BP 2 Unity: Erfassung der Systemarchitektur

2.3 Erfassung der Systemarchitektur

Mappings beschreiben informal Eigenschaften der Architektur des Rechensystems und ihre Auswirkung auf UNITY-Programme.

2.3.1 Asynchrone Prozessoren mit lokal gemeinsamen Speichern (Multiprozessoren)

1. Jede Anweisung wird einem bestimmten Prozessor zugeordnet.
2. Jede Variable wird einem bestimmten Speicher zugeordnet.
3. Für jeden Prozessor wird die Reihenfolge festgelegt, in der er die Anweisungen ausführt.

❑ Diese Abbildung unterliegt folgenden Einschränkungen:

1. Alle Variablen auf der linken Seite einer Zuweisung sind in Speichern hinterlegt, deren Inhalt von dem Prozessor, dem die Zuweisung zugeordnet ist, modifiziert werden kann.
2. Alle Variablen auf der rechten Seite einer Zuweisung sind in Speichern hinterlegt, deren Inhalt von dem Prozessor, dem die Zuweisung zugeordnet ist, gelesen werden kann.
3. Für jeden Prozessor stellt die Ausführungsreihenfolge sicher, daß jede ihm zugewiesene Anweisung unendlich oft zur Ausführung kommt.

BP 2 Unity: Erfassung der Systemarchitektur

2.3.2 Verteilte Systeme (Prozessoren mit lokalem Speicher und Verbindungskanälen)

1. Jede Zuweisung wird einem bestimmten Prozessor zugeordnet.
2. Jede Variable wird dem lokalen Speicher oder einem Kanal zugeordnet.
3. Für jeden Prozessor wird die Reihenfolge festgelegt, in der er die Anweisungen ausführt.

❑ Diese Abbildung unterliegt folgenden Einschränkungen:

1. Jedem Kanal ist höchstens eine Variable zugeordnet und diese Variable ist vom Typ Sequenz (FIFO-Kanal).
2. Eine Variable, die einem Kanal zugeordnet ist, kommt nur in Anweisungen zweier Prozessoren vor und diese haben eine der folgenden Formen:
   - Die Anweisungen auf einem der Prozessoren modifizieren die Variable durch Anfügen eines Wertes, solange die Sequenz eine bestimmte Länge (Pufferlänge) nicht überschreitet.
   - Anweisungen des anderen Prozessors modifizieren die Variable durch Entfernen des ersten Wertes der Sequenz, falls ihre Länge größer als null ist.
   - Die Variable wird auf keine andere Weise benutzt.

BP 2 Unity: Terminierungsentdeckung

2.4 Anwendungsbeispiel: Verteilte Terminierungsentdeckung

2.4.1 Definition des generischen Entdeckungsproblems

2.4.1.1 Programmüberlagerungen

❑ Gegeben sei ein (unterliegendes) Programm F und eine stabile Eigenschaft W dieses Programms.
Durch Überlagerung entstehe daraus ein Programm mit einer neuen Variablen *claim*, so daß gilt: claim detects W.

Unter Überlagerung wird verstanden:
Hinzufügen von Variablen, einschließlich ihrer Initialisierung,
Erweitern der Zuweisungen durch neue parallele Zuweisungen und
Hinzufügen neuer Zuweisungen,
wobei die neuen Zuweisungen keine Variablen von W verändern.

❑ Überlagerungssatz
Alle in den Basisrelationen ausdrückbaren gültigen Prädikate des unterliegenden Programms sind auch jeder Überlagerung erfüllt.

BP 2 Unity: Terminierungsentdeckung

2.4.1.2 Beispiel zur Motivation

W ≡ (die Anzahl der ausgeführten Befehle übersteigt 10)

- **Program** {superposition} P1
  initially count = 0 [] claim = false
  transform
  each statement s in the underlying program to:
  s || count := count + 1
  add claim := (count > 10)
  end {P1}
- Folgerung
  claim *detects* (count > 10)

2.4.2 Heuristiken

1. Direkte Implementierung der Entdeckung

(I) x detects e, wobei x eine überlagerte Variable ist und e ein stabiles Prädikat über dem zugrundeliegenden Programm.

- Anfangs x = false
- Genau ein hinzugefügter Befehl: x := e

Beweis:

a) invariant $x \Rightarrow e$

Aussage am Anfang richtig.

Kann nur falsch werden, wenn e falsch wird oder x wahr.

Da e stabil, ersteres nicht möglich.

Letzteres nur durch Befehl x := e; Nachbedingung ist x = e, also gilt $x \Rightarrow e$.

b) $e \mapsto x$

$(\text{stable } e \Leftrightarrow e \text{ unless false}) \Rightarrow e \text{ unless } x$ und somit $\{e \wedge \neg x\}x := e\{x\}$,

also e ensures x und damit $e \mapsto x$.

(II) Implementierung von $(x \geq k)$ detects $(e \geq k)$, wobei x überlagerte Variable ist und e ein monoton wachsender ganzzahliger Ausdruck.

- Anfangs $x = (\text{Anfangswert von } e)$.
- Genau ein hinzugefügter Befehl: x := e

Beweis:

Mit der Bemerkung, daß die Spezifikation die Eigenschaft invariant $x \leq e$ und $(e = k) \mapsto (x \geq k)$ besitzt, erfolgt der Beweis analog dem vorangehenden.

2. Nutzung der Transitivität von detects

(III) Implementierung von claim detects W(d), wobei es ineffizient ist, W(d) auf der gegebenen Architektur direkt festzustellen (z. B. weil d eine verteilte Datenstruktur ist)

- Auffinden eines günstiger festzustellenden Prädikats p(d') über einer geeigneten Datenstruktur d'.
- Gestaltung des Programms so, daß
  claim detects p(d') und p(d') detects W(d).

Die Invariante, die d' beschreibt, ist normalerweise schwächer als die von d, die Fortschritt-Bedingung stärker.

Beweis:

Transitivität von detects folgt aus der von $\mapsto$.

2.4.3 Anwendungsbeispiel für die Heuristiken

❏ Ausführung des vorangehenden Programms in einem verteilten System

1. Aufspaltung des Zählers nach (III), dazu:
2. Einführung einer überlagerten Variablen u.m für jeden Prozessor u zur Zählung der lokalen Befehlsausführungen.
3. Einführung einer weiteren überlagerten Variablen n, der von Zeit zu Zeit die Summe der u.m zugewiesen wird unter Verwendung von (II).
4. Mittels claim wird n>10 festgestellt nach (I). n>10 ist stabil, weil n monoton steigend ist.

- Formale Beschreibung

  invariant $u.m = (\text{Anzahl der Befehlsausführungen in Prozeß } u)$

  $n \le \langle + u :: u.m \rangle$

  $(\langle + u :: u.m \rangle = k) \mapsto (n \ge k)$

  claim detects $(n > 10)$

- Korrektheitsbeweis der Lösungsstrategie

  $(n > k)$ detects $(\langle + u :: u.m \rangle > k)$

  claim detects $(count > 10)$

  mit $count = (\langle + u :: u.m \rangle)$

  Hier wird deutlich wie die Invariante abgeschwächt und die Fortschritt-Bedingung verstärkt wird:

  claim detects W(count) $\Rightarrow$ claim detects p(n) $\wedge$ p(n) detects W(count)

  invariant $count = \langle + u :: u.m \rangle$ $\Rightarrow$ invariant $n \le \langle + u :: u.m \rangle$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

true $\Rightarrow$ $(\langle + u :: u.m \rangle = k) \mapsto (n \ge k)$

- Ableitung des Programms

  Program {superposition} P2

  initially n = 0 [] claim = false [] ⟨[] u :: u.m = 0⟩

  transform

  for each processor u, transform each statement s in processor u to

  s || u.m := u.m + 1

  add n := ⟨+ u :: u.m⟩ [] claim := (n > 10)

  end {P2}

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

2.4.4 Weitere Anwendung der Heuristiken

- Vorgehensweise

  1. Aufspaltung der Summation über nicht-lokale Variablen in eine Summation über lokale Kopien der Variablen. Dazu:
  2. Einführung weiterer überlagerter Variablen u.r lokal zum entdeckenden Prozessor, die Kopien der Variablen u.m darstellen (II).
  3. Variable n erhält die Summe dieser Zwischenvariablen.
  4. Dadurch Entdeckung mit Zwischenschritt (III).

- Formale Beschreibung

  invariant $u.m = (\text{Anzahl der Befehlsausführungen in Prozeß } u)$

  $(u.r > k)$ detects $(u.m > k)$ (für alle u)

  woraus folgt: $(\langle + u :: u.r \rangle > k)$ detects $(\langle + u :: u.m \rangle > k)$

  $(n > k)$ detects $(\langle + u :: u.m \rangle > k)$

  claim detects $(n > 10)$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- Korrektheitsbeweis der Lösungsstrategie

  Zweimalige Anwendung der Transitivität von *detects*.

- Ableitung des Programms

  Program {superposition} P3

  initially n = 0 [] claim = false [] ⟨[] u :: u.m, u.r = 0, 0⟩

  transform

  for each processor u, transform each statement s in processor u to

  s || u.m := u.m + 1

  add n := ⟨+ u :: u.r⟩ [] claim := (n > 10) [] ⟨[] u :: u.r := u.m⟩

  end {P3}

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

2.4.5 Spezifikation der Terminierungsentdeckung

- Das zugrunde gelegte Programm wird repräsentiert durch einen gerichteten Graphen $G = (V, E)$, wobei V eine (konstante) Menge von Knoten ist, die die Prozesse darstellen, und E eine (konstante) Menge von Pfeilen, die unidirektionale Verbindungskanäle zwischen den Prozessen darstellen.
- Mit jedem Knoten u ist ein Prädikat u.idle verbunden, das anzeigt, ob der entsprechende Prozeß passiv ist.
- Programmstruktur (synchrones Modell!)

Program {structure of the underlying program} R0
  assign ⟨[] u, v :: (u, v) ∈ E :: v.idle = u.idle ∧ v.idle⟩
  [] ⟨[] u :: u ∈ V :: u.idle := true⟩
end {R0}

- Stabiles Prädikat W, das entdeckt werden soll:

$W \equiv \langle \wedge u : u \in V :: u.idle \rangle$

2.4.6 Sequentielle Architektur

- Heuristik I

Program {superposition on R0} R1
  initially claim = false
  add claim := ⟨∧ u :: u.idle⟩
end {R1}

2.4.7 Asynchrone Architektur mit lokal gemeinsamen Speichern

- Informelle Beschreibung
  1. Einführung einer Variablen d, definiert als die Menge der inaktiven Prozesse.
  2. Bei Änderungen des Prozeßzustandes wird d entsprechend aktualisiert.
  3. Wenn d alle Prozesse enthält, ist das Programm terminiert.
- Formale Beschreibung

invariant

$d = \{u \mid u.idle\}$

claim detects $(d = V)$

- Korrektheitsbeweis

Trivial mit $W \equiv (d = V)$.

- Ableitung des Programms

Program {the transformed program} R2
initially claim = false [] d = {u | u.idle}
assign ⟨[] (u, v) :: v.idle = u.idle ∧ v.idle
    || d := d - {v} if ¬u.idle
  ⟩
  [] ⟨[] u :: u.idle = true
    || d := d ∪ {u} if u.idle
  ⟩
  [] claim := (d = V)
end {R2}

Unity: Terminierungsentdeckung

2.4.7.1 Erste Entkopplung der Modifikationen

- Informelle Beschreibung
  1. Abschwächung der Invariante dahingehend, daß d eine Teilmenge der inaktiven Prozesse ist (III).
  2. Verstärkung der Fortschritt-Bedingung: Endliche Zeit, nachdem alle Prozesse inaktiv sind, enthält d alle Prozesse.
  3. Prozesse werden zu d asynchron hinzugefügt, nachdem sie inaktiv geworden sind.
- Formale Beschreibung

invariant

$d \subseteq \{u \mid u.idle\}$

$W \mapsto (d = V)$

claim detects $(d = V)$

- Korrektheitsbeweis

$(d = V) \Rightarrow W$, wegen der ersten Invariante

$(d = V)$ detects W , wegen der zweiten Invariante

claim detects W , wegen der dritten Invariante und der Transitivität von *detects*

Unity: Terminierungsentdeckung

- Ableitung des Programms

Program {the transformed program} R2'

initially claim = false [] d = {u | u.idle}

assign $\langle$[] (u, v) :: v.idle = u.idle $\wedge$ v.idle

|| d := d - {v} if $\neg$u.idle

$\rangle$

[] $\langle$[] u :: u.idle = true

[] d := d $\cup$ {u} if u.idle

$\rangle$

[] claim := (d = V)

end {R2'}

Unity: Terminierungsentdeckung

2.4.7.2 Weitere Entkopplung der unterliegenden und der überlagerten Berechnung

- Informelle Beschreibung
  1. Einführung einer neuen Variablen b mit $b \subseteq d$ zusammen mit lokalen Variablen u.delta, so daß $b \cup \bigcup_u u.delta = d$ ist.
  2. Dadurch Entdeckung mit Zwischenschritt (III).
- Formale Beschreibung

Die Relation *chain* zwischen Prozessen sei zu jedem Zeitpunkt der Berechnung definiert durch:

u chain v $\equiv$ Im unterliegenden Graphen G existiert ein Pfad von u nach v derart, daß für jede Kante (u', v') auf diesem Pfad $v' \in u'.delta$ ist.

invariant $v \in b \Rightarrow [(v.delta = empty) \wedge v.idle] \vee \langle \exists u : u \notin b :: u \text{ chain } v \rangle$

stable $b = V$

$W \mapsto (b = V)$

claim detects $(b = V)$

Unity: Terminierungsentdeckung

- Korrektheitsbeweis der Lösungsstrategie

$(b = V) \Rightarrow W$,wegen der Invariante, da die zweite Alternative nicht zutreffen kann

$(b = V)$ detects W ,wegen der vorangehenden Beziehung und der Fortschritt-Bedingung

claim detects W ,wegen der Transitivität von *detects*

- Ableitung des Programms

Program {the transformed program} R3

initially claim = false [] b = empty [] ⟨[] u :: u.delta = empty⟩

assign ⟨[] (u, v) :: v.idle = u.idle $\wedge$ v.idle

|| u.delta := u.delta $\cup$ {v} if $\neg$u.idle

⟩

[] ⟨[] u :: u.idle = true⟩

[] ⟨[] u :: b, u.delta := b $\cup$ {u} - u.delta, empty if u.idle⟩

[] claim := (b = V)

end {R3}

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- Korrektheitsbeweis des Programms

(i) Invariante

Im Anfangszustand erfüllt, da dort b = empty gilt.

Beweisstruktur: Feststellung der Zustandsübergänge, die die Invariante ungültig machen könnten und Untersuchung entsprechender Anweisungen

	Vorbedingung	Nachbedingung
(1)	$v \notin b$	$v \in b$
(2)	v.delta = empty $\wedge$ v.idle	v.delta $\neq$ empty
(3)	v.delta = empty $\wedge$ v.idle	$\neg$v.idle
(4)	$\exists$u: u $\notin$ b :: u chain v	$\forall$u: u $\notin$ b :: $\neg$(u chain v)

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

(1) Nur durch Befehl

b, v.delta := b $\cup$ {v} - v.delta, empty if v.idle

Erhält die Invariante.

(2) Nur durch Befehl

u.idle = v.idle $\wedge$ u.idle || v.delta := v.delta $\cup$ {u} if $\neg$v.idle

Nach Vorbedingung v.idle, also Zustand unverändert.

(3) Nur durch Befehl

v.idle = u.idle $\wedge$ v.idle || u.delta := u.delta $\cup$ {v} if $\neg$u.idle

Vor Ausführung u.idle: Zustand unverändert.

Vor Ausführung $\neg$u.idle: Wegen der Invariante

(u $\notin$ b) $\vee$ ($\exists$u': u' $\notin$ b :: u' chain u) .

Also nach dem Befehl v $\in$ u.delta und somit u chain v.

Aus u $\notin$ b folgt mit u = u': $\exists$u': u' $\notin$ b :: u' chain v.

Aus $\exists$u': u' $\notin$ b :: u' chain u folgt dies mit

u' chain u $\wedge$ u chain v $\Rightarrow$ u' chain v.

Also bleibt die Invariante erhalten.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

(4) Zwei Fälle

1. Kette hat Verbindungsstück das unterbrochen wird:

Vor Übergang (u chain x) $\wedge$ (y $\in$ x.delta) $\wedge$ (y chain v) ,

danach (y $\notin$ x.delta) $\wedge$ (y chain v) .

Nur durch Befehl **b, x.delta := b $\cup$ {x} - x.delta, empty if x.idle**.

Danach y $\notin$ b und damit v $\notin$ b für y = v und

y $\notin$ b $\wedge$ (y chain v) für y $\neq$ v.

Also Invariante gesichert.

2. Kettenanfang u wird in b aufgenommen:

Vor Übergang: (y $\in$ delta) $\wedge$ (y chain v) .

Nach Übergang: (y $\in$ b) $\wedge$ (y chain v) .

Nur durch Befehl **b, u.delta := b $\cup$ {u} - u.delta, empty if u.idle**.

Nach Ausführung y $\in$ b und damit v $\notin$ b für y = v und

y $\notin$ b $\wedge$ (y chain v) für y $\neq$ v.

Also Invariante gesichert.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

(ii) Fortschritt-Bedingung $W \mapsto (b = V)$

Wenn W gilt, dann können nur die Befehle

$\langle[]\ u :: b, u.delta := b \cup \{u\} - u.delta, empty \quad if\ u.idle\rangle$

$[]\quad claim := (b = V)$

den Zustand ändern.

Daraus folgt:

(a) $W \mapsto (v.delta = empty)$

(b) stable $W \wedge (v.delta = empty)$

(c) $(W \wedge \forall v :: v.delta = empty) \mapsto (u \in b)$

(d) stable $W \wedge (\forall v :: v.delta = empty) \wedge u \in b$

Aus (a) und (b): $W \mapsto (\forall v :: v.delta = empty)$

und daraus mit (c) und (d): $W \mapsto (\forall u :: u \in b)$.

(iii) Stabilität stable $b = V$

$(b = V) \wedge$ Invariante $\Rightarrow (\forall v :: v.idle \wedge v.delta = empty)$.

Damit kann nur die letzte Zeile des Programms den Zustand ändern, also bleibt b auf jeden Fall unverändert.

(iv) claim detects $(b = V)$

Trivial nach Heuristik 1.

2.4.8 Verteilte Architekturen

2.4.8.1 Problembeschreibung

- Grundlegendes Modell wie bei lokal gemeinsamem Speicher.
- Zusätzlich: Ein Kanal von u nach v wird dargestellt durch die Variable (u, v).c.

  Da der Inhalt der Nachrichten des unterliegenden Systems belanglos ist, werden alle seine Nachrichten durch den Wert m dargestellt.
- Anfänglich ist genau der Prozeß *init* aktiv.

- Programmstruktur (asynchrones verteiltes Modell!)

Program {structure of the underlying program} DS

assign

{ u sends message m along channel (u, v).c if u is active}

$\langle[]\ u, v : (u, v) \in E :: (u, v).c := (u, v).c; m \quad if\ \neg u.idle\ \rangle$

{v becomes active by receiving a message along channel (u, v)}

$[]\quad \langle[]\ u, v : (u, v) \in E :: v.idle, m, (u, v).c := false, head((u, v).c), tail((u, v).c)$

$if\ (u, v).c \neq null\ \rangle$

$[]\quad \langle[]\ u :: u.idle := true\ \rangle$

end {DS}

- Stabiles Prädikat W, das entdeckt werden soll:

$W \equiv \langle \wedge u : u \in V :: u.idle \wedge (u, v).c = null\rangle$

- Kanalüberlagerung

  Rückwirkungsfreie Mitbenutzung der vorhandenen Kanäle
  - Durch getypte Nachrichten (vergleichbar den ‘message queues’ von Unix), der Typ legt fest, ob eine Nachricht zum überlagerten oder darunterliegenden Programm gehört.
  - Die unterliegende Programmausführung ignoriert überlagerte Nachrichten.
  - Es wird sichergestellt, daß eine Nachricht des unterliegenden Programms nicht unendlich lange durch überlagerte Nachrichten blockiert wird.
- Informelle Beschreibung der Lösungsstrategie
  - Jeder Prozeß des Systems mit gemeinsamem Speicher wird mit einem Prozeß des verteilten Systems samt seiner sendenden Kanäle gleichgesetzt.
  - Ein inaktiver Prozeß des Systems mit gemeinsamem Speicher entspricht im verteiltemSystem einem inaktiven Prozeß mit leeren sendenden Kanälen.
  - Um festzustellen, ob ein Kanal leer ist, werden Bestätigungsnachrichten verwendet.
  - Für jeden Prozeß u wird ein Prädikat u.e vorausgesetzt mit

    u.e detects $(\forall v : (u, v) \in E :: (u.v).c = null)$

- Abgeleitetes Programm

Program {distributed termination detection - transformed program} R4

initially

claim = false [] b = empty [] ⟨[] u :: u.delta = empty ⟩

assign

{u sends message m along channel (u,v).c if u is active, and u.delta is updated}

⟨[] (u, v) ::

(u, v).c, u.delta := (u,v).c; m, u.delta ∪ {v} if ¬u.idle ⟩

{u receives a message along channel (u, v) after which it becomes active}

[] ⟨[] (u, v) ::

v.idle, m, (u, v).c := false, head((u, v).c), tail((u, v).c) if (u, v).c = null ⟩

[] ⟨[] u :: u.idle := true ⟩

{added statements}

[] ⟨[] u :: b, u.delta := b ∪ {u} - u.delta, empty if u.idle ∧ u.e ⟩

[] claim := (b = V)

end {R4}