Datenflussanalyse als Modelchecking im jABC

Anna-Lena Lamprecht[1], Tiziana Margaria[1], Bernhard Steffen[2]

[1]Service Engineering for Distributed Systems, Universität Göttingen (Germany), alamprec@stud.informatik.uni-goettingen.de, margaria@cs.uni-goettingen.de
[2]Chair of Programming Systems, Universität Dortmund (Germany), steffen@cs.uni-dortmund.de

Abstract: Dieses Papier beschreibt wie das jABC (eine generische Umgebung für bibliotheksbasierte Programmentwicklung) zusammen mit zwei seiner Plugins (der Modelchecker und ein Flussgraph-Konverter) ein Framework, DFA-MC, für intraprozedurale Datenflussanalyse als Modelchecking bildet. Basierend auf Funktionalitäten, die von der Programmanalyseplattform Soot bereitgestellt werden, generiert der Konverter Graphstrukturen aus Java-Klassen. Die Datenflussanalysen werden dann als Formeln im modalen μ-Kalkül ausgedrückt. Die Analyse selbst wird ausgeführt, indem der Modelchecker die Gültigkeit der Formel auf dem Flussgraphen überprüft.

1 Datenflussanalyse als Modelchecking

Statische Programmanalyse [NNH99, Hec77, AU79] hat zum Ziel, Informationen über das Laufzeitverhalten von Programmen anhand des Quelltextes möglichst genau zu bestimmen. Sie besteht aus zwei Grundschritten: Kontrollflussanalyse (hauptsächlich dazu benutzt, einen Kontrollflussgraphen zu generieren, der im nächsten Schritt gebraucht wird) und Datenflussanalyse (DFA), zum Sammeln von Infomationen über das Progamm, die beispielsweise für Optimierungen nützlich sein könnten. Klassische Datenflussanalysen verwenden iterative Algorithmen, die eine bestimmte Eigenschaft für ein gegebenes Programm berechnen und wie folgt charakterisiert werden können [Ste93, Ste91]:

DFA-Algorithmus für eine Eigenschaft: Programm $\rightarrow$ erfüllende Programmpunkte

Modelchecking [CGP01, MOSS99], eine Technik zur automatischen Ermittlung der Zustände in einem endlichen System, die eine bestimmte modale oder temporale Formel erfüllen, kann mit geeigneten Eingaben als Datenflussanalyse genutzt werden [Ste93, Ste91]:

Modelchecker: modale Formel $\times$ Modell $\rightarrow$ erfüllende Zustände

Somit brauchen wir, falls wir einen Modelchecker zur Verfügung haben und eine neue Programmeigenschaft prüfen wollen, nur eine neue modale Formel - und keinen neuen Analysealgorithmus.

Die oben beschriebene Verbindung zwischen Datenflussanalyse und Modelchecking impliziert, was für das Datenflussanalyse als Modelchecking Framework DFA-MC benötigt wird [Ste91, Ste93]: anstelle von Programmen brauchen wir Programmmodelle (siehe Abschn. 1.1), und anstelle von DFA-Algorithmen für verschiedene Eigenschaften brauchen wir einen Modelchecker und verschiedene modale Formeln (siehe Abschn. 1.2). Abschn. 2 beschreibt, wie das DFA-MC innerhalb der jABC-Umgebung umgesetzt wird und Abschn. 3, wie Analysen ausgeführt werden.

1.1 Von Programmen zu Programmmodellen

Geringfügig veränderte Kripke-Transitionssysteme [MOSS99] sind gut geeignet, um sequentielle, iterative Programme für den Zweck der Datenflussanalyse zu modellieren, da sie die implizierten Prädikate prägnant darstellen können [Ste91, Ste93, SS98]. Zwei Varianten sind in unserem Framework verfügbar. Abschnitt 3 verwendet die sogenannten *Precondiction*-Modelle, bei denen die Anweisungen von den Knoten hinunter in die ausgehenden Kanten verschoben werden.

Formal ist ein Precondition-Programmmodell P ein Kripke-Transitionssystem $KTS = (S, Act, \rightarrow, B, \lambda)$, wobei

1. S eine endliche Menge von Knoten oder Programmzuständen,
2. Act eine Menge von Aktionen (möglichen Statements der Programmiersprache),
3. $\rightarrow \subseteq S \times Act \times S$ eine Menge von bezeichneten Transitionen, die den Kontrollfluss von P beschreiben,
4. B eine Menge von atomaren Propositionen und
5. λ eine Funktion $\lambda : S \rightarrow 2^B$, die die Zustände mit Untermengen von B beschriftet

ist. Unser Framework erzeugt seine Modelle direkt aus Kontrollflussgraphen.

1.2 Von DFA-Gleichungen zu modalen Spezifikationen

Obwohl wir die Eigenschaften in Logiken wie CTL [Eme90] ausdrücken können, arbeitet unser Spiel-basierter Modelchecker [MMO-Yoo] intern mit μ-Kalkül-Formeln gemäß der folgenden BNF:

$$\Phi \quad ::= \quad true \mid p \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \overline{[a]}\Phi \mid \mu X.\Phi$$

Hierbei bezeichnen p eine atomare Proposition, a ein Element der Menge aller Aktionen *Act*, X eine Variable aus *Var*, und $[a]$ die Menge aller a-Nachfolger eines Zustands. Der Fixpunktoperator μ bezeichnet den kleinsten Fixpunkt, und der Überstrich kehrt die Betrachtungsrichtung um, d.h. transformiert die Modalität in eine sogenannte Backward-Modalität. Die Semantik ist wie üblich definiert [CGP01].

2 DFA-MC im jABC Framework

Das Java-basierte *jABC* [Nag05] baut auf den Grundlagen des seit 1993 entwickelten, C++-basierten Agent Building Center (ABC) [Mar03] auf und verbindet es mit neuen Ideen aus dem Bereich der Java-Entwicklung. Charakteristisch für das jABC-Framework ist die Verwendung einer graphischen, High-Level-Programmierschicht, in der hierarchische, gerichtete Graphen von speziellen Komponenten, den Service Independent Building Blocks, konstruiert werden. Diese SIBs repräsentieren eine Einheit von Quellcode, der typischer Weise eine bestimmte dem Nutzer angebotene Funktionalität definiert. Für die Anwendung in DFA-MC werden die SIBs in einer besonders feingranularen Form verwendet: pro Anweisung der Zwischensprache Jimple gibt es einen separaten SIB.

Die folgenden drei jABC-Plugins und -Erweiterungen waren die Grundlage für die Realisierung des DFA-MC-Frameworks.

- der **Modelchecker**, den wir im Folgenden als Black Box betrachten werden, so wie es typische Anwender unserer Methode tun,
- das **UnitGraph2SibGraph** Plugin, das es uns auf Basis der Soot-Funktionalitäten ermöglicht, SIB-Graphen aus Java-Quellcode zu erstellen. Soot [VRCG$^+$99] erstellt Kontrollflussgraphen (CFGs) aus Java-Quellcode. Der CFG und die in seinen Knoten enthaltenen Informationen werden dann verwendet, um SIB-Graphen, das Eingabeformat für den Modelchecker, zu generieren.
- der **Formula Builder**, der die komfortable Spezifikation von temporalen Formeln unterstützt. Genauer gesagt unterstützt er die high-level Spezifikation von temporalen Eigenschaften, welche dann ins μ-Kalkül übersetzt werden.

3 Benutzung des DFA-MC-Werkzeugs

Die modale Spezifikation eines DFA-Problems, z.B. für *liveness* von Variablen:

$$\begin{aligned} isLive(x) &= ESU(\neg isDefined(x), isUsed(x)) \\ &= \mu Z.(isUsed(x) \vee \neg isDefined(x) \wedge \langle\,\rangle Z) \end{aligned}$$

ist der Ausgangspunkt. Dann gehen wir wie folgt vor.

Vorbereiten des Modells : Wir verwenden die in das DFA-MC-Werkzeug integrierte Soot-Funktionalität, um aus Jimple-Programmen Precondition-Programmmodelle in Form eines UnitGraphen zu erzeugen. Dabei werden die Statements in die ausgehenden Kanten der Knoten verschoben (siehe Abb. 1 (links)).

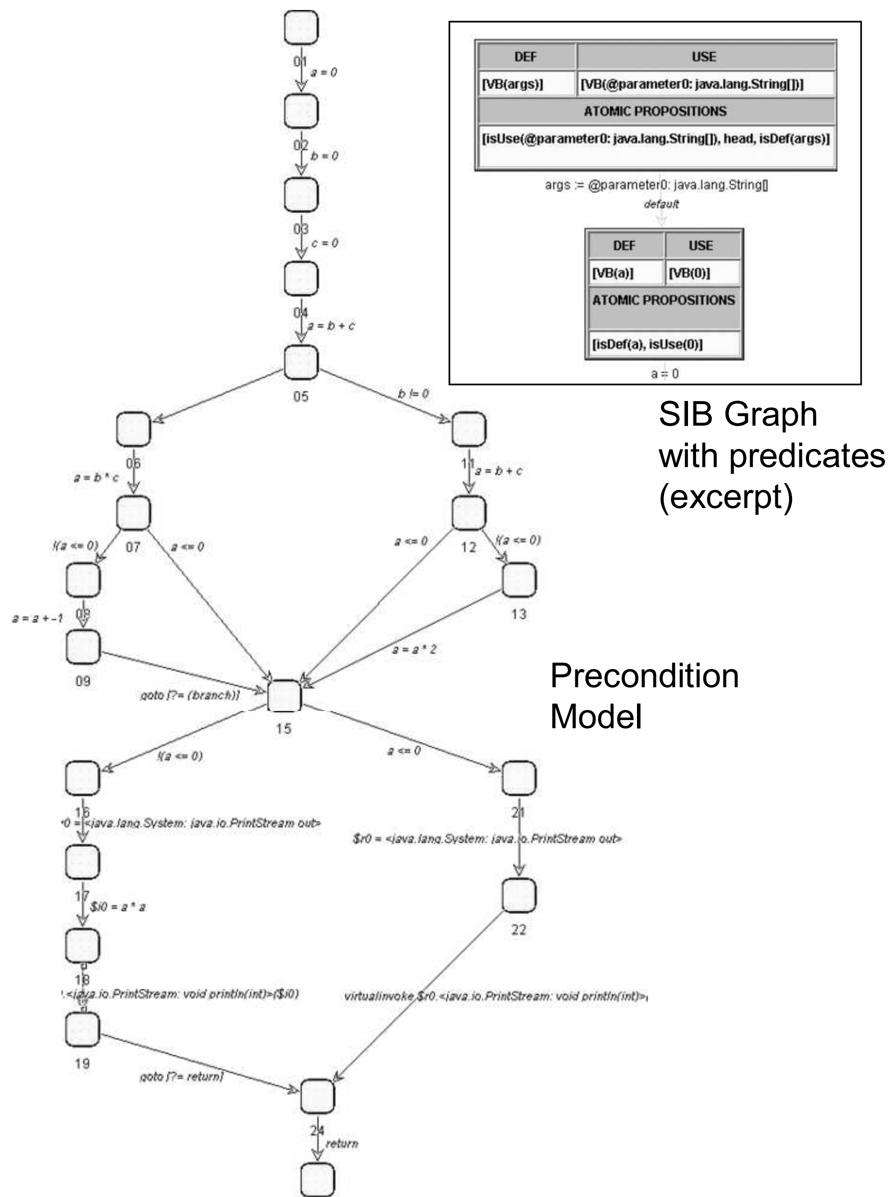

Abbildung 1: Precondition-Programmmodell und Ausschnitt aus dem SIB-Graphen

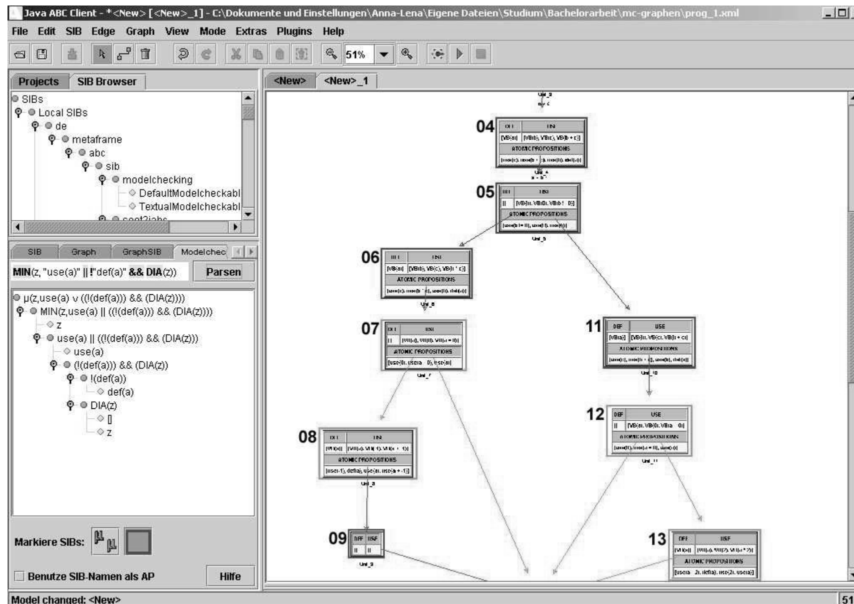

Abbildung 2: Das Analyseergebnis: die Variable *a* ist live in den Knoten 07, 08, 12 und 13, während sie es in 04, 05, 06, 09 und 11 nicht ist.

Anschließend werden die UnitGraphen in jABC-kompatible SIB-Graphen umgewandelt (Abb. 1). Die Vergrößerung auf der rechten Seite zeigt einen im UnitGraphen nicht vorhandenen Einstiegsknoten und den ersten Knoten des Beispiel-UnitGraphen, der nun vom Modelchecker analysiert werden kann. Die Knoten des SIB-Graphen enthalten auch weitere Informationen, wie z.B. die DEF(ined)- und USE(d)-Prädikate, wie sie von Soot mitgeliefert werden, sowie ihre Entsprechungen als atomare Propositionen für den Modelchecker.

Ausführen der Analyse :

1. Laden des *Modelchecking-Plugins* in das jABC,
2. Laden des gerade vorbereiteten *Programmmodells*,
3. Eingabe der zur gewünschten Analyse gehörigen *Formel* und
4. *Starten* des Modelcheckers.

Abb. 2 zeigt, wie die Modelchecking-Funktionalität innerhalb der jABC-Taxonomie erscheint (links) und wie die der gewählten DFA entsprechende μ-Kalkül-Formel im Modelchecking-Fenster (links mittig) in ihre Teilformeln zerlegt wird (links unten). Auf der rechten Seite kennzeichnen die grünen Regionen (hier die Knoten 07, 08, 12 und 13) im Programmmodell die Zustände, in welchen die Analyse „true“ ergibt (in diesem Beispiel sind

das die Bereiche, in denen die Variable a live ist), während die übrigen Regionen, in denen die Formel nicht erfüllt ist, in rot gehalten sind (hier die Knoten 04, 05, 06, 09 und 11).

4 Fazit

Wir haben unser Framework zur intraprozeduralen Datenflussanalyse von Java-Programmen mittels Modelchecking dargestellt. Charakteristisch für unseren Ansatz war die zugrundeliegende Framework-Architektur des jABC, die es uns ermöglichte, die benötigte Funktionalität modular zu realisieren.

Zurzeit muss noch jede Formel einzeln geprüft werden (inkl. dem Anpassen der Parameter per Hand), was den Analyseprozess recht mühsam macht. Wir planen daher, eine Bitvektor-Funktionalität für das Modelchecking-Plugin zu entwickeln, und wir überlegen, den Modelchecker dahingehend zu erweitern, dass er mit Pushdown-Systemen umgehen kann, was uns direkt zu interprozeduralen Analysen befähigen würde.

Danksagungen Vielen Dank an Marco Bakera, Clemens Renner und Marc Njoku für Programmierung und technische Unterstützung.

Literatur

[AU79] A. Aho und J. Ullman. *Principles of Compiler Design.*, Jgg. 3. Addison-Wesley, 1979.

[CGP01] E. Clarke, O. Grumberg und D. Peled. *Model Checking.*, Jgg. 3. The MIT Press, 2001.

[Eme90] E. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, Seiten 995–1072. 1990.

[Hec77] M. Hecht. *Flow Analysis of Computer Programs (Programming Languages Ser. Volume5).* Elsevier Science Ltd, 1977.

[Mar03] T. Margaria. Components, Features, and Agents in the ABC. In *Objects, Agents, and Features*, Vol. 2975 of *LNCS*, Seiten 154–174. Springer, 2003.

[MMO-Yoo] Markus Müller-Olm, Haiseung Yoo: MetaGame: An Animation Tool for Model-Checking Games. Proc. TACAS 2004, LNCS N.2988, Springer 2004, pp. 163-167

[MOSS99] M. Müller-Olm, D. Schmidt und B. Steffen. Model-Checking: A Tutorial Introduction. In Proc. *SAS'99*, Jgg. 1694 of *LNCS*, Seiten 330–354. Springer, 1999.

[Nag05] R. Nagel. Java ABC Framework. http://jabc.cs.uni-dortmund.de, July 2005.

[NNH99] F. Nielson, H. Nielson, C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[SS98] D. Schmidt und B. Steffen. Program Analysis *as* Model Checking of Abstract Interpretations. Proc. *SAS*, Vol. 1503 of *LNCS*, Seiten 351–380. Springer, 1998.

[Ste91] B. Steffen. Data Flow Analysis as Model Checking. In Proc. *TACS*, Vol. 526 of *LNCS*, Seiten 346–365. Springer, 1991.

[Ste93] B. Steffen. Generating Data Flow Analysis Algorithms from Modal Specifications. *Sci. Comput. Program.*, 21(2):115–139, 1993.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam und V. Sundaresan. Soot - a Java bytecode optimization framework. In S. MacKay und J. Johnson, Hrsg., *CASCON*, Seite 13. IBM, 1999.