Sequence: Pipeline Modelling in Pharo

Dmitry Matveev[1]

[1]*Intel Corporation*

Abstract

In the modern computing systems, data processing is often organized as a pipeline: a sequence of operations including data acquisition, different processing stages, and visualization. Normally, these sequences make recurring patterns: as the input data stream produces new data frames in time, the same or nearly the same sequence of operations is applied to this data to produce results.

Modelling pipeline performance is a complex problem and an interesting topic for research. With respect to the computing systems, pipeline stages can run on different parts of the system, so parallel execution is possible. Also, a computing system can run multiple different pipelines concurrently, and as the compute resources are finite, scheduling problems can arise when concurrent pipelines are trying to access the same resource.

Finally, modelling systems at early stages can generate important performance insights for system designers. It applies to both software performing the processing, and hardware executing this software.

This paper introduces *Sequence*: a Pharo package for rapid pipeline execution modelling. Built on powers of Smalltalk and the Pharo interactive environment, Sequence offers a compact syntax to express pipelines and their properties, a sophisticated simulation engine, and Roassal-based visualization for interactive trace inspection.

Keywords

Pipeline modelling, Simulation, Live programming

1. Introduction

Today data processing is all around us. Be it video analytics, audio enhancement, conference calls, document indexing, etc. – it happens everywhere and everytime when data is generated and digitized to be processed by a compute device. Data, software processing this data, and hardware running this software together form a compute system. Modelling such systems generates important insights for system designers:

- For existing systems, their actual measured performance can be compared with theorethical modelling numbers, and possible inefficiencies can be spotted or the observed behavior can be explained;
- For systems at the design stage, modelling can provide projections that can support design decisions, e.g. identify early which parts of the system should be optimized or how resources should be distributed to help system meet its performance expectations.

IWST 2023: International Workshop on Smalltalk Technologies. Lyon, France; August 29th-31st, 2023

dmitry.matveev@intel.com (D. Matveev)
0009-0009-6423-4440 (D. Matveev)

Many compute-intensive audio and video workloads can be represented in the form of a pipeline, where a number of processing stages is executed in order. These pipelines often include input and output: data acquisition and visualization. What makes audio and video workloads interesting for modelling is their regularity: input data is produced at a predefined rate (e.g., by a camera at 30 frames per second), and the number of operations performed is usually the same for every frame. For such systems, properties like *latency* and *throughput* are key to evaluating their performance. Other important metrics include idle time and power consumption. If the system assumes real-time processing or communication, quality of service may become a concern.

Simulation is one of the methods to model systems and evaluate the aforementioned properties. There are many different types of simulations [1]. For compute systems, the most comprehensive ones are cycle-approximate and cycle-accurate simulations[2][3]. Such simulations construct a complete microprocessor pipeline to collect actual timing information down to individual cycles. However, for the evaluation purposes such fine-grain precision is not only rarely needed, but may not be even possible – as it requires excessive and a very detailed description of the modelled system compute resource but also a very detailed description of the modelled system, which may be hard to populate at the prototyping stage. In such cases, methods like discrete event simulation (DES)[4] with more coarse-grain models may provide a reasonable estimate in a much shorter time.

In this paper, we present *Sequence*, a framework for rapid pipeline modelling in Pharo[5]. Sequence combines a compact language to express pipelines and define their execution environment, an event stream-based execution simulator, and an interactive trace visualization powered by Roassal[6], a Smalltalk package for scriptable interactive visualizations. As Sequence is built with Pharo, a modern Smalltalk dialect and environment, it follows the "everything is object" approach where both pipeline definition and the resulting trace are objects. It enables developers and researches with scripting capabilities when synthesizing simulations (e.g., to evaluate different hypotheses or perform a parameter search) as well as when processing the simulation results, all in the same environment and in the same programming language.

The paper is organized as follows: a brief overview of related work and DES simulation tools is given in section 2. Section 3 covers the functionality and different aspects of Sequence and provides a number of examples. Section 4 highlights some key properties of Sequence which make it stand out from other tools. Finally, section 5 concludes the paper and outlines directions for the future work.

Initially, Sequence was designed to model computer vision workloads in heterogeneous systems. However, as those scenarios are quite complex and, at the same time, the resulting framework turned out generic enough, for illustrative purposes this paper lists some real-life scenarios as modelling examples in section 3.

2. Related work

A general overview of open-source DES tools is given in [7]. Most of those tools are generic, which makes them flexible and applicable in different domains, but at the same time it requires them to provide low-level means to express and organize simulations, which brings its extra cost for developers to adopt. Among those, SimPy [8] stands out as one of the closest analogues from the Python world. SimPy follows process-oriented simulation and relies on Python generators and coroutines, calling itself "pseudo-parallel". SimPy is still seen as a low-level tool where additional effort is required to build process simulations atop of it.

OMNeT++[9][10] is a C++ component-based simulation library and framework. Components or modules in OMNeT++ can communicate with message passing; modules can be simple and compound (built from other modules). OMNeT++ also provides its own domain-specific language called NED to describe the simulation component model. The object model in OMNeT++ implies using inheritance to implement module classes; simulations are built into ready-made simulation programs linked with the OMNeT++ simulation kernel. The resulting simulations can be configured with .INI files and visualized through a dedicated graphical runtime environment.

PowerDEVS[11][12] is a C++ toolkit which implements DEVS, the Discrete EVent System formalism. It allows to code atomic DEVS models in C++, and combine them into systems (structures) in a graphical tool. PowerDEVS is a powerful tool to model dynamic and continuous systems, but may be too formal and low-level to simulate simpler DES processes.

`ns-3`[13] is C++ DES framework for computer network simulation. `ns-3` is an example of purpose-built simulation software, where domain specifics are reflected in the library API. Having said that, `ns-3` provides means to express computer networks at the high level, directly operating with a computer network vocabulary: network topologies, protocols, channels, and packet flow simulation.

JaamSim[14] is a free, open-source simulation package written in Java. Similar to OMNeT++, JaamSim specifies its own language to define simulation models, as well as the graphical user interface. The object model can be extended with custom classes in Java. JaamSim supports both process-orientated and event-orientated simulation models.

DESMO-J[15] is positioned as a modern open-source library for Java-based discrete event simulation. DESMO-J expects the model structure (including properties and behavior of all components) to be coded in appropriate Java classes and makes the simulation environment (simulation clock clock, random number generation, event list, reporting) readily available atop of that. Similar to JaamSim, DESMO-J also supports event-oriented and process-oriented perspectives to implement a model.

Finally, Cormas[16] is a generic agent-based modeling platform dedicated to common-pool resource management, written in Smalltalk. In contrast with other tools mentioned in this section, Cormas implements a different paradigm with focus on multi-agent interactive simulation. Cormas is an important example of simulation tool built on powers of Smalltalk programming language and environment, the same as used to implement Sequence.

3. Workload simulation with Sequence

In this section we present our own pipeline simulation package *Sequence*. We explain how to define workloads in Sequence and model various scenarios in it.

Sequence implements DES characteristics as defined in [4] and [1]. Sequence API design is inspired by classic Smalltalk packages Roassal[6] and PetitParser[17]. In contrast with the majority of DES libraries decribed in Section 2, Sequence doesn't require programmer to derive

Listing 1: Defining a pipeline with Sequence

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

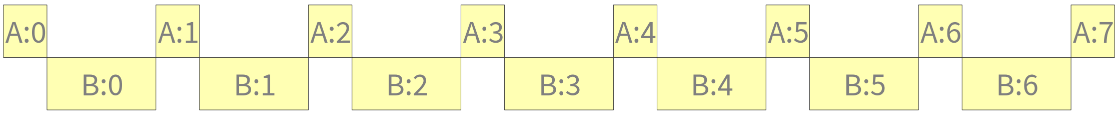


Figure 1: Modelling result for a simple pipeline

new classes to express the simulated domain. Instead, Sequence offers an ad-hoc compositional approach where a simulation can be defined as a script in a single Playground[18] window.

3.1. Blocks and Sequences

Pipeline workloads are defined as sequences of operations. In *Sequence*, operations are called *Blocks*. Objects of class "SeqBlock" are basic building blocks of any sequence.

Every block has two mandatory properties, a name and its execution time. Sequence extends standard Smalltalk classes with new messages to construct blocks and specify this information in a compact manner as shown in Listing 1. Blocks are connected using message `#>>`. After all required blocks are created and configured, a `Sequence` object is constructed and is sent to a 500 ms simulation. The simulation result is a trace represented by class `SeqTrace`; instances of this class support Pharo inspectors and by default open as Roassal canvas as shown in Figure 1.

Trace represents events registered during the simulation. Every box stands for an executed block; boxes are placed in order on a time line. Number suffixes are data frame numbers, we see block `A` executed on frame 0 is followed by block `B` executed on frame 0, as defined in the scenario from Listing 1.

3.2. Data

Input data is an important aspect of modelling. On Figure 1 we see a pipeline of two connected blocks executed repeatedly – Sequence creates data stream implicitly and automatically produces frames 0..7 to feed the pipeline.

In real-world scenarios, pipeline execution may depend on data. For example, every data frame may contain a different amount of information to process, which would affect the block's execution time. Imagine a car workshop where service work ("S") depends on the complexity of the issue, but in the end there is a car wash procedure ("W") which takes nearly the same amount of time for every car. Example of such modelling is shown in Listing 2: `Sequence` accepts a *generator block* which produces data for every frame using a Poisson generator from

Listing 2: Data stream and data-dependent block latency

```
| 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 ]
```

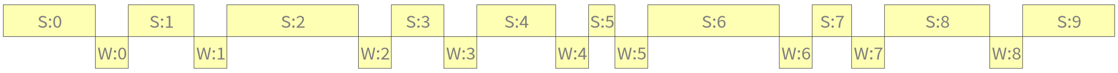


Figure 2: Visualization of car workshop modelling

Listing 3: Data stream and data-dependent execution

```
| scan payment |
scan    := 'S' asSeqBlock latency: 5 ms; tasks: [ :f | f data ].
payment := 'P' asSeqBlock latency: 30 ms.
scan >> payment.
g := PMPoissonGenerator new lambda: 10.
(Sequence startingWith: scan)
    runFor: 500 ms
    on: [ g next ]
```

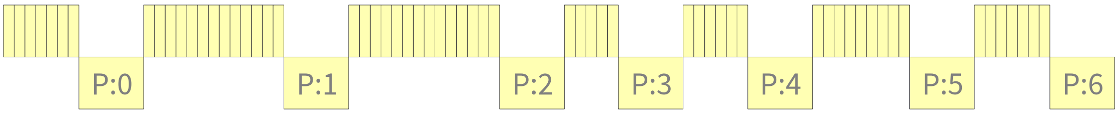


Figure 3: A day in a life of self-service checkout kiosk

PolyMath[1] [19], where a generated number represents repair complexity; latency for a block is specified as a callback (a `BlockClosure`).

Note that in listing 2 and in other examples, the block duration is specified in milliseconds while the domain assumes durations in minutes or hours; this is a known limitation as Sequence was initially designed with millisecond-scale task simulations in mind; this limitation will be addressed in the future versions.

There are cases when input data may spawn multiple tasks to process, for example in self-service checkouts customers need to scan a number of products and then proceed to payment. Sequence allows to specify this behavior using `#tasks:` callback as shown in Listing 3.

[1] https://github.com/PolyMathOrg/PolyMath

Listing 4: Parallel process execution

```
| a b |
a := ’A’ asSeqBlock latency: 27 ms.
b := ’B’ asSeqBlock latency: 30 ms.
SeqNaiveMultiExecutor new
    add: (Sequence startingWith: a);
    add: (Sequence startingWith: b);
    runFor: 500 ms;
    trace.
```

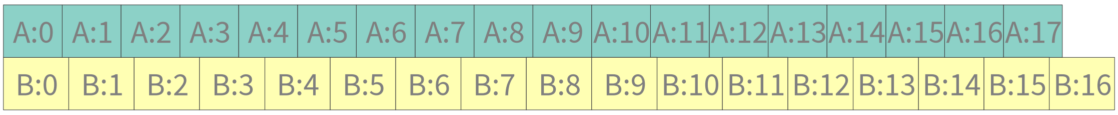


Figure 4: Parallel process execution

3.3. Multi-process execution

In the previous examples we demonstrated how Sequence can model various processes. Sequence, however, is not limited to a single-process modelling – it can run simulation for multiple pipelines in parallel. The way how Sequence runs a simulation depends on an *executor* class, in Listings 1, 2, and 3, a default `SeqNaiveExecutor` was used implicitly. Listing 4 shows how to model two parallel processes using a special executor class, a `SeqNaiveMultiExecutor`.

Visualization of the multi-process scenario is shown on Figure 4. When there are multiple sequences executing in the same simulation, every sequence gets its individual color in the trace.

3.4. Targets and scheduling

In Sequence, *Targets* represent resources where blocks can be executed. On a visualized trace targets are shown as logical rows (defining every block event’s position on Y-axis): if the events are placed in the same row on a trace, it means that they have happened on the same target. By default, every block has its own unique target, this is why Figures 1, 2, 3, and 4 show events only of the same block in every row.

Listing 5 shows how the same target can be specified to two pipeline blocks explicitly. Note Sequence relies on Smalltalk’s *cascading messages* to configure blocks with new details in a compact way.

When two separate processes access the same resource, they become concurrent. In this case, if the resource is already locked by one block, another block assigned to the same resource can’t be executed and has to wait. In this situation, scheduling is required to distribute properly resource time between the concurrent processes. Sequence allows to customize this behavior using *scheduler* objects and provides some basic scheduler implementations out of the box. Figure 5 shows a trace for the case when a simple round-robin scheduler is involved.

Resources may be quantitive; in this case the amount of resources available on target is

Listing 5: Concurrent process execution

```
| a b t |
t := SeqTarget new.
a := 'A' asSeqBlock latency: 27 ms; target: t.
b := 'B' asSeqBlock latency: 30 ms; target: t.
SeqNaiveMultiExecutor new
    add: (Sequence startingWith: a);
    add: (Sequence startingWith: b);
    scheduler: SeqRoundRobinScheduler new;
    runFor: 500 ms;
    trace.
```

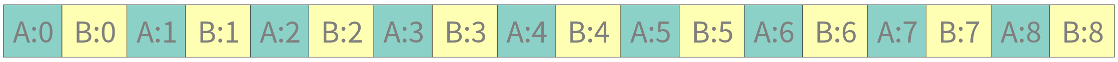


Figure 5: Concurrent process execution

Listing 6: Concurrent execution on a target with lanes

```
| 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.
```

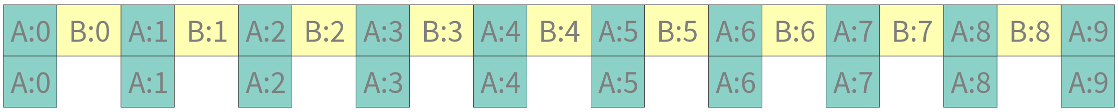


Figure 6: Concurrent execution on a target with lanes

specified by the message `lanes:`, the same message configures how much resource is required for a block. Currently resource allocation is atomic, so if there's N resources (lanes) available on a target, and a block requires such number M of lanes where $M > N$, no allocation will happen.

Example of this situation is given in Listing 6 and Figure 6: a shared target has two lanes, block "A" requires two lanes to run, while block "B" needs only one. The round-robin scheduler balances execution between "A" and "B"; even if there's room available on the target when "B" executes, "A" can't trigger since there's not enough lanes available.

Listing 7: Execution with a live source

```
| 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.
```

3.5. Metrics. Live sources

Sequence *trace* contains comprehensive information about all events registered during modelling. For the convience, Sequence keeps track of the most interesting pipeline properties such as the number of completed frames and pipeline latency.

Pipeline *latency* is calculated as a duration between a point in time when pipeline started processing a frame, and a point in time when pipeline finished processing the frame.

The *start time* of a pipeline is the moment when there is (new) data to process. By default, Sequence assumes new data is available as soon as a pipeline can process a new frame. The *finish time* of the pipeline is the moment when pipeline's last block completes its execution for the given frame. As a classic DES system, Sequence tracks its own simulated clock and advances the time point as new events are handled by the executor.

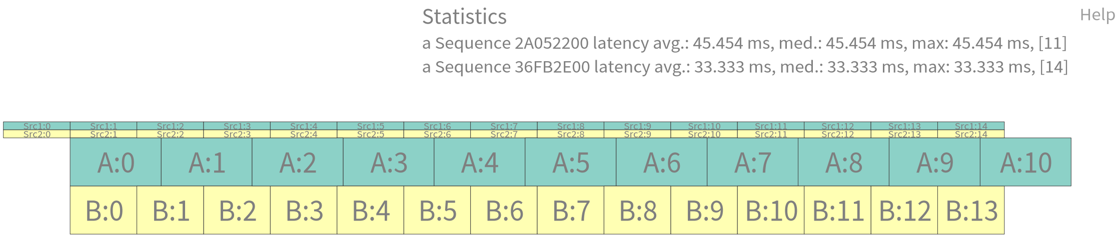
In real-world scenarios, when real-time processes are modelled, input data is generated by the real world and is registered by the system with some sample rate. In this case, the modelling source is called *live* and runs with its own cadence, independently from a processing pipeline. If a pipeline is fast enough to process data until a new data frame arrives, it maintains its real-time property. Otherwise, it will accumulate an output delay or cause a *frame drop*. In the case of a live source, the pipeline start time is taken as the time when pipeline started processing data from a live source (the live source own latency is excluded from that time).

Pipeline latency, pipeline output latency (an inverse of its throughput), number of frames processed and dropped (when enabled by executor) are the metrics Sequence collects and displays by default. As trace is a regular Smalltalk objects containing a collection of all registered events, developers can calculate their own metrics of interest based on this data.

4. Discussion: What makes Sequence special

Sequence combines real-time approach of short scriptable programming paired with quick response based on standard Smalltalk inspectors.

This is unique quality of Smalltalk as a live environment. Sequence provides rapid response so a researcher can adjust parameters and explore different behaviors in an interactive manner.

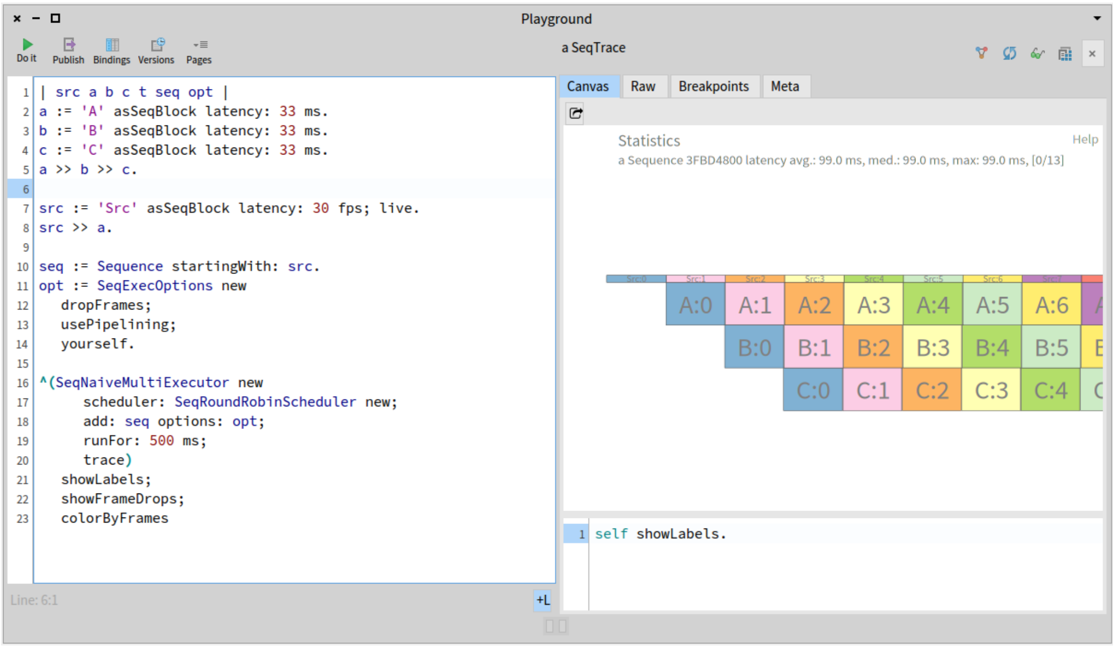


Statistics

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]

Figure 7: Execution with a live source and pipeline metrics. Stream "A" accumulates delay: while data frame #12 is already available, "A" only processes frame #9



```
| src a b c t seq opt |
a := 'A' asSeqBlock latency: 33 ms.
b := 'B' asSeqBlock latency: 33 ms.
c := 'C' asSeqBlock latency: 33 ms.
a >> b >> c.

src := 'Src' asSeqBlock latency: 30 fps; live.
src >> a.

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

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

Statistics

a Sequence 3FBD4800 latency avg.: 99.0 ms, med.: 99.0 ms, max: 99.0 ms, [0/13]

```
self showLabels.
```

Figure 8: Sequence enables ad-hoc simulations directly in Pharo playground

Sequence avoids the "compile-and-run" loop which is specific to C++ and Java-based solutions. This makes Sequence-powered simulations easier to debug: as Sequence stays a Smalltalk library and doesn't introduce its own language, the existing superior Smalltalk debugging tools and workflows[20] can be reused. Sequence doesn't use coroutines and other complicated control flow mechanisms which makes debugging deterministic and clean, as the Sequence's simulation state can be clearly seen in the standard Pharo debugger.

Another strong point behind Sequence is that there is no semantic gap between the simulated data and its visual representation. Objects visualized in the trace are the same objects generated during the simulation, which allows any sorts of analysis or post-processing to be done in the same Smalltalk environment where the simulation has been executed.

At the same time, as a purpose-built package, Sequence remains relatively compact and simple inside, whilst providing powerful pipeline modelling capabilities.

5. Conclusion and future work

Sequence illustrates how a powerful simulation can be built atop of very basic concepts. With programmable interface and inspectable results, Sequence builds a solid foundation on the future work in the field.

One of the logical directions for further development of Sequence may be seen in expanding its applicability to other areas, e.g. modelling of real-life processes, service load, manufacturing, etc.

Section 3 shows that Sequence already can be applied for modelling of different processes, not only limited to computer data processing workloads. While this scaling shouldn't require dramatic changes in the Sequence's event stream execution engine, a revision of supported event durations and visualized trace scale may be required.

Another perspective topic is composable simulations[21]. Currently Sequence implements a flat structure where a pipeline can consist only of terminal blocks. Extending Sequence to handle pipelines of pipelines would enable new types of simulations at different levels: engineers could simulate smaller portions of the system and then combine those into more complex and sophisticated simulations. Also, the same machinery could enable simulating branching and conditional execution within Sequence.

References

[1] A. P. Galvão Scheidegger, T. Fernandes Pereira, M. L. Moura de Oliveira, A. Banerjee, J. A. Barra Montevechi, An introductory guide for hybrid simulation modelers on the primary simulation methods in industrial engineering identified through a systematic review of the literature, Computers & Industrial Engineering 124 (2018) 474–492. URL: https://www.sciencedirect.com/science/article/pii/S0360835218303693. doi:https://doi.org/10.1016/j.cie.2018.07.046.

[2] M. T. Yourst, PTLsim: A cycle accurate full system x86-64 microarchitectural simulator, in: 2007 IEEE International Symposium on Performance Analysis of Systems & Software, IEEE, 2007, pp. 23–34.

[3] J. Yi, D. Lilja, Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations, Computers, IEEE Transactions on 55 (2006) 268 – 280. doi:10.1109/TC.2006.44.

[4] J. Banks, Introduction to Simulation, in: Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1, WSC '99, Association for Computing Machinery, New York, NY, USA, 1999, p. 7–13. URL: https://doi.org/10.1145/324138.324142. doi:10.1145/324138.324142.

[5] A. P. Black, O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by example, Lulu. com, 2010.

[6] A. Bergel, Agile Visualization, LULU Press, 2016. URL: http://AgileVisualization.com.

[7] G. Dagkakis, C. Heavey, A review of open source discrete event simulation software for operations research, Journal of Simulation 10 (2016) 193–206. URL: https://doi.org/10.1057/jos.2015.9. doi:10.1057/jos.2015.9. arXiv:https://doi.org/10.1057/jos.2015.9.

[8] N. Matloff, Introduction to Discrete-Event Simulation and the SimPy Language, Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August 2 (2008) 1–33.

[9] A. Varga, OMNeT++, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 35–59. URL: https://doi.org/10.1007/978-3-642-12331-3_3. doi:10.1007/978-3-642-12331-3_3.

[10] A. Varga, A Practical Introduction to the OMNeT++ Simulation Framework, Springer International Publishing, Cham, 2019, pp. 3–51. URL: https://doi.org/10.1007/978-3-030-12842-5_1. doi:10.1007/978-3-030-12842-5_1.

[11] E. Kofman, M. Lapadula, E. Pagliero, PowerDEVS: A DEVS-based environment for hybrid system modeling and simulation, School of Electronic Engineering, Universidad Nacional de Rosario, Tech. Rep. LSD0306 (2003) 1–25.

[12] F. J. Preyser, An approach to develop a user friendly way of implementing DEV&DESS models in powerDEVS, Masterthesis, TU Wien (2015).

[13] G. F. Riley, T. R. Henderson, The ns-3 Network Simulator, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 15–34. URL: https://doi.org/10.1007/978-3-642-12331-3_2. doi:10.1007/978-3-642-12331-3_2.

[14] D. H. King, H. S. Harrison, Open-source simulation software "JaamSim", in: 2013 Winter Simulations Conference (WSC), 2013, pp. 2163–2171. doi:10.1109/WSC.2013.6721593.

[15] J. Göbel, P. Joschko, A. Koors, B. Page, The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development, 2013, pp. 100–109. doi:10.7148/2013-0100.

[16] P. Bommel, N. Becu, C. Le Page, F. Bousquet, Cormas: An agent-based simulation platform for coupling human decisions with computerized dynamics, in: T. Kaneda, H. Kanegae, Y. Toyoda, P. Rizzi (Eds.), Simulation and Gaming in the Network Society, Springer Singapore, Singapore, 2016, pp. 387–410.

[17] J. Kurs, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, J. Laval, Petitparser: Building modular parsers (2013).

[18] J. Kubelka, R. Robbes, A. Bergel, The Road to Live Programming: Insights from the Practice, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1090–1101. URL: https://doi.org/10.1145/3180155.3180200. doi:10.1145/3180155.3180200.

[19] D. H. Besset, Object-oriented implementation of numerical methods; An introduction with Smalltalk, 2015.

[20] J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 485–495. doi:10.1109/ICSE.2012.6227167.

[21] S. Kasputis, H. C. Ng, Composable simulations, in: 2000 Winter Simulation Conference Proceedings (Cat. No. 00CH37165), volume 2, IEEE, 2000, pp. 1577–1584.