Partial Verification Results

DISS. ETH NO. 22868

Partial Verification Results

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich

(Dr. sc. ETH Zurich)

presented by

Valentin Tobias Wüstholz

MSc ETH CS, ETH Zurich

born on 22.09.1985

citizen of

Germany

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Işil Dillig, co-examiner
Dr. Francesco Logozzo, co-examiner
Prof. Dr. Bertrand Meyer, co-examiner

2015

Abstract

In recent years, program analysis tools have been increasingly applied to real-world software to prevent defects as early as possible. Examples of such tools include both static analyzers and automatic test case generation tools. While the latter traditionally under-approximate the possible program executions to find errors, the former traditionally consider additional program executions that are not actually possible in the analyzed program. This makes it possible to efficiently analyze programs with a large or infinite number of program executions and to prove the correctness of the analyzed program in case the analysis *over-approximates* the possible program executions (i.e., is sound). As a consequence, static analyzers may report *spurious* errors that do not reveal real defects in the analyzed program.

In practice, many static analyzer neither under- nor over-approximate the program executions of the analyzed program. Their *designers* trade soundness for other qualities—such as precision, performance, and automation—by deliberately ignoring certain checks (e.g., that a method respects its write effect specification) or by deliberately making assumptions that do not hold for all program executions (e.g., that no arithmetic overflow occurs).

These characteristics of static analyzers motivate why verification results are often *partial* in practice: (1) some assertions of a program have neither been verified nor have been shown to lead to a defect (i.e., may be spurious errors) and (2) some program executions—including ones that may result in defects—have been ignored due to sources of deliberate unsoundness in the analysis.

To express and share such results with the user or other program analysis tools, we have developed a technique for annotating programs with partial verification results using two new language constructs. For instance, these allow us to express if a property could not definitely be verified and if a property has only been verified under assumptions that might not always hold. We describe several novel use cases for expressing partial verification results from diverse areas of program analysis—such as test case generation, specification inference, counterexample-based error reporting, and static analysis.

In particular, we present an architecture for combining static analyzers with test case generation tools by exchanging programs that have been annotated with partial verification results between tools. By soundly expressing what has already been verified, tools can benefit from the results of other tools and reduce their verification effort.

To evaluate deliberate unsoundness in a practical static analyzer, we identified and documented all sources of deliberate unsoundness in the .Net static analyzer Clousot. Based on this, we developed a wrapper for Clousot that uses our technique for annotating a program using partial verification results. By expressing most sources of unsoundness explicitly and developing a suitable runtime instrumentation, we evaluated whether Clousot's unsound assumptions are violated in practice and whether such violations cause Clousot to miss bugs. Such findings can guide users of static analyzers in using them fruitfully, and help their designers in striking a good balance between soundness and other qualities of an analyzer, such as precision, performance, and automation.

Nowadays, partial verification results are often shown to users within an integrated development environment (IDE). We present the IDE for Dafny—a programming language, verifier, and proof assistant—that addresses two issues present in most state-of-the-art IDEs for program verifiers: low responsiveness and lack of support for understanding non-obvious verification failures. To this end, we present both new techniques and integrate existing technique to improve the user experience. This allows the IDE to provide verification feedback as the user types and to present more helpful information about the program or failed verification attempts in a demand-driven and unobtrusive way. As a result, the user is able to quickly gain insights about the program and the cause of partial verification results.

To increase the responsiveness of the program verifier during such interactions with the user, we designed a system for fine-grained caching of verification results. The caching system uses the program's call graph and control-flow graph to focus the verification effort on just the parts of the program that were affected by the user's most recent modifications. The novelty lies in how the original program is instrumented with partial verification results from the cache to avoid unnecessary work for the verifier. By using our technique for expressing partial verification results, we are able to reuse some cached verification results even if assumptions in the program (e.g., due to modular reasoning about calls by means of the callee's postcondition) are affected by a change.

Zusammenfassung

In den letzten Jahren wurden Werkzeuge zur Programmanalyse mehr und mehr dazu eingesetzt, um Defekte in praxisnaher Software so früh wie möglich zu verhindern. Beispiele für solche Werkzeuge sind Werkzeuge zur statischen Analyse und zur automatischen Testfall-Generierung. Während Letztere traditionell die möglichen Programmausführungen unterapproximieren um Fehler zu finden, betrachten Erstere weitere Programmausführungen, die nicht tatsächlich im analysierten Programm auftreten. Dies ermöglicht es, Programme mit einer grossen oder unendlichen Zahl von Programmausführungen effizient zu analysieren und die Korrektheit des analysierten Programms zu beweisen, falls die Analyse die möglichen Programmausführungen *überapproximiert* (d.h. sound ist). Folglich kann es dazu kommen, dass Werkzeuge zur statischen Analyse *unechte* Fehler melden, die gar nicht zu echten Defekten im analysierten Programm führen.

In der Praxis betrachten viele Werkzeuge zur statischen Analyse weder eine Unter- noch eine Überapproximation der möglichen Programmausführungen. Ihre *Designer* wägen Soundness gegen andere Qualitäten—wie etwa Präzision, Effizienz, und Automatisierung—ab, indem sie absichtlich gewisse Überprüfungen auslassen (zum Beispiel, dass eine Methode ihre Write-Effect-Spezifikation erfüllt) oder absichtlich Annahmen treffen, die nicht in allen Programmausführungen zutreffen (zum Beispiel, dass kein arithmetischer Overflow eintritt).

Diese Charakteristika von Werkzeugen zur statischen Analyse begründen, weshalb Verifikationsresulte in der Praxis oft *partiell* sind: (1) einige Assertions des Programms wurden weder verifiziert noch wurde gezeigt, dass sie zu einem Defekt führen (d.h. sie könnten unechte Fehler sein) und (2) einige Programmausführungen—inklusive solcher, die zu Defekten führen könnten—wurden ignoriert aufgrund von absichtlicher Unsoundness in der Analyse.

Um solche Resultat auszudrücken und mit dem Benutzer oder anderen Werkzeugen zur Programmanalyse zu teilen, haben wir eine Technik entwickelt, die es erlaubt, mittels zweier neuer Sprachkonstrukte das Programm mit partiellen Verifikationsresulten zu annotieren. Beispielsweise lässt sich damit ausdrücken, dass eine Eigenschaft nicht mit Sicherheit verifi-

ziert werden konnte und dass eine Eigenschaft lediglich unter Annahmen verifiziert wurde, die nicht zwangsläufig zutreffen müssen. Wir beschreiben mehrere Fälle aus verschiedenen Gebieten der Programmanalyse—wie zum Beispiel Testfall-Generierung, Spezifikationsinferenz, Gegenbeispiel-basierte Fehlermeldeverfahren und statische Analyse—in denen sich partielle Verifikationsresulte ausdrücken lassen.

Im Speziellen stellen wir eine Architektur vor, um Werkzeuge zur statischen Analyse mit solchen zur Testfall-Generierung zu kombinieren, indem Programme ausgetauscht werden, die mit partiellen Verifikationsresulten annotiert wurden. Indem sound ausgedrückt wird, was bereits verifiziert wurde, können Werkzeuge von den Resultaten anderer Werkzeuge profitieren und ihren Verifikationsaufwand senken.

Um absichtliche Unsoundness in einem praktischen Werkzeug für statische Analyse auszuwerten, haben wir alle Quellen von absichtlicher Unsoundness in Clousot, einem Werkzeug zur statische Analyse für .Net, identifiziert und dokumentiert. Basierend darauf haben wir einen Wrapper für Clousot entwickelt, der unsere Technik zur Annotation von Programmen mit partiellen Verifikationsresulten einsetzt. Indem die Mehrzahl aller Quellen von absichtlicher Unsoundness explizit gemacht werden und durch die Entwicklung einer geeigneten Laufzeitinstrumentierung, haben wir untersucht, ob Clousots unsounde Annahmen in der Praxis verletzt werden und ob solche Verletzungen dazu führen, dass Clousot Fehler übersieht. Solche Erkenntnisse können Designern von Werkzeugen zur statischen Analyse sowohl dabei helfen, diese nutzbringend einzusetzen, als auch eine gute Balance zwischen Soundness und anderen Qualitäten solcher Werkzeuge, wie beispielsweise Präzision, Effizienz und Automatisierung, zu finden.

Heutzutage werden partielle Verifikationsresulte dem Benutzer oft in einer integrierten Entwicklungsumgebung (IDE) angezeigt. Wir stellen die IDE für Dafny—eine Programmiersprache, ein Verifikationswerkzeug, und ein Beweisassistent—vor, welche zwei Probleme in aktuellen IDEs für Programmverifikationswerkzeuge angeht: geringe Reaktionsfreudigkeit und die mangelhafte Verständnisförderung bei nicht offensichtlichen Verifikationsfehlern. Zu diesem Zweck stellen wir neue Techniken vor und integrieren existierende Techniken zur Verbesserung der Benutzerfreundlichkeit. Dies erlaubt der IDE dem Benutzer Verifikationsrückmeldungen anzuzeigen, während er tippt, und ihm unauffällig und je nach Bedarf hilfreiche Informationen über das Programm oder fehlgeschlagene Verifikationsversuche zukommen zu lassen. Das führt dazu, dass es dem Benutzer möglich ist schnell Erkenntnisse über das Programm oder die Ursache der partiellen Verifikationsresulte zu erlangen.

Um die Reaktionsfreudigkeit des Programmverifikationswerkzeugs während solcher Interaktionen mit dem Benutzer zu erhöhen, haben wir ein System entwickelt, um feinkörnig

Verifikationsresultate zwischenzuspeichern. Dieses System benutzt sowohl den Call-Graph wie auch den Control-Flow-Graph des Programms um die Verifikationsbemühungen auf die Teile des Programms zu konzentrieren, die von den letzten Änderungen des Benutzers betroffen sind. Das Novum liegt dabei darin, wie das ursprüngliche Programm mit partiellen Verifikationsresultaten aus dem Zwischenspeicher instrumentiert wird, um unnötige Arbeit für das Verifikationswerkzeug zu vermeiden. Indem wir unsere Technik zum Ausdrücken von partiellen Verifikationsresultaten einsetzen, sind wir in der Lage Verifikationsresultate aus dem Zwischenspeicher selbst dann zu verwenden, wenn Annahmen im Programm (zum Beispiel aufgrund von modularer Beweisführung über Aufrufe mittels der Nachbedingung des Aufgerufenen) von der Änderung betroffen sind.