Ergebnisbewertung konservativer statischer Data-Race-Analysen

Sandro Degiorgi Martin Wittiger

Universität Stuttgart, Institut für Softwaretechnologie
Universitätsstr. 38, 70569 Stuttgart
{sandro.degiorgi, martin.wittiger}@informatik.uni-stuttgart.de

Zusammenfassung

Data-Races sind ein ernstes Problem der nebenläufigen Software-Entwicklung. Sie lassen sich mit der Analyse-Suite Bauhaus durch konservative statische Analyse aufspüren. Die Ergebnisse müssen manuell klassifiziert werden, da False-Positives auftreten. Die hier beschriebene Analysesoftware unterstützt und beschleunigt diesen zeitaufwändigen und teuren Prozess durch die automatisierte heuristische Bewertung aller Warnungen. Dieses Papier zeigt die Mach- und Skalierbarkeit des Ansatzes.

1 Einleitung

Das Projekt Bauhaus, gegründet 1996, beschäftigt sich mit Software-Architektur, Software-Reengineering und Programmverstehen [1]. Entwickelt wurde eine umfangreiche Werkzeugsammlung zur Analyse von sequenziellen und nebenläufigen Programmen.

In nebenläufigen Programmen treten Race-Conditions auf. Eine Teilmenge dieser Race-Conditions sind die Data-Races. Man spricht von einem Data-Race, wenn mindestens zwei nebenläufige Threads auf denselben Speicherbereich zugreifen, wobei mindestens einer der Zugriffe schreibend ist und die Gleichzeitigkeit der Zugriffe nicht explizit ausgeschlossen wird [4]. Data-Races können Ausführungsfehler (z. B. verlorene Aktualisierung oder Verwendung inkonsistenter Daten) verursachen oder undefiniertes Verhalten des Programms bewirken.

Die Untersuchung von Programmen auf Data-Races mittels statischer Softwareanalyse ist ein aktuelles Forschungsthema, auch weil es für Software-Tests und andere dynamische Methoden schwer ist, die teils nur sporadisch auftretenden Data-Races sicher zu finden. Die Data-Race-Analyse von Bauhaus ist konservativ. Sie setzt sich aus verschiedenen Teilanalysen zusammen (z. B. Zeigeranalyse, verfeinert durch eine Escape-Analyse). Diese Analyseteile schätzen Programmverhalten so ab, dass sie alle Data-Races finden – False-Negatives treten nicht auf. Da diese Analysen im Allgemeinen jedoch unentscheidbar sind, müssen False-Positives in Kauf genommen werden [3].

Eine Schärfung der Analyse unter Ausnutzung spezifischer Eigenschaften von Softwaresystemen verringert die Zahl der False-Positives. Keul [3] gelingt mit diesem Ansatz eine Reduzierung um 39 %. Der Einsatz des Werkzeugs in der Praxis zeigt, dass gerade bei größeren Softwaresystemen zu viele Falschmeldungen auftreten.

Die Data-Race-Analyse gibt die betroffenen globalen Variablen und Speicherstellen, jeweils mit einer Liste von Zugriffstellenpaaren, aus. Wie in [5] beschrieben, müssen diese Data-Race-Warnungen manuell klassifiziert werden. Dies ist zeitaufwändig und damit auch kostspielig.

Um diesen Prozess zu unterstützen, wurde eine Analysesoftware entwickelt, die es einem Software-Ingenieur ermöglicht, die tatsächlich schädlichen Data-Races schneller zu identifizieren.

Unser Ansatz basiert darauf, heuristisch relevante Muster zu erkennen und daran eine Bewertung der Warnungen auszurichten. Die Bewertung einer Warnung soll hoch sein, wenn die Betrachtung lohnend ist. Das Betrachten von True-Positives ist lohnender als das Betrachten von False-Positives, da man nur im ersten Fall einen Fehler finden kann. Außerdem spielt es eine Rolle, wie schwerwiegend die Auswirkungen eines Data-Race sind. Gibt es Hinweise darauf, dass ein bestimmtes Data-Race besonders verheerendes Fehlverhalten verursachen kann, soll es höher bewertet werden, als wenn sein Auftreten nur von begrenzter Bedeutung ist.

2 Mustererkennung, Filter und Bewertung

Die Implementierung bewertet alle Data-Race-Warnungen. Hierzu wurden verschiedene Filter entwickelt, die jeweils ein Muster erkennen und für jede Warnung eine Einzelbewertung berechnen. Somit sind am Ende der Berechnung für jedes Muster Einzelbewertungen verfügbar, die zu einer heuristischen Gesamtbewertung kombiniert werden können.

2.1 Filterimplementierungen

Filter 1 untersucht die Anzahl und Art (schreibend oder lesend) der Zugriffe aller Data-Race-Warnungen auf Komponenten eines Objekts. Aufgrund funktionaler Abhängigkeiten zwischen den einzelnen Komponenten geht dieser Filter davon aus, dass es sich umso eher um ein schädliches Data-Race handelt, je höher die Anzahl der Zugriffe ist. Hierbei werden schreibende Zugriffe stärker gewichtet als lesende.

Sei W_ω diejenige Teilmenge der Warnungen, die auf

Diese Arbeit wurde im Rahmen des Projektes ARAMiS durch das Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01IS11035 gefördert. Die Verantwortung für den Inhalt liegt bei den Autoren.

ein Objekt ω zugreifen, dann ergibt sich die Zahl der anscheinend problematischen Zugriffe auf ein Objekt ω wie folgt:

$$Uses_\omega = \left| \{ w \in W_\omega \mid Has_{Use}(w) \} \right| \quad (1)$$

$$Defs_\omega = 2 \cdot \left| W_\omega \right| - Uses_\omega \quad (2)$$

Filter 2 arbeitet analog zu Filter 1, basiert jedoch auf den funktionalen Abhängigkeiten zwischen den Variablen eines Moduls, d. h. statt Zugriffen auf Komponenten eines Objekts werden die Zugriffe auf Variablen eines Moduls gezählt. Auch hier werden die Warnungen desto höher bewertet, je mehr anscheinend problematische Zugriffe es auf Variablen des gleichen Moduls gibt.

Ist W_μ diejenige Teilmenge der Warnungen, die Variablen betreffen, die im Modul μ deklariert werden, können die Gleichungen 1 und 2 entsprechend verwendet werden.

Filter 3 sucht Warnungen, bei denen die Zugriffe in einem der Threads unter Schutz, in einem anderen Thread ohne oder mit disjunktem Schutz stattfinden. Je mehr Zugriffe mit unterschiedlichem Schutz der Filter findet, umso höher bewertet er die Warnung.

Der Filter untersucht für jede von Warnungen betroffene Speicherstelle, unter welchem Schutz die Zugriffe erfolgen. Dabei wird gezählt, wie viele unterschiedliche Schutzarten auftreten.

Sei Z_v die Menge der Zugriffe auf die Variable v und $P : Acc \rightarrow Prot$ die Funktion, die jedem Zugriff seinen Schutz zuweist (ungeschützten Zugriffen wird der leere Schutz zugewiesen), dann berechnet die folgende Funktion f eine Bewertung im Intervall $[0;1[$.

$$f(v) = 1 - \left| P(Z_v) \right|^{-1} \quad (3)$$

2.2 Weitere Filterkonzepte

- Dieser Filter betrachtet die Zugriffe von Data-Race-Warnungen auf Variablen genauer, die vom Speicherbus nicht atomar behandelt werden können (z. B. Verbundtypen oder Ganzzahlen, deren Größe nicht der Busbreite entspricht). Wird eine Variable nicht atomar geschrieben, kann es dazu kommen, dass Werte gelesen werden, die so nie geschrieben wurden. Dies spricht für eine besondere Gefährlichkeit dieser Data-Races. Liegen die konkreten Typinformationen der beteiligten Variablen und die nötigen Informationen zum Zielsystem vor, kann ein Filter eine entsprechende Bewertung der Warnung vornehmen.

- Ist erkennbar, dass eine Warnung in einem Kontext auftritt, der mit großer Wahrscheinlichkeit der Unschärfe von Teilanalysen (z. B. einer Kontext-insensitiven Zeigeranalyse) geschuldet ist, so kann die Warnung durch eine negative Einzelbewertung herabgestuft werden. Ganz allgemein ist eine Suche nach Mustern sinnvoll, die negative Bewertungen rechtfertigen. Analog zur Auszeichnung einer erkennbar lohnenden Warnung ist es möglich, Einzelbewertungen vorzunehmen, die für eine deutlich begrenzte Bedeutung stehen.

2.3 Prototypische Implementierung

Die in Kapitel 2.1 skizzierten Filter sind in einem Kommandozeilen-Werkzeug implementiert. Das Werzeug wurde bereits bei mehreren Analysen eingesetzt (vgl. Tabelle 1). Die Einzel- und Gesamtbewertungen der Data-Race-Warnungen können als CSV-Datei zur weiteren Verarbeitung gespeichert werden. Zusätzlich kann mittels eines C-Interfaces selektiv auf die Bewertungen zugegriffen werden. Eine verwandte Arbeit [2] nutzt dies, um die Informationen grafisch darzustellen, und ermöglicht so interaktives Arbeiten mit den Bewertungen.

Die maschinelle Weiterverarbeitung ist sinnvoll, da bereits bei Programmen mit wenigen Hundert Zeilen Quelltext die manuelle Überprüfung der Warnungen sehr aufwändig ist. Auf Basis der Einzelbewertungen können spezielle Aggregationen der Bewertungen synthetisiert werden.

	SLoCs	#Vars	#DRW	Laufzeit
System A	40 k	$\sim$ 400	374 k	68 s
System B	41 k	$\sim$ 400	735 k	21 s
System C	55 k	$\sim$ 700	152 k	31 s

Tabelle 1: Ergebnisse der Untersuchung dreier eingebetteter Systeme aus dem Automotive-Bereich.

3 Fazit und Ausblick

Der Ansatz der automatisierten Ergebnisbewertung ist vielversprechend und zeigt in den ersten Ergebnissen bereits, dass die Filter gut skalieren und auch auf umfangreichen Quelltexten performant operieren.

Neben der Entwicklung weiterer geeigneter Muster soll der Ansatz im nächsten Schritt qualitativ evaluiert werden. Die bereits implementierten Filter müssen auf Nützlichkeit untersucht werden; es muss gezeigt werden, dass sie in ihrer Gesamtheit gute Ergebnisse liefern.

Literatur

[1] Thomas Eisenbarth, Rainer Koschke, Erhard Plödereder, Jean-François Girard und Martin Würthner. Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In: *Workshop Software Reengeneering*, Fachberichte Informatik, Seiten 17–26, 1999.

[2] Timm Felden und Torsten Görg. Werkzeugunterstützte Eliminierung von Data-Races in Eclipse (unveröffentlicht), 2013.

[3] Steffen Keul. Tuning Static Data Race Analysis for Automotive Control Software. In: *11th IEEE International Working Conference on Source Code Analysis and Manipulation*, Seiten 45–54, 2011.

[4] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro und Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. In: *ACM Transactions on Computer Systems*, Band 15 (4), Seiten 391–411, 1997.

[5] Martin Wittiger und Steffen Keul. Extraktion von Interthread-Kommunikation in eingebetteten Systemen. In: Erhard Plödereder et al. (Hg.), *Automotive – Safety & Security 2012*, *LNI*, Band P-210, Seiten 55–67. Köllen Druck + Verlag GmbH, Bonn, 2012.