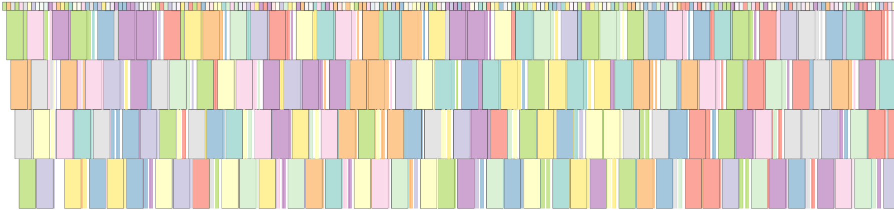
Sequence: Pipeline modelling in Pharo

IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France

Dmitry Matveev
Intel Corporation

August 31, 2023



Outline

- Introduction
- Sequence overview
- Implementation
- Future

The background

What we did

- We developed platforms: SoC for IoT.
- Platforms bring various capabilities, e.g:
  - Camera (MIPI), Media (codec), AI, DSP, etc.
- Capabilities are utilized by workloads.
- Workloads enable use-cases and their KPIs:
  - "Handle `N` streams at `K` FPS while doing `X`".

The environemnt

How we did it

- Every component had its own team:
  - Development, testing, benchmarking done individually;
- The integration team pulled all components together to make a BKC:
  - "Best-Known Configuration".

Pre-production hardware is fun

- A very limited number of units.
- May be not very stable in the beginning.

The problem

How to evaluate the system performance without having the complete system ready?

- Simulate it!

Why it matters?

- Understand how every individual component contributes to the overall performance:
  - *What* (*where*) to optimize next?
- Understand the application flow:
  - Are there execution gaps or stalls?
  - How to reorganize the application flow to meet the criteria?

Seeking for a solution

Simulator expectations

- Allow to describe system resources;
- Allow to define building blocks using that resources;
- Allow to build scenarios atop of these blocks;
- Collect information of interest;
- Present the interactive system trace for inspection;
- Provide rapid feedback for "what-if?" and "trial-and-error" analysis.

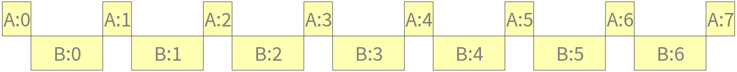
Crafting the solution

Why Smalltalk

- There was no good out-of-the box solution;
- Pharo itself was closest to become a solution:
  - Smalltalk is a nice language to describe things;
  - Pharo Playground (`Ctrl+G`) is all we need for rapid feedback;
  - Roassal is a nice visualization framework.
- Only a simulator engine itself was missing.

Sequence: A minimum example

```
| a b |
a := 'A' asSeqBlock latency: 20 ms.
b := 'B' asSeqBlock latency: 20 fps.
a >> b.
(Sequence startingWith: a)
    runFor: 500 m
```



Sequence: Data stream example

```
| s w g |
s := 'S' asSeqBlock latency: [ :f | (f data * 10) ms  ].
w := 'W' asSeqBlock latency: 25 ms.
s >> w.
g := PMPoissonGenerator new lambda: 5.
(Sequence startingWith: s)
    runFor: 800 ms
    on: [ g next ]
```

Sequence: Resource sharing example

```
| a b t |
t := SeqTarget new lanes: 2.
a := 'A' asSeqBlock latency: 25 ms; target: t; lanes: 2.
b := 'B' asSeqBlock latency: 30 ms; target: t; lanes: 1.
SeqNaiveMultiExecutor new
  add: (Sequence startingWith: a);
  add: (Sequence startingWith: b);
  scheduler: SeqRoundRobinScheduler new;
  runFor: 500 ms;
  trace.
```

A:0	B:0	A:1	B:1	A:2	B:2	A:3	B:3	A:4	B:4	A:5	B:5	A:6	B:6	A:7	B:7	A:8	B:8	A:9
A:0		A:1		A:2		A:3		A:4		A:5		A:6		A:7		A:8		A:9

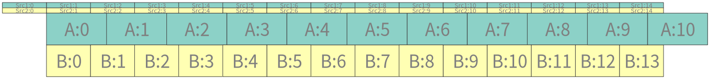
Sequence: Real-time processing example

```
| s1 a s2 b |
s1 := 'Src1' asSeqBlock latency: 30 fps; live.
a  := 'A'    asSeqBlock latency: 22 fps.
s2 := 'Src2' asSeqBlock latency: 30 fps; live.
b  := 'B'    asSeqBlock latency: 30 fps.
s1 >> a.
s2 >> b.
SeqNaiveMultiExecutor new
    add: (Sequence startingWith: s1);
    add: (Sequence startingWith: s2);
    runFor: 500 ms;
    trace.
```

Statistics Help

a Sequence 2A052200 latency avg.: 45.454 ms, med.: 45.454 ms, max: 45.454 ms, [11]

a Sequence 36FB2E00 latency avg.: 33.333 ms, med.: 33.333 ms, max: 33.333 ms, [14]



Sequence: Advanced examples

Pipelining

```
| src a b c seq opts |

src := 'Src' asSeqBlock
  latency: 30 fps;
  live.
a := 'A' asSeqBlock latency: 33 ms.
b := 'B' asSeqBlock latency: 33 ms.
c := 'C' asSeqBlock latency: 33 ms.
src >> a >> b >> c.

seq := Sequence startingWith: src.
opts := SeqExecOptions new
  usePipelining;
  dropFrames.

(SeqNaiveMultiExecutor new
  scheduler: SeqRoundRobinScheduler new;
  add: seq options: opts;
  runFor: 500 ms;
  trace) showFrameDrops; colorByFrames
```

Streams

```
| src a b c xpu seq opts |

src := 'Src' asSeqBlock
  latency: 33 ms;
  live.
a := 'A' asSeqBlock latency: 10 ms.
b := 'B' asSeqBlock latency: 45 ms.
c := 'C' asSeqBlock latency: 10 ms.
src >> a >> b >> c.

xpu := SeqTarget new lanes: 2.
b target: xpu; streams: 2.

seq := Sequence startingWith: src.
opts := SeqExecOptions new
  usePipelining;
  dropFrames.

(SeqNaiveMultiExecutor new
  scheduler: SeqRoundRobinScheduler new;
  add: seq options: opts;
  runFor: 500 ms;
  trace) showFrameDrops; colorByFrames.
```

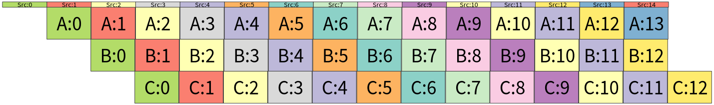
Sequence: Advanced examples

Pipelining example

Statistics

Help

a Sequence 6948700 latency avg.: 99.0 ms, med.: 99.0 ms, max: 99.0 ms, [0/13]

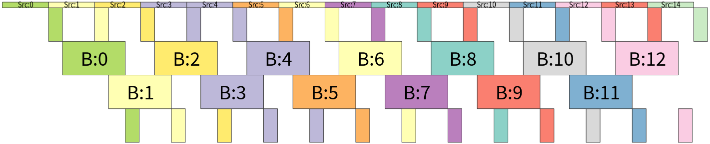


Streams example

Statistics

Help

a Sequence 3A3BA700 latency avg.: 65.0 ms, med.: 65.0 ms, max: 65.0 ms, [0/13]



Sequence: Characteristics of a DES

Sequence is as Discrete Event Simulation (DES) system

- It registers Events that happen during the simulation;
  - Events are essentially facts that particular blocks could execute;
- System state is defined by Targets which are free or locked at the given time point;
- Targets are essentially the Resources;
- Simulation time is discrete, the current time pointer is advanced by the coming events.

More on DES

- J. Banks, Introduction to Simulation (1999)

Sequence inside

- The core of Sequence is a simulation executor.
  - There may be multiple but `SeqNaiveMultiExecutor` is the most advanced at the time.
- Simulation executor represents sequences as running processes.
  - Processes are running periodically with no termination criteria.
  - The system-level termination criteria is simulation time (`#runFor:`).
- Normally, a sequence object maps to a single execution process, but there are exceptions:
  - Sequences with live sources map to two processes, one for a source block and one for the sequence;
  - Pipelined sequences map to `N` interconnected processes: every block gets its own process (or `K` processes if `streams: k` property is assigned).

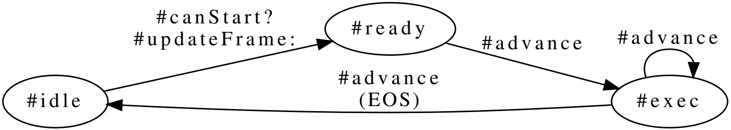
Sequence inside: Event streams

`SeqEventStream`: the heart of the simulation system

- Represents a running periodic process.
- FSM inside, handles one block (one target) at time.
- Provides compact API for the executor to manage:
  - `#canWork`: answer executor if this particular stream can do the work, depending on its state.
  - `#updateFrame:`: sent implicitly within executor when stream's input data is available (a new input frame is generated). Stream updates its internal *block stream* based on this data.
  - `#advance`: make next step (progress) in the process. Internally, either lock or release the current resource for the current block.
  - `#nextTick`: answer the time of the next event in this stream.
  - `#updateTimePoint:`: let the event stream know what is the simulation time right now.

Sequence inside: Event stream state machine

	#idle	#ready	#exec
Meaning	No data available	Data is available Not entered execution yet	Data is available Executing (may be blocked waiting for a resource)
#canWork	Asks canStartBlock	This/next block can lock its target	Lock acquired: true No lock: - See #canWork @ #ready
#advance	Call startBlock Move to #ready	Call startBlock if not yet Enter a new block: - See #advance @ #exec	Lock acquired: leave block - Release resource - Record event ...If at the last block: - Record completion - Move to #idle No lock: - Peek next block - Acquire a new lock



Sequence inside: Simulation executor loop

Simulation executor loop becomes straightforward with the `SeqEventStream` abstraction defined above:

- Update time point for all streams;
- Ask which streams can work;
  - Filter out streams which can work *right now* (time point is not in the future);
  - Let *Scheduler* decide which stream will be advanced;
  - Advance the selected stream (`#advance:`) and update the time point (sometimes with 0 increment).
- Repeat.

Note: advancing one stream invalidates invariants so some streams which could work now may not work anymore in the next iteration.

Sequence inside: Scheduler

- User-configurable object (can implement your own);
- Integrated into:
  - Executor loop: asked which stream to prefer if there're multiple candidates to run right now.
    - `#decide: aCollectionOfStreams`.
  - Target (resource) locking: asked if a stream can lock this resource.
    - `#askLockOn: aTarget for: aStream at: aSeqBlock`: targets consult with scheduler if an *available* resource can be given on request;
    - `#waitlist: aStream to: aTarget`: sent by a Stream to inform it is interested in locking the target, if resource lock request has been rejected.
- Available: `SeqDumbScheduler`, `SeqRoundRobinScheduler`, `SeqPriorityRRScheduler`.

Next steps on Seqence

Short-term plan

- Extend test coverage, close the functionality gaps.
- Interoperability: *export* to Perfetto format.
- Prepare a formal release.

Mid-term plan

- Interoperability: *import* from Perfetto format.
- Introduce *annotations*: some way to mark parts of the sequence we're especially interested in, to see it in the trace.
- Introduce *monitors*: custom hooks to probe and collect simulation run-time information about the running processes and resource state.

Sequence: Vision

Long-term vision

- Extend the time domain from *milliseconds* to arbitrary.
- Extend to model non-periodic processes.
- Consider *Queues* as the right abstraction to access targets?
- Composable simulations:
  - What if a Sequence block is also simulation inside?

Thanks!

Sequence is already Open Source:

- Currently hosted at Github: `https://github.com/dmatveev/sequence`.
- ~2.5KLOC of code, ~1.5KLOC of tests.
- MIT License.

Try it today!

```
Metacello new
    baseline: 'Sequence';
    repository: 'github://dmatveev/sequence';
    load.
```