Ludwig-Maximilians-Universität München
Institut für Informatik
Dr. S. Merz

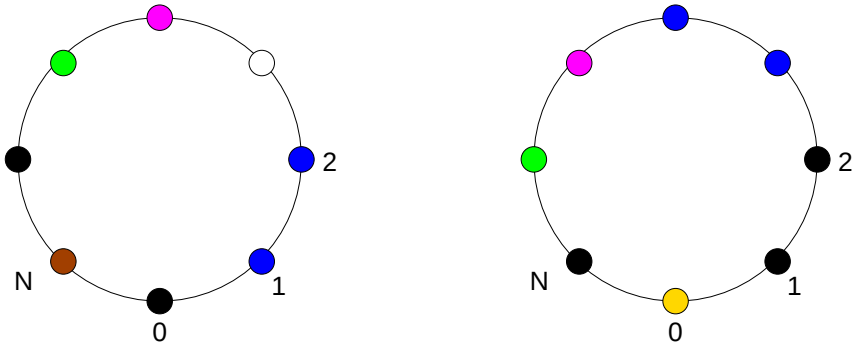
SS 2001
Begleitblatt 13

Temporale Logik

Ein größeres Verifikationsbeispiel

Ein Algorithmus heißt *selbst-stabilisierend*, wenn er ausgehend von jedem beliebigen Zustand nach endlich vielen Schritten einen „stabilen“ Zustand erreicht und danach immer „stabil“ bleibt. Protokolle zur Selbst-Stabilisierung finden Anwendung zur Initialisierung verteilter Systeme und zur Sicherung der Fehlertoleranz. Im folgenden soll ein auf Dijkstra [1] zurückgehendes selbst-stabilisierendes Netzwerkprotokoll modelliert und verifiziert werden.

Ein Ring-Netzwerk besteht aus $N + 1 \geq 2$ Knoten, die mit $0, 1, \ldots, N$ nummeriert sind. Die folgende Abbildung illustriert die Netz-Struktur:



(a) mögliche Ausgangskonfiguration (b) Konfiguration nach einigen Schritten

Abbildung 1: Ablaufbeispiel von Dijkstras Protokoll

Jeder Knoten i enthält ein Register r_i, das eine Zahl aus der Menge $\{0, 1, \ldots, M\}$ speichern kann, dabei gelte $M \geq N$. In obiger Abbildung sind verschiedene Registerwerte durch verschiedene Grautöne gekennzeichnet. Jeder Knoten kann auf sein Register lesend und schreibend und außerdem auf das Register seines linken Nachbarn lesend zugreifen.

Wir nennen eine Belegung der Register mit Werten eine *Konfiguration*. Das Netz startet mit einer beliebigen Ausgangskonfiguration. Konfigurationen ändern sich, indem ein Knoten einen Schritt ausführt:

- Knoten 0 kann einen Schritt ausführen, wenn $r_0 = r_N$ gilt; er erhöht dann r_0 um 1 modulo $M + 1$. Alle übrigen Registerinhalte bleiben unverändert.
- Knoten i (für $i \in \{1, \ldots, N\}$) kann einen Schritt ausführen, wenn $r_i \neq r_{i-1}$ gilt; er kopiert dann den Wert r_{i-1} in sein eigenes Register. Alle übrigen Registerinhalte bleiben unverändert.

In Pseudocode-Notation kann das Protokoll folgendermaßen beschrieben werden:

var $r_0, \ldots, r_N : [0..M]$;
do
$\quad r_0 = r_N \quad \longrightarrow \quad r_0 := (r_0 + 1) \bmod (M + 1)$
$\square_{i<N} \quad r_{i+1} \neq r_i \quad \longrightarrow \quad r_{i+1} := r_i$
od

Eine Konfiguration heißt *stabil*, wenn es genau einen Knoten gibt, der einen Schritt ausführen kann. Das Protokoll stellt sicher, dass in jedem Ablauf eine stabile Konfigurationen erreicht wird und alle Folgekonfigurationen dann ebenfalls stabil sind — allerdings ist der Beweis dafür alles andere als offensichtlich [2,3].

Formalisierung als FSTS

Die Signatur SIG enthalte die Sorte VAL, die Konstanten $0^{(\varepsilon, VAL)}$ und $1^{(\varepsilon, VAL)}$ sowie das zweistellige Funktionszeichen $\oplus^{(VAL\ VAL, VAL)}$, das infix geschrieben wird. (Die Signatur wird später noch erweitert werden.) Ferner sei $\mathbb{S}$ eine Struktur für SIG, in der die Sorte VAL durch $\{0, 1, \ldots, M\}$, die Konstanten durch 0 und 1 und die Funktion $\oplus$ durch die Addition modulo $M + 1$ interpretiert werde.

Aktionen: $\mathcal{A} = \{\lambda_0, \lambda_1, \ldots, \lambda_N\}$, dabei entspreche λ_i einem Schritt von Prozessor i

Systemvariablen: $X = \{r_0, r_1, \ldots, r_N\}$, $V = \{\text{exec } \lambda_0, \ldots, \text{exec } \lambda_N\}$

Zustände: Z sei die Menge von Abbildungen

$$\eta : \left\{ \begin{array}{l} X \to \{0, 1, \ldots, M\} \\ V \to \{\mathbf{f}, \mathbf{t}\} \end{array} \right\} \quad \text{mit}$$

- $\eta(\text{exec } \lambda_i) = \mathbf{t}$ für höchstens ein i,
- falls $\eta(\text{exec } \lambda_0) = \mathbf{t}$, dann $\eta(r_0) = \eta(r_n)$,
- falls $\eta(\text{exec } \lambda_{i+1}) = \mathbf{t}$, dann $\eta(r_{i+1}) \neq \eta(r_i)$ für $i = 0, \ldots, N-1$.

Transitionsrelation: Es sei $(\eta, \eta') \in T \subseteq Z \times Z$ genau dann, wenn

- falls $\eta(\text{exec } \lambda_0) = \mathbf{t}$, dann $\eta'(r_0) = (\eta(r_0) + 1) \bmod M + 1$ und $\eta'(r_i) = \eta(r_i)$ für $i = 1, \ldots, N$,
- falls $\eta(\text{exec } \lambda_{i+1}) = \mathbf{t}$, dann $\eta'(r_{i+1}) = \eta(r_i)$ und $\eta'(r_j) = \eta(r_j)$ für $i = 0, \ldots, N-1, j = 0, \ldots, N$, $j \neq i + 1$.

Ausführbarkeitsbedingungen:

$$\begin{array}{rcl} \text{enabled}_{\lambda_0} & \equiv & r_0 = r_N \\ \text{enabled}_{\lambda_{i+1}} & \equiv & r_{i+1} \neq r_i \quad (i = 0, \ldots, N-1) \end{array}$$

Die Γ-Theorie ist gegeben durch die Axiome (data) sowie

(action)	exec $\lambda_i \to$ enabled$_{\lambda_i}$ $(i = 0, \ldots, N)$
(fair)	$\Box\Diamond$enabled$_{\lambda_i} \to \Diamond$exec λ_i $(i = 0, \ldots, N)$
(intl)	$\neg(\text{exec } \lambda_i \wedge \text{exec } \lambda_j)$ $(i, j = 0, \ldots, N, i \neq j)$
(act$_0$)	exec $\lambda_0 \to r_0' = r_0 \oplus 1$
(act$_{i+1}$)	exec $\lambda_{i+1} \to r_{i+1}' = r_i$ $(i = 1, \ldots, N-1)$
(unch)	exec $\lambda_i \to r_j' = r_j$ $(i, j = 0, \ldots, N, i \neq j)$

Wir definieren ferner die Formel

$$stable \equiv \bigvee_{i=0}^{N} \left(\text{enabled}_{\lambda_i} \wedge \bigwedge_{j=0, j\neq i}^{N} \neg\text{enabled}_{\lambda_j} \right)$$

1. Schritt: Ein stabiles System bleibt stabil

Zunächst beweisen wir, dass ausgehend von einer stabilen Konfiguration nur stabile Konfigurationen erreichbar sind. Dazu beweisen wir zunächst die Hilfsaussagen

(D1)	enabled$_{\lambda_0} \vee$ enabled$_{\lambda_1} \vee \ldots \vee$ enabled$_{\lambda_N}$
(D2)	exec $\lambda_i \to \bigcirc\neg$enabled$_{\lambda_i}$ $\quad (i = 0, \ldots, N)$
(D3)	exec $\lambda_i \to (\bigcirc$enabled$_{\lambda_j} \leftrightarrow$ enabled$_{\lambda_j})$ $\quad (i, j = 0, \ldots, N, j \neq i, j \neq (i+1) \bmod (N+1))$

und folgern dann

(D4) $\mathbb{A}_\Gamma \vdash stable \to \Box stable$

Beweis von (D1). Angenommen, die Aktionen $\lambda_1, \ldots, \lambda_N$ seien nicht ausführbar. Dann gilt $r_1 = r_0$, $r_2 = r_1$, $\ldots$, $r_N = r_{N-1}$, also folgt $r_0 = r_n$. Das heißt, λ_0 ist ausführbar. Also ist in jedem Zustand mindestens eine Aktion ausführbar.

(1)	$\big(\bigwedge_{i=1}^{N} \neg\text{enabled}_{\lambda_i}\big) \rightarrow \bigwedge_{i=1}^{N} r_{i-1} = r_i$	(taut)
(2)	$\big(\bigwedge_{i=1}^{N} r_{i-1} = r_i\big) \rightarrow r_0 = r_N$	(pred)
(3)	$r_0 = r_N \rightarrow \text{enabled}_{\lambda_0}$	(taut)
(4)	$\bigvee_{i=0}^{N} \text{enabled}_{\lambda_i}$	(prop)(1)–(3)

Beweis von (D2). Unmittelbar aus der Definition von λ_i. Im Fall $i = 0$ benutzen wir die Annahme, dass $M \geq N \geq 1$ gilt, und daher $r_0 \oplus 1 \neq r_0$ gilt.

(1)	exec $\lambda_{i+1} \rightarrow r'_{i+1} = r_i \wedge r'_i = r_i \quad (i = 0, \ldots, N-1)$	(act$_{i+1}$)(unch)
(2)	exec $\lambda_i \rightarrow \bigcirc\neg\text{enabled}_{\lambda_i} \quad (i = 1, \ldots, N-1)$	(1)
(3)	exec $\lambda_0 \rightarrow r_0 = r_N \wedge r'_0 = r_0 \oplus 1 \wedge r'_N = r_N$	(action)(act$_0$)(unch)
(4)	$\neg(r_0 \oplus 1 = r_0)$	(data)
(5)	exec $\lambda_0 \rightarrow \bigcirc\neg\text{enabled}_{\lambda_0}$	(3)(4)

Beweis von (D3). Die Ausführbarkeitsbedingung enabled$_{\lambda_i}$ hängt nur von r_i und $r_{(i-1) \bmod (N+1)}$ ab, während die Ausführung von λ_i nur den Inhalt von Register r_i verändert. Die formale Herleitung ist sehr ähnlich zu der von (D2) und bleibt als Übung.

Beweis von (D4). Wir zeigen $stable$ **invof** $\mathcal{A}$. Wenn die Ausgangskonfiguration stabil ist, entsteht die Folgekonfiguration durch einen Schritt des einzigen Knotens i, für den enabled$_{\lambda_i}$ gilt. Nach (D2) gilt im Folgezustand $\neg$enabled$_{\lambda_i}$, und nach (D3) ausserdem $\neg$enabled$_{\lambda_j}$ für alle j ausser dem rechten Nachbarn von i, also gilt enabled$_{\lambda_k}$ für höchstens ein k. Andererseits gilt enabled$_{\lambda_k}$ nach (D1) für mindestens ein k, und damit ist die Folgekonfiguration wieder stabil.

(1)	$stable \rightarrow stable$	(taut)
(2)	exec $\lambda_i \rightarrow$ enabled$_{\lambda_i} \quad (i = 0, \ldots, N)$	(action)
(3)	enabled$_{\lambda_i} \wedge stable \rightarrow \neg$enabled$_{\lambda_j} \quad (i, j = 0, \ldots, N, i \neq j)$	(taut)
(4)	exec $\lambda_i \rightarrow \bigcirc\neg$enabled$_{\lambda_i} \quad (i = 0, \ldots, N)$	(D2)
(5)	$\bigcirc$enabled$_{\lambda_0} \vee \ldots \vee \bigcirc$enabled$_{\lambda_N}$	(D1)(nex)(T16)
(6)	exec $\lambda_0 \wedge stable \rightarrow \bigcirc\neg$enabled$_{\lambda_j} \quad (j = 2, \ldots, N)$	(2)(3)(D3)
(7)	exec $\lambda_0 \wedge stable \rightarrow \bigcirc stable$	(4)(5)(6)
(8)	exec $\lambda_i \wedge stable \rightarrow \bigcirc\neg$enabled$_{\lambda_0} \quad (i = 1, \ldots, N-1)$	(2)(3)(D3)
(9)	exec $\lambda_i \wedge stable \rightarrow \bigcirc\neg$enabled$_{\lambda_j} \quad (i, j = 1, \ldots, N-1, j \neq i, j \neq i+1)$	(2)(3)(D3)
(10)	exec $\lambda_i \wedge stable \rightarrow \bigcirc stable \quad (i = 1, \ldots, N-1)$	(4)(5)(8)(9)
(11)	exec $\lambda_N \wedge stable \rightarrow \bigcirc\neg$enabled$_{\lambda_j} \quad (j = 1, \ldots, N-1)$	(2)(3)(D3)
(12)	exec $\lambda_N \wedge stable \rightarrow \bigcirc stable$	(4)(5)(11)
(13)	$stable$ **invof** $\mathcal{A}$	(7)(10)(12)
(14)	$stable \rightarrow \Box stable$	(inv)(1)(13)

2. Schritt: Das System konvergiert zu einer stabilen Konfiguration

Der eigentliche Kern des Beweises besteht darin zu zeigen, dass von jeder Ausgangskonfiguration aus eine stabile Konfiguration erreicht wird, d.h. dass die Formel $\Diamond stable$ Γ-gültig ist. Diesen Beweis unterteilen wir in zwei Schritte:

1. Angenommen, r_0 enthalte einen Wert, der in keinem der übrigen Register vorkommt. (Eine solche Konfiguration ist in Abb. 1(b) dargestellt.) Nun kann Prozessor 0 so lange keinen Schritt ausführen, bis $r_N = r_0$ gilt, während die Aktionen der übrigen Prozessoren dafür sorgen, dass sich der Wert von r_0 im Ring fortpflanzt. Dabei entsteht ein immer länger werdendes "Anfangsstück" des Rings, in dem alle Register den Wert r_0 enthalten. Gilt schließlich $r_0 = r_1 = \ldots = r_N$, so ist eine stabile Konfiguration erreicht.

2. Es reicht also zu zeigen, dass irgendwann eine Konfiguration wie unter (1) beschrieben erreicht wird, falls nicht zuvor bereits eine stabile Konfiguration erreicht wurde. Dazu bemerken wir, dass nach M und N so gewählt waren, dass $M+1 \geq N+1 > N$ gilt und es daher in jeder Konfiguration einen Wert $i \in \{0, \ldots, M\}$ gibt, der in den Registern $r_1, r_2, \ldots, r_N$ nicht vorkommt ("Taubenschlagprinzip"). Sei m der (zyklisch) kleinste solche Wert oberhalb vom Wert von r_0 in der Ausgangskonfiguration. Da nur Schritte von Prozessor 0 neue Werte in den Ring einführen können, während die übrigen Prozessoren lediglich kopieren, wird m in den Registern $r_1, \ldots, r_N$ so lange nicht vorkommen, bis $r_0 = m$ gilt, und dann ist eine Konfiguration wie in (1) erreicht.

Die folgenden Herleitungen präzisieren und formalisieren diese Überlegungen. Zunächst definieren wir einige Hilfsformeln. Dabei nehmen wir an, dass die Signatur SIG das Prädikatszeichen $<$ enthalte, das in $\mathbb{S}$ durch die „kleiner als"-Relation interpretiert werde.

(pref) $\quad pref_i \equiv \bigwedge_{j=0}^{i} r_j = r_0 \qquad (i = 0, \ldots, N)$

(oth) $\quad oth(x) \equiv \bigvee_{j=1}^{N} (r_j = x \wedge \neg pref_j)$

(min) $\quad min(x) \equiv \neg oth(r_0 \oplus x) \wedge \forall\, y(y < x \rightarrow oth(r_0 \oplus y))$

Zu den (data)-Axiomen, die wir im folgenden benutzen, gehört insbesondere

(min-ex) $\quad \exists x\ min(x)$

Denn nach Definition kann $oth(x)$ nur für höchstens N verschiedene Werte von $x \in \mathbb{Z}_M$ gelten, während der Ausdruck $r_0 \oplus x$ (unabhängig vom aktuellen Wert von r_0) alle Werte in $\{0, \ldots, M\}$ annehmen kann, und daher muss $\neg oth(r_0 \oplus x)$ für mindestens ein $x \in \mathbb{Z}_M$ zutreffen. Daher gibt es auch einen kleinsten Wert x, für den $\neg oth(r_0 \oplus x)$ zutrifft, und für diesen gilt nach Definition gerade $min(x)$.

Die folgenden Aussagen und Herleitungen gelten jeweils für alle $i = 0, \ldots, N-1$.

(D5) $\quad \text{exec}\ \lambda_{i+1} \wedge pref_j \rightarrow \circ pref_j \quad (j = 0, \ldots, N)$

Idee: Wird λ_{i+1} ausgeführt und gilt $pref_j$, so muss $i+1$ größer als j sein, denn sonst hätten wir $r_{i+1} = r_i = r_0$, und Prozessor $i+1$ könnte keinen Schritt ausführen. Doch dann folgt $r'_k = r_k$ für alle $k \leq i$, und insbesondere $r'_k = r'_0$ für alle $k \leq j$.

(1)	$\text{exec}\ \lambda_{i+1} \rightarrow r_{i+1} \neq r_i$	(action)
(2)	$\text{exec}\ \lambda_{i+1} \rightarrow \neg pref_{i+1}$	(1)(pref)
(3)	$\text{exec}\ \lambda_{i+1} \wedge \neg pref_{i+1} \wedge pref_j \rightarrow \bigwedge_{k=0}^{j} r'_k = r_k$	(unch)
(4)	$\bigwedge_{k=0}^{j} r'_k = r_k \rightarrow \big((\bigwedge_{k=0}^{j} r_k = r_0) \leftrightarrow (\bigwedge_{k=0}^{j} r'_k = r'_0)\big)$	(pred)
(5)	$\text{exec}\ \lambda_{i+1} \wedge pref_j \rightarrow \circ pref_j$	(2)(3)(4)

(D6) $\quad \text{exec}\ \lambda_{i+1} \wedge \circ oth(x) \rightarrow oth(x)$

Idee: Wird λ_{i+1} ausgeführt und gilt $\circ oth(x)$, so ist $x = r'_j$ für ein $j \in \{1, \ldots, N\}$, so dass $\circ\neg pref_j$ gilt. Mit (D5) folgt $\neg pref_j$ und daher $oth(r_j)$. Im Fall $i+1 \neq j$ folgt die Behauptung, da dann $r_j = r'_j = x$ gilt. Ist $j = i+1$, so folgt $x = r'_j = r_i$ sowie $\neg pref_i$, und damit ebenfalls $oth(x)$.

(1)	$\circ oth(x) \rightarrow \bigvee_{j=1}^{N} (x = r'_j \wedge \circ\neg pref_j)$	(pref)
(2)	$\text{exec}\ \lambda_{i+1} \wedge \circ\neg pref_j \rightarrow \neg pref_j \qquad (j = 1, \ldots, N)$	(D5)

(3) exec $\lambda_{i+1} \rightarrow r'_j = r_j \qquad (j = 1, \ldots, N, j \neq i+1)$ (unch)

(4) $r'_j = r_j \wedge x = r'_j \wedge \neg \mathit{pref}_j \rightarrow \mathit{oth}(x) \qquad (j = 1, \ldots, N)$ (pred)

(5) exec $\lambda_{i+1} \wedge x = r'_j \wedge \circ\neg \mathit{pref}_j \rightarrow \mathit{oth}(x) \qquad (j = 1, \ldots, N, j \neq i+1)$ (2)(3)(4)

(6) exec $\lambda_{i+1} \rightarrow r'_{i+1} = r_i \wedge r'_i = r_i$ (act$_{i+1}$)(unch)

(7) exec $\lambda_{i+1} \wedge \mathit{pref}_i \rightarrow \circ \mathit{pref}_i$ (D5)

(8) $\circ \mathit{pref}_i \wedge r'_{i+1} = r'_i \rightarrow \circ \mathit{pref}_{i+1}$ (pref)

(9) exec $\lambda_{i+1} \wedge \neg\circ \mathit{pref}_{i+1} \rightarrow \neg \mathit{pref}_i$ (6)(7)(8)

(10) $x = r'_{i+1} \wedge r'_{i+1} = r_i \wedge \neg \mathit{pref}_i \rightarrow \mathit{oth}(x)$ (oth)(pred)

(11) exec $\lambda_{i+1} \wedge x = r'_{i+1} \wedge \neg\circ \mathit{pref}_{i+1} \rightarrow \mathit{oth}(x)$ (9)(10)

(12) exec $\lambda_{i+1} \wedge \circ \mathit{oth}(x) \rightarrow \mathit{oth}(x)$ (1)(5)(11)

(D7) exec $\lambda_{i+1} \wedge \mathit{min}(x) \rightarrow \circ\exists\, y(y \leq x \wedge \mathit{min}(y))$

Idee: Schritte von Knoten $i+1$ ändern nicht den Wert von r_0 und vergrößern nach (D6) auch nicht die Menge der Werte x, für die $\mathit{oth}(x)$ gilt. Gilt daher vor dem Schritt $\mathit{min}(x)$, und somit insbesondere $\neg \mathit{oth}(r_0 \oplus x)$, so gilt nach dem Schritt immer noch $\neg \mathit{oth}(r_0 \oplus x)$. Aber dann muss sicher $\mathit{min}(y)$ für ein $y \leq x$ gelten.

(1) $\mathit{min}(x) \rightarrow \neg \mathit{oth}(r_0 \oplus x)$ (min)

(2) $\mathit{min}(x) \wedge r'_0 = r_0 \wedge \forall\, y(\circ \mathit{oth}(y) \rightarrow \mathit{oth}(y)) \rightarrow \circ\neg \mathit{oth}(r_0 \oplus x)$ (1)(pred)

(3) $\neg \mathit{oth}(r_0 \oplus x) \rightarrow \exists\, y(y \leq x \wedge \mathit{min}(y))$ (min)(data)

(4) $\circ\neg \mathit{oth}(r_0 \oplus x) \rightarrow \circ\exists\, y(y \leq x \wedge \mathit{min}(y))$ (3)(T25)

(5) exec $\lambda_{i+1} \rightarrow r'_0 = r_0$ (unch)

(6) exec $\lambda_{i+1} \rightarrow \forall\, y(\circ \mathit{oth}(y) \rightarrow \mathit{oth}(y))$ (D6)(pred)

(7) exec $\lambda_{i+1} \wedge \mathit{min}(x) \rightarrow \circ\exists\, y(y \leq x \wedge \mathit{min}(y))$ (2)(4)(5)(6)

Die folgenden Aussagen beziehen sich auf die Schritte von Knoten 0.

(D8) exec $\lambda_0 \wedge \circ \mathit{oth}(x) \rightarrow \mathit{oth}(x) \vee x = r_0$

Idee: Die Annahme $\circ \mathit{oth}(x)$ impliziert, dass $x = r'_j$ gilt für ein $j > 0$ mit $\circ\neg \mathit{pref}_j$. Der Wert von r_j ändert sich nicht durch Schritte von Knoten 0, allerdings kann zuvor pref_j gegolten haben. Wir unterscheiden daher zwei Fälle: Ist $x = r_j = r_0$, so folgt die Aussage unmittelbar. Andernfalls folgt aber $\neg \mathit{pref}_j$ und damit $\mathit{oth}(x)$.

(1) $\circ \mathit{oth}(x) \rightarrow \bigvee_{j=1}^{N} r'_j = x$ (oth)

(2) exec $\lambda_0 \wedge r'_j = x \rightarrow r_j = x \qquad (j = 1, \ldots, N)$ (unch)(pred)

(3) $r_j \neq r_0 \rightarrow \neg \mathit{pref}_j \qquad (j = 1, \ldots, N)$ (pref)

(4) $\neg \mathit{pref}_j \wedge r_j = x \rightarrow \mathit{oth}(x) \qquad (j = 1, \ldots, N)$ (oth)

(5) exec $\lambda_0 \wedge r'_j = x \rightarrow \mathit{oth}(x) \vee x = r_0 \qquad (j = 1, \ldots, N)$ (2)(3)(4)

(6) exec $\lambda_0 \wedge \circ \mathit{oth}(x) \rightarrow \mathit{oth}(x) \vee x = r_0$ (1)(5)

(D9) exec $\lambda_0 \wedge \mathit{min}(x) \wedge x > 0 \rightarrow \circ\exists\, y(y < x \wedge \mathit{min}(y))$

Idee: Dies ist der zentrale Punkt im Beweis der Eigenschaft $\diamond \mathit{stable}$ mittels der Regel (wfo): Schritte von Knoten 0 verkleinern die Differenz von r_0 zum kleinsten „freien“ Wert (der nicht außerhalb des Präfix vorkommt), so lange diese Differenz positiv ist, d.h. so lange der Wert von r_0 nicht außerhalb des Präfix gleicher Registerwerte enthalten ist. Zum Beweis dieser Aussage reicht es zu zeigen, dass der Wert $r'_0 \oplus (x-1) = r_0 \oplus x$ nicht außerhalb des Präfix vorkommt, und gemäß (D8) gilt dies, falls $r'_0 \oplus (x-1) \neq r_0$ gilt. Dies ist allerdings für $x > 0$ der Fall (wegen $M \geq 2$).

(1) exec $\lambda_0 \rightarrow r_0' = r_0 \oplus 1$ (act$_0$)

(2) $min(x) \rightarrow \neg oth(r_0 \oplus x)$ (min)

(3) $x > 0 \rightarrow r_0 \oplus x = (r_0 \oplus 1) \oplus (x - 1)$ (data)

(4) $x > 0 \rightarrow r_0 \oplus x \neq r_0$ (data)

(5) exec $\lambda_0 \wedge \neg oth(r_0' \oplus (x-1)) \wedge r_0' \oplus (x-1) \neq r_0 \rightarrow \neg\bigcirc oth(r_0 \oplus (x-1))$ (D8)

(6) $x > 0 \wedge \neg oth(r_0 \oplus (x-1)) \rightarrow \exists\, y(y < x \wedge min(y))$ (min)(data)

(7) $x > 0 \wedge \neg\bigcirc oth(r_0 \oplus (x-1)) \rightarrow \bigcirc\exists\, y(y < x \wedge min(y))$ (6)(T25)(T15)(ltl6)

(8) exec $\lambda_0 \wedge min(x) \wedge x > 0 \rightarrow \bigcirc\exists\, y(y < x \wedge min(y))$ (1)–(7)

(D10) $min(0) \wedge \text{enabled}_{\lambda_{j+1}} \rightarrow \neg\text{enabled}_{\lambda_0} \qquad (j = 0, \ldots, N-1)$

Idee: Diese Aussage wird gebraucht, um zu zeigen, dass Knoten 0 blockiert ist, so lange der Wert von r_0 nur innerhalb des Präfix vorkommt, aber noch nicht Knoten N erreicht hat. Der Beweis ist einfach: Aus enabled$_{\lambda_{j+1}}$ folgt $oth(r_N)$, und aus $min(0)$ folgt $\neg oth(r_0)$. Also müssen die Werte von r_0 und r_N verschieden sein. In der folgenden Herleitung sei $0 \leq j \leq N-1$.

(1) $r_{j+1} \neq r_j \rightarrow \neg pref_N$ (pref)

(2) $\neg pref_N \rightarrow oth(r_N)$ (oth)

(3) $min(0) \rightarrow \neg oth(r_0)$ (min)

(4) $min(0) \wedge \text{enabled}_{\lambda_{j+1}} \rightarrow r_0 \neq r_N$ (1)(2)(3)

(5) $min(0) \wedge \text{enabled}_{\lambda_{j+1}} \rightarrow \neg\text{enabled}_{\lambda_0}$ (4)

In Eigenschaft (D9) haben wir eine Größe gefunden, die durch die Schritte von Knoten 0 verkleinert wird. Zur Anwendung von (wfo) fehlt noch eine Größe, die durch $\lambda_1, \ldots, \lambda_N$ verkleinert wird. Aus (D2) und (D3) wissen wir, dass jeder solche Schritt enabled$_{\lambda_i}$ falsch macht, während enabled$_{\lambda_j}$ für $1 \leq j < i$ unverändert bleibt. Die Idee besteht darin, daraus einen Term zu konstruieren, der mit jedem Schritt kleiner wird. Dazu betrachten wir Tupel $\langle e_1, \ldots, e_N \rangle$, wobei für $0 \leq i \leq N-1$ gilt

$$e_{i+1} = \begin{cases} 1 & \text{falls enabled}_{\lambda_{i+1}} \text{ gilt} \\ 0 & \text{sonst} \end{cases}$$

Die Ausführung eines λ_{i+1} verringert dann den Wert dieses Tupels bezüglich der lexikographischen Ordnung auf $\{0, 1, \ldots, M\}^N$. Zur Vervollständigung des Beweises erweitern wir das Tupel um eine Komponente e_0, so dass $min(e_0)$ gilt.

Formal erweitern wir die Signatur SIG um eine neue Sorte SEQ, $N+1$ Funktionssymbole $SEL_i^{(SEQ, VAL)}$ für $0 \leq i \leq N$ und das Prädikatensymbol $\preceq^{(SEQ\ SEQ)}$. In der Struktur $\mathbb{S}$ gelte

$$\mathbb{S}(SEQ) = \{0, \ldots, M\}^{N+1} \qquad \text{(also } N+1\text{-Tupel über } |\mathbb{S}|_{VAL}\text{)}$$
$$\mathbb{S}(SEL_i)(\langle m_0, \ldots, m_N \rangle) = m_i$$

ferner sei $\mathbb{S}(\preceq)$ die lexikographische Ordnung auf N-Tupeln; dies ist eine fundierte Ordnung. Damit wird die Regel (wfo) anwendbar, wobei SEQ die Rolle der Sorte wf spielt. Es sei z eine Variable der Sorte SEQ, dann definieren wir die Formeln

$$B \;\equiv\; min(SEL_0(z)) \wedge \bigwedge_{i=1}^{N} \big((\text{enabled}_{\lambda_i} \rightarrow SEL_i(z) = 1) \wedge (\neg\text{enabled}_{\lambda_i} \rightarrow SEL_i(z) = 0)\big)$$

und

$$A \;\equiv\; \Box\neg stable \wedge B$$

(Wir beweisen $\Box\neg stable \rightarrow \Diamond stable$, damit folgt dann aussagenlogisch $\Diamond stable$.)

(1)	exec $\lambda_i \rightarrow$ enabled$_{\lambda_i} \wedge \bigcirc\neg$enabled$_{\lambda_i}$ $\quad (i = 1, \ldots, N)$	(action)(D2)
(2)	exec $\lambda_i \rightarrow \bigwedge_{j=1}^{i-1}$(enabled$_{\lambda_j} \leftrightarrow \bigcirc$enabled$_{\lambda_j}$) $\quad (i = 1, \ldots, N)$	(D3)
(3)	exec $\lambda_i \wedge B \rightarrow \bigcirc\exists\, \bar{z}(\bar{z} \prec z \wedge B_z(\bar{z}))$ $\quad (i = 1, \ldots, N)$	(1)(2)(D7)
(4)	$\neg stable \rightarrow \bigvee_{i=1}^{N}$ enabled$_{\lambda_i}$	(D1)
(5)	exec $\lambda_0 \rightarrow$ enabled$_{\lambda_0}$	(action)
(6)	exec $\lambda_0 \wedge \neg stable \rightarrow \forall\, x(min(x) \rightarrow x > 0)$	(4)(5)(D10)(data)
(7)	exec $\lambda_0 \wedge \neg stable \wedge B \rightarrow \bigcirc\exists\, \bar{z}(\bar{z} \prec z \wedge B_z(\bar{z}))$	(6)(D9)
(8)	exec $\lambda_i \wedge A \rightarrow \bigcirc(\Diamond stable \vee \exists\, \bar{z}(\bar{z} \prec z \wedge A_z(\bar{z}))$ $\quad (i = 0, \ldots, N)$	(3)(7)
(9)	$\bigvee_{i=0}^{N}$ exec λ_i	(D1)(progress)
(10)	$(\exists\, z\ A) \rightarrow \Diamond stable$	(wfo)(8)(9)(T7)(T11)
(11)	$(\exists\, z\ B) \rightarrow \Diamond stable$	(prop)(10)
(12)	$\exists\, z\ B$	(min-ex)(ltl4)
(13)	$\Diamond stable$	(11)(12)
(14)	$\Diamond stable \rightarrow \Diamond\Box stable$	(D4)(T26)
(15)	$\Diamond\Box stable$	(13)(14)

Literaturhinweise

[1] E. W. Dijkstra: Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), S. 643–644, Nov. 1974.

[2] E. W. Dijkstra: A belated proof of self-stabilization. *Distributed Computing* 1, S. 5–6, 1986.

[3] S. Qadeer, N. Shankar: Verifying a Self-Stabilizing Mutual Exclusion Algorithm. D. Gries, W.-P. de Roever (Hrsg.): *Programming Concepts and Methods*, S. 424–443, Chapman & Hall, 1998