AG Betriebssysteme FB3
Jan Peleska, Kirsten Berkenkötter

Implizite Spezifikation

1 Kreissäge

Das Ein- und Ausschalten der Kreissäge wird über die Events *on* und *off* signalisiert. Die Säge hat einen Schutzmechanismus, der hoch- und runtergeklappt werden kann. Dies wird über die Events *guard.up* und *guard.down* modelliert.

1. Der Schutz kann innerhalb von 5 Zeiteinheiten nach dem Ausschalten nicht abgenommen werden:

   $\forall t \in \mathbb{R}^+ \bullet (t, guard.up) in\, s \Rightarrow off \notin \sigma(s \uparrow (t, t+5))$

2. Der Schutz alterniert zwischen den beiden möglichen Positionen, im Startzustand gilt *guard.down*:

   $s \downarrow \{guard.up\} \leq s \downarrow \{guard.down\} \leq s \downarrow \{guard.up\} + 1$

3. Ein- und Ausschalten der Säge ist alternierend, zu Beginn ausgeschaltet:

   $s \downarrow \{off\} \leq s \downarrow \{on\} \leq s \downarrow \{off\} + 1$

4. Die Säge kann nicht eingeschaltet werden, wenn der Schutz nicht geschlossen ist:

   $off(s) \Leftrightarrow s \downarrow on = s \downarrow off$
   $guard\_down(s) \Leftrightarrow s \downarrow guard.down = s \downarrow guard.up$
   $\neg off(s) \Rightarrow guard\_down(s)$

5. Der Schutz muss bereits 8 Zeiteinheiten geschlossen sein, bevor die Säge eingeschaltet werden kann:

   $\neg off(s) \Rightarrow end(s \upharpoonright \{guard.down\}) + 8 \leq end(s \upharpoonright \{on\}) \vee s \upharpoonright \{guard.up\} = <>$

2 Fischer's Protokoll

2.1 Vereinfachte Version

Annahme Zunächst wird eine vereinfachte Form des Protokolls verwendet: alle versuchen, in die kritische Region zu kommen, aber nur einer schafft es. k wird niemals auf 0 gesetzt.

1. Wir wollen zeigen, dass nur ein Prozess die kritische Region betritt:

   $SYSTEM\, sat\, s \downarrow \{enter\} \leq 1$

2. Ein Prozess beschreibt die shared-Variable k. Hier soll gelten (zugesichert aufgrund des verwendeten Betriebssystems), dass ein gelesener Wert immer der zuletzt geschriebene ist, oder – falls noch nicht geschrieben wurde – 0:

   $$\begin{aligned} VARPROCESS\, sat\, s \quad = \quad & s1 \frown < (t, read.j) > \frown s2 \Rightarrow \\ & (last(s1 \upharpoonright write) = write.j \wedge j \neq 0) \vee ((s1 \upharpoonright write) = <> \wedge j = 0) \end{aligned}$$

   In diesem vereinfachten Fall können wir also auch sagen:

$VARPROCESS\ sat\ read.0\ at\ t \Rightarrow no\ write\ at\ [0,t) \ \vee \ write.0\ in\ s$

Wenn dies für den Prozess *VARPROCESS* gilt, muss es auch für den Prozess *SYSTEM* gelten, da beide Prozesse über *read*– und *write*–Events synchronisiert sind:

$SYSTEM\ sat\ read.0\ at\ t \Rightarrow no\ write\ at\ [0,t) \ \vee \ write.0\ in\ s$

3. Wir spezifizieren, dass jeder Prozess nur seine eigene ID schreibt:

   $PROCESS_i\ sat\ \sigma(s \upharpoonright write) \subseteq \{write.i\}$

   Daraus können wir folgern, dass für $i \neq j \Rightarrow enter.i \notin \sigma(PROCESS_j)$.

   Für die Parallelkomposition gilt also: $i \neq j \Rightarrow enter.i \notin \sigma(|||_{j \neq i}\ PROCESS_j)$.

   Für die Parallelkomposition aller Prozesse gilt also, dass jedes *write.i* aus dem Prozess $PROCESS_i$ stammt:

   $PROCESSES\ sat\ \forall i,j \in \{1..2\} \bullet i \neq j \Rightarrow write.i \notin \sigma(|||_{j \neq i}\ PROCESS_j)$

4. Jeder Prozess schreibt seine ID maximal einmal:

   $PROCESS_i\ sat\ s \downarrow write.i \leq 1$

   Damit gilt also für die Parallelkomposition unter Berücksichtigung von 3. :

   $PROCESSES\ sat\ s \downarrow write.i \leq 1$ und auch

   $SYSTEM\ sat\ s \downarrow write.i \leq 1$

   Das bedeutet u.a., dass kein Prozess ein *write.0* macht, da kein Prozess die ID 0 hat:

   $PROCESS_i\ sat\ no\ write.0\ in\ s$

   Dies muss natürlich auch für die Parallelkomposition gelten sowie für das Gesamtsystem:

   $SYSTEM\ sat\ no\ write.0\ in\ s$

   Mit 2. können wir dann folgern:

   $SYSTEM\ sat\ read.0\ at\ t \Rightarrow no\ write.\{1..2\}\ at\ [0,t)$

5. Jeder Prozess kann nur für sich selbst ein *enter* machen:

   $PROCESS_i\ sat\ \sigma(s \upharpoonright enter) \subseteq \{enter.i\}$

   Für die Parallelkomposition gilt also analog zu 3.:

   $PROCESSES\ sat\ \forall i,j \in \{1..2\} \bullet i \neq j \Rightarrow enter.i \notin \sigma(|||_{i \neq j}\ PROCESS_j)$

6. Jeder Prozess macht nur einmal ein *enter*:

   $PROCESS_i\ sat\ s \downarrow enter.i \leq 1$

   Für die Parallelkomposition gilt also:

   $PROCESSES\ sat\ s \downarrow enter.i \leq 1$

   $SYSTEM\ sat\ s \downarrow enter.i \leq 1$

7. Nun müssen die Zeitbedingungen spezifiziert werden. Wenn die kritische Region von einem Prozess erreicht wurde, d.h. ein *enter*–Event aufgetreten ist, müssen vorher *read*– und *write*–Events mit den festgelegten Zeitabständen aufgetreten sein:

$$\begin{aligned} PROCESS_i\ sat\ T_i \quad = \quad & enter.i\ in\ s \Rightarrow \\ & \exists\, t_i, t_i', t_i'' \bullet read.0\ at\ t_i \wedge write.i\ at\ t_i' \wedge read.i\ at\ t_i'' \wedge \\ & t_i' - t_i = a \wedge t_i'' - t_i' = b \wedge t_i \leq t_i' \leq t_i'' \end{aligned}$$

   Die Prozesse $PROCESS_i$ laufen alle unsynchronisiert. Das Event *enter.i* wird nur vom jeweiligen Prozess $PROCESS_i$ ausgeführt. Aus diesem Grund erfüllt auch die Parallelkomposition aller Prozesse $PROCESS_i$ diese Spezifikation. Diese ist über alle Events *read* und *write* mit dem Prozess *VARPROCESS* synchronisiert, so dass diese Komposition *SYSTEM* ebenfalls die Spezifikation erfüllen muss.

8. An dieser Stelle wissen wir nun folgendes: *SYSTEM* garantiert,

   - dass der gelesene Wert immer der zuletzt geschriebene ist oder am Anfang die 0 (1).
   - dass jede ID nur einmal geschrieben wird (2,3).
   - dass 0 nur gelesen werden kann, wenn noch kein Prozess seine ID geschrieben hat (4).
   - dass jeder Prozess i nur einmal *enter.i* ausführen kann (5,6).
   - dass die Zeitbedingungen eingehalten werden (7).

   Nun muss gezeigt werden, dass nur ein einziges *enter* geschieht, wie in 1. gefordert wird. Wie zeigen dies, indem wir zunächst annehmen, dass zweimal ein *enter* auftritt und diese Annahme zum Widerspruch führen.

   s_0 sei eine beliebige Trace, in der *enter.i* und *enter.j* vorkommen mit $i \neq j$ (lt. 5. und 6.). Dann müsste gelten:

   $read.0\,at\,t_i \wedge write.i\,at\,t_i' \wedge read.i\,at\,t_i'' \wedge$
   $read.0\,at\,t_j \wedge write.j\,at\,t_j' \wedge read.j\,at\,t_j'' \wedge$
   $t_i' - t_i = a \wedge t_i'' - t_i' = b \wedge t_i \leq t_i' \leq t_i'' \wedge t_j' - t_j = a \wedge t_j'' - t_j' = b \wedge t_j \leq t_j' \leq t_j''$

   write.i und *write.j* müssen in der Trace in irgendeiner Weise geordnet sein, selbst wenn sie zum gleichen Zeitpunkt registriert werden. Wir nehmen (ohne Verlust der Allgemeinheit) an, dass *write.j* das letzte Auftreten von *write* ist:

   $write.j = last(s_0 \restriction \{write.i, write.j\})$

   Es gilt nun für s_0:

   $read.0\,at\,t_i$
   $read.0\,at\,t_j\,und$
   $write.i\,at\,t_i'$

   Also muss gelten:

   $t_j \leq t_i'$

   Für die Trace s_0 sieht es nun folgendermaßen aus:

   $s_0 = s_1 \frown <(t_i'', read.i)> \frown s_2 \wedge write.i = last(s_1 \restriction write)$

   Das letzte *write* in s_1 muss *write.i* gewesen sein, damit *read.i* stattfinden kann (nach 2.).

   Wir haben ferner festgelegt, dass *write.j* nach *write.i* passiert. In s_1 kann es demnach nicht auftauchen, sondern muss in s_2 sein. Daraus folgt:

   $t_i'' \leq t_j'$

   Es gilt also:

   (a) $t_j \leq t_i'$
   (b) $t_i'' \leq t_j'$
   (c) $b > a$ (lt. Voraussetzung für Fischer's Protocol)

   Daraus folgt:

   $t_j \leq t_i' = t_i'' - b$
   $\Rightarrow t_j \leq t_j' - b$
   $\Rightarrow t_j < t_j' - a$, da b > a

   Diese Aussage steht im Widerspruch zur Annahme $t_j = t_j' - a$ Dadurch ist bewiesen, dass nur ein *enter*–Event auftritt, wie in 1. gefordert.

   Die einzigen Zusicherungen, die unser Programm erfüllen muss, ist dass jeder Prozss:

   (a) nur seine eigene ID schreibt,

(b) seine ID maximal einmal schreibt,

(c) nur für sich selbst *enter* aufruft,

(d) enter nur einmal aufruft, und

(e) die Zeitbedingung einhält.

Zusammen mit der Zusicherung, dass der zuletzt gelesene Wert der zuletzt geschriebene ist und zu Beginn den Wert 0 hat, genügt dies völlig aus, um zu garantieren, dass nur ein *enter* auftritt.

2.2 Allgemeine Version

Auf der vereinfachten Version des Protokolls aufbauend, soll nun gezeigt werden, dass auch die allgemeine Version wie erwartet funktioniert, sofern die einzelnen Komponenten des Systems ihrer Spezifikation entsprechen.

1. Zu zeigen ist diesmal, dass immmer nur ein Prozess in der kritischen Region ist (wenn die Zusicherung gilt, dass jeder Prozess nur für sich selbst ein *enter* macht):

   $SYSTEM\, sat\, s \downarrow exit \leq s \downarrow enter \leq s \downarrow exit + 1$

   In zwei Teilschritten:

   $SYSTEM\, sat\, s \downarrow exit \leq s \downarrow enter$

   $SYSTEM\, sat\, s \downarrow exnter \leq s \downarrow exit + 1$

2. Die Spezifikation für den Prozess, der die shared-Variable k modelliert sieht ähnlich aus wie zuvor, nur dass auch der Wert 0 geschrieben werden kann:

$$\begin{aligned} VARPROCESS\, sat\, s \quad = \quad & s1 \frown < (t, read.j) > \frown s2 \Rightarrow \\ & (last(s1 \upharpoonright write) = write.j \wedge j \neq 0) \vee \\ & (((last(s1 \upharpoonright write) = write.0) \vee (s \upharpoonright write) = <>) \wedge j = 0) \end{aligned}$$

   Es gilt also:

   $VARPROCESS\, sat\, read.0\, at\, t \Rightarrow no\, write\, at\, [0, t) \vee write.0\, in\, s$

   Wenn dies für den Prozess *VARPROCESS* gilt, muss es auch für den Prozess *SYSTEM* gelten, da beide Prozesse über *read*– und *write*–Events synchronisiert sind:

   $SYSTEM\, sat\, read.0\, at\, t \Rightarrow no\, write\, at\, [0, t) \vee write.0\, on\, s$

$$\begin{aligned} SYSTEM\, sat\, s \quad = \quad & s1 \frown < (t, read.j) > \frown s2 \Rightarrow \\ & (last(s1 \upharpoonright write) = write.j \wedge j \neq 0) \vee \\ & (((last(s1 \upharpoonright write) = write.0) \vee (s \upharpoonright write) = <>) \wedge j = 0) \end{aligned}$$

3. Es gilt weiterhin, dass jeder Prozess nur seine eigene ID schreibt:

   $PROCESS_i\, sat\, \sigma(s \upharpoonright write) \subseteq \{write.i\}$

   Daraus können wir folgern, dass für $i \neq j \Rightarrow enter.i \notin \sigma(PROCESS_j)$.

   Für die Parallelkomposition gilt also: $i \neq j \Rightarrow enter.i \notin \sigma(|||_{j \neq i}\, PROCESS_j)$.

   Für die Parallelkomposition aller Prozesse gilt also, dass jedes *write.i* aus dem Prozess $PROCESS_i$ stammt:

   $PROCESSES\, sat\, \forall i, j \in \{1..2\} \bullet i \neq j \Rightarrow write.i \notin \sigma(|||_{j \neq i}\, PROCESS_j)$

4. Außer seiner eigenen ID kann jeder Prozss auch 0 schreiben. Die Bedingungen, dass nur einmal geschrieben wird, sowie dass niemals der Wert 0 geschrieben wird, gelten natürlich nicht mehr:

   $PROCESS_i \, sat \, \sigma(s \upharpoonright write) \subseteq \{write.i, write.0\}$

   $PROCESSES \, sat \, \forall i, j \in \{1..2\} \bullet i \neq j \Rightarrow write.i \notin \sigma(|||_{i \neq j} \, PROCESS_j)$

5. Jeder Prozess kann nur für sich selbst ein *enter* machen:

   $PROCESS_i \, sat \, \sigma(s \upharpoonright enter) \subseteq \{enter.i\}$

   Für die Parallelkomposition gilt also analog zu 3.:

   $PROCESSES \, sat \, \forall i, j \in \{1..2\} \bullet i \neq j \Rightarrow enter.i \notin \sigma(|||_{i \neq j} \, PROCESS_j)$

6. Jeder Prozess kann nur für sich selbst ein *exit* machen:

   $PROCESS_i \, sat \, \sigma(s \upharpoonright exit) \subseteq \{exit.i\}$

   Für die Parallelkomposition gilt also analog zu 3.:

   $PROCESSES \, sat \, \forall i, j \in \{1..2\} \bullet i \neq j \Rightarrow exit.i \notin \sigma(|||_{i \neq j} \, PROCESS_j)$

7. Für jeden Prozess sind *enter* und *exit* alternierend:

   $PROCESS_i \, sat \, s \downarrow exit.i \leq s \downarrow enter.i \leq s \downarrow exit.i + 1$

   Mit den vorigen Bedingungen muss deshalb auch gelten:

   $PROCESSES \, sat \, \forall i \in \{1, 2\} \downarrow exit.i \leq s \downarrow enter.i \leq s \downarrow exit.i + 1$

8. *write*.0 wird ausschließlich nach einem *exit* ausgeführt:

   $PROCESS_i \, sat \, \forall t \in \mathbb{R}^+ \bullet (t, write.0) ins \Rightarrow s = s1^\frown < (t, write.0) > {}^\frown s2 \wedge last(s1) = exit$

9. Die Zeitbedingung gilt wie vorher:

$$\begin{aligned} PROCESS_i \, sat \, T_i \quad = \quad & enter.i \, in \, s \Rightarrow \\ & \exists \, t_i, t_i', t_i'' \bullet read.0 \, at \, t_i \wedge write.i \, at \, t_i' \wedge read.i \, at \, t_i'' \wedge \\ & t_i' - t_i = a \wedge t_i'' - t_i' = b \wedge t_i \leq t_i' \leq t_i'' \end{aligned}$$

10. Nun muss gezeigt werden, dass immer nur ein Prozess in der kritischen Region ist. Wir wissen aus (2), dass der gelesene Wert immer der zuletzt geschriebene ist (bzw. 0 zu Beginn). Jeder Prozess schreibt auch nur seine eigene Id und nicht die eines anderen (3,4) und *entry.i*, *exit.i* für seine PID(5,6). Dies tut er nur abwechselnd (7).

    Die einzige Möglichkeit ist also, dass ein anderer Prozess *j* ein *entry.j* macht, nachdem Prozess *i* es für sich getan hat. Dazu muss dieser Prozess read.j für seine ID machen. Es gilt hier also wieder der gleich Beweis wie im vereinfachten Fall, der zum Wiederspruch führt.

    Wir wissen also, dass immer nur ein Prozess ein entry macht. Die zu zeigende Bedingung gilt also, da $0 \leq 1 \leq 1$. Für jeden Prozess sind *entry* und *exit* alternierend beginnend mit *entry* (7), dass heisst, dass nur ein *exit* von dem Prozess gemacht werden kann, der in der kritischen Region ist. Auch hier gilt die zu zeigende Bedingung: $1 \leq 1 \leq 2$. Vor jedem *entry* muss ein *read.0* passieren, dies geht nur zu Beginn oder nach einem *write.0* (2,9). Das *write.0* passiert aber nur nach einem exit. Damit ist der "Anfangszustand" wiederhergestellt, für jedes weitere *entry* und *exit* gilt dies analog. Damit gilt die in (1) aufgestellte Behauptung.

    Es müssen insgesamt also folgende Zusicherungen für jeden Prozess gelten:

    (a) Jeder Prozess schreibt nur seine eigene ID oder $0(3, 4)$.

    (b) Jeder Prozess schreibt *enter.i* und *exit.i* nur für seine eigene ID (5,6).

    (c) *enter.i* und *exit.i* sind alternierend (7).

(d) Nach einem *exit.i* wird *write.0* geschrieben (8).

(e) Die Zeitbedingung wird eingehalten (9).

Zusätzlich muss natürlich wieder gelten, dass der zuletzt gelesene Wert der zuletzt geschriebe ist (2).