Parallelitätsanalyse für Slicing von Java Threads

Erweitertes Abstract

Christian Hammer
Universität Passau, hammer@fmi.uni-passau.de

1 Einleitung

Für automatisches Reengineering benötigt man garantierte Semantikerhaltung. Notwendige Voraussetzung ist hierbei z.B. dass der Slice eines bestimmten Codestücks gleich bleibt, d.h. dass es von den gleichen Anweisungen abhängig bleibt. Dadurch kann gewährleistet werden, dass alle Daten eines verschobenen Codestücks bei dessen Berechnung auch wirklich aktuell sind. Refactoring ist ein Beispiel, bei dem Slicing benutzt wird um die Semantik zu erhalten. Für viele aktuelle Programmiersprachen gibt es aber keine etablierte Theorie für deren Abhängigkeitsanalyse zum Program Slicing

Java z.B. bietet dem Programmierer eine moderne Programmiersprache mit Objektorientierung und Threads zur parallelen Ausführung von Programmteilen. Vor Kurzem konnten wir ein Verfahren vorstellen, mit dem die Abhängigkeiten aus Objektzugriffen berechnet werden können [1].

Für parallele Programme kann man aber keine optimalen Slices mehr finden, wie es für prozedurale Programme der Fall ist. Trotzdem lassen sich Näherungen berechnen, die einige nicht-erfüllbare Pfade durch das Programm ausschließen können und die damit den Reengineering-Prozess vereinfachen. Leider gibt es aber bis jetzt noch kein etabliertes Verfahren zur Berechnung der Abhängigkeiten, die durch Nutzung des gemeinsamen Speichers bei der parallelen Ausführung von Threads entstehen (Interferenz-Abhängigkeiten).

2 Interferenz

Interferenz entsteht immer dann, wenn ein Thread eine Zelle des gemeinsamen Speichers modifiziert, die von einem anderen Thread wieder ausgelesen wird. In Java ist dies nur mit globalen Variablen oder Feldern gemeinsamer Objekte möglich. Durch Synchronisation kann es aber vorkommen, dass eine Änderung an einer gemeinsamen Variable keinen direkten Einfluss auf andere Threads hat. Dies ist immer dann der Fall, wenn eine Modifikation nicht das Ende eines atomar ausgeführten Blocks erreicht, weil sie vorher von einer anderen Modifikation ausgelöscht wird. Um zu berechnen, welche Teile eines Threads atomar ausgeführt werden und welche mit anderen Statements aus nebenläufigen Threads parallel ausgeführt werden könnten, gibt es mehrere Verfahren.

3 Parallelitätsanalyse

Das beste bis dato veröffentlichte Verfahren zur Berechnung welche Teile der Threads wirklich parallel ausgeführt werden können („May Happen in Parallel“, MHP [3]) und damit, welche Teile potentiell Einfluss auf den gemeinsamen Speicher haben können, ist nicht allgemein einsetzbar. Der Grund hierfür ist, dass es mit Inling arbeitet und deswegen Rekursion nicht modelliert werden kann. Vor kurzem wurde zwar eine Variante [2] veröffentlicht, die Rekursion prinzipiell analysieren kann, aber nur, wenn darin keine für die Analyse interessanten Statements (z.B. Synchronisation) auftauchen, so dass man die Rekursion im Prinzip ignorieren kann.

Abbildung 1 zeigt den Flussgraphen eines Beispielprogramms. In Abbildung 2 werden die Ergebnisse der Parallelitäts-Analyse dargestellt. So kann z.B. Knoten 11 nur während des Wartens in Knoten 4 ausgeführt werden.

Unser Ansatz kombiniert MHP mit den Erkenntnissen aus der Pointer-Analyse, die auch nur intra-prozedural definiert wurde und durch Cloning kontext-sensitiv gemacht werden kann. Dazu wurde das Tool *bddbddb* verwendet, das Pointer-Analyse mit Hilfe von BDDs (Binary Decision Diagram) berechnet. BDDs wurden bisher vor allem beim Model Checking verwendet, um die Explosion des Zustandsraumes besser im Speicher eindämmen zu können. In letzter Zeit finden sie aber immer mehr Anhänger in der Programmanalyse-Community.

Durch diesen Ansatz wird das Problem der Rekursion bereinigt. Die Grundlage hierfür bildet die Fähigkeit des Algorithmus, Schleifen analysieren zu können. Rekursion kann aber als eine Schleife mit zusätzlichem Stack für die Variablen implementiert werden. In unserem Fall werden die Variablenwerte dadurch konservativ approximiert. Die Approximation ist aber sowieso nötig, weil context- und synchronisationssensitve Analysen unberechenbar sind. Diese Approximation findet schon in der Pointer-Analyse statt, wo Rekursion auch als eine zyklische Abhängigkeit modelliert wird. Es muss auf die konsistente Modellierung von Pointer-Analyse und MHP Analyse geachtet werden.

4 Eigener Beitrag

Da MHP Analyse ursprünglich für Ada entworfen wurde, fehlen im ursprünglichen Entwurf einige Modellierungen von Java-Eigenheiten. So wird gefordert, dass Aliasing durch Cloning von Code-Teilen eliminiert wurde, bevor der MHP Algorithmus ausgeführt werden kann. Leider ist Aliasing für Java nicht entscheidbar, wodurch die Elimination durch Cloning nicht durchführbar ist. Cloning ist aber auch nicht sinnvoll, da davon auszugehen ist, dass dadurch der zugrunde liegende Graph explodieren würde. Unsere Analyse verwendet die Ergebnisse einer kontextsensitiven Pointer-Analyse um festzustellen, ob zwei Variablen potentielle Aliase sind. Diese Information genügt an den meisten Stellen der Analyse.

Allerdings gibt es auch Teile der Analyse, die mehr als nur (may-) Aliasing Information (also ob zwei Variablen auf den selben Speicher zeigen *können*) benötigen. Der Algorithmus ist ein Standard Datenfluss-Algorithmus mit *gen*– und *kill*–Mengen. Wie immer genügt bei den *gen*–Mengen may-Aliasing, während bei den *kill*–Mengen must-Aliasing Information nötig ist (d.h. ob zwei Variablen immer auf den selben Speicher zeigen.) Im Gegensatz zur Datenabhängigkeitsanalyse, wo die Vernachlässigung der *kill*–Mengen ca. 5% Genauigkeitsverlust bedeutet, zieht die MHP-Analyse den Großteil ihrer Genauigkeit aus den *kill*–Mengen. Schließlich wird in Java die Synchronisation nur ausgelöst, wenn das Lock-Objekt wirklich das gleiche ist, weil dann dessen Monitor nur von einem Thread betreten werden kann. Kann es aber sein, dass das Lock-Objekt nicht dasselbe ist, wird der Code unter Umständen parallel ausgeführt, was dann als die konservative Approximation generell angenommen werden muss. In diesem Fall kann also der Block nicht als atomar betrachtet werden, so dass keine Modifikationen von nachfolgenden ausgelöscht werden.

5 Interprozedurales Must-Aliasing

Zu must-Aliasing allgemein und zu interprozeduralem im Speziellen gibt es praktisch keine Literatur. Wir folgen deswegen einem Vorschlag von Li et al., die erkannten, dass must-Aliasing vorliegt, wenn beide Variablen auf nur eine Objekt-Allokationsstelle (`new`-Aufruf) zeigen, die im ganzen Programm nur ein einziges Mal aufgerufen werden kann. Dies ist unter anderem der Fall, wenn die Allokation nicht in einer Schleife oder einer Rekursion stattfindet. Um dies jedoch prüfen zu können, muss der gesamte Call-Graph auf Rekursionen untersucht werden.

6 Ausblick

Dieses Verfahren ermöglicht die Anwendung des MHP Algorithmus auf rekursive Java Programme. Das Framework *bddbddb* erlaubt dabei die Formulierung in einer Hochsprache (datalog). Die Ergebnisse aus der Analyse fließen direkt in die Berechnung der

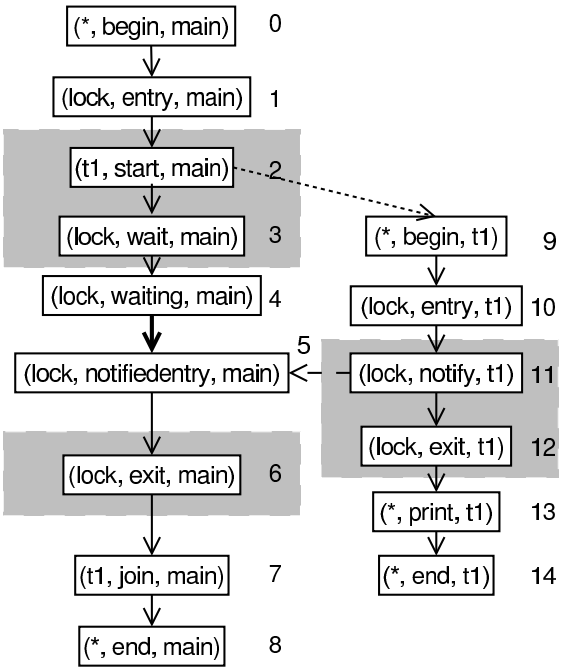

Abbildung 1: Flussgraph für die Parallitätsanalyse

```
wait(3):    begin(9), entry(10)
waiting(4): begin(9), entry(10), notify(11)
n.-entry(5):exit(12), print(13), end(14)
exit(6):    print(13), end(14)
join(7):    print(13), end(14)

begin(9):  wait(3),waiting(4)
entry(10): wait(3),waiting(4)
notify(11):waiting(4)
exit(12):  notifiedentry(5)
print(13): notifiedentry(5), exit(6), join(7)
end(14):   notifiedentry(5), exit(6), join(7)
```

Abbildung 2: Ergebnisse der Parallelitätsanalyse

Interferenz-Abhängigkeiten für Program Slicing ein, worauf zahlreiche Reengineering Verfahren aufbauen, um die Semantikerhaltung der Methode zu erreichen.

Literatur

[1] C. Hammer and G. Snelting. An improved slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop PASTE'04*, pages 17–22. ACM Press, 2004.

[2] L. Li and C. Verbrugge. A practical MHP information analysis for concurrent Java programs. In *Proceedings of LCPC'04*, LNCS. Springer Verlag, Sept. 2004.

[3] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent java programs. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 338–354. Springer-Verlag, 1999.