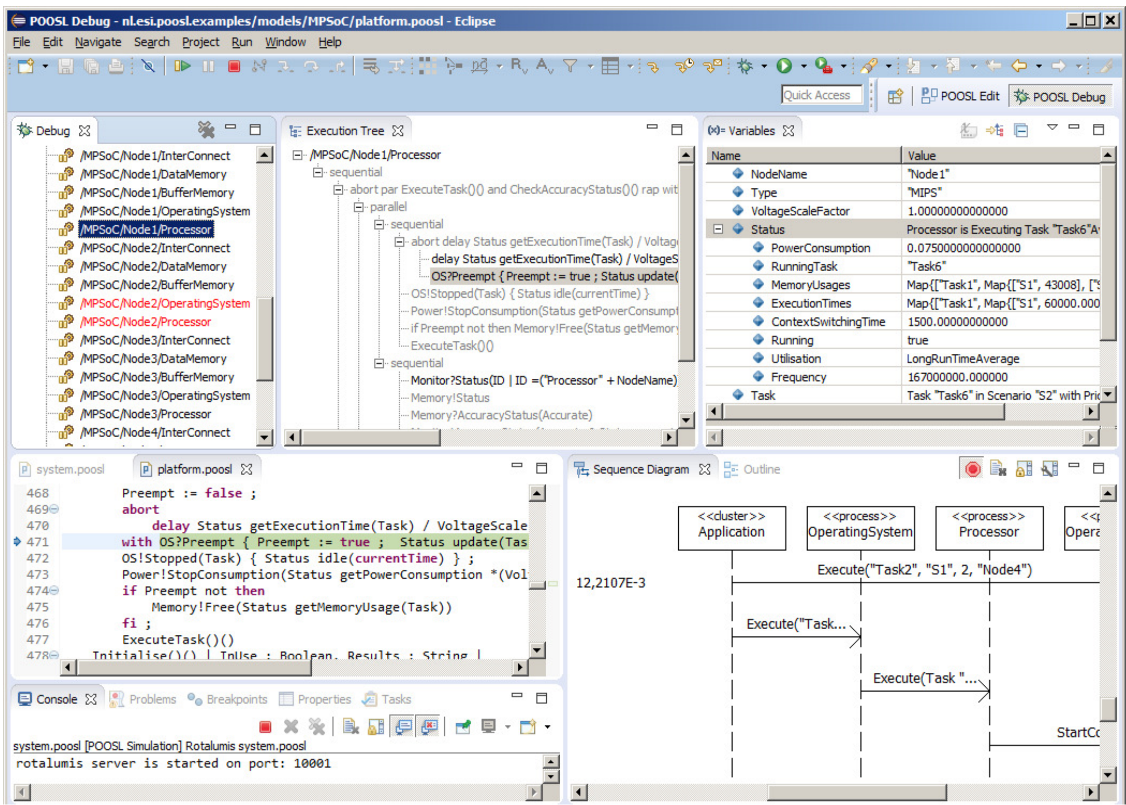
Embedded Systems Innovation BY TNO

COMMIT/ A public- private research community

ARTEMIS

CRYSTAL

POOSL workshop



Arjan Mooij

Bart Theelen

Jozef Hooman

POOSL - Parallel Object-Oriented Specification Language

Language for light-weight modeling and analysis of systems, including both software and digital hardware

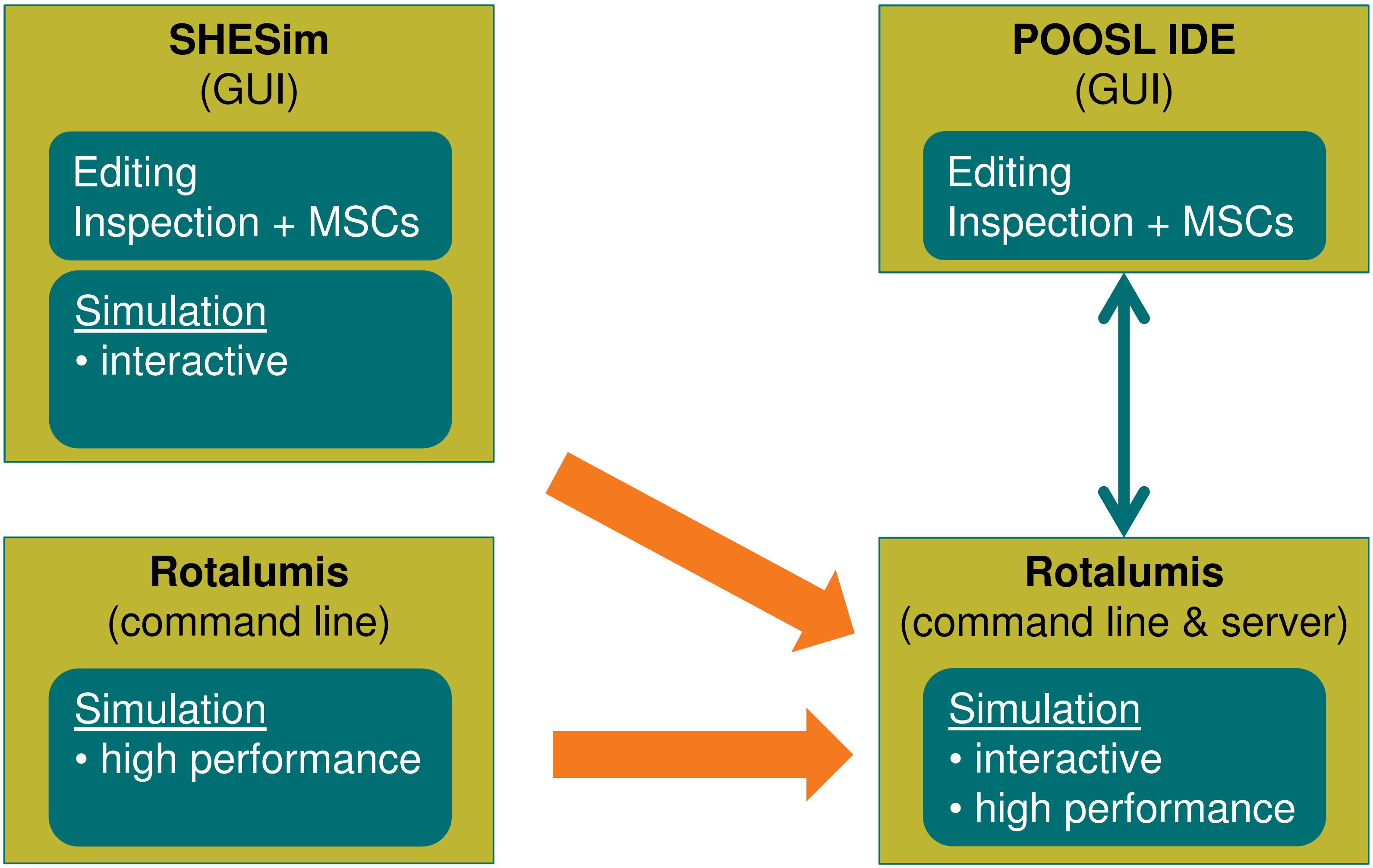
- Developed at Eindhoven Univ. of Tech., Electrical Engineering
- Stable since 2002

Object-oriented modeling language with

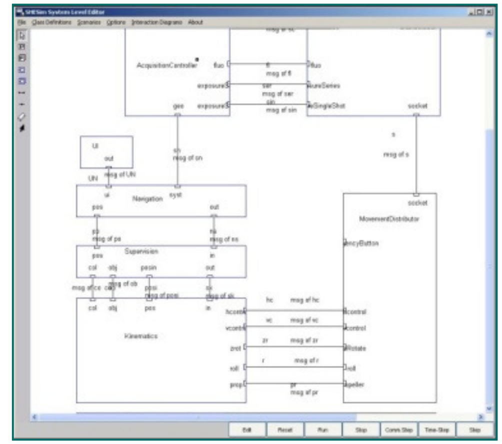
- Concurrent parallel processes
- Synchronization: message passing & shared memory
- Timing
- Hierarchical structure
- Object-oriented data structures
- Stochastic behaviour

Supported by simulation tools, e.g., for performance analysis

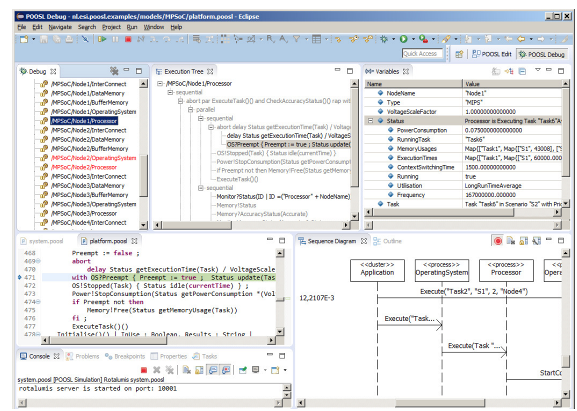
POOSL tool landscape



POOSL tool positioning



- General user experiences with **SHESim**:
  - Positive
    - Used in many industrial projects at TNO-ESI
    - Expressive modeling language (POOSL)
    - Interactive simulation
  - Negative
    - Usability aspects: Many windows, many mouse operations, inline errors, …
    - Early fault detection: Most faults are only detected during simulation



➢ Initial focus for the new **POOSL IDE**:

- Textual editing
- Early fault detection
- Eclipse-based environment

Start new POOSL project

Create **POOSL project**

- Click on File -> New -> Project… and select **POOSL project**
- Next; give name: “workshop.example”; Finish

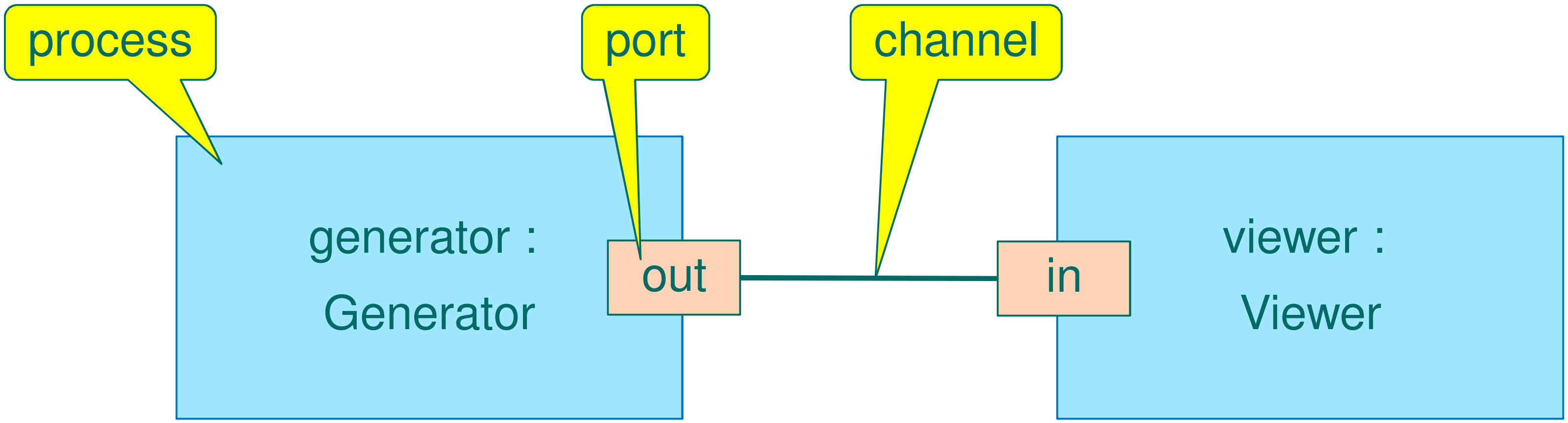
Create POOSL model

- Right-click on directory “models”; select New -> POOSL Model
- Next; give name: “stream1a.poosl”; Finish

Edit file: double-click on it (or drag to edit window)

- <CTRL>-<SPACE> content assist
- <CTRL>-<SHIFT>-<F> automatic formatting
- <CTRL>-S: save
- Comments: // and /* .. */

Stream example



Synchronous communication: send and receive statements are only executed if a matching statement is ready to execute

Matching: same message name and number of parameters

Examples of matching communication pairs (in & out are connected):

out ! hi ()	→	in ? hi ()
out ? talk (stringvar)	←	in ! talk (“hello”)
out ! msg (“store”, 1+2)	→	in ? msg (command, val)

! : send

? : receive

stream1a.poosl - Generator

- Use <CTRL>-<SPACE> and select process class
- Change the template to:

```
process class Generator()
ports

messages
	out!message(Integer)
variables
	number : Integer
init
	initialize()()
methods
	initialize()()
		number := 1 ;
		sendMessage()()
	sendMessage()()
		out!message(number) ;
		number := number + 1 ;
		sendMessage()()
```

assignment

; means sequential composition

Note:
- Errors detected
- Click on the quick-fix of last one

stream1a.poosl - Viewer

- Similarly create a Viewer process

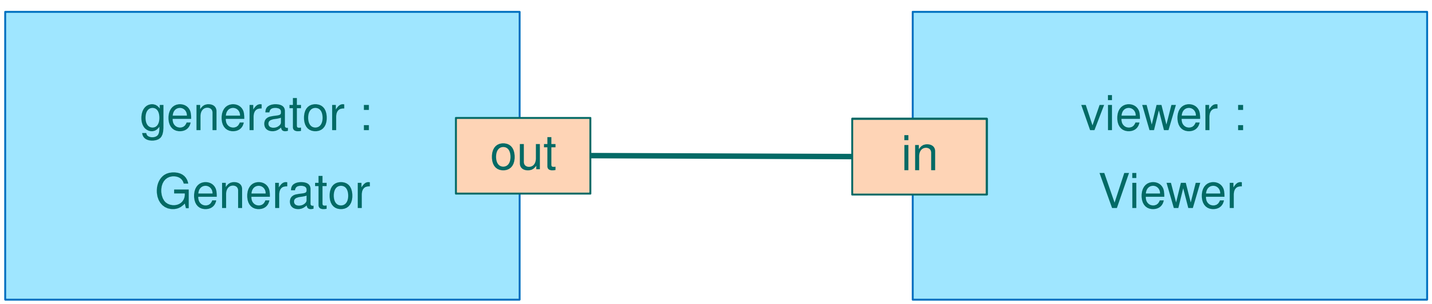
```
process class Viewer()
ports
    in
messages
    in ? message(Integer)
variables

init
    receiveMessage()()
methods
    receiveMessage()() | counter : Integer |
        in ? message(counter);
        receiveMessage()()
```

local variable

stream1a.poosl - system

- Finally define the system – use content assist: **system class**

```
system
instances
    generator : Generator()
    viewer    : Viewer()
channels
    { generator.out, viewer.in }
```



Simulation

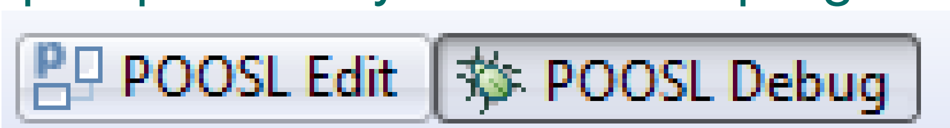
POOSL model can be simulated by

- right click on file name or in editor of file, select Run As > POOSL Simulation

For this model there is no visible output, so use debug mode:

- right click on file name or in editor of file, select Debug As -> POOSL Simulation

This opens Debug perspective; switch between edit and debug perspective by buttons in top-right corner:

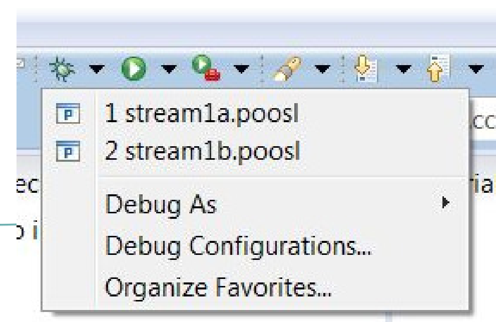


or (without text)

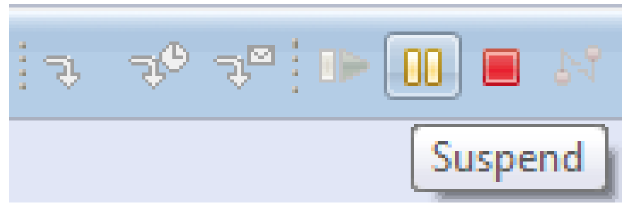
Note: models can also be edited in Debug perspective

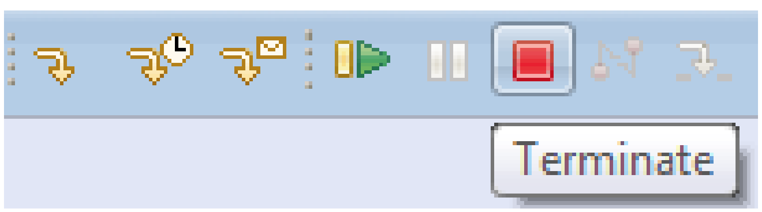
(first stop simulation!) – start simulation in debug mode as above

After the first simulation, it can be restarted using the drop down menu right of the “debug” button:

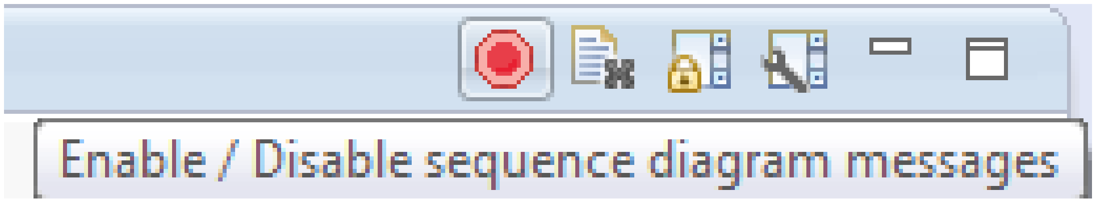


Running and stepping





Sequence diagrams



Press red button to record

Reordering of life lines: click and hold to drag the lifeline

Hands on stream1a.poosl

- Create a new POOSL project
- Create and edit stream1a.poosl
- Simulate the model in debug mode
  - Try different ways of simulating steps
  - Observe the sequence diagram, try reordering of life lines
- Experiment in editor with errors and quick fixes

Notion of time

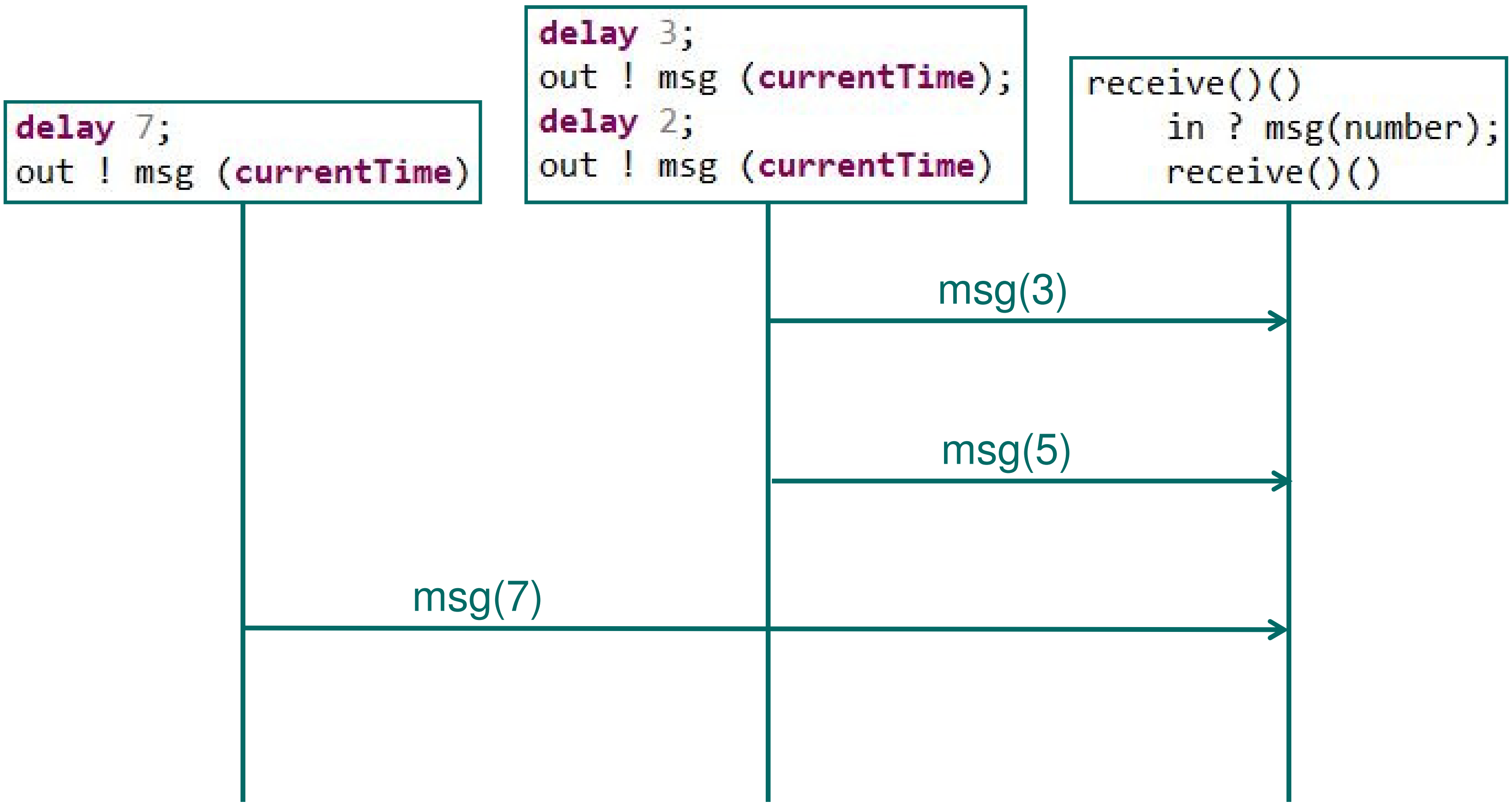
Time can be represented in POOSL by statement **delay** d

- It postpones the execution of the process by “d” time units
- All other statements do not take time

Delay statements are only executed if no other statement can be executed

Note: `currentTime` denotes the current simulation time

Delay example



stream1b.poosl - timing

Add time to Generator and Viewer to represent time needed to generate a message and to display a message, resp., e.g.:

```
sendMessage()()
    delay 3;
    out!message(number) ;
    number := number + 1 ;
    sendMessage()()
```

```
receiveMessage()() | counter : Integer |
    in ? message(counter);
    delay 5;
    receiveMessage()()
```

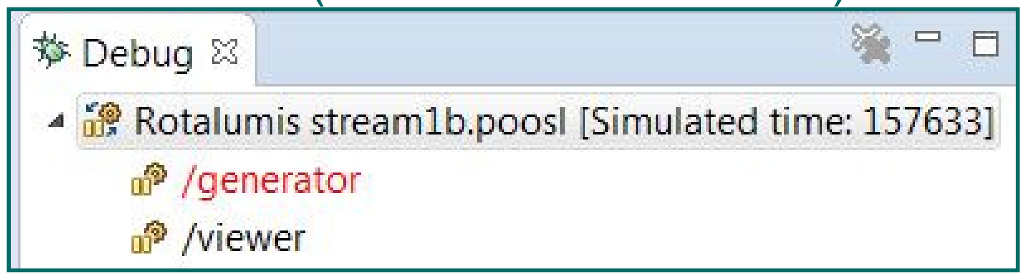
Save and simulate the new model in debug mode

Observe time stamp left of sequence diagram

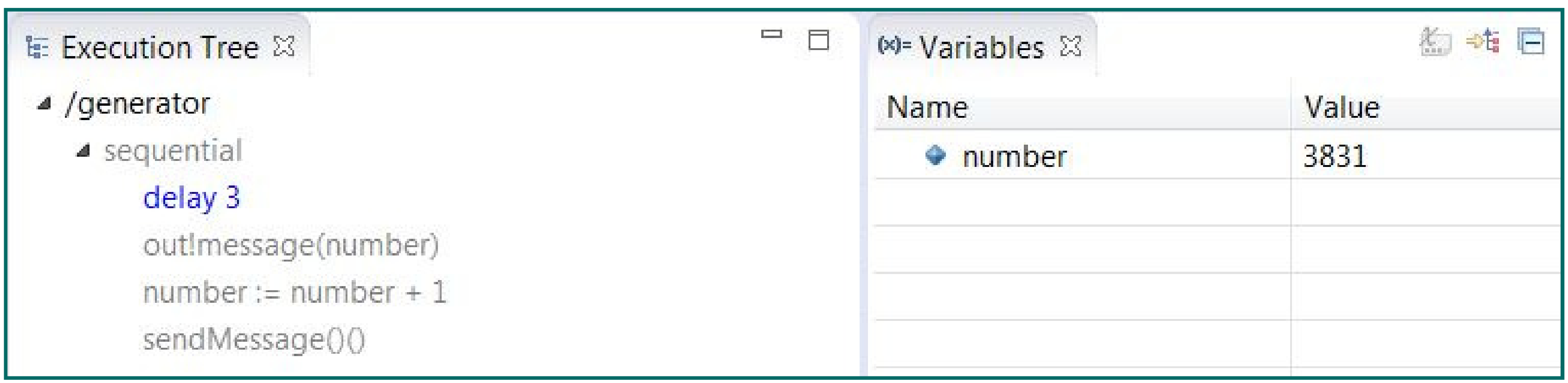
Debug perspective

Debug view shows all process instances (and simulation time):

- Red: can do step
- Blue: can do time step
- Black: no step possible

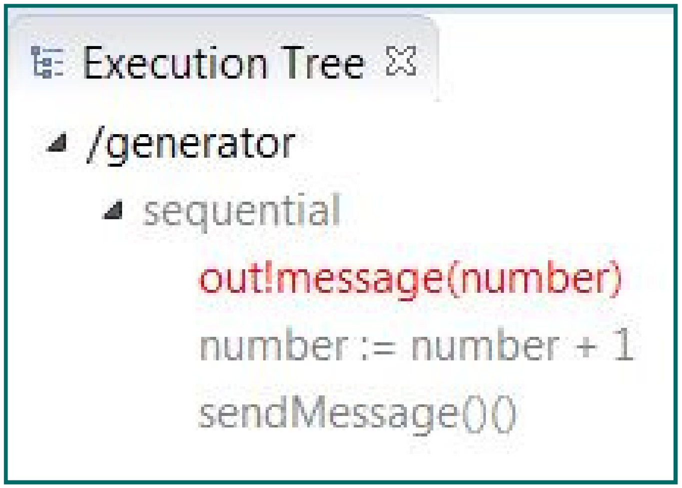


When not simulating, click on instance to show its execution tree and its variables

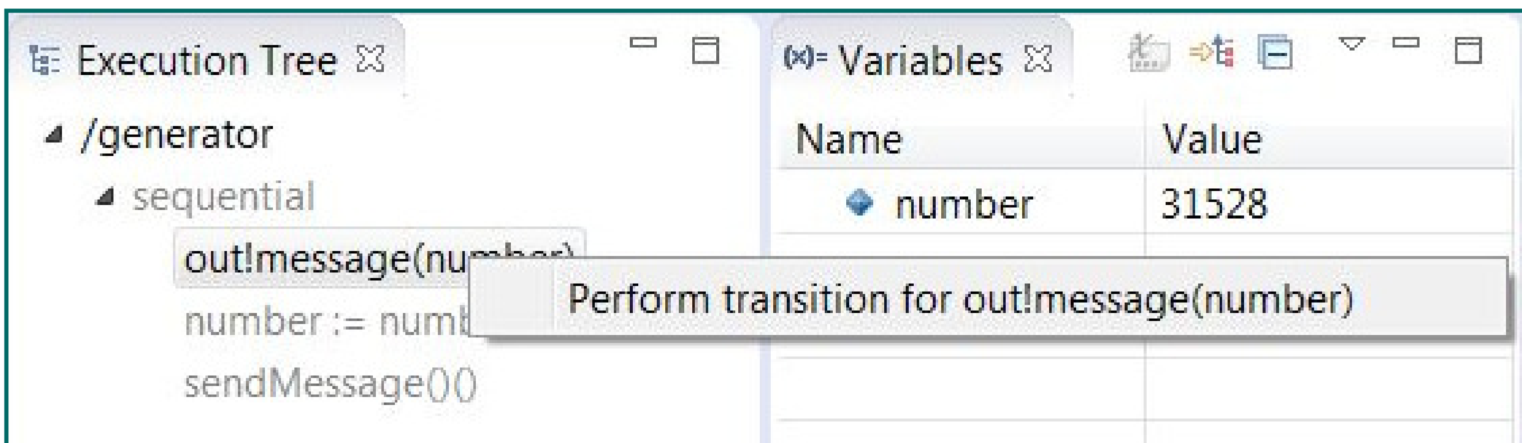


Select step to execute from execution tree

To execute a step of a particular instance, double click on it



or right-click on it and click on the "Perform transition …" message:



Hands on stream1b.poosl

- [Stop simulation of stream1a]
- Copy stream1a.poosl to stream1b.poosl
- Edit stream1b.poosl, add delays
- Simulate stream1b in debug mode
  - Try time steps, inspect variables
- Investigate changes in delay values
- Try the delay example:

```
delay 7;
out ! msg (currentTime)
```

```
delay 3;
out ! msg (currentTime);
delay 2;
out ! msg (currentTime)
```

```
receive()()
    in ? msg(number);
    receive()()
```

stream2.poosl – process parameters & multiple instances

Copy file to stream2 and add parameters to processes

Generator:

```
process class Generator(id: String, prepareTime: Integer)
ports
    out
messages
    out ! message(String,Integer)
```

```
sendMessage()()
    delay prepareTime;
    out ! message(id,number) ;
    number := number + 1 ;
    sendMessage()()
```

Viewer:

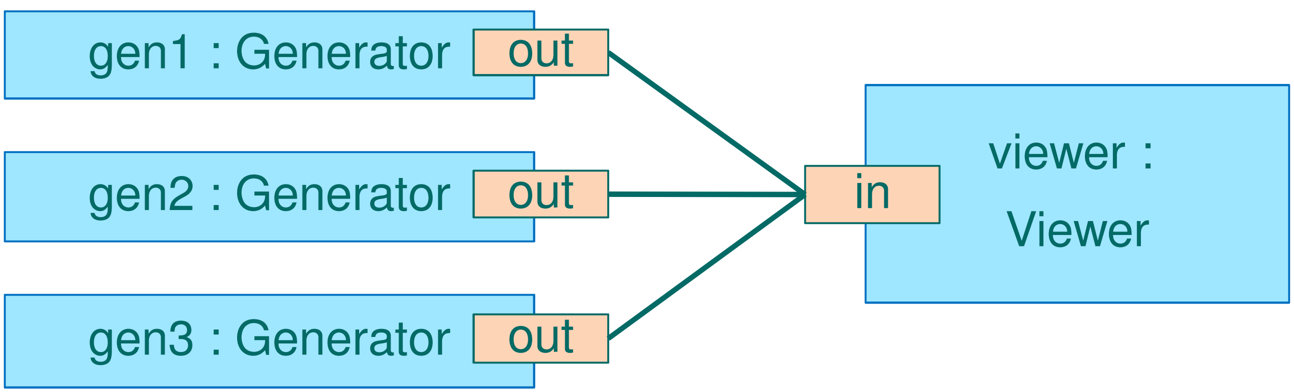
- add time parameter “receiveTime” to class definition
- receive messages with 2 parameters: String and Integer

Embedded Systems Innovation BY TNO

stream2.poosl – process parameters & multiple instances

Define system as follows:

```
system
instances
    gen1:   Generator(id := "First", prepareTime := 7)
    gen2:   Generator(id := "Second", prepareTime := 4)
    gen3:   Generator(id := "Third", prepareTime := 3)
    viewer: Viewer(receiveTime := 5)
channels
    { gen3.out, gen1.out, gen2.out, viewer.in }
```



Hands on stream2.poosl

- Copy stream1b.poosl to stream2.poosl
- Edit stream2
- Simulate the model in debug mode
  - Observe the sequence diagram
- Change timing parameters
  - For which values of timing parameters will all generators send their messages regularly?

Data Objects

- Passive sequential entities
- Can be created dynamically
- Encapsulate their attributes
- Accessible only through method calls
- Single inheritance and polymorphism

stream3.poosl – define data class

Copy to stream3.poosl and add data class:

```
data class Message extends Object
variables
    Identifier : String,
    Number : Integer
methods
    setIdentifier(I : String) : Message
        Identifier := I ;
        return self
    setNumber(n : Integer) : Message
        Number := n ;
        return self
    getIdentifier() : String
        return Identifier
    getNumber() : Integer
        return Number
    printString() : String
        return "[ Id: " + Identifier + " Nr: " + Number printString + " ]"
```

use of pre-defined printString for Integer

stream3.poosl – use data class

Generator

```
messages
    out ! message(Message)
```

methods calls returning message

```
sendMessage()() | m : Message |
    delay prepareTime;
    m := new(Message) setIdentifier(id) setNumber(number);
    out ! message(m) ;
    number := number + 1 ;
    sendMessage()()
```

Viewer

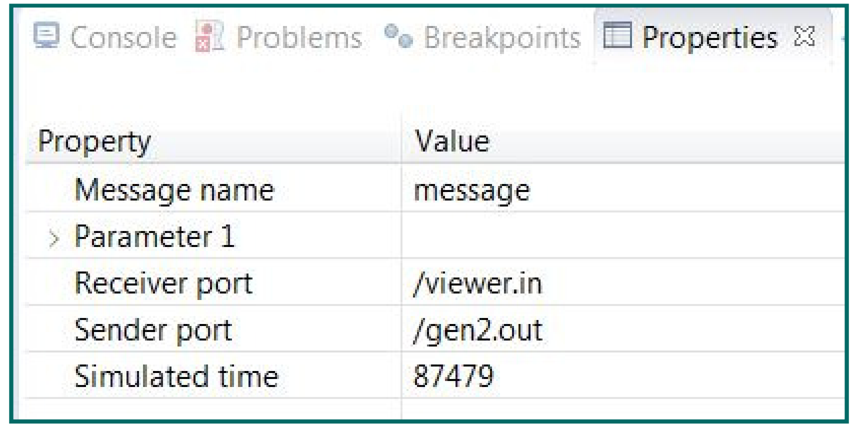
```
messages
    in ? message(Message)
```

```
receiveMessage()() | m : Message |
    in ? message(m);
    delay receiveTime ;
    receiveMessage()()
```

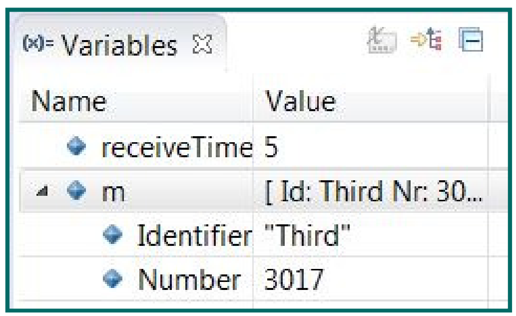
stream3.poosl

Simulate the model in debug mode

- Inspect sequence diagram;
  see property view in lower left corner,
  click on messages in sequence diagram



- Inspect received message
  in Variables view

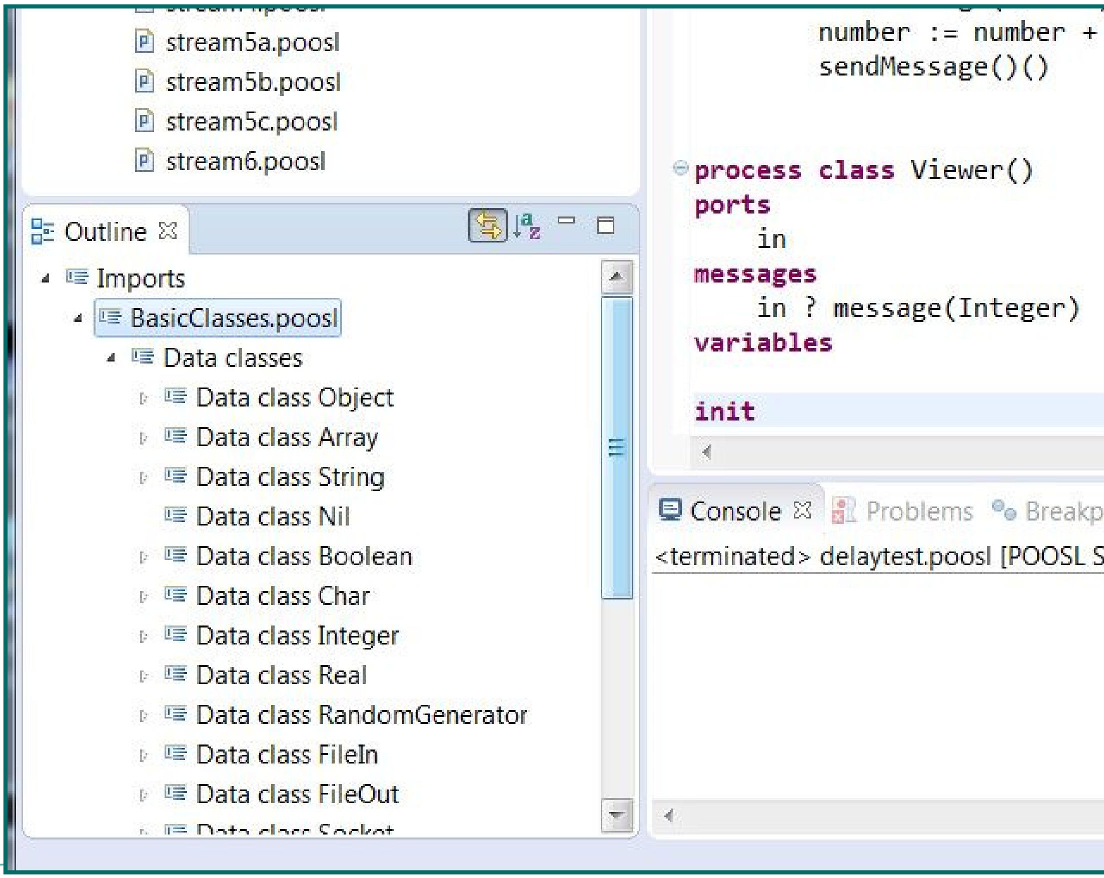


- Change printString of messages and
  see how it is used in sequence diagram

Predefined data classes

Implicitly imported: BasicClasses.poosl ;

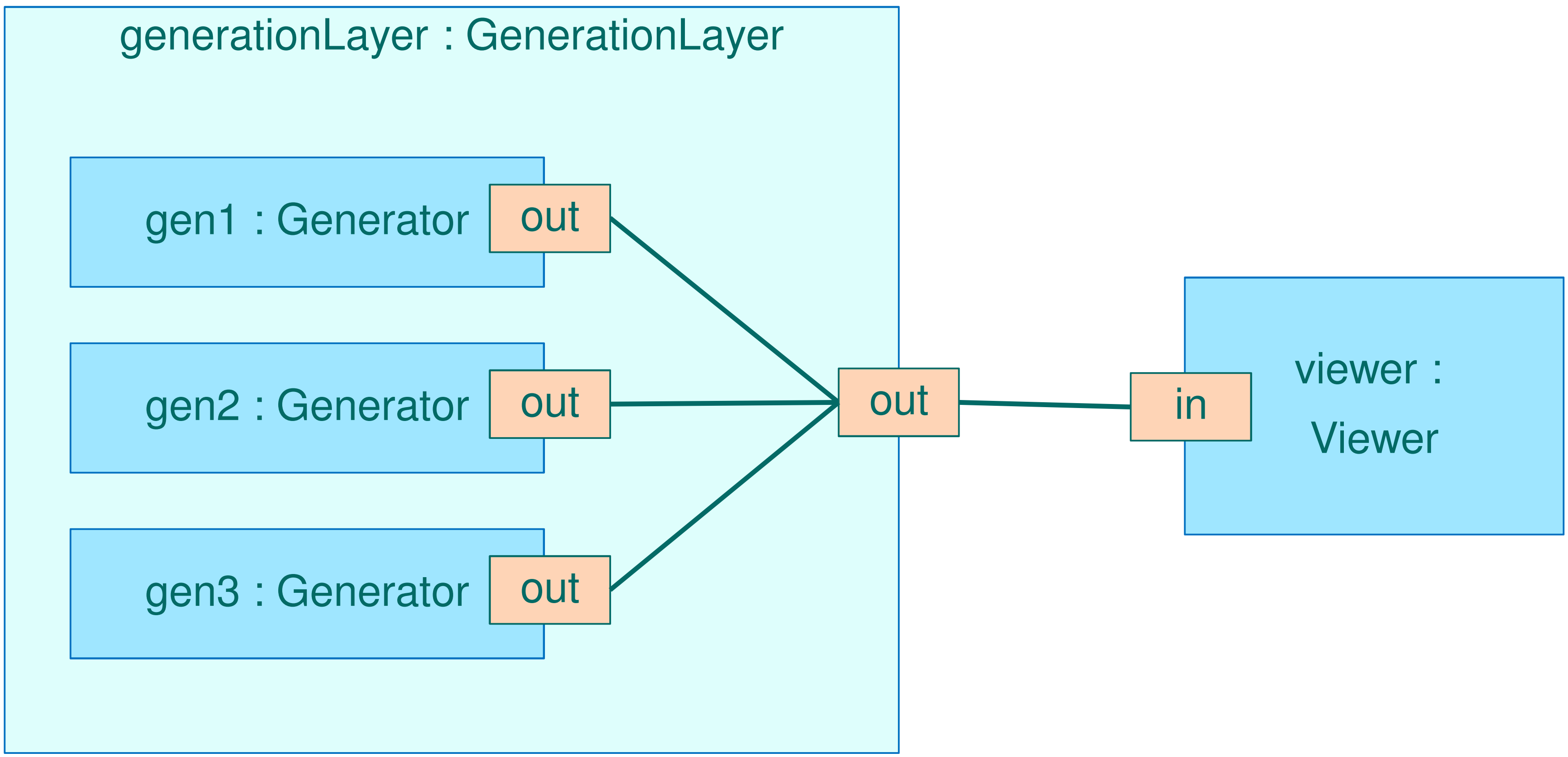
- See Outline view in lower left corner of editor



Hands on stream3.poosl

- Copy stream2.poosl to stream3.poosl
- Edit stream3
- Simulate the model in debug mode
  - Observe the sequence diagram, inspect the Properties view
  - Inspect received messages in the Variables view
- Experiment with changes in the printString method of messages
  - Observe the use in the sequence diagram
- Inspect the contents of BasicClasses.poosl

stream4.poosl - cluster



stream4.poosl - cluster

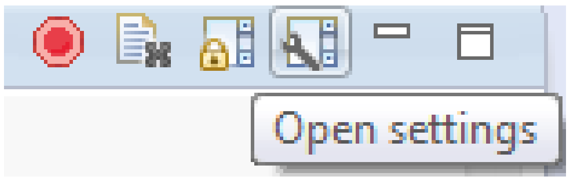
```
cluster class GenerationLayer()
ports
    out
instances
    gen1:   Generator(id := "First", prepareTime := 14)
    gen2:   Generator(id := "Second", prepareTime := 18)
    gen3:   Generator(id := "Third", prepareTime := 25)
channels
    {gen1.out, gen2.out, gen3.out, out}

system
instances
    generationLayer : GenerationLayer()
    viewer: Viewer(receiveTime := 5)
channels
    { generationLayer.out, viewer.in }
```

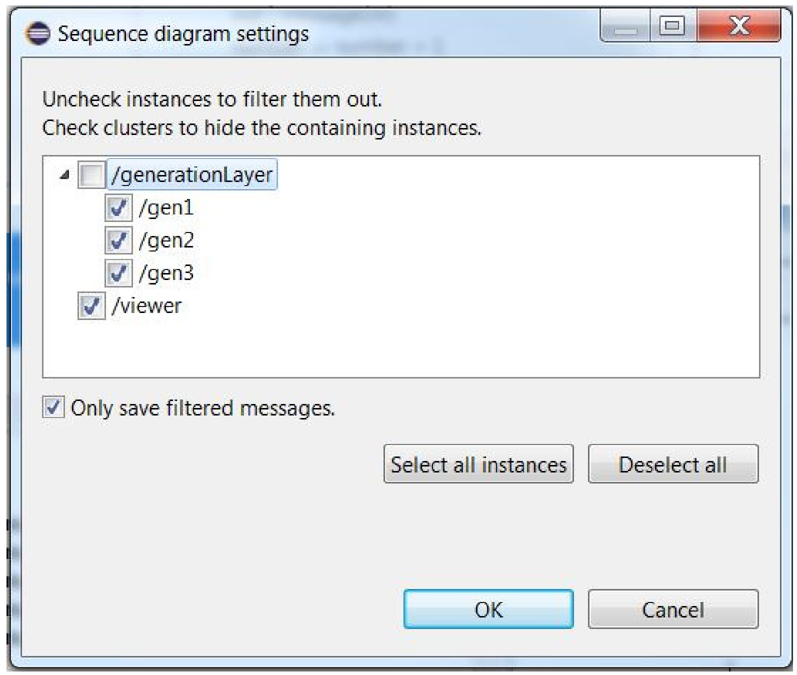
Hands on stream4.poosl

Copy stream3.poosl to stream4.poosl

Simulate the model with a cluster in debug mode.

Experiment with changes in the setting of the sequence diagram





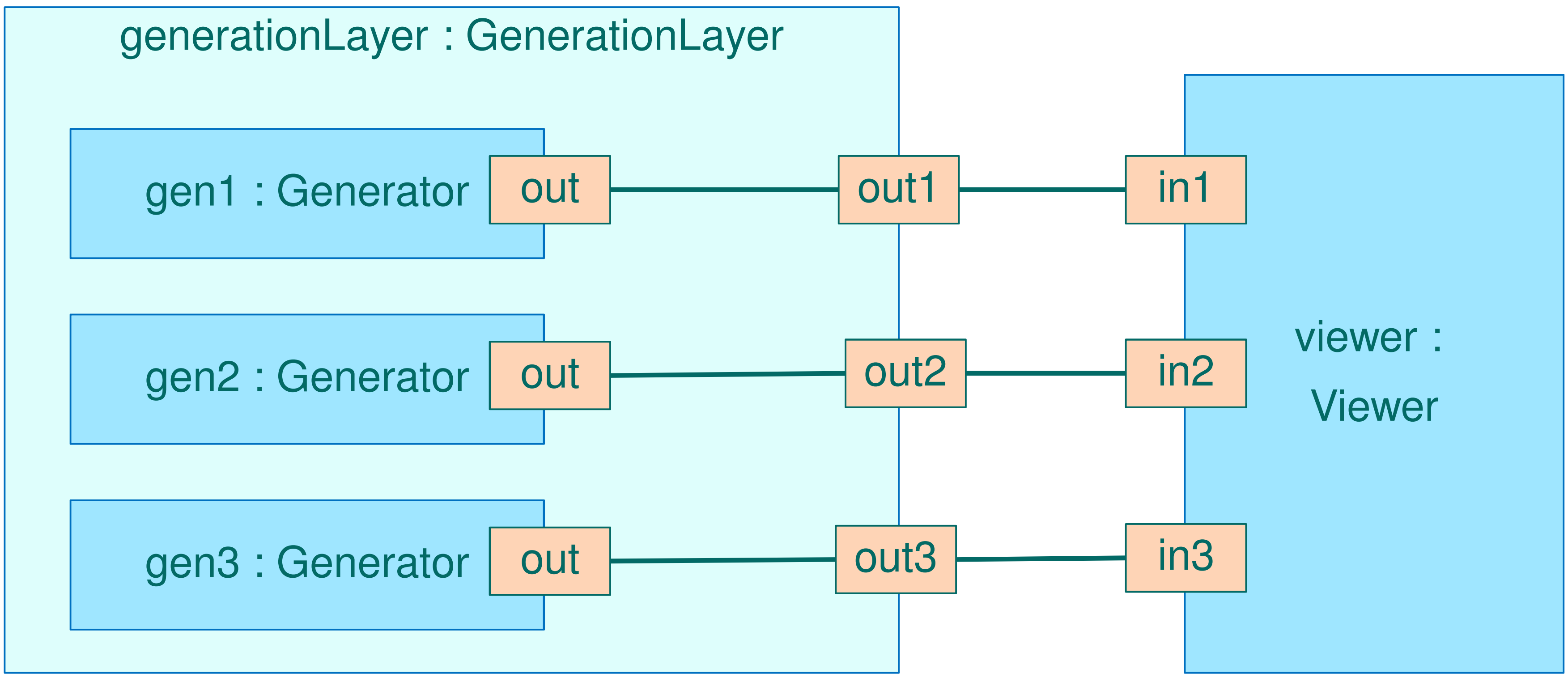
Also try right click on instance in sequence diagram and

- Hide
- Collapse
- Expand

Note: by default, message buffer contains last 1000 messages

stream5a.poosl - multiple channels

Adapt the **cluster** to obtain three output channels



stream5a.poosl - select input

Viewer – use three input channels in1, in2, in3

```
receiveMessage()() | m : Message |
    sel
        in1?message(m)
    or
        in2?message(m)
    or
        in3?message(m)
    les ;
    delay receiveTime;
    receiveMessage()()
```

Use <CTRL>-<SPACE>
to insert select statement

Use quick-fix to correct
“messages” part

Adapt **system definition** to connect all three to input of viewer

stream5b.poosl - guards

Add **guards** to the receive statements in the viewer, e.g.

```
receiveMessage()() | m : Message |
    sel
        [next = 1] in1?message(m); next := 2
    or
        [next = 2] in2?message(m); next := 3
    or
        [next = 3] in3?message(m); next := 1
    les ;
    delay receiveTime;
    receiveMessage()()
```

Add declaration of “next” and an initialization method, e.g.,

- init()() which initializes “next” and calls receiveMessage()()

stream5c.poosl– conditional receive

Restrict received messages by conditional receive, e.g.,

```
receiveMessage()() | m : Message |
    sel
        in1?message(m | m getNumber < 5)
    or
        in2?message(m | m getNumber < 10)
    or
        in3?message(m)
    les ;
    delay receiveTime ;
    receiveMessage()()
```

Hands on stream5.poosl

Save stream4.poosl for later use and copy it to stream5a.poosl

- Edit stream5a
  - Simulate in debug mode; observe that in the sequence diagram the channel is visible in Properties view in lower left corner
- Edit stream5b
  - Simulate in debug mode
- Edit stream5c
  - Simulate in debug mode

stream6.poosl – asynchronous communication add queue

Create library with queue model

- Right click on project name “workshop.example” > New > Folder
  - name: lib
- Right click on folder name > New > POOSL Model
  - name: queue.poosl
- Open nl.esi.poosl.example/models/MPSoC/common.poosl
- Copy data classes Element and Queue to queue.poosl

Make a copy of model stream4.poosl and name it: stream6.poosl

In stream6.poosl insert on the first line:

```
import "../lib/queue.poosl"
```

stream6.poosl – change Viewer to include queue

```
process class Viewer(receiveTime: Integer)
ports
    in
messages
    in ? message(Message)
variables
    messageQueue : Queue
init
    initialize()()
methods
    initialize()()
        messageQueue := new(Queue) init() ;
        par
            receiveMessages()()
        and
            inspectMessages()()
        rap

    receiveMessages()() | m : Message |
        in ? message(m);
        messageQueue add(m);
        receiveMessages()()

    inspectMessages()() | m : Message |
        [messageQueue notEmpty()] m := messageQueue remove();
        delay receiveTime;
        inspectMessages()()
```

parallel construct

guard

Hands on stream6.poosl

- Copy stream4 to stream6.poosl
- Edit stream6
- Simulate stream6.poosl
  - Observe the behaviour, including the queue size
  - Change timing behaviour to get more / less elements in the queue

Random generator

See Outline (lower left corner of editor)

Imports > BasicClasses > Data classes >

Data class RandomGenerator

- random(): Real ; uniform distribution, value in [0,1)
- randomInt(i: Integer): Integer ; value in { 0, 1, …, i-1} (i>0)
- randomSeed(): RandomGenerator
- seed(i: Integer): RandomGenerator ; set seed to i

Documentation on BasicClasses:

http://poosl.esi.nl/downloads/manuals/BasicClasses.pdf

See, for instance, FileIn, FileOut, Socket

stream7.poosl - use random generator

```
process class Generator(id: String, min, max: Real)
ports
    out
messages
    out ! message(Message)
variables
    number : Integer,
    randomGenerator : RandomGenerator
init
    initialize()()
methods
    initialize()()
        number := 1 ;
        randomGenerator := new(RandomGenerator) randomiseSeed;
        sendMessage()()
    sendMessage()() | m : Message |
        delay min + (max * randomGenerator random);
        m := new(Message) setIdentifier(id) setNumber(number);
        out ! message(m) ;
        number := number + 1 ;
        sendMessage()()
```

stream7.poosl – system definition

Cluster is not used

notation for Real

```
system
instances
    gen:   Generator(id := "First", min := 5.0, max := 10.0)
    viewer: Viewer(receiveTime := 15)
channels
    { gen.out, viewer.in }
```

- Simulate and observe Queue size in Variables window

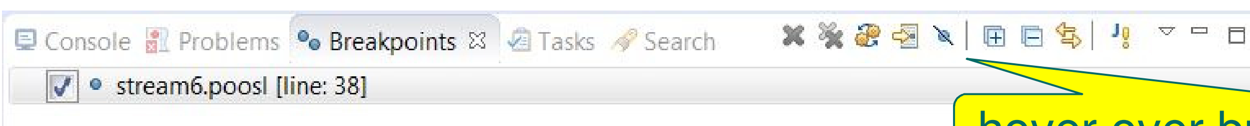
Breakpoints in process methods

3 ways to add breakpoints:

- Double click ruler in front of line in editor
- Right click ruler in front of line and select Toggle Breakpoint.
- Use shortcut Ctrl+Shift+B to set a breakpoint on current line.

```
37          out ! message(m) ;
38          number := number + 1 ;
39          sendMessage()()
```

- Breakpoints are visible in Breakpoints view (bottom part)



hover over buttons to see function

- “When a breakpoint is hit during simulation, simulation will be suspended.

Atomicity brackets

```
process class atomicityTest()
ports
messages
variables
    temp1, temp2, number : Integer
init
    init()()
methods
    init()()
        number := 0;
        par
            update1()()
        and
            update2()()
        rap
    update1()()
        temp1 := number; number := temp1+1
    update2()()
        temp2 := number; number := temp2+1

system
instances
    test : atomicityTest()
channels
```

Observe that number can be 1 at the end
Compare with a version where **atomicity brackets** are added:

```
update1()()
    {temp1 := number; number := temp1+1}
update2()()
    {temp2 := number; number := temp2+1}
```

Other language constructs

```
if (id = "first")
then
      id := "second"
else
    id := "third"
fi
```

```
while (index <= 10)
do
      doTask()();
      index := index + 1
od
```

```
abort
      doTask()()
with
      in?stopTask()
```

```
interrupt
      doTask()()
with
      somethingUrgent()()
```

Other editor features

- Task markers: // TODO // FIXME // XXX
- Searchable outline tree with model structure (including imports)
- Refactor, search-and-replace
- Print, undo, redo, …

As separate plug-in:

- Import from SHESim XML
- Export to SHESim XML (with default graphical layout)

Library & Examples

See: nl.esi.poosl.examples/libraries

- **distributions.poosl**

  distribution functions:
  - Bernoulli, Beta, Beta4, DiscreteUniform, Exponential, Gamma, Discrete, Normal, PERT, Triangle, Uniform, Weibull, Histogram

- **performance.poosl**

  function to observe performance
  - PerformanceMonitor, LongRunSampleAverage, LongRunSampleVariance, LongRunTimeAverage, LongRunTimeVariance, LongRunRateAverage, ConfidenceInterval

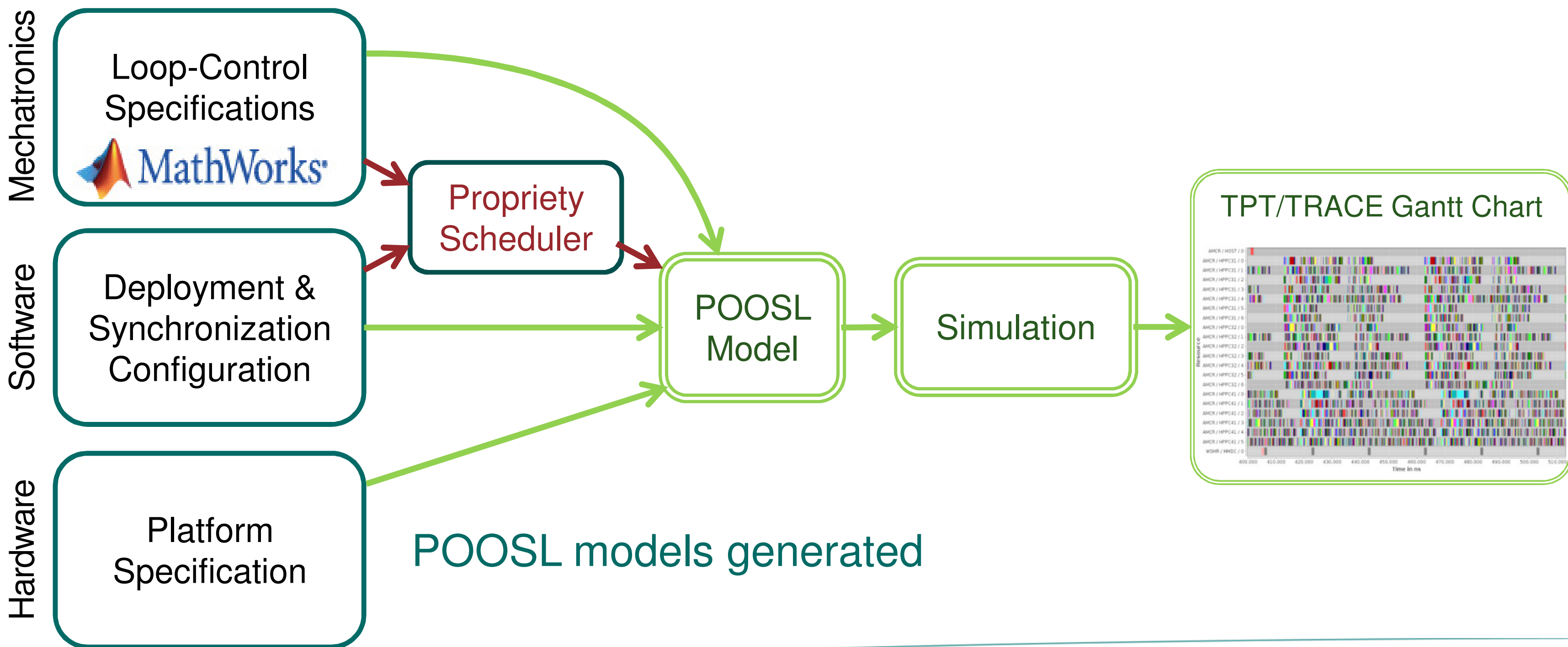
Industrial Applications of POOSL

Application of system level modeling using POOSL	Industry
Design Space Exploration for Advanced Driver Assistance E/E Architectures	NXP
Performance Analysis and DSE for Mixed-Criticality Computing Systems in Automotive	TNO and NXP
Rapid Prototyping of a Hybrid Architecture for Movement Control of iXR Systems	Philips Healthcare
Rapid Prototyping of a Proposed New Architecture for Movement Control of iXR Systems	Philips Healthcare
Performance Analysis of the Imaging Chain in iXR Systems	Philips Healthcare
Rapid Prototyping of the Requirements and Design of the Pedal Handling Component of iXR Systems	Philips Healthcare
Simulation of Distributed Lighting Systems	Philips Lighting
Variability Analysis in Fixed-Order Multi-Core Schedules	ASML
Performance Prediction and Design-Space Exploration for Wafer Scanners	ASML
Performance Modeling of SmartTV Systems	TPVision
Performance Analysis of Ethernet AVB	NXP
Load Regulation Modeling of Variability 3P Applications for Tacticos	Thales Navel Systems
Performance Analysis of a Printer Data Path	Océ
Performance Analysis of Compact Picking Systems	Vanderlande Industries
Simulation Model of Automatic Case Picking System Concept	Vanderlande Industries

Real-time performance analysis @ ASML

Goal: bottleneck identification & exploration of design alternatives

- Focus: real-time performance for (multi-rate & multi-core) loop control
- Scale (medium size case): 1500 sensors/actuators, 200 loop control networks with 4500 tasks, 60 processors in 6 racks



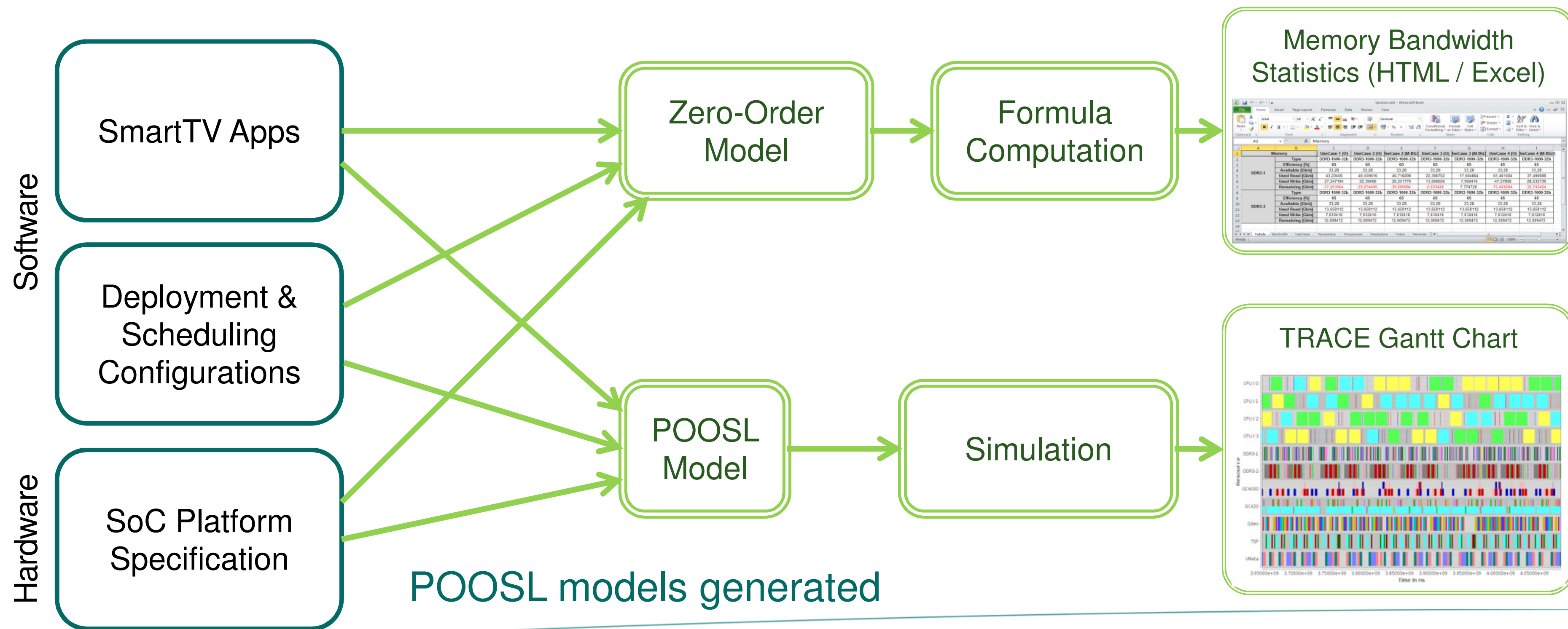
POOSL models generated

Embedded Systems Innovation BY TNO

Performance & memory @ TPVision (Smart TVs)

Goal: bottleneck identification & exploration of design alternatives

- Focus: real-time performance & memory bandwidth utilization
- Scale: 50 tasks in video/graphics pipelines operating in 5 use cases, 1 multi-core CPU, 2 GPUs, 8 accelerators, 2 main memories

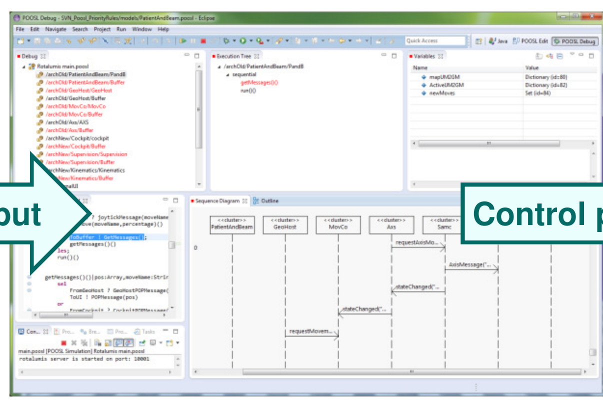


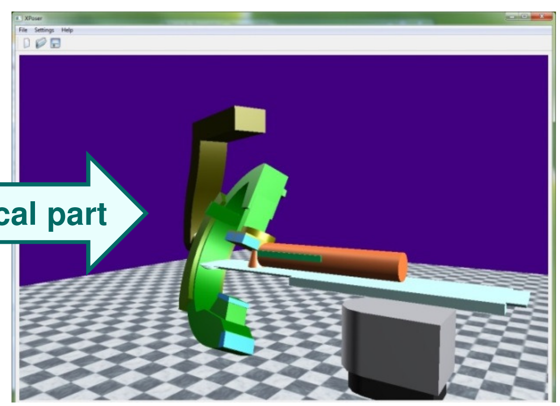
Embedded Systems Innovation BY TNO

Architecture validation @ Philips Healthcare

Use of pre-defined sockets to connect to

- Java program
- Ogre 3D graphics engine



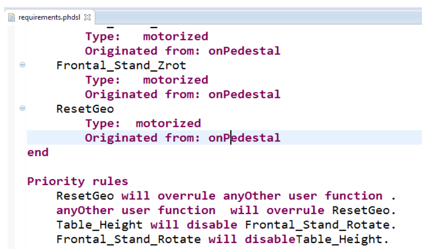


User Input **POOSL model and simulation** **Visualization**

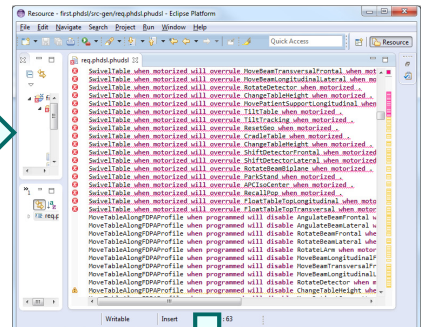
Requirements validation @ Philips Healthcare

- Generate part of POOSL model from Domain Specific Language

Edit requirements

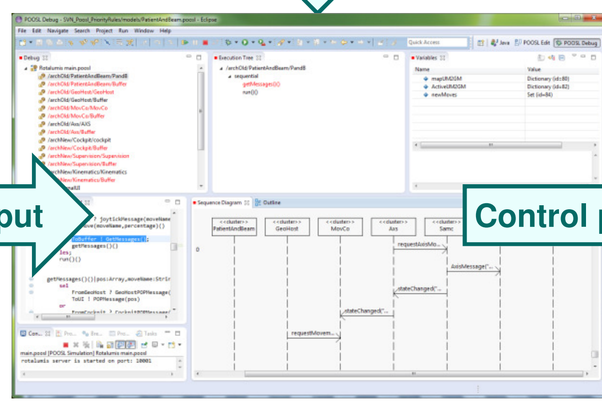


Analyze completeness of requirements

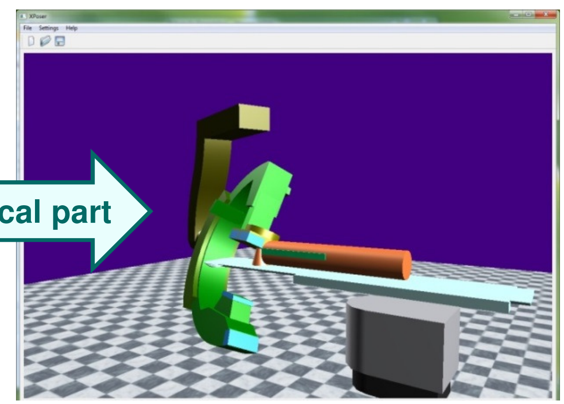


Generate POOSL model

User input



Control physical part



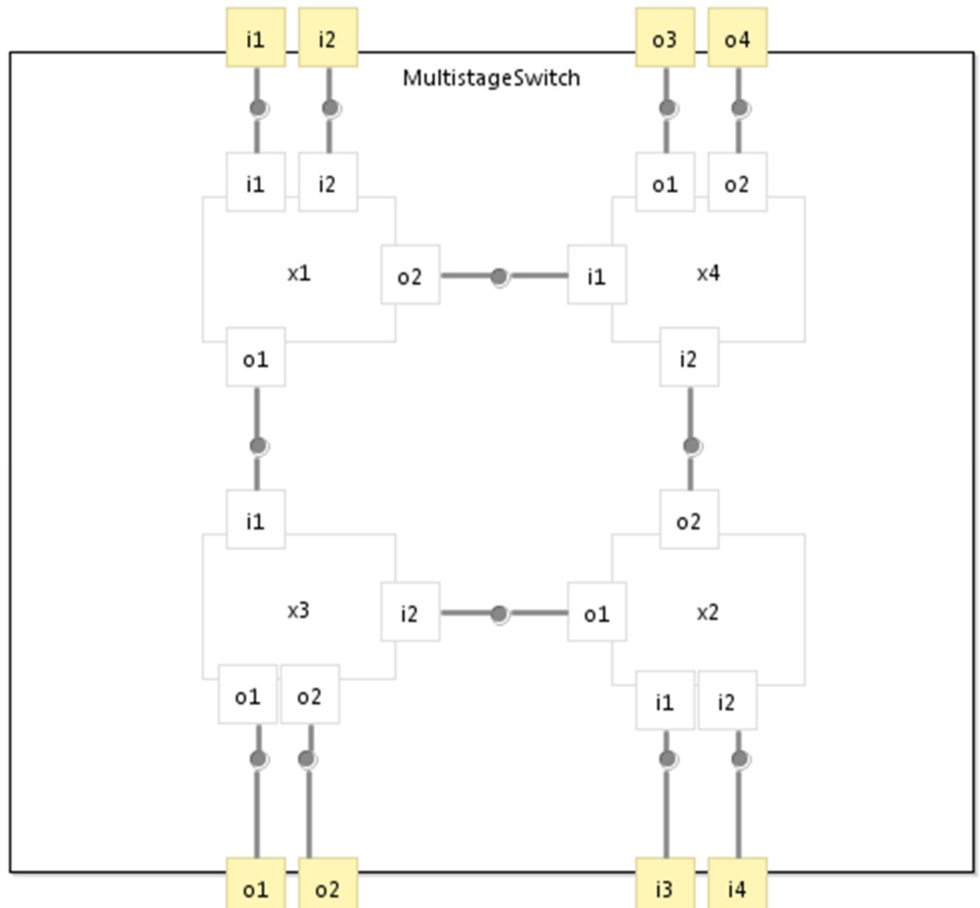
User Input **POOSL model and simulation** **Visualization**

Current work – textual + graphical editor

Integration of Xtext and Sirius to edit system structure graphically

Xtext

Sirius



More information

nl.esi.poosl.examples

- examples
  - SocketExample: client-server using SocketProcess (and random)
  - ATMSwitch: using [Bounded] FIFO buffer
  - MarsRover: tasks, mutex, processor
  - SoCInterconnects: bus with arbitration
  - MPSoC: use of distributions and performance libraries for mapping of applications to execution platform with scheduling, memory, battery, ….

http://poosl.esi.nl/

jozef.hooman@tno.nl
arjan.mooij@tno.nl
bart.theelen@tno.nl