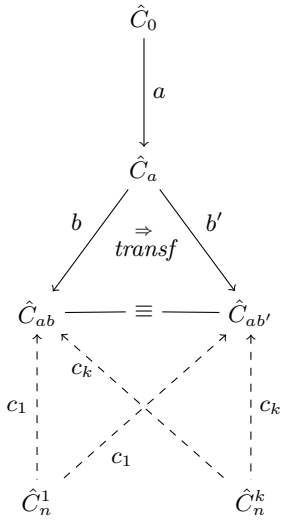

Figure 5: Computation

Figure 5 gives an outline of what it takes to prove a static optimization correct by showing Continuation Equivalence. Assume we want to prove correctness of the transformation *transf*. To do so, we first show that $\hat{C}_0$ is a sound over-approximation of C_0, as likewise is $\hat{C}_a$ for C_a, and so on. To show this, one needs to argue why the flow function along a, b and b' transfers one soundly over-approximated configuration into another. Crucially this part of the proof could be conducted as an instance of a generalized proof framework (and application framework), which we propose to develop in the near future. Such a framework would allow researchers to reason about continuations in a fully abstract manner, leaving the construction of approximations over these continuations to the framework itself. In particular, researchers could abstract from complicated control-flow related issues such as loops, exceptions and recursion. The author's dissertation [3] gives a first idea of what such a framework could look like.

In the end, we wish to prove that the configurations $\hat{C}_{ab}$ and $\hat{C}_{ab'}$ obtained this way are continuation equivalent, i.e., any possible difference between $\hat{C}_{ab}$ and $\hat{C}_{ab'}$ won't matter, given the possible continuations of control flow. We want to prove:

$$\hat{C}_{ab} \equiv \hat{C}_{ab'}$$

The exact definition of this equivalence relation "$\equiv$" depends on the specific dynamic analysis at hand. To be able to prove *transf* correct, however, the relation must be defined such that if $\hat{C}_{ab} \equiv \hat{C}_{ab'}$ holds, then indeed the program transformation on b cannot affect the outcome of the dynamic analysis on any continuation.

The equivalence relation hence must be based on information about the program path's possible future, which is most conveniently computed through a backwards analysis (in the figure depicted by the dashed arrows). This analysis must be path sensitive: configurations for different continuations must not be confused. For instance, in our analysis instantiation for optimizing finite-state monitors, we may have $2 \equiv 3$ along one possible continuation and $1 \equiv 3$ along another (where $1, 2, 3$ are state numbers) but this does *not* imply that $1 \equiv 2$ holds as well. In any case, the equivalence relation must be cased on a "must" analysis: two configurations are only continuation equivalent if they yield equivalent configurations along all executions, not just along some execution.

Discussing a complete proof is outside the scope of this idea paper. The author's dissertation [3, Appendix 2] gives a complete proof for our static optimization of runtime monitors [4], and we refer the interested reader to this document.

4. CONCLUSION

We have presented the idea of basing correctness proofs for static optimizations of dynamic analyses on a notion of Continuation Equivalence. An optimization is correct if it alters the program in such a way that the optimized dynamic analysis ultimately yields the same analysis result as the un-optimized analysis. During parts of the execution, the optimization may cause the analysis to be in a configuration that does not accurately reflect the current program state. According to Continuation Equivalence, this is no problem, as long as the dynamic analysis of the remainder of the program execution must ultimately lead to the same configuration as if the optimization had not been applied.

We have successfully used Continuation Equivalence to proof correct different static optimizations for the purpose of runtime verification [4–6], but our investigation suggests that the same notion can be used for correctness proofs of optimizations of other kinds of static analyses as well.

5. REFERENCES

[1] 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, `http://www.aspectbench.org/`, March 2006.

[2] 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.

[3] Eric Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009.

[4] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 5–14, New York, NY, USA, 2010. ACM.

[5] 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.

[6] Eric Bodden and Klaus Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering (TSE)*, 36(4):509–527, July 2010.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Workshop and conference series on Runtime Verification. `http://runtime-verification.org/`.

[13] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and platform-independent calling context profiling for the Java virtual machine. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2011.

[14] Suan Yong and Susan Horwitz. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design*, 27:313–334, 2005. 10.1007/s10703-005-3401-0.