# (AP 3.8) Deliverable Arbeitspaket 3.8

28. 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 Produzent-Konsument-Kreislauf, in welchem mehrere Threads über einen Ringpuffer kommunizieren.

## 1 Sortieralgorithmus

### 1.1 Beschreibung des Szenarios

Im ersten Testszenario wird ein festes Array mittels Quicksort sortiert. Der Sortiervorgang wird viele Male wiederholt. Das Testszenario ist mit Absicht sehr einfach gehalten, um die generelle Funktion des Gesamtsystems zu testen. Insbesondere verwendet das Programm keinerlei Bibliotheken.

### 1.2 Eigenschaft

Die Eigenschaft, die für dieses Szenario spezifiziert wurde, beschreibt eine typisches Erwartung an den Quicksort-Algorithmus: Er sollte nach maximal $\frac{n \cdot (n-1)}{2}$ rekursiven Aufrufen terminieren, wobei $n$ die Länge des Arrays ist, das sortiert werden soll. Die folgende Eigenschaft beschreibt, dass es bei jeder Wiederholung maximal diese Anzahl an Aufrufen geben darf, bis die nächste Wiederholung gestartet wird.

```
--
-- Macros
--
define onTrue(x) := onIf(changeOf(x), x)

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

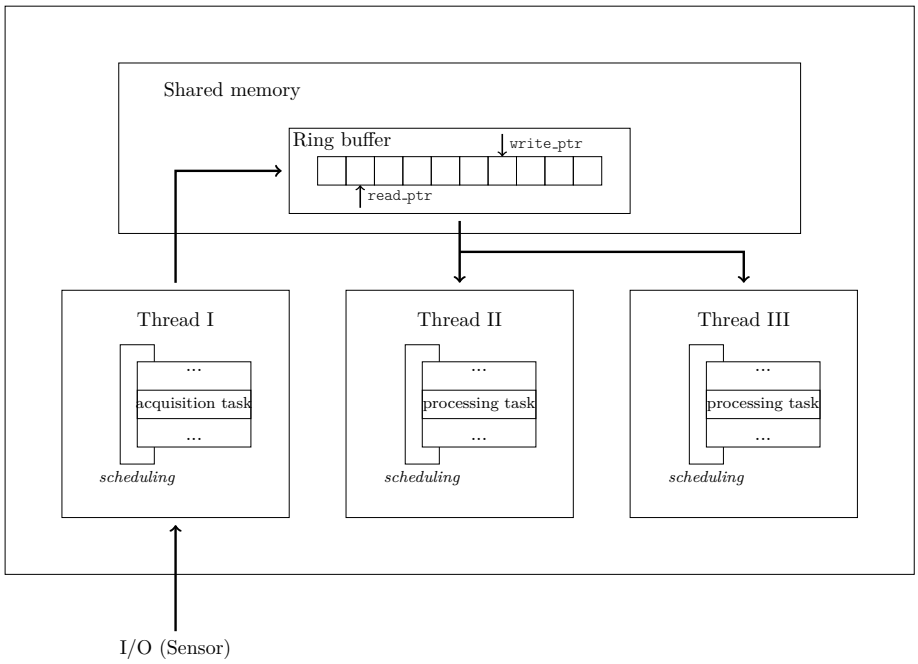


Abbildung 1: Übersichtsbild für die Architektur des Ringpuffer-Testszenarios mit einem Produzenten (Thread I) sowie zwei Konsumenten (Threads II und III).

```
--
-- Spec
--
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 fester Größe (5) betrachtet, in den Elemente geschrieben und aus dem Elemente gelesen und verarbeitet werden. Es gibt genau einen Thread, der als Produzent agiert und Elemente in den Ringpuffer hineinschreiben kann. Weiterhin gibt es eine (potenziell beliebige) Anzahl an Konsumenten, die Elemente aus dem Ringpuffer lesen und verarbeiten. Die Konsumenten sind ebenfalls als (gleichartige) Threads implementiert und laufen damit parallel zum Produzenten und zueinander. Der Ringpuffer ist im Haldenspeicher abgelegt und wird über Zeiger referenziert. Des Weiteren können der Produzent und die Konsumenten durch externe Kontrolleingaben angehalten und wieder gestartet werden. Der Aufbau der einzelnen Komponenten ist grafisch

in Abbildung 2.1 dargestellt.

## 2.2 Eigenschaften

Für dieses Testszenario wurden drei unterschiedliche Eigenschaften spezifiziert, die mögliche Fehler bei der Verwendung des Ringpuffers oder bei der Verarbeitung der Daten beschreiben. Eine Eigenschaft bezieht sich auf die Anzahl der geschriebenen und verarbeiteten Elemente sowie die Größe des Puffers, eine auf Echtzeitanforderungen der Verarbeitung und eine temporale Eigenschaft auf die Kontrolle der Threads.

### 2.2.1 Elemente und Größe des Puffers

Diese Eigenschaft beschreibt zwei mögliche, ähnliche Fehlerfälle dieses Szenarios. Zum Einen, dürfen nicht mehr Elemente verarbeitet werden, als in den Ringpuffer geschrieben wurden. Zum Anderen wird darf die Menge der geschriebenen, aber noch nicht verarbeiteten Elemente die Größe des Puffers nicht überschreiten, da sonst der Puffer überläuft oder, in diesem Szenario, Daten unverarbeitet verworfen (überschrieben) werden.

```
--
-- Inputs
--
define writeElement := exec(2)     -- write to buffer (pointer)
define processElement := exec(13) -- call to process_data() routine

--
-- 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)))
```

Diese und die folgenden Spezifikationen verwenden eine Präfixnotation für das `on-if-yield`-Konstrukt, da Infixnotation im Prototypen des Compilers bisher nicht realisiert wurde.

### 2.2.2 Verarbeitungszeit

Diese Eigenschaft beschreibt, dass die Verarbeitungszeit eines Konsumenten für ein Element nicht mehr als zwei Sekunden beträgt. Dafür wird die Zeit zwischen dem Aufruf der Verarbeitungsmethode und dessen Rückgabe betrachtet. Es wird zunächst davon ausgegangen, dass es höchstens drei Konsumenten gibt.

```
--
-- Macros
--
define sample(s, e) := ifThen(e, s)
define onTrue(x) := onIf(changeOf(x), 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 Konsumenten angehalten und danach wieder gestartet werden. Sobald die Konsumenten angehalten wurden, darf der Lesezeiger für den Ringpuffer nicht verändert werden, bis diese wieder gestartet wurden. Ansonsten würden die Konsumenten aus dem Ringpuffer lesen, obwohl sie angehalten wurden.

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

--
-- 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
)
```

Das prop-Konstrukt ist nötig, da die monitor-Funktion im Prototypen des Compilers noch keine Eventströme mit booleschen Werten als Propositionen verarbeiten kann.