Temporale Verifikation mit Transitionsinvarianten

Andrey Rybalchenko

Max-Planck-Institut für Informatik
Ecole Polytechnique Fédérale de Lausanne
andrey.rybalchenko@epfl.ch

Abstract: Temporale Verifikation von Programmen befasst sich mit der Erstellung von Korrektheitsaussagen über Programmausführungen. Dabei besteht die Hauptherausforderung in der Synthese von geeigneten Hilfsaussagen, welche die Gültigkeit einer temporalen Eigenschaft implizieren. Es existieren bereits Verifikationswerkzeuge, die Sicherheitseigenschaften, d.h. die Abwesenheit von unerwünschten Ereignissen, automatisch nachweisen können. Diese Werkzeuge verwenden Abstraktion, um die Programmkomplexität zu bewältigen. Für Lebendigkeitseigenschaften, die das Auftreten von erwünschten Ereignissen garantieren, gab es bislang kein automatisches Verifikationswerkzeug, da die notwendigen Hilfsaussagen (Ranking-Funktionen) schwer zu berechnen sind und die klassische Abstraktion für die Lebendigkeitseigenschaften nicht anwendbar ist. Transitionsinvarianten stellen eine neue Art von Hilfsaussagen für die Verifikation von Lebendigkeitseigenschaften dar. Sie sind einfacher zu finden als Ranking-Funktionen und können mit Abstraktion kombiniert werden. Dadurch schaffen Transitionsinvarianten eine Grundlage für die Konstruktion von automatischen Werkzeugen, die ihre Praxistauglichkeit erfolgreich nachwies.

1 Einleitung

Sicherheit und Lebendigkeit Jeder PC-Benutzer hat es schon erlebt: ein Programm geht plötzlich schief, eine Fehlermeldung wird ausgegeben und das Programm muss geschlossen werden. Manchmal erscheint überhaupt keine Fehlermeldung, das Programm reagiert nicht auf die Benutzerkommandos und man sieht eine sich drehende Sanduhr.

Für solche Fehlverhalten gibt es unterschiedliche Ursachen, die man in zwei Gruppen zusammenfassen kann. In die erste Gruppe gehören Fehler, die durch eine unerlaubte Operation verursacht werden. Die typischen Beispiele sind Division durch Null, Überlauf bei arithmetischen Operationen und Zugriff auf Speichergebiete außerhalb der vorgesehenen Grenzen. Man spricht dabei von Sicherheitseigenschaften, die das Auftreten von unsicheren Ereignissen ausschließen. In der Regel können die Verletzungen von Sicherheitseigenschaften während der Programmausführung erkannt und dem Benutzer gemeldet werden.

Die zweite Gruppe besteht aus Fehlern, die durch die Abwesenheit von bestimmen erwünschten Ereignissen verursacht werden. Zu solchen Ereignissen zählen z.B. die Ausführung einer gestellten Anfrage und die Terminierung einer Berechnung. Man spricht dabei von Lebendigkeitseigenschaften, die das Auftreten von erwünschten Ereignissen

garantieren. In der Regel kann man auf die Verletzungen von Lebendigkeitseigenschaften nicht durch eine Fehlermeldung hinweisen, da es zu jedem Zeitpunkt unbekannt ist, ob die gewünschten Ereignisse eventuell noch später auftreten könnten.

Um Fehler bei der Programmausführung zu vermeiden, versucht man, die möglichen Fehlerursachen zu finden und zu beheben, bevor das Programm in Betrieb genommen wird. Hier kommt das Testen, welches die am meisten benutzte Methode ist, zum Einsatz. Dabei wird das Programm auf einer großen Menge von möglichen unterschiedlichen Eingaben ausgeführt und sein Verhalten beobachtet. So können mehrere Fehler rechtzeitig entdeckt werden. Leider kann man durch das Testen keine hundertprozentige Garantie für die Fehlerfreiheit erzielen. Zum einen kann die Anzahl der Kombinationen der Eingabewerte sehr groß sein, so dass deren vollständige Aufzählung zu viel Zeit in Anspruch nehmen würde. Zum anderen sieht man sich beim Testen von Lebendigkeitseigenschaften mit der Ungewissheit konfrontiert, wie lange man auf das Auftreten von erwünschten Ereignissen zu warten hat.

Temporale Verifikation Abhilfe soll die so genannte Temporale Verifikation schaffen, die sich mit der automatisierten Überprüfung von Computerprogrammen beschäftigt. Das Ziel besteht darin, festzustellen, ob ein Programm eine gegebene Eigenschaft erfüllt oder verletzt. Dabei werden mathematische und logische Methoden angewandt, um Aussagen über Programmausführungen zu treffen. Man benutzt Formeln, um die Ausführungen abstrakt aber präzise zu beschreiben.[1] Dadurch kann man sehr große, sogar unendliche Mengen von Ausführungen repräsentieren und untersuchen.

Die Forschung auf dem Gebiet der temporalen Verifikation fing gleichzeitig mit der Erfindung von Computerprogrammen an. Bedeutende Fortschritte wurden erzielt, sowohl in der Theorie als auch in der Praxis. Es existieren bereits automatische Werkzeuge, die bei der Verifikation von Sicherheitseigenschaften erfolgreich eingesetzt werden können. Dazu zählen u. a. SLAM [BMMR01] (Microsoft Research, Redmond), BLAST [HJMM04] (UC Berkeley) und MAGIC [CCG+03] (CMU, Pittsburgh). Ein Hauptmerkmal solcher Werkzeuge ist die Benutzung von Abstraktion [CC77], um die Komplexität der Verifikationsaufgaben zu bewältigen. Dabei werden die unwesentlichen Einzelheiten des zu verifizierenden Programms weggelassen. Die Werkzeuge können den notwendigen Abstraktionsgrad automatisch ableiten. Allerdings konnten diese Werkzeuge bislang nur zur Überprüfung von Sicherheitseigenschaften eingesetzt werden. Der Grund hierfür ist der Verlust von Lebendigkeitseigenschaften durch die verwendete Abstraktionsart. Außerdem verlangen die Lebendigkeitsbeweise so genannte Ranking-Funktionen, die abschätzen, wann das nächste Auftreten von erwünschten Ereignissen stattfindet. Die Berechnung solcher Funktionen für komplexe Programme ist wegen der damit verbundenen hohen Schwierigkeiten nur begrenzt möglich. Die existierenden Lebendigkeitswerkzeuge für spezielle Klassen von Programmen und Rechenmethoden, z. B. Termersetzungssysteme [GTSKF04], logische und funktionale Programmierung [CT99, LJBA01, LSS97] und imperative Programme mit spezifischen arithmetischen Operationen [BMS05, CS02, Cou05a] bieten keine Skalierbarkeit und/oder keine ausreichende Unterstützung der Eigenschaften von praktischen

[1] Vergleiche dies mit einem Physiker, der mit Hilfe von Differentialgleichungen die Natur in der Welt der Mathematik abbildet.

Programmiersprachen.

Trotz der intensiven, über dreißig Jahre andauernden Forschung, blieb das Problem ungelöst, wie man die Abwesenheit von Lebendigkeitsfehlern automatisch nachweisen kann. Insbesondere stellt sich die Frage, wie man die Argumentation über unendliche Programmausführungen, die bei der Verifikation von Lebendigkeitseigenschaften entscheidend ist, mit Abstraktion kombiniert und wie man solche Argumentation auf komplexe Programme anwendet.

Transitionsinvarianten Die Dissertation "Temporale Verifikation mit Transitionsinvarianten" [Ryb04] gibt eine Antwort auf diese Fragen. Sie führt eine neue Art von logischen Hilfsaussagen ein, die als Transitionsinvarianten bezeichnet werden. Der entscheidende Vorteil von Transitionsinvarianten, gegenüber den bereits existierenden Methoden, liegt in der Tauglichkeit für die automatische Synthese unter Verwendung von Abstraktion.

Die Theorie von Transitionsinvarianten stellt sowohl einen formalen Rahmen zur Benutzung dieser Hilfsaussagen bei der temporalen Verifikation [PR04b, PPR05, PR03], als auch die notwendigen Methoden und Algorithmen, die eine Verwirklichung von diesem Rahmen in einem automatischen Werkzeug ermöglichen [PR04a, PR05, CPR05].

Die praktische Anwendbarkeit von Transitionsinvarianten wurde mit Hilfe von zwei automatischen Werkzeugen, ARMC und TERMINATOR, erfolgreich nachgewiesen. ARMC ist ein Forschungsprototyp, der am Max-Planck-Institut für Informatik entwickelt wurde [Ryb06a]. ARMC wird u.a. benutzt, um die Terminierung von C Programmen [Con03] und durch ganzzahlige Zähler-erweiterten endlichen Automaten [IBB+06] zu beweisen. TERMINATOR wurde in Zusammenarbeit mit dem Forschungslabor von Microsoft in Cambridge gebaut [CPR06b] und erfolgreich zur Verifikation von Gerätetreibern aus dem Windows-Betriebssystem eingesetzt [CPR06a].

Im Abschnitt 2 wird ein kurzer Überblick über die Theorie der Transitionsinvarianten und deren Einsatz bei der Verifikation von Lebendigkeitseigenschaften geboten. Die Algorithmen für die automatische Synthese von Transitionsinvarianten werden im Abschnitt 3 beschrieben. Das Beispiel im Abschnitt 4 veranschaulicht die Anwendung der vorgestellten Algorithmen auf ein einfaches Programm. Die durch die Forschung an Transitionsinvarianten entstandenen Werkzeuge werden im Abschnitt 5 vorgestellt.

Danksagung Ich danke Prof. Dr. Andreas Podelski für seine hervorragende Betreuung, wertvolle Hinweise und spannende Diskussionen.

2 Theorie der Transitionsinvarianten

Die allgemeine Vorgehensweise zur Verifikation von temporalen Eigenschaften von Programmen besteht darin, die Argumentation über die Programmausführungen (Sequenzen von Programmzuständen) auf die Argumentation über Hilfsaussagen, wie z.B. Invarianten und Ranking-Funktionen, zu reduzieren. Invarianten beschreiben Mengen von erreichba-

ren Programmzuständen und werden zur Verifikation von Sicherheitseigenschaften eingesetzt. Ranking-Funktionen bilden Programmzustände in eine wohlfundierte Domäne ab, so dass bei jedem Ausführungsschritt des Programms der Wert der Ranking-Funktion strikt abnimmt. Sie werden zur Verifikation von Lebendigkeitseigenschaften benutzt. Die größte Herausforderung in der Automatisierung der Verifikationmethoden liegt in der automatischen Synthese dieser Hilfsaussagen. Die Theorie der abstrakten Interpretation [CC77] wird eingesetzt, um die Komplexität der Programme zu bewältigen und dadurch die Invariantensynthese zu erleichtern. Leider bleiben bei der Abstraktion keine Lebendigkeitseigenschaften erhalten, so dass die Ranking-Funktionssynthese ohne Abstraktion betrieben werden muss und schwer automatisierbar ist.

Transitionsinvarianten stellen eine neue Art von Hilfsaussagen dar, die für die Lebendigkeitsbeweise geeignet sind und sich mit Hilfe von abstrakter Interpretation synthetisieren lassen [PR04b]. Eine Menge von Paaren der Programmzustände ist eine Transitionsinvariante, falls sie den transitiven Abschluss der Übergangsrelation des Programms enthält. Eine Transitionsinvariante heißt disjunktiv wohlfundiert, falls sie als eine endliche Vereinigung von wohlfundierten Relationen darstellbar ist. (Eine Relation ist wohlfundiert, falls man keine unendliche Kette von Elementen bilden kann, in der die benachbarten Elemente in der Relation enthalten sind.)

Wir charakterisieren die Gültigkeit einer Lebendigkeitseigenschaft durch die Existenz einer disjunktiv wohlfundierten Transitionsinvariante. Das hierfür angeführte Korrektheitsargument benutzt das Ramsey-Theorem für unendliche Graphen [Ram30] und basiert auf dem folgenden Satz [PR04b]: Eine Übergangsrelation ist genau dann wohlfundiert, wenn sie eine disjunktiv wohlfundierte Transitionsinvariante besitzt. Dieser Satz erlaubt eine Vereinigung von mehreren wohlfundierten Relationen zu einem Terminierungsargument: der Transitionsinvariante. Die einzelnen Relationen enthalten Teile des transitiven Abschlusses der Übergangsrelation und können unabhängig voneinander berechnet werden. Wir erkennen, dass keine ähnliche Zusammensetzung von Ranking-Funktionen zu einem Terminierungsargument führen kann, da die Vereinigung von wohlfundierten Relationen im Allgemeinen nicht wohlfundiert ist.

Wir führen ein Induktionsprinzip ein, das uns erlaubt, eine gegebene Relation als eine Transitionsinvariante zu identifizieren. Eine Transitionsinvariante ist induktiv, falls sie die Übergangsrelation des Programms enthält und unter der relationalen Komposition mit der Übergangsrelation abgeschlossen ist. Die disjunktive Wohlfundiertheit und das Induktionsprinzip stellen die Basis unserer Beweisregel für Lebendigkeitseigenschaften dar.

Die meisten Lebendigkeitseigenschaften (nebenläufiger) Programme gelten nur unter bestimmten Fairness-Anforderungen, wie z.B. den Anforderungen, dass jeder Prozess irgendwann ausgeführt wird oder ein Kommunikationskanal irgendwann erfolgreich eine Nachricht übermittelt. Fairness-Anforderungen werden in der Regel als Bedingungen an das Vorkommen von Programmübergängen oder -zuständen in Programmausführungen spezifiziert. Es wird verlangt, dass z.B. in jeder unendlichen Ausführung jeder Programmübergang unendlich oft genommen wird, oder dass keine Ausführung eine bestimmte Zustandsmenge je verlässt. Das Einbeziehen von Fairness-Anforderungen erschwert die Verifikation, da das Auftreten von unterschiedlichen Mengen bestimmter Programmzustände berücksichtigt werden muss. Dies führt zu komplizierten Ranking-

Funktionen, die synthetisiert werden müssen.

Um eine direkte Berücksichtigung von den an die Programmzustände gestellten Fairness-Anforderungen zu ermöglichen, führen wir so genannte markierte Transitionsinvarianten ein, die eine Erweiterung von Transitionsinvarianten darstellt [PPR05]. Die Mengen von Markierungen, die an die einzelnen Teilrelationen einer markierten Transitionsinvariante angehängt werden, beinhalten die Indizes der erfüllten Fairness-Anforderungen. Wir schwächen das Kriterium der disjunktiven Wohlfundiertheit ab, indem wir die Wohlfundiertheit nur für diejenigen Relationen einer endlichen Vereinigung voraussetzen, deren Mengen von Markierungen die Indizes aller Fairness-Anforderungen enthalten. Wir entwickeln eine entsprechende Beweisregel und automatisieren diese mit Hilfe der abstrakten Interpretation.

Das Induktionsprinzip erlaubt die Darstellung des Syntheseproblems von (markierten) Transitionsinvarianten als die Berechnung von kleinsten Fixpunkten einer Funktion, die durch den relationalen Kompositionsoperator definiert wird. Diese Darstellung führt zur Anwendbarkeit der Methoden und Algorithmen zur abstrakten Fixpunktberechnung, die bereits maßgeblich zum Erfolg der Werkzeuge für die Sicherheitseigenschaften beitrugen.

3 Automatische Synthese

Transitionsinvarianten können durch Berechnungsmethoden für kleinste Fixpunkte konstruiert werden. Seit [CC77] sind solche Berechnungen ein Bestandteil von automatischen Methoden zur Verifikation von Sicherheitseigenschaften. Da eine Fixpunktberechnung für Programme mit unendlichen Zustandsräumen im Allgemeinen nicht terminiert und für endliche aber sehr große Zustandsmengen sehr aufwendig ist, ist Abstraktion notwendig, um die Berechnung effektiv und effizient durchführen zu können. Wir stellen eine geeignete Abstraktionsart vor [PR05], die eine Verallgemeinerung der klassischen Abstraktion darstellt. Die Bausteine hierfür werden automatisch berechnet [CPR05]. Um die Gültigkeit der Lebendigkeitseigenschaft anhand einer gegebenen Transitionsinvariante zu überprüfen, müssen Wohlfundiertheitstests auf die Komponenten der Transitionsinvariante angewandt werden. Wir entwickeln einen effizienten Algorithmus für diese Aufgabe [PR04a].

Abstraktion durch Transitionsprädikate Die klassischen Abstraktionsmethoden, die durch eine Äquivalenzrelation mit endlichem Index über Programmzustände definiert werden, erzeugen ein abstraktes Programm, in welchem die Übergänge zwischen den Äquivalenzklassen stattfinden. Die Überprüfung der Eigenschaft findet auf dem abstrakten Programm statt. Leider gehen bei einer solchen Abstraktion die Lebendigkeitseigenschaften verloren. Der Grund dafür liegt in den unberechtigten Schleifen, die in dem abstrakten Programm entstehen und durch keine Verfeinerung entfernt werden können.

Wir schlagen eine neue Abstraktionsart vor, die das Entstehen von unberechtigten Schleifen zu vermeiden imstande ist. Dafür führen wir zwei neue Begriffe ein: “Abstraktion durch Transitionsprädikate” und “abstraktes Transitionsprogramm”. Wir benutzen diese

Begriffe, um eine automatische Methode für den Beweis der Terminierung unter Fairness-Anforderungen zu entwickeln [PR05]. Transitionsprädikate sind binäre Relationen über Programmzustände. Abstrakte Transitionsprogramme sind endliche gerichtete Graphen, deren Knoten durch Transitionsprädikate und deren Kanten durch Programmübergänge markiert sind. Wir geben einen Algorithmus zur automatischen Synthese eines abstrakten Transitionsprogramms für ein gegebenes Programm an. Wir argumentieren über Terminierung anhand der Knotenmarkierung. Fairness-Anforderungen werden mit Hilfe der Kantenmarkierung berücksichtigt.

Abstraktionsverfeinerung Unsere Abstraktion ist definiert durch eine Menge von Transitionsprädikaten. Diese Menge wird benutzt, um eine Transitionsinvariante durch eine Fixpunktberechnung zu konstruieren. Falls die Menge der verwendeten Transitionsprädikate nicht adäquat ist, wird das Ergebnis nicht disjunktiv wohlfundiert sein, d.h. einige Komponenten sind keine wohlfundierten Relationen. Dabei werden die Pfade durch den Kontrollflussgraphen in Betracht gezogen, die zur Verletzung der disjunktiven Wohlfundiertheit beitrugen. Die Abstraktion wird durch zusätzliche Transitionsprädikate verfeinert, falls die Übergangsrelationen dieser Pfade wohlfundiert sind. Dabei dienen die Ranking-Funktionen als Basis für die zusätzlichen Transitionsprädikate [CPR05]. Andernfalls, d.h. wenn es einen Pfad mit der nicht wohlfundierten Übergangsrelation gibt, stellt dieser Pfad den Nachweis der Eigenschaftsverletzung dar und kann für die eventuelle Fehlerbeseitigung benutzt werden.

Synthese von Ranking-Funktionen Wir stellen Teilrelationen einer (markierten) Transitionsinvariante und die Knotenmarkierungen eines abstrakten Transitionsprogramms mit Hilfe von ‘single while’ Programmen dar. Diese Programme bestehen aus einer While-Schleife, die nur (möglicherweise nichtdeterministische) Zuweisungen enthält. Die meisten für die Praxis relevante ‘single while’ Programmen enthalten arithmetische Operationen. Wir entwickeln einen Algorithmus zur Synthese linearer Ranking-Funktionen für lineare ‘single while’ Programme [PR04a]. Damit automatisieren wir die Wohlfundiertheitsbeweise, die bei der Verwendung von Transitionsinvarianten durchgeführt werden müssen.

4 Beispiel

Wir veranschaulichen die Theorie der Transitionsinvarianten mit Hilfe des einfachen Programms AUSWAHL und beweisen, dass AUSWAHL terminierend ist. Das Programm ist nichtdeterministisch. Bei jedem Schleifendurchlauf kann entweder die erste oder die zweite Anweisung ausgeführt werden. Die Übergangsrelation R wird durch die Disjunktion $(x' = x - 1 \wedge y' = y) \vee (x' = x \wedge y' = y - 1)$ dargestellt.[2] Im

$$\textbf{while } x \geq 0 \wedge y \geq 0 \textbf{ do} \left[\begin{array}{c} x := x - 1 \\ \textbf{or} \\ y := y - 1 \end{array}\right]$$

Programm AUSWAHL.

[2] Wir lassen die Formeln $x \geq 0$ und $y \geq 0$ während des gesamten Beispiels aus Platzgründen weg.

Folgenden simulieren wir vereinfacht die mögliche Vorgehensweise eines Werkzeugs, das Transitionsinvarianten durch Abstraktion mit Transitionsprädikaten berechnet.

Am Anfang ist die Menge der Transitionsprädikate $\mathcal{P}$ leer. Der mit dieser Menge berechnete abstrakte Fixpunkt ist die leere Konjunktion $\wedge\emptyset$. Sie stellt die Menge aller Paare von Programmzuständen dar und ist nicht disjunktiv wohlfundiert. Wir nehmen an, dass die Anweisung $x := x - 1$ dazu beitrug. Wir berechnen eine Ranking-Funktion r für die entsprechende Relation. Die Ranking-Funktion $r(x, y) = x$ bildet die Programmzustände in die Domäne der natürlichen Zahlen ab. Wir verfeinern die Abstraktion durch das Aufnehmen der Ranking-Relation $x' \leq x - 1$ und ihrer Relaxation $x' \leq x$ in die Transitionsprädikatenmenge $\mathcal{P}$.

Die neue Abstraktion führt zum Fixpunkt, der aus den Relationen $x' \leq x - 1$ und $\wedge\emptyset$ besteht. Die erste Relation ist wohlfundiert. Die zweite, nicht wohlfundierte Relation ist durch die Anweisung $y := y - 1$ entstanden. Die Ausführung des Abstraktionsschrittes fügt die Transitionsprädikate $y' \leq y - 1$ und $y' \leq y$ zu $\mathcal{P}$ hinzu.

Jetzt haben wir $\mathcal{P} = \{x' \leq x - 1, x' \leq x, y' \leq y - 1, y' \leq y\}$. Diese Abstraktion erlaubt uns, einen disjunktiv wohlfundierten Fixpunkt zu berechnen. Die dadurch repräsentierte Transitionsinvariante besteht aus zwei Relationen: $x' \leq x - 1 \wedge y' \leq y$ und $y' \leq y - 1 \wedge x' \leq x$. Die erste (bzw. zweite) Relation enthält alle Paare von Zuständen (s, s') aus dem transitiven Abschluss R^+, so dass s in s' mit mindestens einer Anwendung der ersten (bzw. zweiten) Anweisung übergeht.

5 Werkzeuge

Die Theorie der Transitionsinvarianten und die oben erwähnten Algorithmen fuhren zur Entstehung der nachfolgenden Werkzeuge zur automatischen Verifikation von Lebendigkeitseigenschaften.

ARMC ist ein automatisches Werkzeug zur Verifikation von Sicherheits- und Lebendigkeitseigenschaften von Programmen [Ryb06a]. ARMC steht für "Abstraction Refinement Model Checker". Es ist ein Forschungsprototyp. ARMC wurde in Sicstus Prolog unter Verwendung eines Constraint-Solvers für die lineare Arithmetik über die rationale Zahlen implementiert. ARMC akzeptiert als Eingabe Kontrollflussgraphen, die mit linearen arithmetischen Ausdrücken versehen sind. Dabei unterliegt die Form der dargestellten Übergangsrelation keinen Einschränkungen. Insbesondere dürfen die Zuweisungen nichtdeterministisch sein. Ein weiteres Merkmal von ARMC liegt in der uniformen Implementierung der Prozedur zur Berechnung von kleinsten Fixpunkten für Sicherheits- und Lebendigkeitseigenschaften.

Im Rahmen des Projektes Verisoft [Con03] wird ARMC benutzt, um die Terminierung von C Programmen zu beweisen. Da ARMC allgemeine lineare Übergangsrelationen als Eingabe akzeptiert, wandte man ARMC auf die durch ganzzahlige Zähler-erweiterten endlichen Automaten [IBB$^+$06] an.

TERMINATOR ist ein automatisches Werkzeug zur Verifikation von Lebendigkeitseigenschaften von Systemsoftware [CPR06a, CPR06b]. Es wurde in Zusammenarbeit mit dem Forschungslabor von Microsoft in Cambridge gebaut.

TERMINATOR wurde erfolgreich zur Verifikation von Lebendigkeitseigenschaften von Gerätetreibern aus dem Windows-Betriebssystem benutzt. Gerätetreiber sind spezielle Programme, die den Zugriff auf angeschlossene Computerkomponenten ermöglichen. Dabei ist die Korrektheit von Gerätetreibern unabdingbar für das stabile Verhalten des gesamten Betriebssystems. TERMINATOR konnte erfolgreich Gerätetreiber, die aus 5.000 bis 35.000 Programmzeilen bestehen und in der Programmiersprache C geschrieben sind, verifizieren und in manchen Fällen unbekannte Lebendigkeitsfehler feststellen [CPR06a].

RANKFINDER implementiert den Algorithmus zur Synthese linearer Ranking-Funktionen für 'single while' Programme [PR04a]. Es wurde unter Verwendung eines Constraint-Solvers für die lineare Arithmetik über die rationalen Zahlen implementiert [Ryb06b]. RANKFINDER wird zum Testen der disjunktiven Wohlfundiertheit im TERMINATOR [CPR06a, CPR06b] und seinen Erweiterungen [BCDO06] eingesetzt.

6 Zusammenfassung

Transitionsinvarianten liefern eine neue Basis für die automatisierte Verifikation von Lebendigkeitseigenschaften. Transitionsinvarianten stellen eine neue Art von temporalen Hilfsaussagen dar. Sie sind einfacher zu finden als Ranking-Funktionen und können mit durch abstrakte Fixpunktberechnung konstruiert werden. Dadurch schaffen die Transitionsinvarianten eine Grundlage für die Konstruktion von automatischen Werkzeugen für Lebendigkeitseigenschaften. Wir wiesen die Praxistauglichkeit der Transitionsinvarianten durch die erfolgreichen Werkzeuge ARMC und TERMINATOR nach.

Literatur

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano und Peter O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In Ball und Jones [BJ06]. to appear.

[BJ06] Thomas Ball und Robert Jones, Hrsg. *CAV'2006: 18th Int. Conf. on Computer Aided Verification*, LNCS. Springer, 2006. to appear.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein und Sriram Rajamani. Automatic predicate abstraction of C programs. In *PLDI'2001: ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jgg. 36 of *ACM SIGPLAN Notices*, Seiten 203–213. ACM Press, 2001.

[BMS05] Aaron Bradley, Zohar Manna und Henny Sipma. Termination of polynomial programs. In Cousot [Cou05b], Seiten 113–129.

[CC77] Patrick Cousot und Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'1977: 4th ACM Symp. on Principles of Programming Languages*, Seiten 238–252. ACM Press, 1977.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha und Helmut Veith. Modular verification of software components in C. In *ICSE'2003: 25th Int. Conf. on Software Engineering*, Seiten 385–395. IEEE, 2003.

[Con03] The Verisoft Consortium. The Verisoft project website. `http://www.verisoft.de`, 2003.

[Cou05a] Patrick Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In Cousot [Cou05b], Seiten 1–24.

[Cou05b] Radhia Cousot, Hrsg. *VMCAI'2005: 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, Jgg. 3385 of *LNCS*. Springer, 2005.

[CPR05] Byron Cook, Andreas Podelski und Andrey Rybalchenko. Abstraction refinement for termination. In *SAS'2005: 12th Int. Static Analysis Symp.*, Jgg. 3672 of *LNCS*, Seiten 87–101. Springer, 2005.

[CPR06a] Byron Cook, Andreas Podelski und Andrey Rybalchenko. Termination proofs for systems code. In *PLDI'2006: ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM Press, 2006. to appear.

[CPR06b] Byron Cook, Andreas Podelski und Andrey Rybalchenko. Terminator: Beyond safety (tool description). In Ball und Jones [BJ06]. to appear.

[CS02] Michael Colón und Henny Sipma. Practical methods for proving program termination. In *CAV'2002: 14th Int. Conf. on Computer Aided Verification*, Jgg. 2404 of *LNCS*, Seiten 442–454. Springer, 2002.

[CT99] Michael Codish und Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1), 1999.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp und Stephan Falke. Automated termination proofs with AProVE. In *RTA'2004: 15th Int. Conf. on Rewriting Techniques and Applications*, Jgg. 3091 of *LNCS*, Seiten 210–220. Springer, 2004.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar und Kenneth L. McMillan. Abstractions from proofs. In *POPL'2004: 31st ACM SIGPLAN–SIGACT Symp. on Principles of Programming Languages*, Seiten 232–244. ACM Press, 2004.

[IBB+06] Radu Iosif, Marius Bozga, Ahmed Bouajjani, Peter Habermehl, Pierre Moro und Tomáš Vojnar. Programs with lists are counter automata. In Ball und Jones [BJ06]. to appear.

[LJBA01] Chin Soon Lee, Neil D. Jones und Amir M. Ben-Amram. The size-change principle for program termination. In *POPL'2001: 28th Annual ACM SIGPLAN–SIGACT Symp. on Principles of Programming Languages*, Seiten 81–92. ACM Press, 2001.

[LSS97] Naomi Lindenstrauss, Yehoshua Sagiv und Alexander Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *CAV'1997: 9th Int, Conf. on Computer-Aided Verification*, Jgg. 1254 of *LNCS*, Seiten 444–447. Springer, 1997.

[PPR05] Amir Pnueli, Andreas Podelski und Andrey Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *TACAS'2005: 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Jgg. 3440 of *LNCS*, Seiten 124–139. Springer, 2005.

[PR03] Andreas Podelski und Andrey Rybalchenko. Software model checking of liveness properties via transition invariants. Forschungsbericht 2003-2-004, Max-Planck-Institut für Informatik, 2003.

[PR04a] Andreas Podelski und Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'2004: 5th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, Jgg. 2937 of *LNCS*, Seiten 239–251. Springer, 2004.

[PR04b] Andreas Podelski und Andrey Rybalchenko. Transition invariants. In *LICS'2004: 19th Annual IEEE Symp. on Logic in Computer Science*, Seiten 32–41. IEEE, 2004.

[PR05] Andreas Podelski und Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'2005: 32nd Annual ACM SIGPLAN–SIGACT Symp. on Principles of Programming Languages*, Seiten 132–144. ACM Press, 2005.

[Ram30] Frank P. Ramsey. On a problem of formal logic. In *Proc. London Math. Soc.*, Jgg. 30, Seiten 264–285, 1930.

[Ryb04] Andrey Rybalchenko. *Temporal verification with transition invariants*. Dissertation, Universität des Saarlandes, 2004.

[Ryb06a] Andrey Rybalchenko. The ARMC project web page. `http://www.mpi-sb.mpg.de/~rybal/armc`, 2006.

[Ryb06b] Andrey Rybalchenko. The RankFinder project web page. `http://www.mpi-sb.mpg.de/~rybal/rankfinder`, 2006.

Andrey Rybalchenko ist Forscher am Max-Planck-Institut für Informatik in Saarbrücken und an der Ecole Polytechnique Fédérale in Lausanne. Er studierte (1999–2002) und promovierte (2002–2005) in Informatik an der Universität des Saarlandes. Während des Studiums war er als wissenschaftliche Hilfskraft am Deutschen Forschungszentrum für Künstliche Intelligenz (1999–2002) und am Max-Planck-Institut für Informatik (2002) tätig. Seine Forschungsinteressen umfassen temporale Verifikation reaktiver Systeme, Programmanalyse und -verifikation, abstrakte Interpretation, automatisierte Deduktion und logische Programmierung mit Constraints. Während seines Aufenthaltes am Forschungslabor von Microsoft in Cambridge entwickelte und implementierte er das erste automatische Werkzeug zur Verifikation von Lebendigkeitseigenschaften von Systemsoftware. Er erhielt die Günther-Hotz-Medaille für seine Diplomarbeit und seine Doktorarbeit wurde mit der Note summa cum laude ausgezeichnet.