(AP 3.8) Deliverable Arbeitspaket 3.8

27. Juni 2016

1 Beschreibung des Szenarios

Als Szenario wird 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.

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.

2.1 Größe des Puffers

Diese Eigenschaft beschreibt, dass die Größe des Puffers, die mit 5 angegeben war, nicht 5 überschreitet. Ansonsten gibt es einen Überlauf der Puffers und Daten gehen verloren.

```
--
-- Macros
--
define onIf(trig, cond) := filter(trig, cond: Signal<Boolean>)
define geq(x,y) := not(lt(x,y))
define lt(x,y) := gt(y,x)

--
-- Inputs
--
--define writeElement: Events<Unit> := instruction_executions("main.c:49")
define writeElement: Events<Unit> := tracePointID
```

```
--define processElement: Events<Unit> := function_calls("main.c:process_data")
define processElement: Events<Unit> := tracePointID

--
-- Spec
--
define diffProcWrite := sub(eventCount(processElement), eventCount(writeElement))
--define error := on processElement if geq(diffProcWrite,1)
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 Verarbeitungszeit

Diese Eigenschaft beschreibt, dass die Verarbeitungszeit eines Consumers für ein Element nicht mehr als zwei Sekunden beträgt. Ansonsten braucht der Consumer zu lang.

```
--
-- General macros
--
define onIf(trig, cond) := filter(trig, cond: Signal<Boolean>)
define onYield(trig, value) := ifThen(trig,value)
define on(trig, cond, value) := onYield(onIf(trig, cond: Signal<Boolean>), value)

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

define geq(x,y) := not(lt(x,y)): Signal<Boolean>
define lt(x,y) := gt(y,x): Signal<Boolean>
define leq(x,y) := not(gt(x:Signal<Int>,y:Signal<Int>):Signal<Boolean>): Signal<Boolean>
define ne(x,y) := not(eq(x,y): Signal<Boolean>): Signal<Boolean>

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

--
-- Coniras specific definitions
--

define now: Signal<Int> := mrv(input_vector_timestamps,0)
define inPast(time, event) := leq(
sub(
now: Signal<Int>,
mrv(timestamps(event): Events<Int>, 0: Int):Signal<Int>
): Signal<Int>,
constantSignal(time): Signal<Int>
): Signal<Boolean>

-- HW implementation would be more reasonable since signals are represented in
```

```
-- terms of update events anyway!
define changeOf(s, initial) := onIf(anyEvent, ne(s, mrv(delay(sample(s, anyEvent)), initial))) --assu

--
-- Inputs
--

define ids := mrv(input_vector_ownerships, 0)

--define startC1 := on(function_calls("main.c:process_data"), eq(ids, constantSignal(1)), --constantSignal(true)
define startC1 := on(tracePointID, eq(ids, constantSignal(1)), constantSignal(true))
--define startC2 := on(function_calls("main.c:process_data"), eq(ids, constantSignal(2)), --constantSignal(true)
define startC2 := on(tracePointID, eq(ids, constantSignal(2)), constantSignal(true))
--define startC3 := on(function_calls("main.c:process_data"), eq(ids, constantSignal(3)), --constantSignal(true)
define startC3 := on(tracePointID, eq(ids, constantSignal(3)), constantSignal(true))
--define endC1 := on(function_returns("main.c:process_data"), eq(ids, constantSignal(1)), --constantSignal(false)
define endC1 := on(tracePointID, eq(ids, constantSignal(1)), constantSignal(false))
--define endC2 := on(function_returns("main.c:process_data"), eq(ids, constantSignal(2)), --constantSignal(false)
define endC2 := on(tracePointID, eq(ids, constantSignal(2)), constantSignal(false))
--define endC3 := on(function_returns("main.c:process_data"), eq(ids, constantSignal(3)), --constantSignal(false)
define endC3 := on(tracePointID, eq(ids, constantSignal(3)), constantSignal(false))

--
-- Spec
--

------- inFuture is not implemented! -------------
--define errorC1 := onIf(startC1, not(inFuture(endC1,2s)))
--define errorC2 := onIf(startC2, not(inFuture(endC2,2s)))
--define errorC3 := onIf(startC3, not(inFuture(endC3,2s)))
-------

-- not as accurate alternative --
define errorC1 := onIf(endC1, not(inPast(2000: Int, startC1): Signal<Boolean>): Signal<Boolean>): Even
define errorC2 := onIf(endC2, not(inPast(2000: Int, startC2): Signal<Boolean>): Signal<Boolean>): Even
define errorC3 := onIf(endC3, not(inPast(2000: Int, startC3): Signal<Boolean>): Signal<Boolean>): Even

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

-- more accurate alternative (not semantically equivalent,
-- yet operationally equivalent in the Coniras system)
define runningC1 := mrv(merge(startC1, endC1): Events<Boolean>, false): Signal<Boolean>
define timeOutC1 := on(anyEvent, and(runningC1,not(inPast(2000, startC1): Signal<Boolean>): Signal<Boo
define earlyErrorC1 := onTrue(mrv(merge(timeOutC1, neg(startC1): Events<Boolean>): Events<Boolean>, fa

define runningC2 := mrv(merge(startC2, endC2): Events<Boolean>, false): Signal<Boolean>
define timeOutC2 := on(anyEvent, and(runningC2,not(inPast(2000, startC2): Signal<Boolean>): Signal<Boo
define earlyErrorC2 := onTrue(mrv(merge(timeOutC2, neg(startC2): Events<Boolean>): Events<Boolean>, fa

define runningC3 := mrv(merge(startC3, endC3): Events<Boolean>, false): Signal<Boolean>
```

```
define timeOutC3 := on(anyEvent, and(runningC3,not(inPast(2000, startC3): Signal<Boolean>): Signal<Boo
define earlyErrorC3 := onTrue(mrv(merge(timeOutC3, neg(startC3): Events<Boolean>): Events<Boolean>, fa

define earlyError := merge(merge(earlyErrorC1,earlyErrorC2),earlyErrorC3)
```

2.3 Anhalten der Consumer

Diese Eigenschaft beschreibt das korrekte Verhalte eines Moduswechsels. Sobald die Consumer angehalten wurden, darf der Lesepointer für den Ringbuffer von den Consumern nicht verändert werden, bis diese wieder gestartet wurden. Ansonsten würde die Consumer aus dem Ringbuffer lesen, obwohl sie angehalten sind.

```
--
-- Macros
--
define prop(e1,e2) := mrv(merge(ifThen(e1, constantSignal(true)), ifThen(e2, constantSignal(false))),

--
-- Input
--
--define readPointerAddr := variable_values("main.c:read_idx")
define readPointerChanged := tracePointID
--define stopConsumer := function_calls("main.c:stopConsumers")
define stopConsumer := tracePointID
--define startConsumer := function_calls("main.c:startConsumers")
define startConsumer := tracePointID


--
-- Spec
--
--define readPointerChanged := changeOf(readPointerAddr)
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
)

-- out monitor_output
```