13. Robustheit

Bouajjani, A., Mohlmann ICALP '11
Bouajjani, Derevenetc, M. ESOP '13.

Behauptung:
- Beim Programmieren denkt man in SC.
- Also sollte jedes Verhalten, das von SC abweicht, als Programmierfehler angesehen werden.

Korrektheits-Formulierung:
- Das Programmverhalten unter TSO sollte mit dem SC-Verhalten übereinstimmen:

$$B_{TSO}(c) = B_{SC}(c).$$

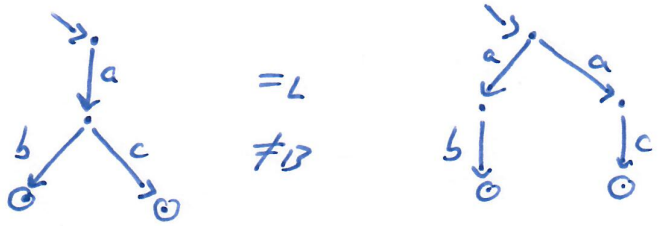
- In dem Fall sagen wir, das Programm ist <u>robust</u> gegenüber (Ausführungen unter) TSO.
- Der Begriff der Robustheit ist <u>abhängig</u> vom Begriff des Verhaltens.

Trade-Off:
- Wir wollen einen <u>schwachen</u> Verhaltensbegriff, der selbst starke Fehlverhalten als SC-Verhalten auffasst.
  Damit maximieren wir den Performancegewinn, den die TSO-Architektur bringt.
- Je schwächer wir allerdings die Korrespondenz zwischen TSO und SC wählen, desto <u>schwer</u> (Komplexität) ist sie algorithmisch zu prüfen.

Als Analogon lässt sich
die Äquivalenz endlicher Automaten betrachten:

Sprachgleichheit ($=_L$): schwach aber PSPACE-vollständig.

Bisimulationsäquivalent ($=_B$): stark aber in PTIME prüfbar.



Optionen: · Zustandsbasierte Robustheit, $Read_{TSO}(C) = Read_{SC}(C)$

Erreichbare Kontrollzustände

↳ sehr gut: sehr schwache Äquivalenzbegriff,
die beliebige Pufferverwendung zulässt,
solange halt keine neuen Zustände hinzukommen.

↳ schlecht: genauso schwer wie Erreichbarkeit unter TSO,
was nicht-primitiv-rekursiv (Atig und Ponk)
(also immerhin entscheidbar) ist → Atig et al. POPL '10.

Bei algorithmischen Betrachtungen
werden Heap und Datendomäne
als endlich angenommen,
sonst ist sofort alles unentscheidbar. (Warum?)

↳ Lamport '79.

· Happens-Before-basierte Robustheit, $Traces_{TSO}(C) = Traces_{SC}(C)$

Definition wir gleich.

↳ gut: immer noch starke Pufferverwendung möglich.

↳ sehr gut: nun PSPACE-vollständig, das ist
die Standardkomplexität für SC-Verifikation.

Ziel: (1) Definiere Happens-Before.

(2) Formuliere das Lokalitätsprinzip
für Happens-Before-basierte Robustheit.
Memberships in PSPACE folgt daraus.

13.1 Happens-Before

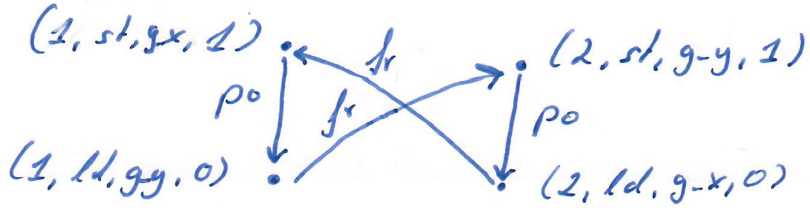
Idee: Die Happens-Before-Relation extrahiert
aus einer total geordneten Berechnung
die wirklich wichtigen Abhängigkeiten.

Wirklich wichtig:
- Hat eine andere Berechnung τ'
  dieselben Happens-Before-Abhängigkeiten wie τ,
  und ist τ ausführbar,
  dann ist auch τ' ausführbar und
  liefert denselben Zustand (Control-Stack + Shared-Heap).
- Die nicht wirklich wichtigen Abhängigkeiten / Reihenfolgen
  der Pfeile
  sind nur entstanden,
  weil eine TSO-Berechnung halt total geordnet ist.

Beispiel:
Betrachte die Berechnung des Dekker-Beispiels:

HBTrace $\langle (1, su).(1, ld, g\text{-}y, 0).\ (2, isu).\ (2, st, y\text{-}y, 1).\ (2, ld, g\text{-}x, 0).(1, st, g\text{-}x, 1)\rangle$

$=$



Definition (Happens-Before-Trace):

Sei $\tau \in [\![c_0 \| \dots \| c_n]\!]_{ISO}$.

Die zugehörige Happens-Before-Trace

$$HBTrace(\tau) = (N, \lambda, \to_{po}, \to_{co}, \to_{rf})$$

ist ein Graph mit

- N = Menge an Knoten
- $\lambda : N \to Act$ Knotenbeschriftung
- $\to_{po}$ = Program-Order // Totale Ordnung pro Thread
  $\to_{co}$ = Coherence-Order, auch Store-Order genannt, // Totale Ordnung pro Adresse.
  $\to_{rf}$ = Reads-From-Relation.

Die Definition ist per Induktion nach der Länge der Berechnung:

↳ $HBTrace(\varepsilon) := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

↳ Angenommen $HBTrace(\tau) = (N, \lambda, \to_{po}, \to_{co}, \to_{rf})$.

Dann ist

$$HBTrace(\tau . act) := (N \cup \{n\}, \lambda', \to_{po}', \to_{co}', \to_{rf}'),$$

wobei die Wahl von n und

die Änderung der Relationen von act abhängen:

$act = (t, st, adr, val)$:

Sei n der minimale Knoten in $\to_{po}^{\varepsilon}$

mit Beschriftung $\lambda(n) = (t, isu)$.

Wir setzen $\lambda' := \lambda[n \mapsto act]$ und $\to_{po}' := \to_{po}$.

// Falls wir ein Store sehen, wählen wir den Issue-Moment.

act $\neq (t, st, adr, val)$:

Wir fügen zur Trace ein neuen Knoten $n \notin N$ hinzu.

Wir setzen:

$$\lambda' := \lambda \cup \{(n, act)\}$$

$$\rightarrow_{po}' := \rightarrow_{po} \cup \{(max(\rightarrow_{po}^{t}), n)\}.$$

Coherence:

Wird nur für Stores (t, st, adr, val) geändert,

$$\rightarrow_{co}' := \rightarrow_{co} \cup \{(max(\rightarrow_{co}^{adr}), n)\}$$

Sonst $\rightarrow_{co}' := \rightarrow_{co}$.

Reads-From:

Wird nur für Loads und Stores geändert:

Load (t, ld, adr, val):

$$\rightarrow_{rf}' := \rightarrow_{rf} \cup \{(max(\rightarrow_{co}^{adr}), n)\}$$

Store (t, st, adr, val):

Ändere die Reads-From-Relation für alle Loads, die early vom diesem Store lesen:

$\forall m \in N$ mit $n \rightarrow_{po}^{*} m$ und $\lambda(m) = (t, ld, adr, val)$:

$$\rightarrow_{rf}' := (\rightarrow_{rf} \setminus \{(*, m)\}) \cup \{(n, m)\}.$$

Definition (From-Read):

Betrachte $HBTrace(\bar{t}) = (N, \lambda, \rightarrow_{po}, \rightarrow_{co}, \rightarrow_{rf})$.

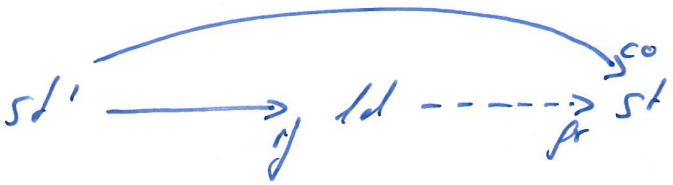
Die From-Read-Relation, auch Conflict genannt,

ist von $\rightarrow_{rf}$ und $\rightarrow_{co}$ abgeleitet:

ld $\rightarrow_{fr}$ st, falls $\exists st'.\ st' \rightarrow_{rf} ld \wedge st' \rightarrow_{co} st$

oder ld liest den Initialwert und
st ist der erste Store auf der Adresse.

Illustration:



Definition:

- Die (SC-) Happens-Before-Relation der Berechnung $\tau \in [\![c_1 \| \dots \| c_n]\!]_{TSO}$ ist

$$\rightarrow_{hb} := \rightarrow_{po} \cup \rightarrow_{co} \cup \rightarrow_{rf} \cup \rightarrow_{fr}.$$

- Die Menge an Happens-Before-Traces unter $MM \in \{SC, TSO\}$ ist

$$Traces_{MM}(c_1 \| \dots \| c_n) := \{ HBTrace(\tau) \mid \tau \in [\![c_1 \| \dots \| c_n]\!]_{MM} \}.$$

Das Entscheidungsproblem ist Folgendes:

ROBUST:

Gegeben: Ein zweielles Programm $c_1 \| \dots \| c_n$ mit endlichem Heap und endlicher Datendomäne.

Frage: $Traces_{TSO}(c_1 \| \dots \| c_n) = Traces_{SC}(c_1 \| \dots \| c_n)$?

Bemerkung:

Diese Happens-Before-basierte Robustheit ist wirklich stärker als Zustandsbasierte Robustheit.

Lemma:

Falls $Traces_{TSO}(c_1 \| \ldots \| c_n) = Traces_{SC}(c_1 \| \ldots \| c_n)$,

dann gilt $Reads_{TSO}(c_1 \| \ldots \| c_n) = Reads_{SC}(c_1 \| \ldots \| c_n)$.

Die Umkehrung gilt nicht.

Bemerkung:

- Die Inklusion $Traces_{TSO}(c_1 \| \ldots \| c_n) \supseteq Traces_{SC}(c_1 \| \ldots \| c_n)$ gilt immer.
  Wir müssen also die Umkehrung prüfen.
- Wie prüft man also $Traces_{TSO}(c_1 \| \ldots \| c_n) \subseteq Traces_{SC}(c_1 \| \ldots \| c_n)$?
  Für Traces gibt es ja keine Methoden,
  die wir komplementieren könnten.

Lemma (Shasha & Snir, TOPLAS 1988):

Betrachte $\tau \in \Pi_{c_1 \| \ldots \| c_n} \mathcal{T}_{TSO}$.

Dann gilt

$HBTrace(\tau) \in Traces_{SC}(c_1 \| \ldots \| c_n)$ gdw. $\rightarrow_{hb}$ ist azyklisch.

Beweis:

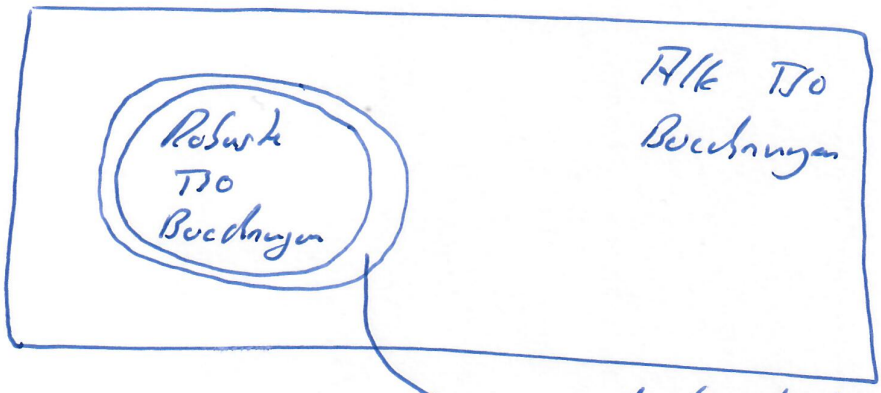
„$\Rightarrow$" SC generiert nur Ausführungen,
deren Happens-Before-Relation azyklisch ist.

„$\Leftarrow$" Jede partielle Ordnung lässt sich
zu einer totalen Ordnung fortsetzen.
Diese totale Ordnung ist die SC-Ausführung. $\square$

Beobachtung:

- Die Aufgabe der From-Read-Relation ist genau, die Zyklen in den Abhängigkeiten sichtbar zu machen.
- Das Resultat von Shasha & Snir ist semantischer Natur. Es gibt uns keinen Algorithmus, um Berechnungen mit zyklischer Happens-Before-Relation zu finden.

Ansatz:



(1) Versuche die Gestalt minimaler Robustheitsverletzungen.

Lokalität (*):
Wenn ein Program nicht robust ist, gibt es verletzende Berechnungen, bei denen nur ein Thread seinen Puffer nutzt.

Kombinatorik.

(2) Entwickle ein Algorithmus, um die Existenz von Berechnungen dieser Form zu prüfen.

Algorithmik.

Wenn Sie einen Beweis für (*) finden, melden Sie sich bitte.

Berechnung:

- Lokalität gilt für die Begriffsberechnung.
- Übrigens lässt sich die Kreis in der TSO-Berechnung entdecken: