(AP 3.8) Deliverable Arbeitspaket 3.8

27. Juni 2016

In diesem Dokument werden die Testszenarien protokolliert, die zum Testen der Integration der Teilsysteme verwendet wurden. Das erste Testszenario besteht aus einem einfachen Sortieralgorithmus und das zweite aus einem Producer / Consumer-Setting mit einem Ringpuffer, welches mehrere Threads verwendet.

1 Sortieralgorithmus

1.1 Beschreibung des Szenarios

Als erstes Testszenario wurde ein Quicksort implementiert, der ein festes Array sortiert. Dieser Quicksort wird viele male aufgerufen. Das gesamte Testszenario läuft in einem einzigen Thread und ist damit mit Absicht sehr einfach gehalten, um die generelle Funktion des Gesamtsystems zu testen.

1.2 Eigenschaft

Die Eigenschaft, die für dieses Szenario spezifiziert wurde, beschreibt ein typisches Verhalten eines Quicksort-Algorithmus. Ein solcher Algorithmus sollte nach maximal $\frac{n \cdot (n-1)}{2}$ rekursiven Aufrufen fertig sein, wobei n die Länge des Arrays ist, das sortiert werden soll. Deswegen beschreibt diese Eigenschaft, dass es bei jeder Iteration maximal diese Anzahl an Aufrufen geben darf, bis die nächste Iteration gestartet wird.

```
define onTrue(x) := filter(changeOf(x), x)

define exec(x) := filter(on(input_vector_ir_ids),
  eq(mrv(input_vector_ir_ids, 0), constantSignal(x)))

define callQS := exec(17) -- call to quick_sort()
define repeat := exec(45) -- start new iteration round

define callCount := eventCount(callQS, repeat)
define error := onTrue(gt(callCount, constantSignal(div(mul(20,19),2))))
```

2 Ringpuffer

2.1 Beschreibung des Szenarios

Als zweites Testszenario wurde ein Ringpuffer der Größe 5 betrachtet, in den Elemente geschrieben und aus dem Elemente herausgelesen und verarbeitet werden können. Es gibt genau einen Producer, der Elemente in den Ringpuffer hineinschreiben kann. Außerdem gibt es eine beliebige Anzahl an Consumern, die Elemente aus dem Ringpuffer herauslesen und auf eine beliebige Art verarbeiten. Sowohl der Producer als auch die Consumer sind im zugehörigen C-Programm Threads und laufen damit parallel zueinander. Der Ringpuffer besteht aus Speicherstellen. Des Weiteren können der Producer und die Consumer angehalten und wieder gestartet werden.

2.2 Eigenschaften

Für dieses Test-Szenario wurden drei unterschiedliche Eigenschaften spezifiziert, die mögliche Fehler des Ringpuffers beschreiben. Dazu gehört eine Eigenschaft über die Größe des Puffers, eine Echtzeiteigenschaft sowie eine temporallogische Eigenschaft.

2.2.1 Größe des Puffers

Diese Eigenschaft beschreibt zwei mögliche, ähnliche Fehlerfälle dieses Szenarios. Zum Einen, dass nicht mehr Elemente verarbeitet werden dürfen, als in den Ringpuffer geschrieben wurden. Zum Anderen wird auch beschrieben, dass die Größe des Puffers, die mit 5 angegeben war, 5 nicht überschreitet. Ansonsten gibt es einen Überlauf des Puffers und Daten gehen verloren.

```
--
-- Macros
--
define onIf(trig, cond) := filter(on(trig), cond)
define geq(x,y) := not(lt(x,y))
define lt(x,y) := gt(y,x)
define exec(x) := filter(on(input_vector_ir_ids),
  eq(mrv(input_vector_ir_ids, 0), constantSignal(x)))

--
-- Inputs
--
define writeElement := exec(2)
define processElement := exec(13)

--
-- Spec
--
define diffProcWrite := sub(eventCount(processElement),
  eventCount(writeElement))
```

```
define doubleProcessing := onIf(processElement,
  geq(diffProcWrite,constantSignal(1)))

define diffWriteProc := sub(eventCount(writeElement),
  eventCount(processElement))
define bufferOverflow := onIf(writeElement,
  geq(diffWriteProc,constantSignal(6)))
```

2.2.2 Verarbeitungszeit

Diese Eigenschaft beschreibt, dass die Verarbeitungszeit eines Consumers für ein Element nicht mehr als zwei Sekunden beträgt. Ansonsten braucht der Consumer zu lang. Dafür wird die Zeit zwischen dem Aufruf der Verarbeitungsmethode und dessen Rückgabe betrachtet. Es wird davon ausgegangen, dass es genau drei Consumer gibt.

```
--
-- Macros
--
define onIf(trig, cond) := filter(on(trig), cond)
define onYield(trig, value) := ifThen(trig,value)
define onIfYield(trig, cond, v) := onYield(filter(trig, cond), v)

define sample(s, e) := ifThen(e, s)

define geq(x,y) := not(lt(x,y))
define lt(x,y) := gt(y,x)
define leq(x,y) := not(gt(x,y))
define ne(x,y) := not(eq(x,y))

define onTrue(x) := onIf(changeOf(x), x)

define now := mrv(input_vector_timestamps,0)
define inPast(time, event) := leq(sub(now,mrv(timestamps(event), 0)),
  constantSignal(time))

define owner_valid := filter(input_vector_RegChangeMessageID,
  eq(mrv(input_vector_RegChangeMessageID, 1), constantSignal(0)))
define threadID := mrv(ifThen(owner_valid,
  mrv(input_vector_RegChangeMessageValue, 0)), 0)

define exec(x) := filter(on(input_vector_ir_ids),
  eq(mrv(input_vector_ir_ids, 0), constantSignal(x)))

--
-- Inputs
--
define startC1 := onIfYield(exec(1),
  eq(threadID, constantSignal(1)), constantSignal(true))
```

```
define startC2 := onIfYield(exec(1),
  eq(threadID, constantSignal(2)), constantSignal(true))
define startC3 := onIfYield(exec(1),
  eq(threadID, constantSignal(3)), constantSignal(true))
define endC1 := onIfYield(exec(1),
  eq(threadID, constantSignal(1)), constantSignal(false))
define endC2 := onIfYield(exec(1),
  eq(threadID, constantSignal(2)), constantSignal(false))
define endC3 := onIfYield(exec(1),
  eq(threadID, constantSignal(3)), constantSignal(false))

--
-- Spec
--
define errorC1 := onIf(endC1, not(inPast(2000, startC1)))
define errorC2 := onIf(endC2, not(inPast(2000, startC2)))
define errorC3 := onIf(endC3, not(inPast(2000, startC3)))

define error := merge(merge(errorC1,errorC2),errorC3)
```

2.2.3 Anhalten der Consumer

Diese Eigenschaft beschreibt das korrekte Verhalten, wenn die Consumer angehalten und danach wieder gestartet werden. Sobald die Consumer angehalten wurden, darf der Lesezeiger für den Ringpuffer von den Consumern nicht verändert werden, bis diese wieder gestartet wurden. Ansonsten würde die Consumer aus dem Ringpuffer lesen, obwohl sie angehalten wurden.

```
--
-- Macros
--
define prop(e1,e2) := mrv(merge(ifThen(e1, constantSignal(true)),
  ifThen(e2, constantSignal(false))), false)
define exec(x) := filter(on(input_vector_ir_ids),
  eq(mrv(input_vector_ir_ids, 0), constantSignal(x)))

--
-- Input
--
define readPointerChanged := exec(1)
define stopConsumer := exec(2)
define startConsumer := exec(3)

--
-- Spec
--
define clk := occursAny(occursAny(stopConsumer, readPointerChanged),
  startConsumer)
define stop := prop(stopConsumer, clk)
```

```
define start := prop(startConsumer, clk)
define change := prop(readPointerChanged, clk)

define monitor_output := monitor(
  "always(p1 implies (not(p2) until p3))",
  p1 := stop,
  p2 := change,
  p3 := start,
  clock := clk
)
```