Integrating a Neutral Action Language in a DEVS Modelling Environment

Bruno Barroca\textsuperscript{1}, Sadaf Mustafiz\textsuperscript{1}, Simon Van Mierlo\textsuperscript{2}, Hans Vangheluwe\textsuperscript{2}

August 24\textsuperscript{th}, 2015, Athens, Greece
SIMUtools – Eight EAI International Conference on Simulation Tools and Techniques
Motivation

• DEVS
  • a universal simulation “assembly language”
  • enables modular combination of different modelling languages and their simulators
• Need for an integrated DEVS environment (e.g., visualization)
• Need for standard DEVS libraries (e.g., to be easily reused)
• Need for platform-independent DEVS formalisms
• Need for integration of neutral action code


DEVS Background

• Compositional modelling and simulation of discrete event systems

• **Atomic model**
  
  • $M = < X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, \tau_a >$
  
  • where $X =$ input set, $Y =$ output set, $S =$ state set, $\delta_{\text{int}} =$ internal transition function, $\delta_{\text{ext}} =$ external transition function, $\delta_{\text{conf}} =$ confluent function, $\lambda =$ output function, $\tau_a =$ time advance function

• **Coupled model**
  
  • composition of several concurrent submodels (either atomic or coupled)
  
  • submodels have ports connected by *channels*
  
  • channels define *transfer functions* to translate output to input


DEVS Formalisms

- **DEVSPro**
  - based on DEVS theory and at an abstraction level closer to programming languages (Python-like syntax)
  - allows expert users to exploit the full power of DEVS
  - basis for synthesizing simulation-specific code
  - useful for exchange and re-use (between different DEVS simulators)

- **DEVSLang**
  - a restricted user-friendly variant of DEVS (to match the limited expressiveness of visual DEVS with finite states)
  - blended textual/visual modelling
  - a textual DEVS representation to describe models in a more user-friendly fashion

- **Action Code**
  - explicitly modelled action language using HUTN (Human Usable Textual Notation)
DEVS IM&S Environment

[Diagram showing the DEVS IM&S Environment, including components like DEVSLang, Transformation Metamodel, DEVSPro, etc.]

Linguistic Conformance
Concrete Syntax Mapping
Transformation
Deployment
DEVSLang: Producer-Consumer Example Model (1)

```python
## Producer-Consumer example model in HUTN DEVSLang concrete syntax

# State definitions
statedef GeneratorState()
statedef ProcessorState(job=None)
statedef CollectorState(nr_of_jobs)

# Event definitions
event Job(jobSize)
```
# Functional components

## atomic Generator(
outports p_out

initial generating:

state generating:
  time advance:
    return 1
  internal \rightarrow generating
  output:
    return \{p_out: \{Job(0.3)\}\}

## atomic Processor(
outports p_out

initial idle

state idle:
  time advance:
    return \infty
  external \rightarrow processing:
    action:
      return \{job: p_in[0]\}

state processing:
  time advance:
    return job.jobSize
  internal \rightarrow idle
  output:
    job.jobSize /= 1
    return \{p_out: \{job\}\}

## atomic Collector(
outports p_in

initial waiting

state waiting:
  time advance:
    return \infty
  external \rightarrow waiting:
    action:
      return \{nr_of_jobs: nr_of_jobs + 1\}

## coupled CoupledProcessor(
outports p_out

instances:
  p1 = Processor()
  p2 = Processor()

connections:
  from p_in to p1.p_in
  from p1.p_out to p2.p_in
  from p2.p_out to p_out

## coupled Root(

instances:
  g = Generator(a,b)
  cp = CoupledProcessor()
  p = Processor()
  c = Collector()

connections:
  from g.p_out to cp.p_in
  from cp.p_out to p.p_in
  from p.p_out to c.p_in
DEVSLang Meta-model
DEVSPRO: Producer-Consumer Example Model

top:
    import sys

statedef GeneratorState:
    constructor(name):
        self.name = name

statedef ProcessorState:
    constructor(name, job=None):
        self.name = name
        self.job = job

atomic Processor:
    constructor():
        self.state = ProcessorState('idle')
        self.outports = {'p_out'}
        self.inports = {'p_in'}

    timeAdvance():
        if self.state == ProcessorState('idle') and self.state.name == 'idle':
            return INFINITY
        else:
            if self.state == ProcessorState('processing') and self.state.name == 'processing':
                return self.state.job.jobSize

    outputFnc():
        if self.state == ProcessorState('idle') and self.state.name == 'idle':
            return {}
        else:
            if self.state == ProcessorState('processing') and self.state.name == 'processing':
                return {self.outports['p_out'][self.state.job]: [self.state.job]}

    intTransition():
        if self.state == ProcessorState('idle') and self.state.name == 'processing':
            return ProcessorState('idle')

    extTransition(inputs):
        if self.state == ProcessorState('idle') and self.state.name == 'processing':
            return ProcessorState('processing', job=inputs[self.inports['p_in']][0])
DEVSPro Meta-model
## Mapping and Merging

<table>
<thead>
<tr>
<th>DEVSLang Concepts</th>
<th>DEVSPRO Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>statedef SDName(pars..)</code></td>
<td><code>statedef SDName: constructor(pars..):</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>attribute CName() with SDName:</td>
<td><code>atomic CName: constructor():</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>initial ISName</td>
<td><code>self.state = SDName(ISName)</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>atomic CName(args..) ..</td>
<td><code>atomic CName: constructor(args..):</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>atomic CName() ..</td>
<td><code>atomic CName: constructor():</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>imports p, q...</td>
<td><code>self.imports = {'p', 'q', ..}</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>outputs r, s...</td>
<td><code>self.outputs = {'r', 's', ..}</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>atomic CName():</td>
<td><code>atomic CName: timeAdvance():</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>state SName:</td>
<td><code>if self.state == 'SName':</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>timeAdvance:</td>
<td><code>OO(Statements..)</code></td>
</tr>
<tr>
<td>Statements..</td>
<td><code>..</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>atomic CName() with SDN:</td>
<td><code>atomic CName:</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>state SN:</td>
<td><code>extTransition(inputs):</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>external -&gt; SON:</td>
<td><code>if self.state == 'SN'</code></td>
</tr>
<tr>
<td><code>..</code></td>
<td><code>..</code></td>
</tr>
<tr>
<td>condition:</td>
<td><code>and Cond(Stms..):</code></td>
</tr>
<tr>
<td>Stms..</td>
<td><code>OO(Stms')..</code></td>
</tr>
<tr>
<td>action:</td>
<td><code>return SDN('SON', args)</code></td>
</tr>
<tr>
<td>Stms'..</td>
<td><code>..</code></td>
</tr>
</tbody>
</table>
| `..` | `..`
Action Code Language Meta-model
Validating DEVS Models

- Static semantics analysis
- Conformance checking

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Time Advance Function</th>
<th>Internal Function Condition</th>
<th>Internal Function Action</th>
<th>External Function Condition</th>
<th>External Function Action</th>
<th>Confluent Function Condition</th>
<th>Confluent Function Action</th>
<th>Output Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>State variables</td>
<td>Float</td>
<td>Boolean</td>
<td>Dictionary</td>
<td>Boolean</td>
<td>Dictionary</td>
<td>Boolean</td>
<td>Dictionary</td>
<td>Read Only</td>
</tr>
<tr>
<td>Import variables</td>
<td>Read Only</td>
<td>Read Only</td>
<td>Read/Write</td>
<td>Read Only</td>
<td>Read/Write</td>
<td>Read Only</td>
<td>Read/Write</td>
<td>Read Only</td>
</tr>
<tr>
<td>Output variables</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Elapsed variable</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Read Only</td>
<td>N/A</td>
<td>Read Only</td>
<td>N/A</td>
<td>Write Only</td>
</tr>
</tbody>
</table>
Action Code Analysis: DEVSPro

```python
atomic Generator:

    constructor():
        self.state = GeneratorState('generating')
        self.outports = {'p_out'}

    timeAdvance():
        if self.state == GeneratorState and self.state.name == 'generating':
            return 1

    outputFnc():
        if self.state == GeneratorState and self.state.name == 'generating':
            return {'p_out'}:

    intTransition():
        if self.state == GeneratorState and self.state.name == 'generating':
            return GeneratorState('generating')
```

macroots-mbp:HUTN bbbarroca$ python devssim.py -l devspro models/ProducerConsumer.devsp
Syntax error at line 35 and column 29. Read: ' '.

Action Code Analysis: Textual DEVSLang

# Functional components
atomic Generator() with GeneratorState:
  outports p_out

  initial generating

state generating
  time advance:
    return 1

internal -> generating
  output:
    return {p_out: [Job(0.3)]}

state processing:
  time advance:
    return job.jobSize

internal -> idle
  output:
    if job.jobSize != 1:
      return {p_out: [job]}

macroots-mbp:HUTN bbarroca# python devssin.py -l devslang models/ProducerConsumer.devs
Context error in atomic 'Processor':
cannot modify state 'job' variable inside output function.

macroots-mbp:HUTN bbarroca# python devssin.py -l devslang models/ProducerConsumer.devs
Syntax error at line 18 and column 9. Expected 't' or 'space'.
Instead read: 't'.
DEVSLang Analysis in AToMPM (1)
DEVSLang Analysis in AToMPM (2)
FTG+PM for DEVS

FTG+PM for DEVS

The DEVS Models can be mapped from UML class diagrams, petri nets, statecharts, etc. and can hence be transformed automatically to DEVS models.

Extract legacy code in languages such as C or Java, can be reverse engineered to DEVSPro models.

Mapping of properties (defined in a CTL-like language) to Uppaal, and an optimization loop in the process model have not been shown in this version of our model.
FTG+PM for DEVS

1. Translate to TA
2. Uppaal
3. Analyze Behaviour
4. Analysis Trace
5. Check Analysis Trace
6. DEVSPro
7. PyPDEVs
8. Simulate DEVs
9. Simulation Trace
10. Check Simulation Trace
11. Boolean

12. Feedback to AToPM
13. Refine DEVSLang Model
14. Translate to TA
15. Correct AToMPM Model
16. Analyze Behaviour
17. Check Analysis Trace
18. Errors?
19. Boolean
20. Generate DEVSPro
21. DEVSPro
22. PyPDEVs
23. Simulate DEVs
24. Simulation Trace
25. Boolean
Conclusion

• Textual DEVS languages, DEVSPro and DEVSLang (textual/visual), with neutral action code
• Validation of DEVS models: syntax and semantic analysis of action code
• Integrated modelling, analysis, simulation, and debugging environment for DEVS
Questions?

bbarroca@cs.mcgill.ca
DEVS Background (3)

An abstract simulator for Parallel DEVS computes the next state of the system (a “step”) until its end condition is satisfied.

- compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components);
- execute the output function for each imminent component, causing events to be generated on the output ports;
- route events from output ports to input ports, translating them in the process by executing the transfer functions;
- determine the type of transition to execute for the atomic DEVS model, depending on it being imminent and/or receiving input;
- execute, in parallel, all enabled internal, external, and confluent transition functions; and
- compute, for each atomic DEVS model, the time of its next internal transition (specified by its time advance function).