On the Prototype-Based Object Orientation in Modeling and Simulation

ASIS 2006

Vladimír Janoušek

Outline

- Context
- Class-Based and Prototype-Based Object Orientation
- Exploratory programming (and image-based systems)
- DEVS formalism for modeling
- Prototype-Based OO Modeling, Exploratory Modeling
- Summary

Context

- OOPN/PNtalk
  high-level visual formalism used as a full-featured programming language
- SmallDEVS
  hierarchical component framework based on systems theory featuring openness, reflectivity, interactivity
- PNtalk is now being nested to SmallDEVS as one of high-level languages for component specification
- App. area:
  Model-based development, Model continuity
  Multiparadigm modeling
  Evolvable and reflective models
- Prototype-Based OO and reflectivity are the SmallDEVS aspects being discussed in this talk

Approaches to the Object Orientation

- Class-based OO
  - Simula
  - Smalltalk (classes are objects, methods are objects, image),
  - C++, Java - mainstream
- Prototype-based OO
  - Self (Smalltalk-like system)
  - JavaScript, ... (for scripting)

Prototype-Based Object Orientation

- No key feature of class-based approach is lost
- More flexibility in object building, reusability and behavior sharing.
- Real bottom-up development - abstracions are obviously constructed after some experience with the concrete individuals.
- One problem: Not in mainstream
  - hardly integrable with file-based approach,
  - not supported by UML sufficiently yet,
  - nothing is static, everything can change - more metamodeling is needed more sofisticated tools are needed

Exploratory programming

- Exploring a state of a running system, editing live objects at run time

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects
- Image-Based systems - live objects in persistent object memory

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects
- Image-Based systems - live objects in persistent object memory
- Source code is not important - text can be generated from live objects

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects
- Image-Based systems - live objects in persistent object memory
- Source code is not important - text can be generated from live objects
- Incremental development (classes and methods in Smalltalk, objects and slots in Self)

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects
- Image-Based systems - live objects in persistent object memory
- Source code is not important - text can be generated from live objects
- Incremental development (classes and methods in Smalltalk, objects and slots in Self)
- Tools for programming - browsers, inspectors, workspaces, outliners

Exploratory programming

- Exploring a state of a running system, editing live objects at run time
- Reflectivity - objects can explore and edit objects
- Image-Based systems - live objects in persistent object memory
- Source code is not important - text can be generated from live objects
- Incremental development (classes and methods in Smalltalk, objects and slots in Self)
- Tools for programming - browsers, inspectors, workspaces, outliners
- The gap between ’what is programmed’ and ’what is running’ is eliminated.

DES Modeling and Simuation

Popular approaches

- Quasi-parallel processes - Simula, ...
- State-centered formalisms - DEVS, Petri nets
  - well suitable for reflectivity studies
  (simplicity, explicit state, clonable)

DES Modeling and Simuation

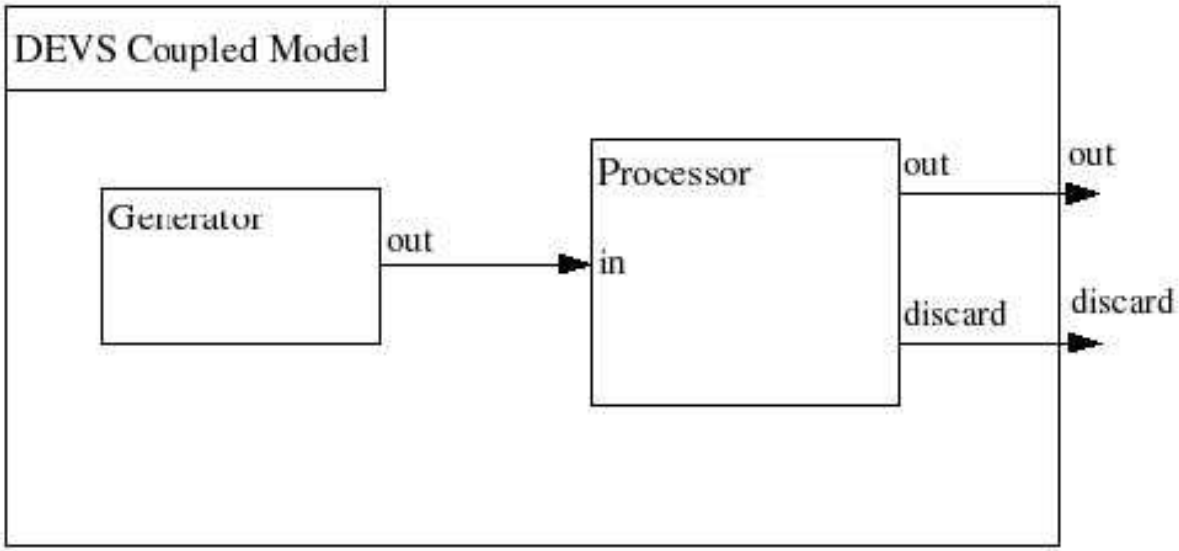
Popular approaches

- Quasi-parallel processes - Simula, ...
- State-centered formalisms - DEVS, Petri nets
  - well suitable for reflectivity studies
  (simplicity, explicit state, clonable)

$$DEVS = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$$

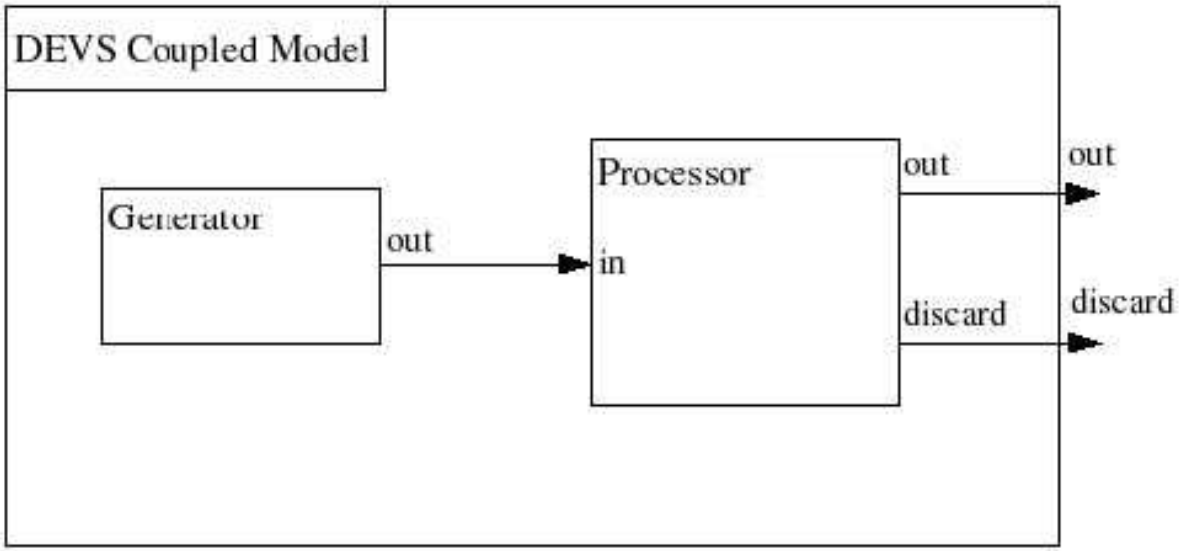
X is a set if input values
S is a set of states
Y is a set of output values
$\delta_{int} : S \longrightarrow S$ is the internal transition function
$\delta_{ext} : Q \times X \longrightarrow S$ is the external transition function, where
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of all states
e is the time passed since the last transition
$\lambda : S \longrightarrow Y$ is the output function
$ta : S \longrightarrow R^{+}{}_{0,\infty}$ is the time advance function

DEVS



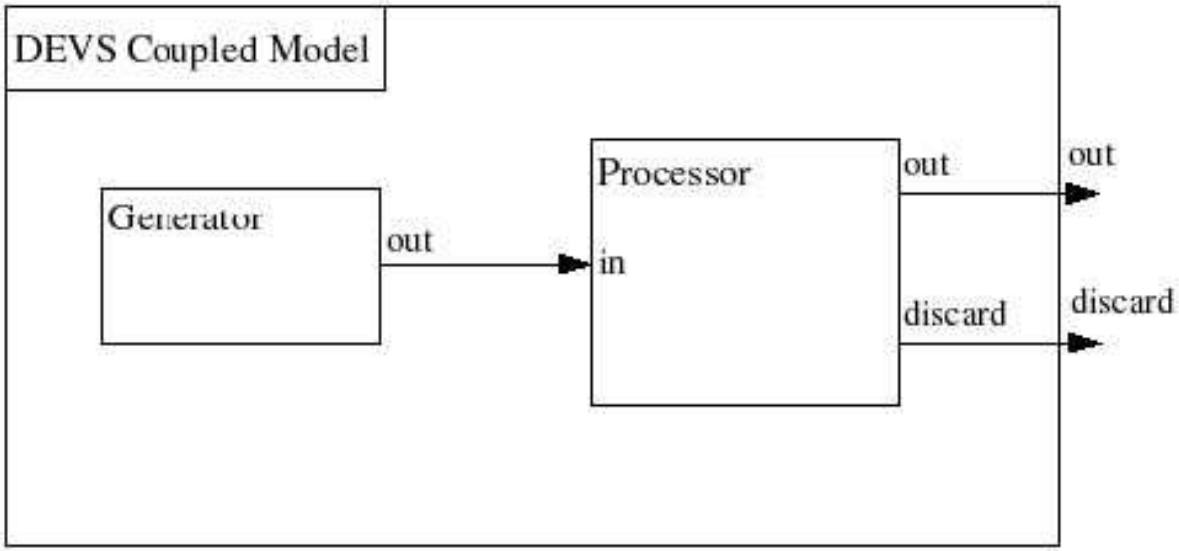
- hierarchical component architecture, static or dynamic structure

DEVS



- hierarchical component architecture, static or dynamic structure
- state machines represent a lower level approach than processes, but it is very well suited for exploring and modification (in theory, as well as in real implementations)

DEVS



- hierarchical component architecture, static or dynamic structure
- state machines represent a lower level approach than processes, but it is very well suited for exploring and modification (in theory, as well as in real implementations)
- higher-level paradigms can be mapped or wrapped (processes, PNs, statecharts)

DEVS Implementation

- Mainstream approach:
  - Atomic components are defined as classes
  - Coupled components as well
  - Structure of coupled components can obviously change (ports, coupling, instantiation)
  - No new atomic components can be introduced at run time

DEVS Implementation

- Mainstream approach:
  - Atomic components are defined as classes
  - Coupled components as well
  - Structure of coupled components can obviously change (ports, coupling, instantiation)
  - No new atomic components can be introduced at run time
- SmallDEVS approach:
  - Both Class-based and prototype-based OO modeling supported
  - All components can arbitrarily change at runtime, new components can arise

Prototypes in Smalltalk

- Object creation:

  aPrototypeObject := PrototypeObject new.
  anotherPrototypeObject := aPrototypeObject clone.

Prototypes in Smalltalk

- Object creation:

  aPrototypeObject := PrototypeObject new.
  anotherPrototypeObject := aPrototypeObject clone.

- Slots and methods exploring:

  aPrototypeObject slotNames
  aPrototypeObject methodNames
  aPrototypeObject perform: aSlotName
  aPrototypeObject methodSourceAt: methodName

Prototypes in Smalltalk

- Object creation:

  aPrototypeObject := PrototypeObject new.
  anotherPrototypeObject := aPrototypeObject clone.

- Slots and methods exploring:

  aPrototypeObject slotNames
  aPrototypeObject methodNames
  aPrototypeObject perform: aSlotName
  aPrototypeObject methodSourceAt: methodName

- Slots and methods editing:

```
aPrototypeObject addSlots:{
                'name1' -> anObject.
                'name2' -> anotherObject}.
aPrototypeObject addMethod:
                'messageSelector codeOfTheMethod'.
aPrototypeObject removeSlots: { 'name1'.'name2'}.
aPrototypeObject removeMethod: 'messageSelector'.
```

Behavior Sharing

traits + delegation (dynamic inheritance):

```
aPrototypeObject addDelegates:{
                'name1' -> aTrait.
                'name2' -> anotherTrait}.
aPrototypeObject removeDelegates:{ 'name1'. 'name2' }
aPrototypeObject delegateNames.
```

well-known objects (traits and prototypes) are stored in some globally available structure

Atomic DEVS Incremental Construction

```
model := AtomicDEVSPrototype new.
model addSlots: {...}.
model addInputPorts: {...}.
model addOutputPorts: {...}.
model addDelegates: {...}.
model intTransition: '...'.
model extTransition: '...'.
model outputFnc: '...'.
model timeAdvance: '...'.
```

Exploring and editing slots, ports, methods, delegates:

slotNames, removeSlots, ...

Coupled DEVS Incremental Construction

```
model := CoupledDEVSPrototype new.
model addInputPorts: { name1. name2.. ... }.
model addOutputPorts: { name1. name2. ... }.
model addComponents: {
        name1 -> aComponent1.
        name2 -> aComponent2. ... }.
model addCouplings: {
        #(component1 port1) -> #(component2 port2).
        #(component3 port3) -> #(component4 port4). .... }.
```

Exploring and editing ports, components, couplings:

inputPortNames, removeInputPorts, couplings, removecouplings ....

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
aModel2 := aModel copy.
aSimulation2 := aSimualtion copy.
```

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
aModel2 := aModel copy.
aSimulation2 := aSimualtion copy.
```

Editig ports, couplings, slots, methods, delegates is possible even during simulation

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
aModel2 := aModel copy.
aSimulation2 := aSimualtion copy.
```

Editig ports, couplings, slots, methods, delegates is possible even during simulation

A snapshot of a simulation can be used as a model

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
aModel2 := aModel copy.
aSimulation2 := aSimualtion copy.
```

Editig ports, couplings, slots, methods, delegates is possible even during simulation

A snapshot of a simulation can be used as a model

Persistent repository for models and simulations:

```
MyRepository at: '/Simulations/MySimulation' put: s.
(MyRepository at: '/Simulations/MySimulation') inspect.
```

Operating system

Support for manipulation with models and simulations

```
s := model getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
aModel2 := aModel copy.
aSimulation2 := aSimualtion copy.
```

Editig ports, couplings, slots, methods, delegates is possible even during simulation

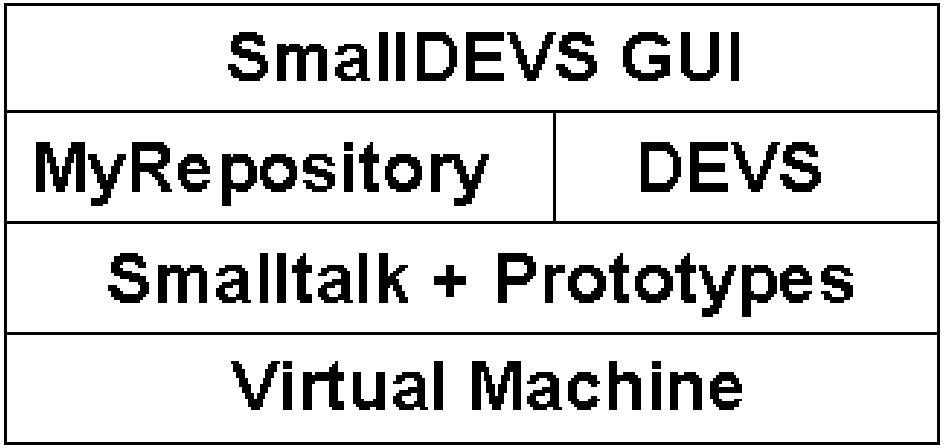
A snapshot of a simulation can be used as a model

Persistent repository for models and simulations:

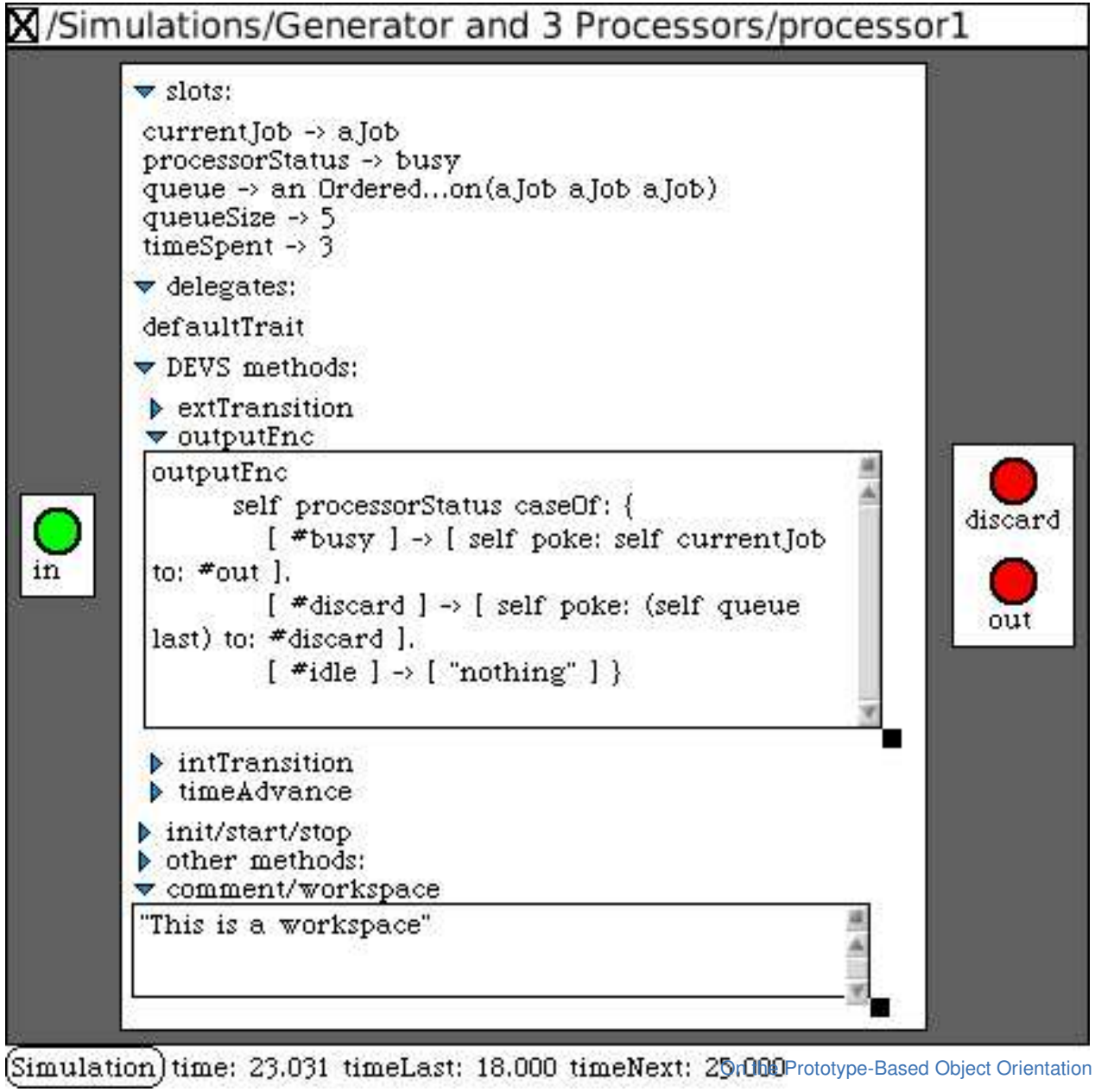
```
MyRepository at: '/Simulations/MySimulation' put: s.
(MyRepository at: '/Simulations/MySimulation') inspect.
```

Serialization of models and simulations
(for storing to disk or for migration)

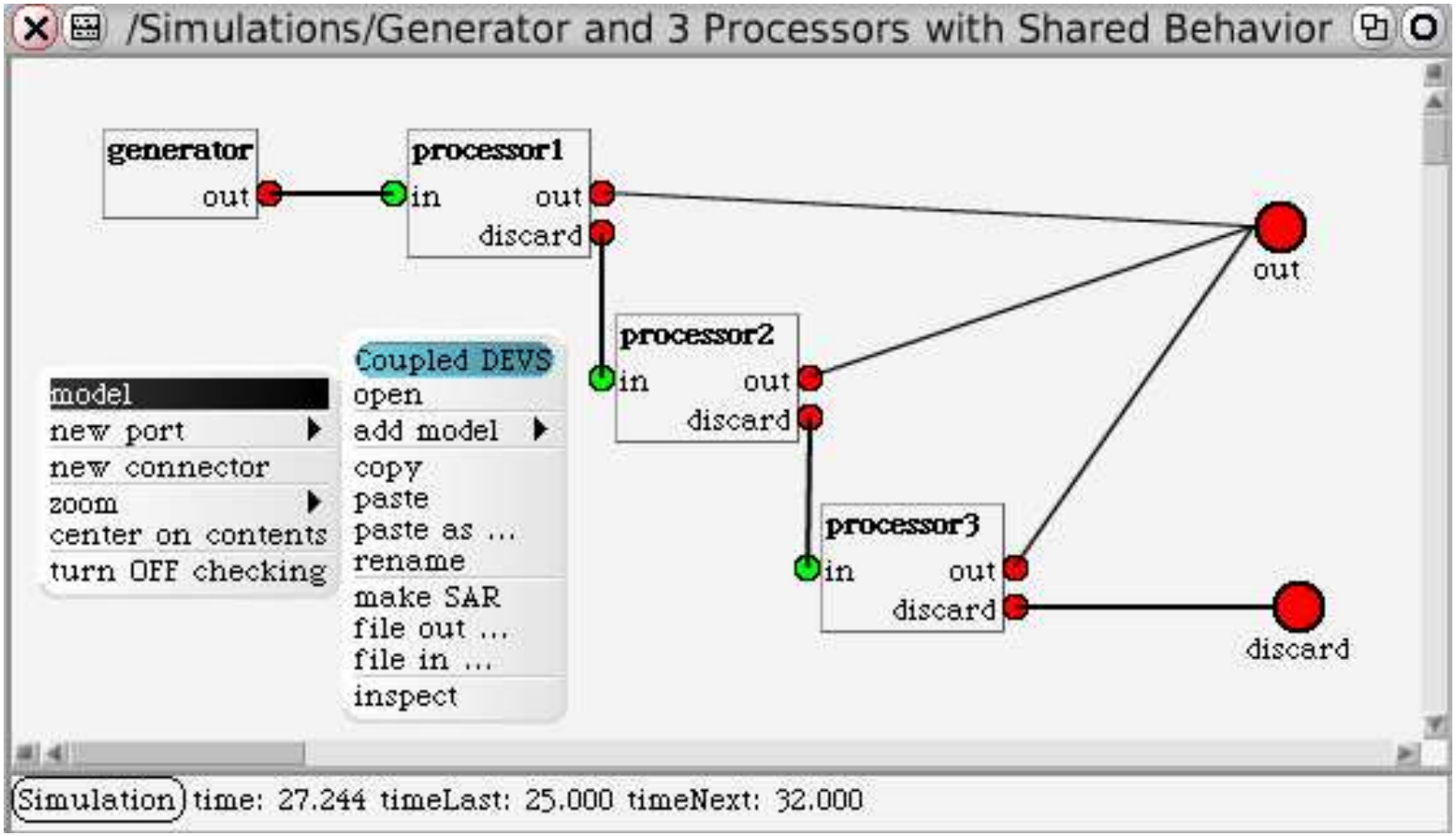
SmallDEVS System



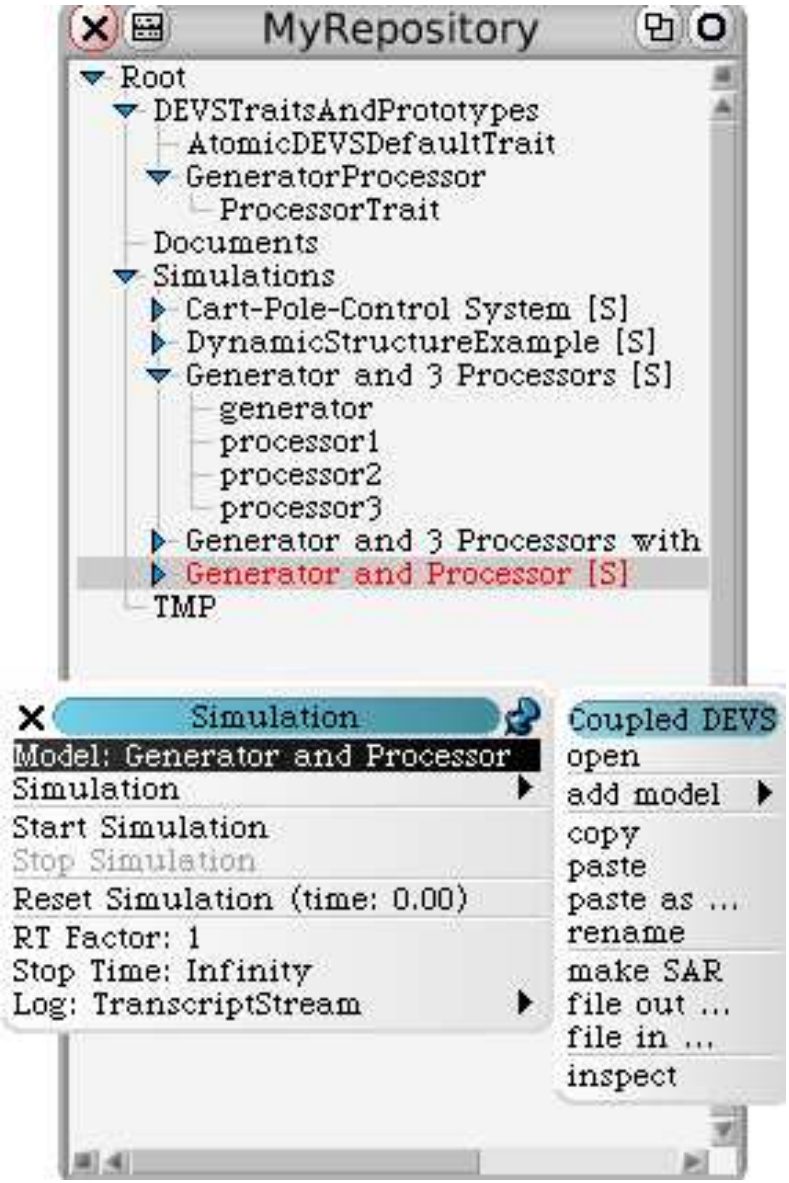
Visual tools for exploratory modeling



Visual tools for exploratory modeling



Visual tools for exploratory modeling



Conclusion

Why DEVS?

- computer supported systems theory
- inteligent systems modeling, simulation, design (NASA)
- DEVS standardization in progress, HLA compatibility
- other formalism can be mapped and/or wrapped

Conclusion

Why DEVS?

- computer supported systems theory
- inteligent systems modeling, simulation, design (NASA)
- DEVS standardization in progress, HLA compatibility
- other formalism can be mapped and/or wrapped

Why prototype-based exploratory modeling with SmallDEVS?

- real bottom-up approach (from concrete examples to abstractions)
- "understanding by modeling" is more concrete - live objects, no "dead source code"
- reflectivity and concretness of prototype-based approach makes no difference between a model and any snapshot of a running simulation
- almost unlimited automatic evolution of models during simulation is possible and the resulting model is fully available for exploration by standard tools for modeling

Relations with PNtalk

Object orientation vs. DEVS

- OO deals with dynamically appearing and disappearing instances of classes.
- Message sendings.
- No explicit of object interconnections, no visible structure.
- Only classes are static. Only classes are maintainable.

- DEVS deals with static hierarchical structures of objects. Dynamic structure???
- No direct message sending. Explicit, visible connections.

Relations with PNtalk

Merging dynamic objects (e.g. PNtalk) and DEVS

- DEVS focusses to objects (similarly to Prototype Objects). Classes are not needed.
- DEVS can be animated using reflection (dynamic structure, dynamic atoms). Structure remains visible.
- DEVS offers component structure to object systems
  - atomic component can encapsulate communicating objects
  - atoms can be simple state machines as well as complex and dynamic object structures
  - simple event-based component interface allows for effective composability
  - hierarchical component structures can be dynamic and visible at the same time
  - component level is easily maintainable and has sound theoretical background
  - atomic level maintainability depends on particular formalism (FSA, ASM, Statecharts, Petri nets, LISP, Prolog, ...)

PNtalk TO DO

To be fully interoperable with SmallDEVS, verification tools and external world

- Asynchronous ports compatible with DEVS ports
- Reflective access to state for clonning and serialization
- ...

Partially done.