Linear constraints layout in graph grammars, layout algorithms, and scoped UI layout

Presented by Denis Dubé

August 27, 2005

McGill

MSDL

Overview

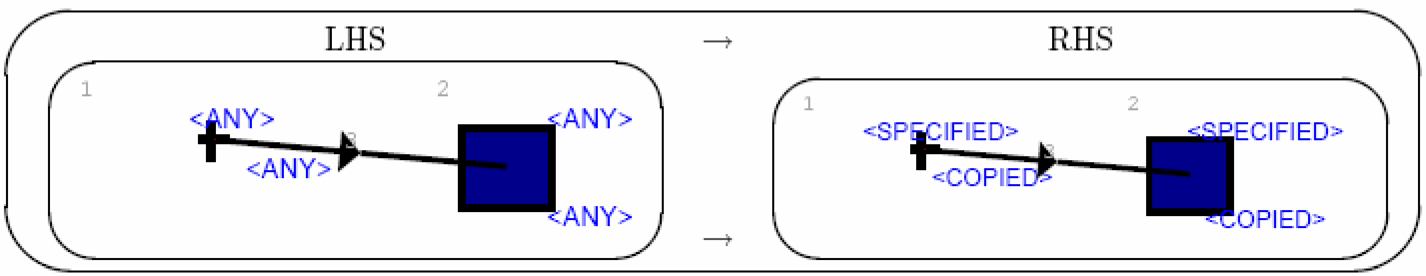
1. Automatic graph layout
   - Graph grammars and QOCA linear constraints
   - General layout algorithms
     - Hierarchical, Force transfer, Spring-electrical, Circle, Tree-like, Import/Export

2. Graphical user interfaces and statecharts

Graph grammars: previous work

- Automatic latex code documentation generator
- Eliminates lack of or inaccurate documentation

Rule 2 (Order 2): road2sink

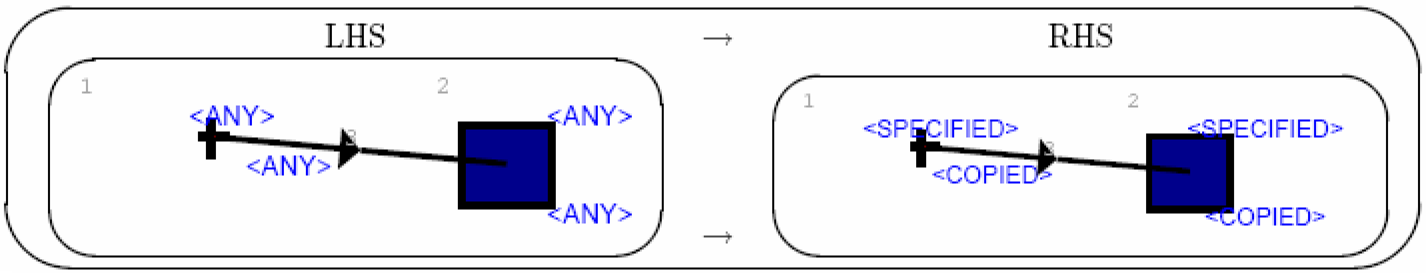


Precondition:

```
# Proceed only if we have cars to move to the sink
currentNumCars = self.getMatched(graphID, self.LHS.nodeWithLabel(1)).num_vehicles.getValue()
if( currentNumCars > 0 ):
    return True
else:
    return False
```

Automatic documentation

Rule 2 (Order 2): road2sink



Post action:

```
# Road segment capacity increases as car leaves it
roadSegNode = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
for in_link in roadSegNode.in_connections_:
    if( isinstance( in_link, CapacityOf) and isinstance(in_link.in_connections_[0], Capacity ) ):
        capNode = in_link.in_connections_[0]
        capNode.setGenValue( 'capacity', capNode.capacity.getValue() + 1 )
```

Specify: *RoadSection #1*

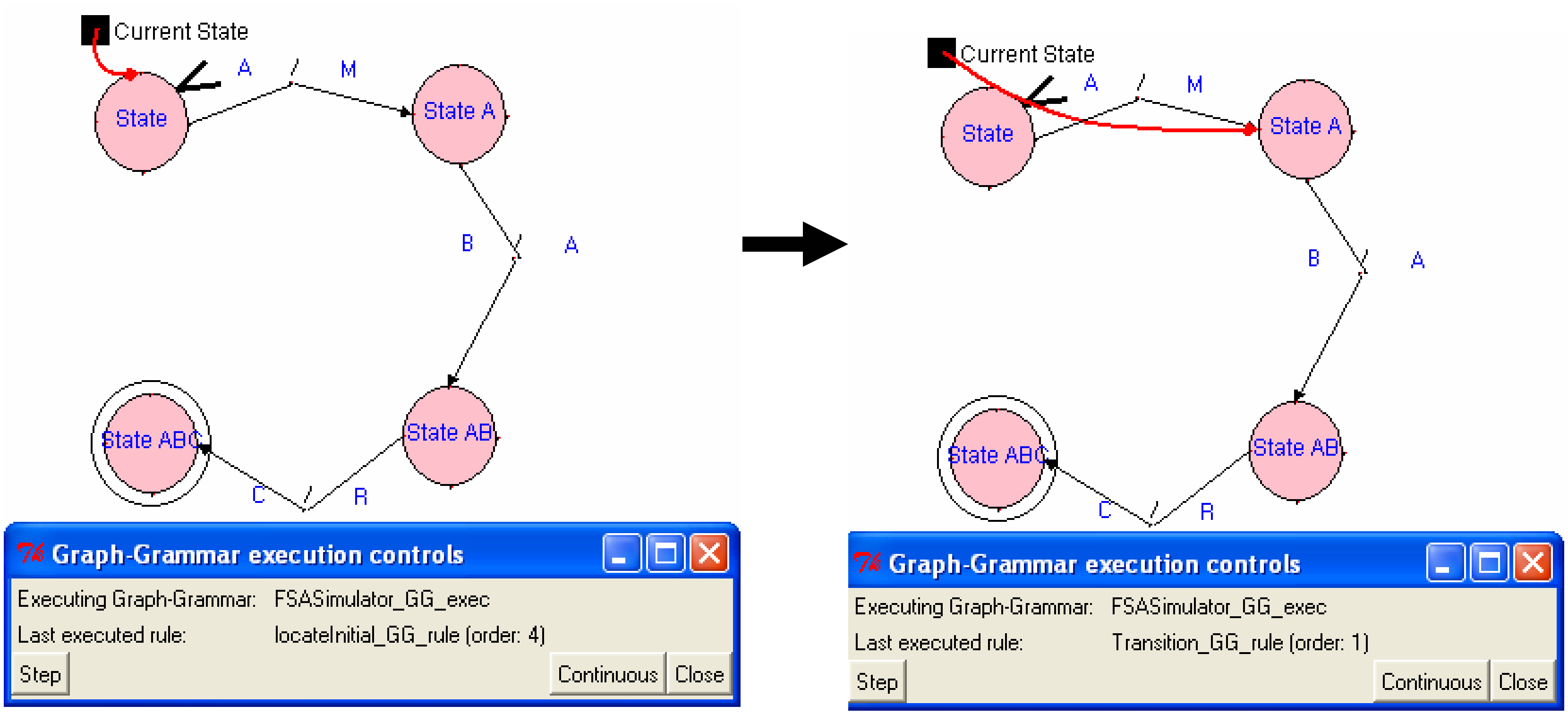
```
# Road segement loses one car
currentNumCars = self.getMatched(graphID, self.LHS.nodeWithLabel(1)).num_vehicles.getValue()
return currentNumCars - 1
```

Specify: *Sink #2*

```
# Sink gets one more car
currentNumCars = self.getMatched(graphID, self.LHS.nodeWithLabel(2)).num_vehicles.getValue()
return currentNumCars + 1
```

Why linear constraints?

- FSA graph grammar based simulator
  - A simple linear constraint could have moved the “Current State” box over the active state



What is QOCA? Why use it?

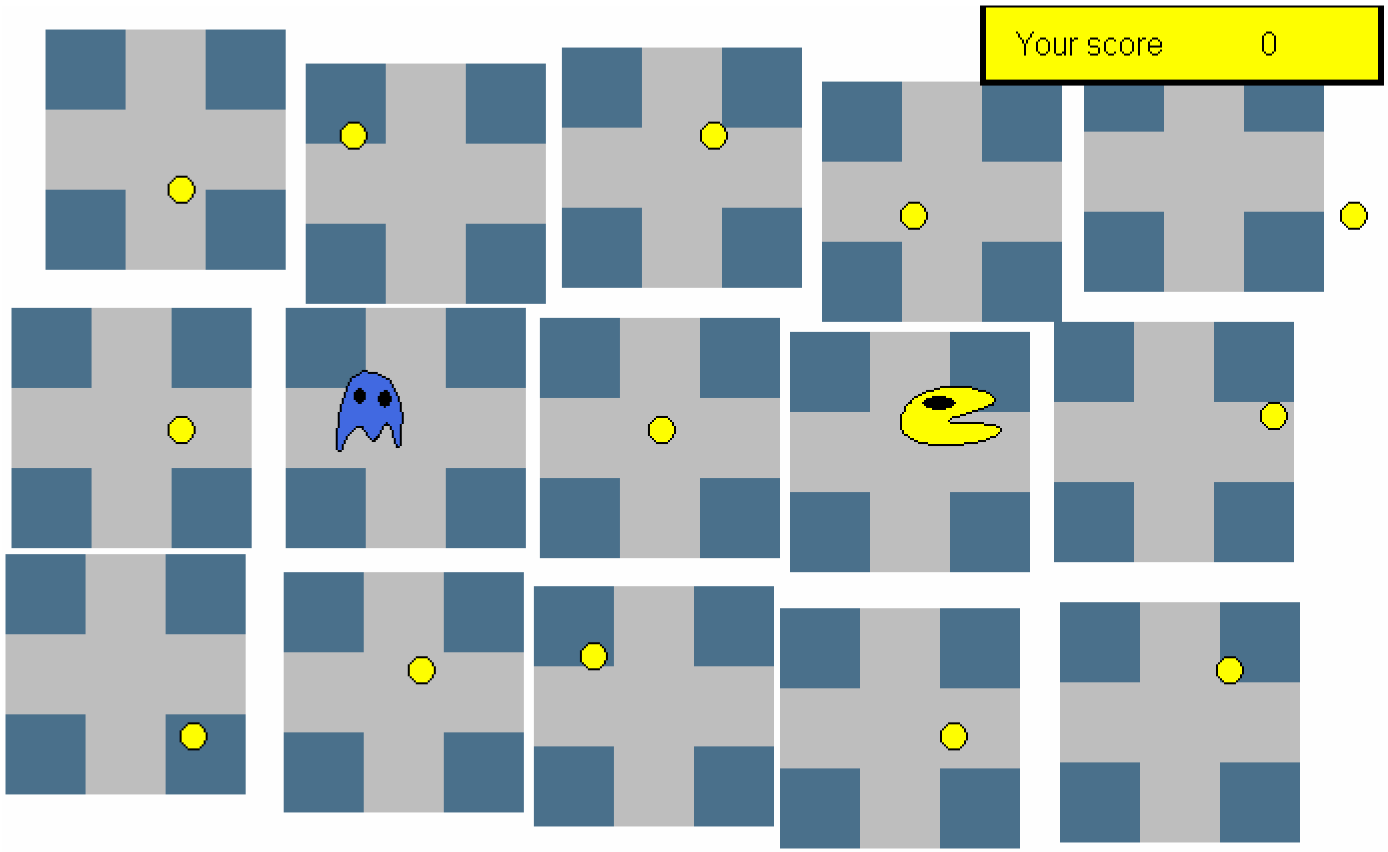
- QOCA is an object-oriented constraint solving toolkit whose source code is available in C++ and Java

- QOCA is worth using because:
  - Makes building a custom solver unnecessary
  - Unlike general purpose constraint solvers, it works incrementally, allowing for rapid re-solving of constraints when small changes occur

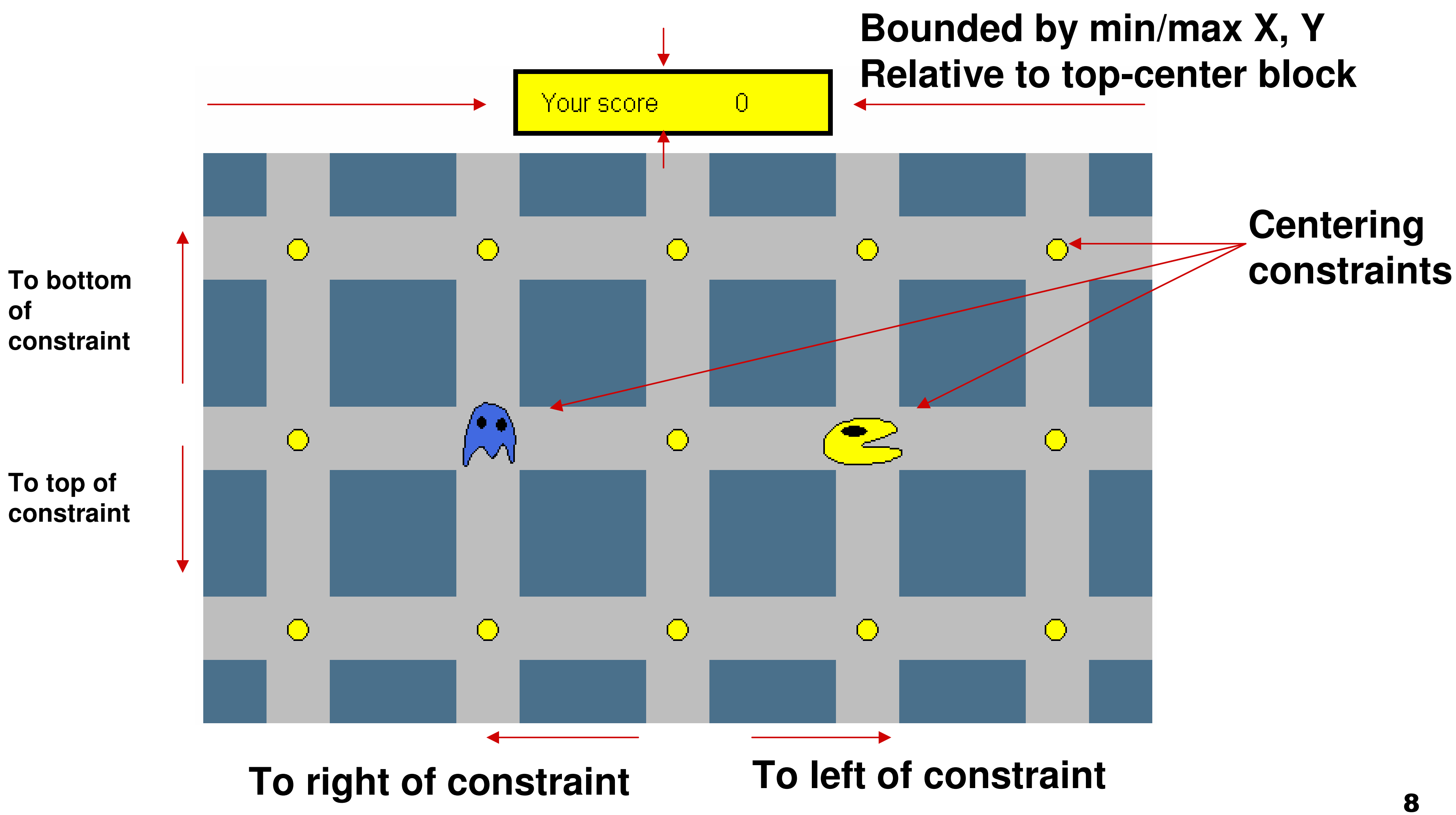
  (Example: user drags a node)

Pacman example: QOCA off

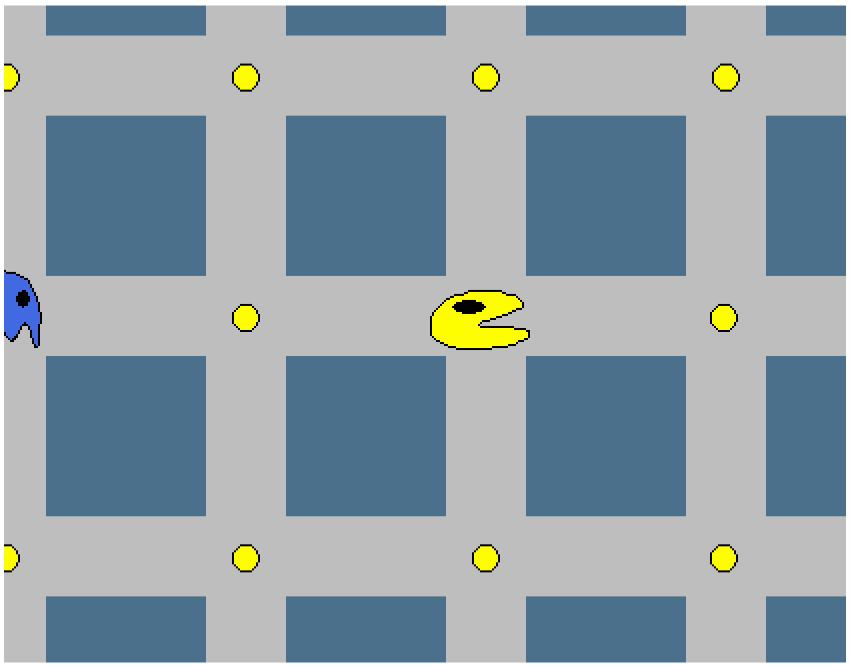
- NOTE: Connections are not visible (but present)

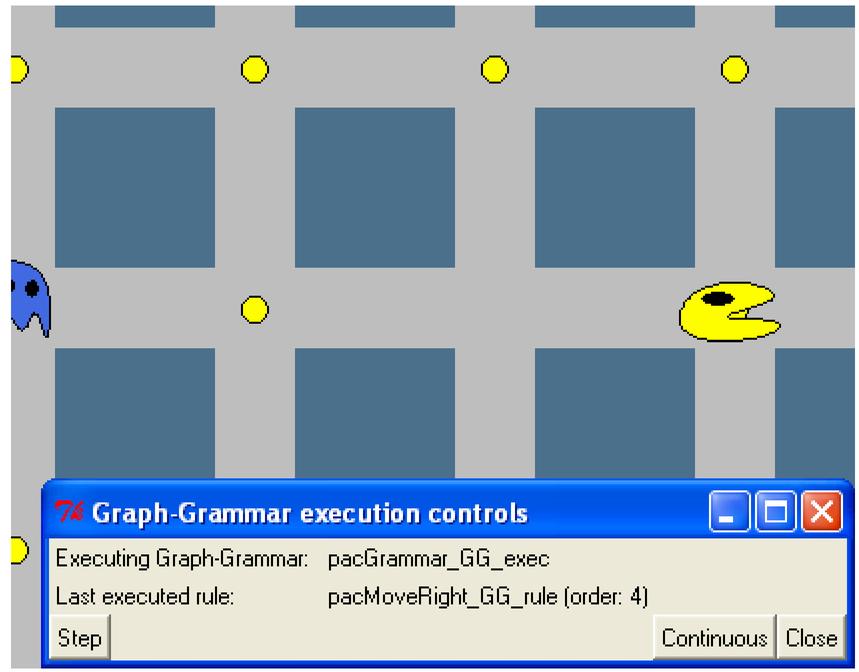


Pacman example: QOCA on

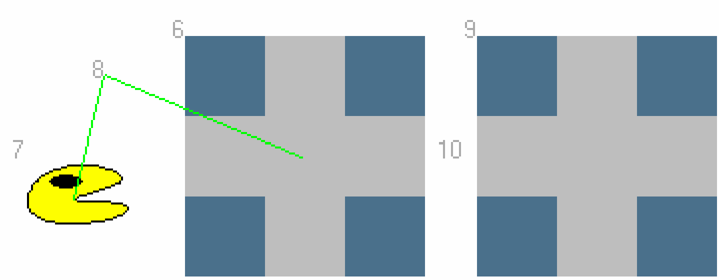


Pacman simulation grammar

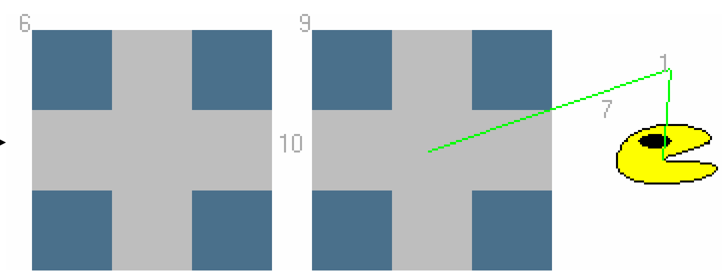




LHS of GG rule



RHS of GG rule



QOCA Pros/Cons

- Pros
  - High level constraints are easy to set by formalism developers, no special layout knowledge required
  - The incremental constraint solver is fast
- Cons
  - Linear constraints are not sufficient to capture many aesthetic constraints such as:
    - Crossing minimization
    - Overlap prevention

General layout algorithms

- Clearly QOCA cannot solve all our layout woes

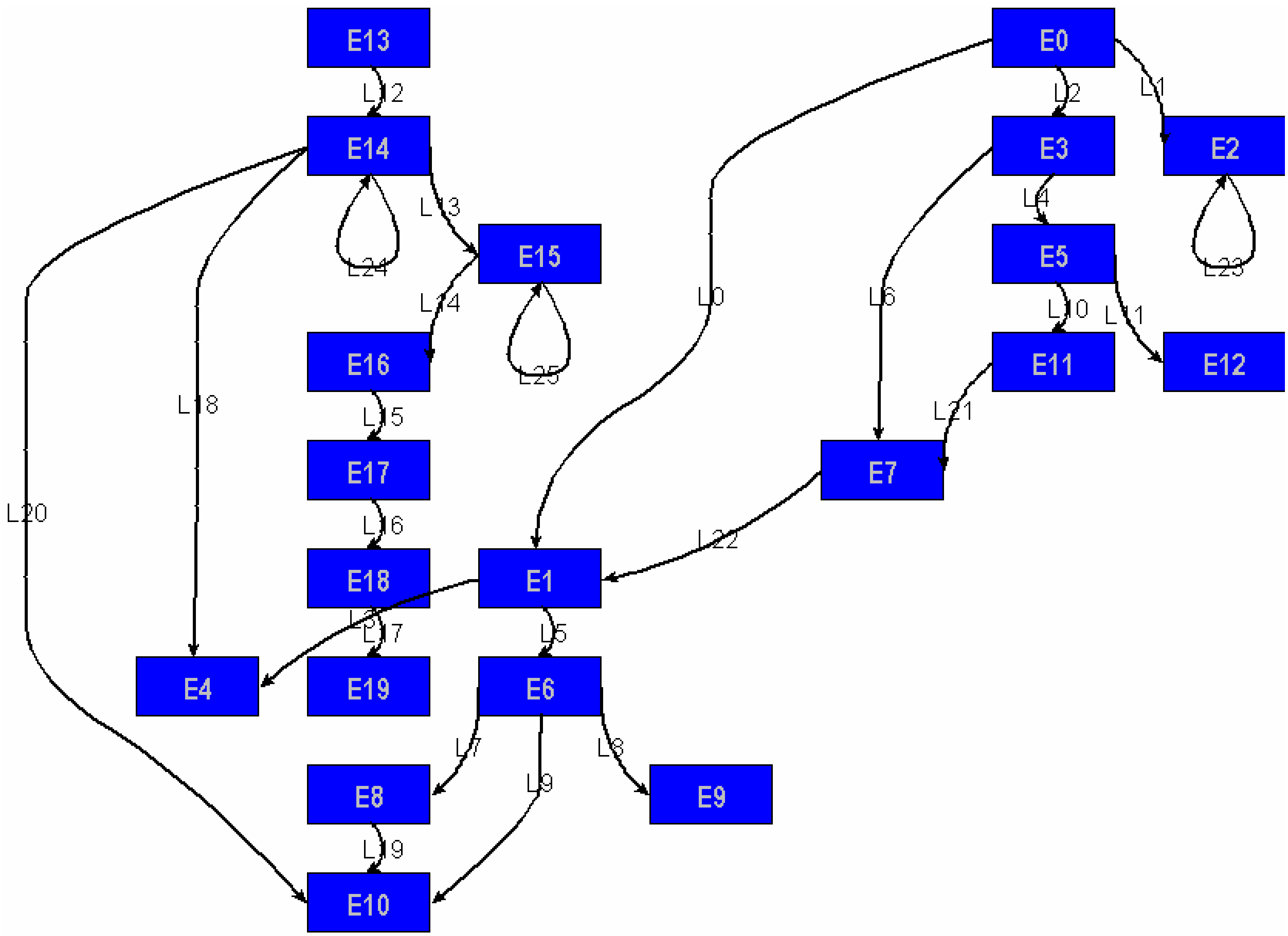
- Indeed, the NP-Complete nature of satisfying virtually every aesthetic criteria singly, let alone all at once, indicates a need for many different heuristic strategies

Hierarchical Layout

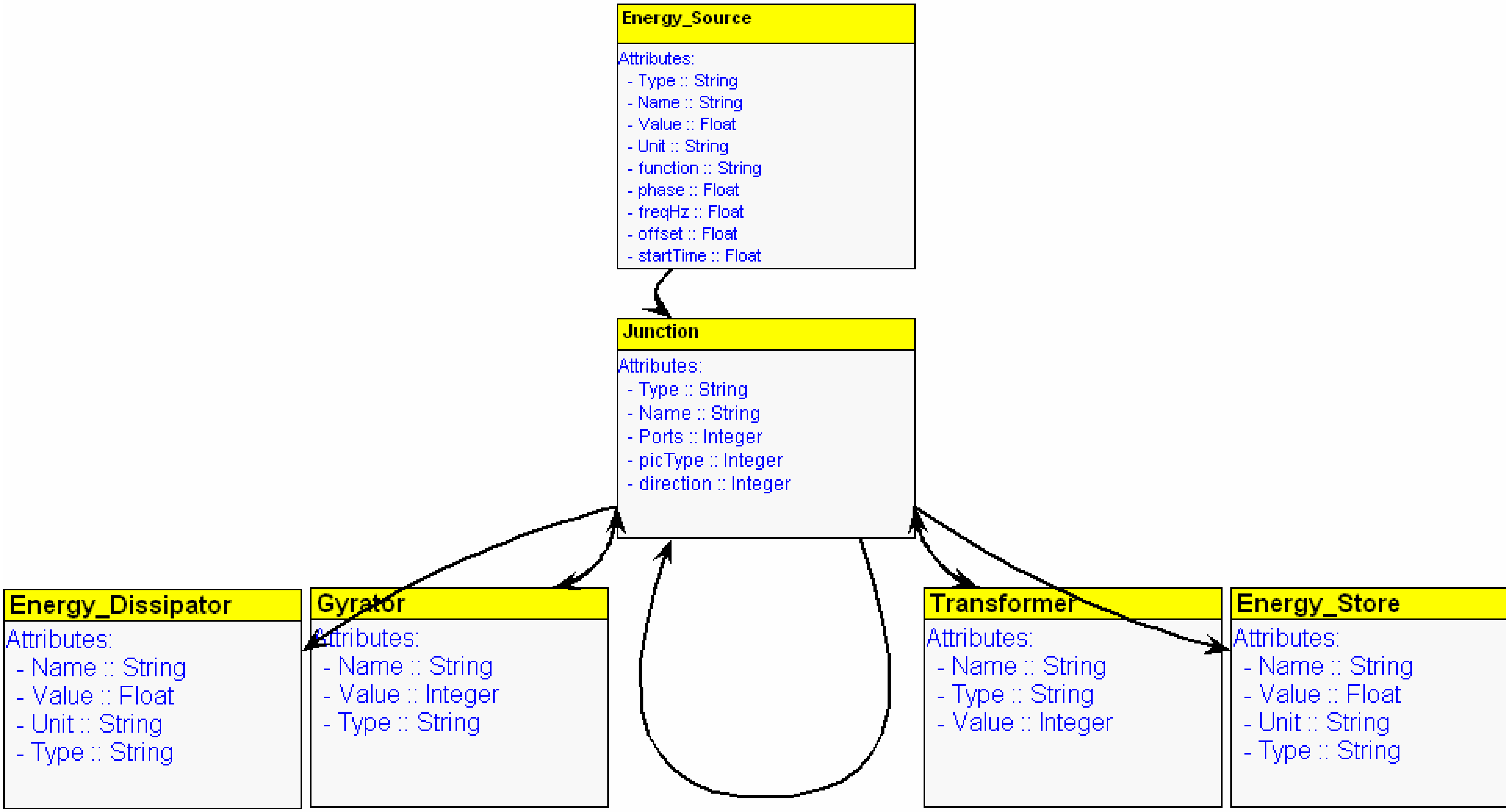
- A Sugiyama-based algorithm implementation, it works well on many graph types

- Hierarchical layout algorithm sketch:
  - Layers nodes from root to leaves and remove cycles
  - Swaps nodes on a given layer to minimize crossing
  - Places nodes on a grid, aligns them

Hierarchical: Random test graph



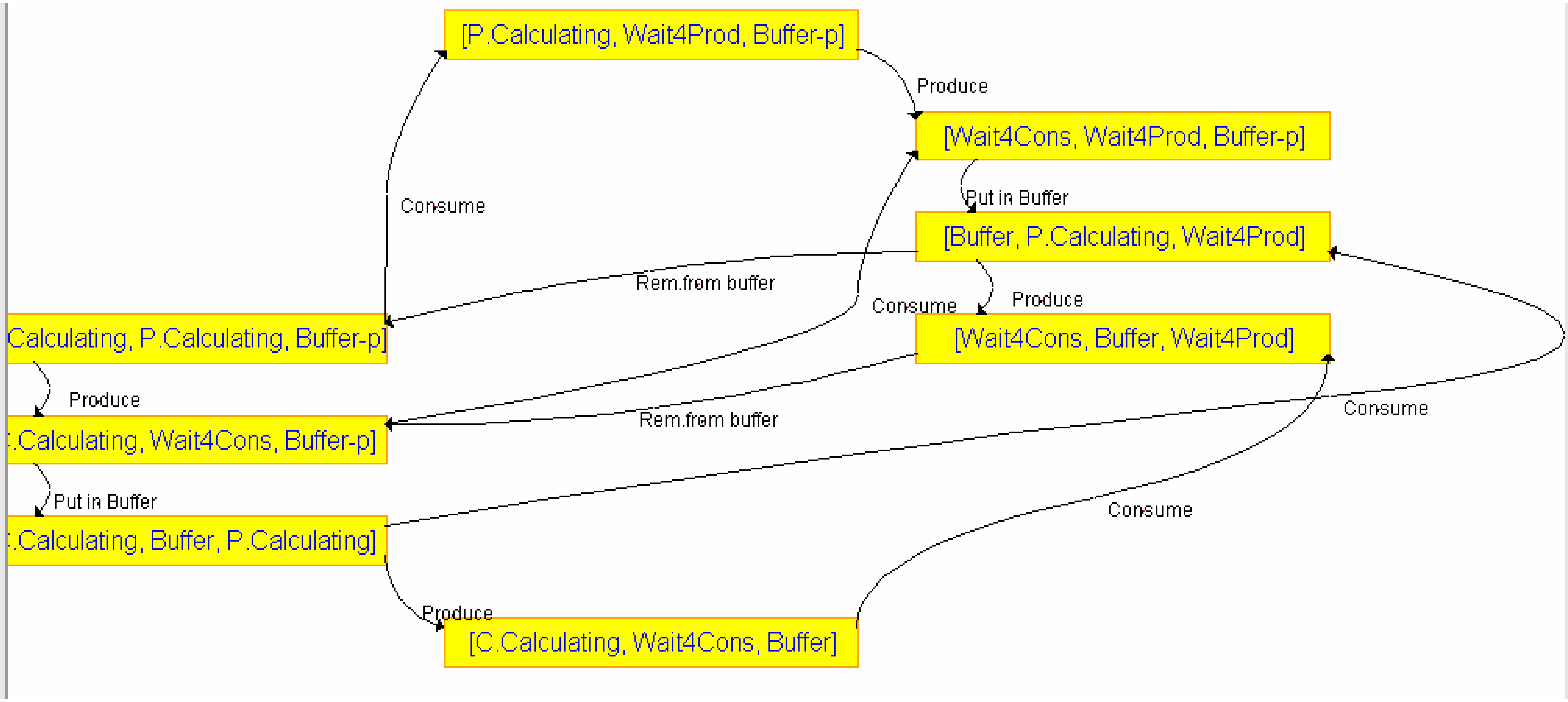
Hierarchical: Class diagrams



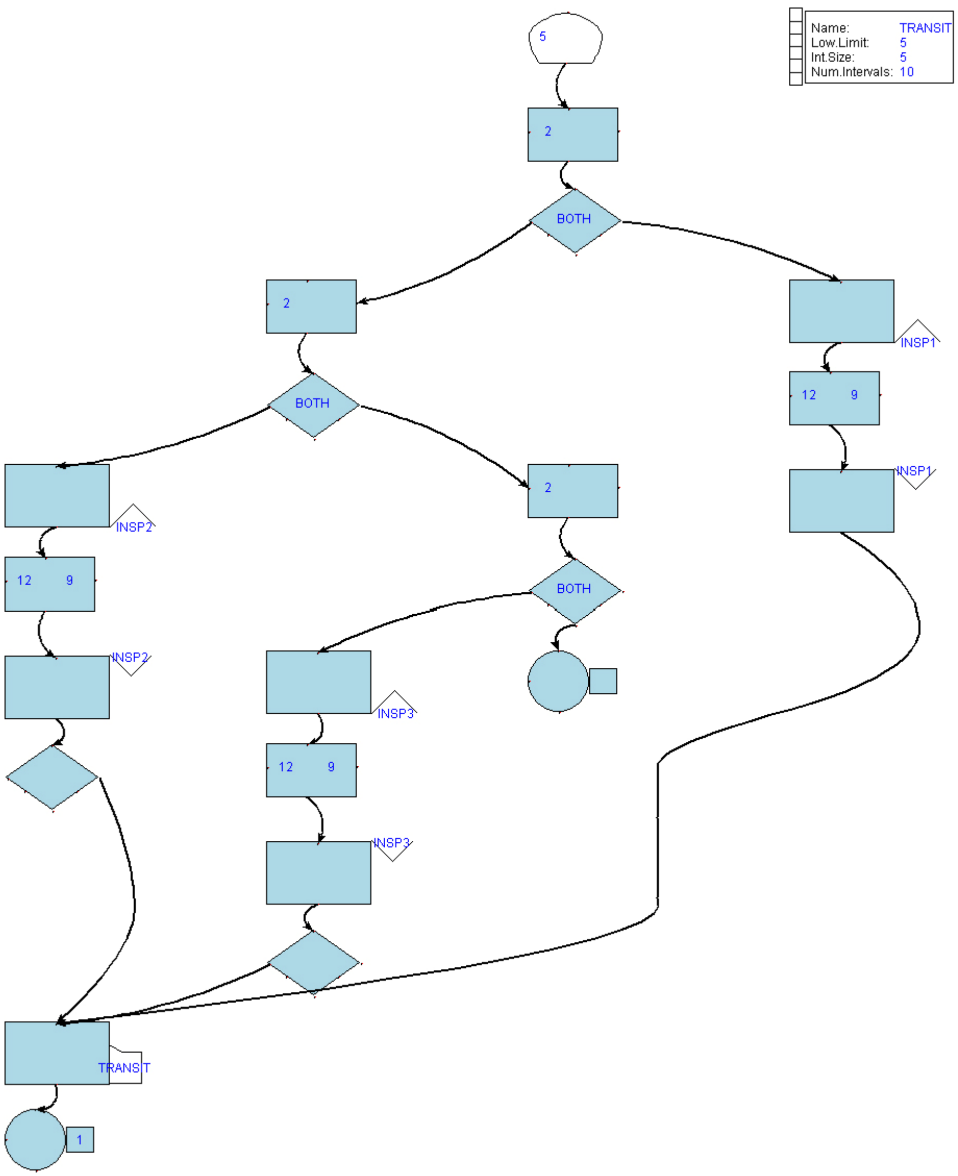
Bond graph meta-model by Sagar Sen

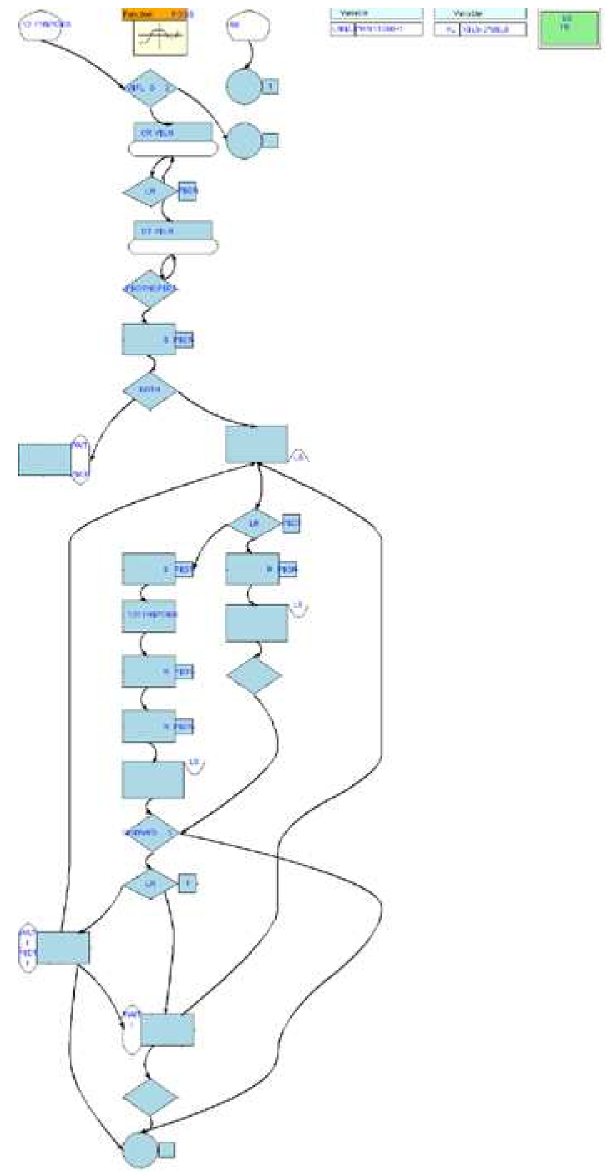
Hierarchical: Reachability graph

- Graph generated from a Petri-Net

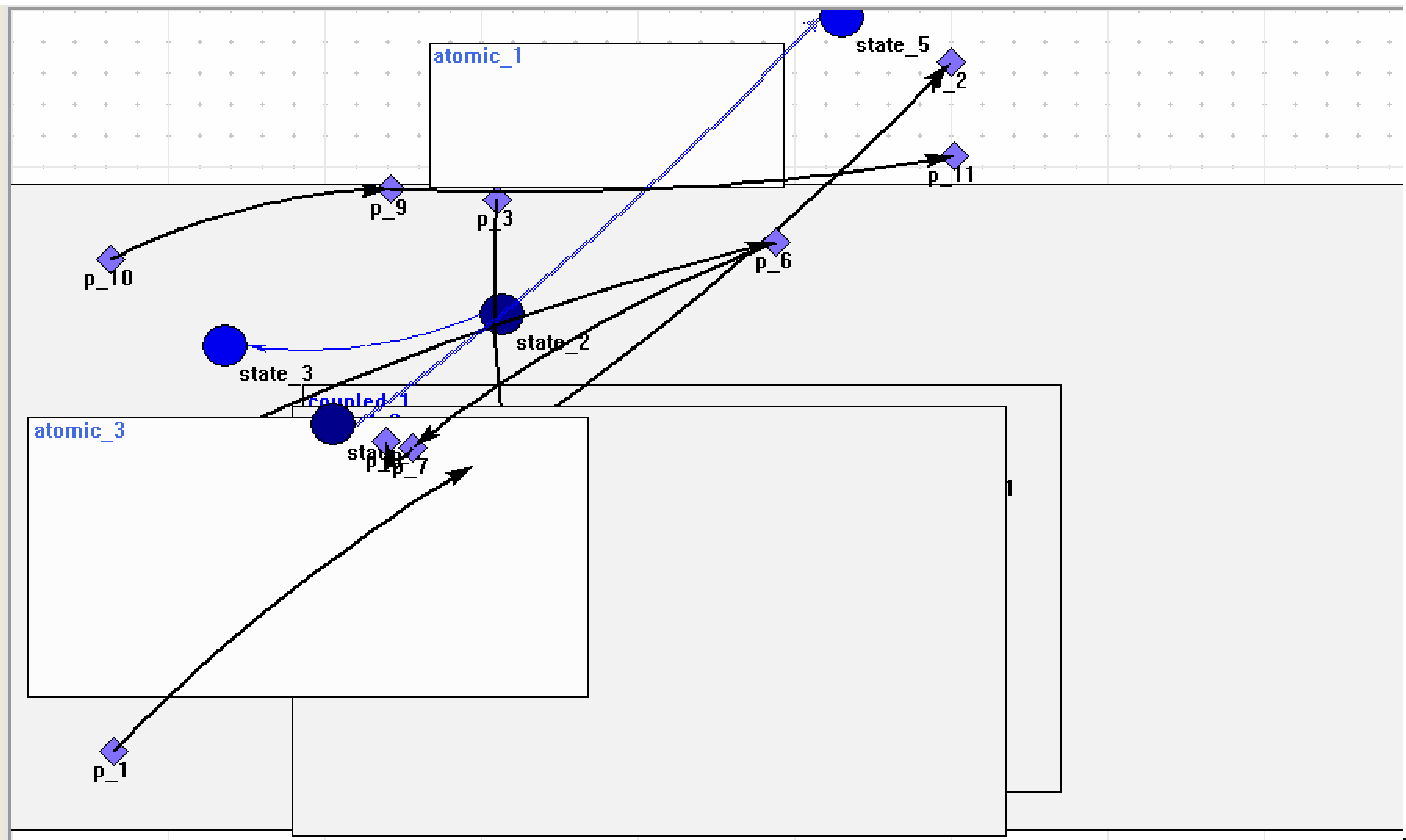


Hierarchical: GPSS models

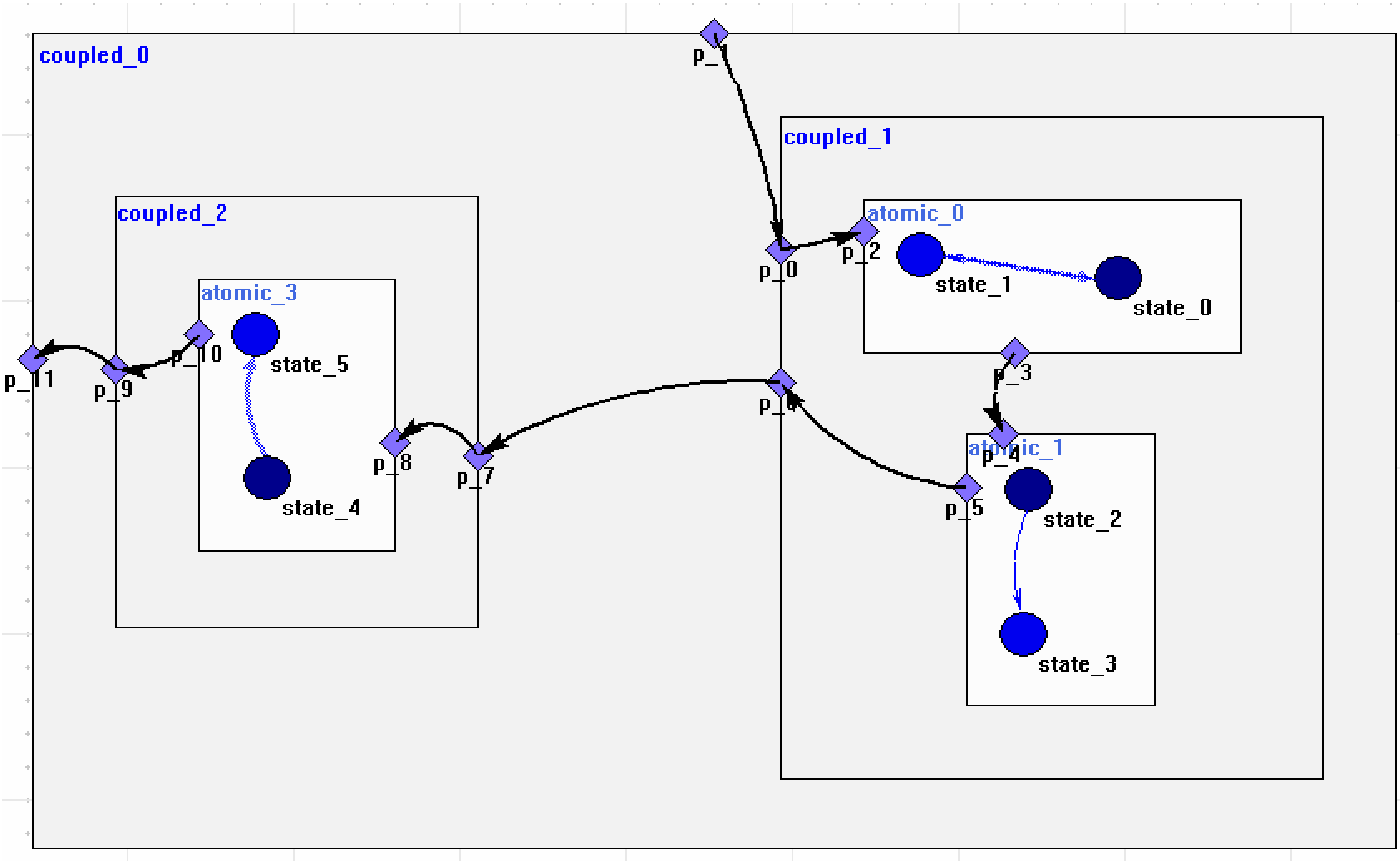




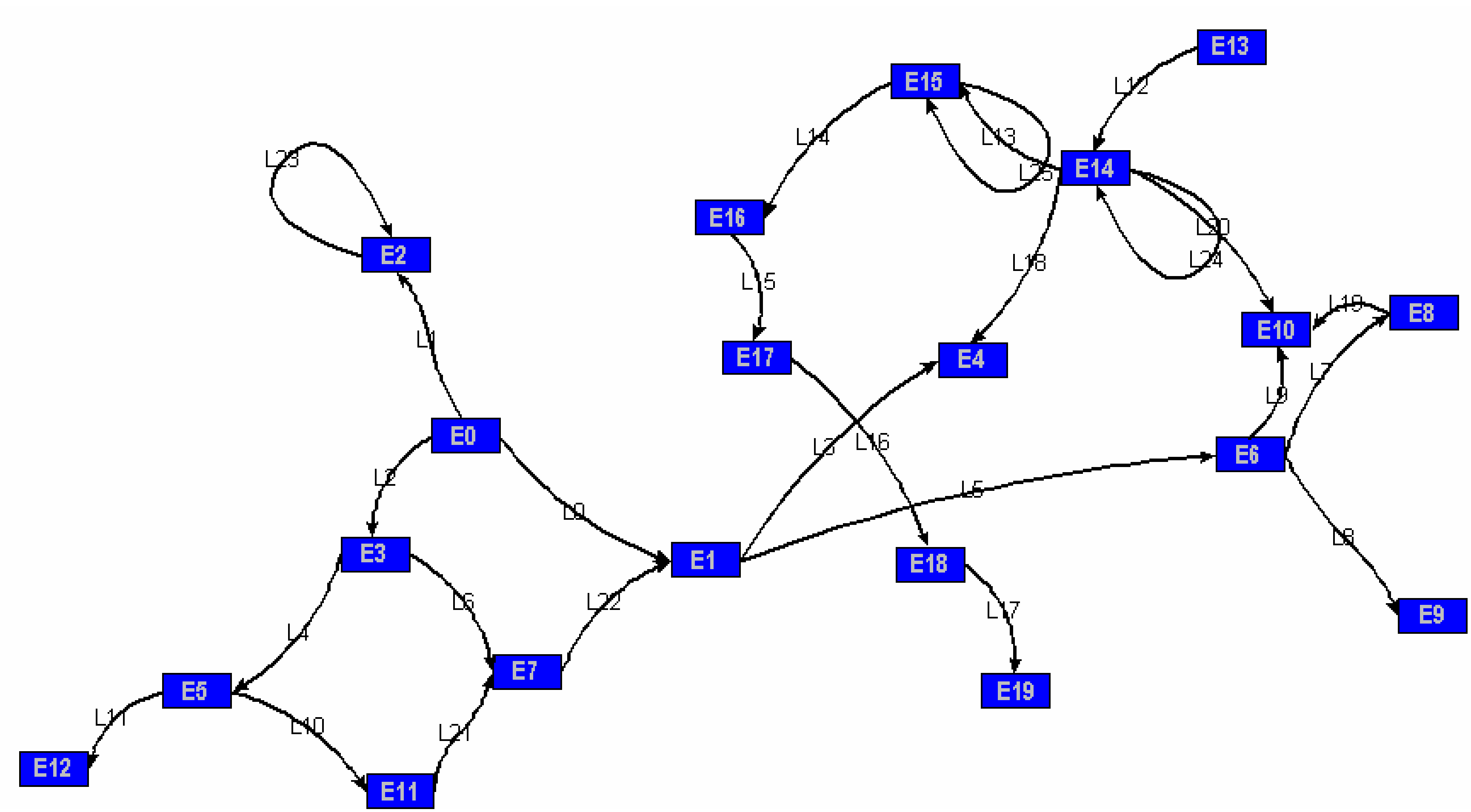
Random layout: Devs model



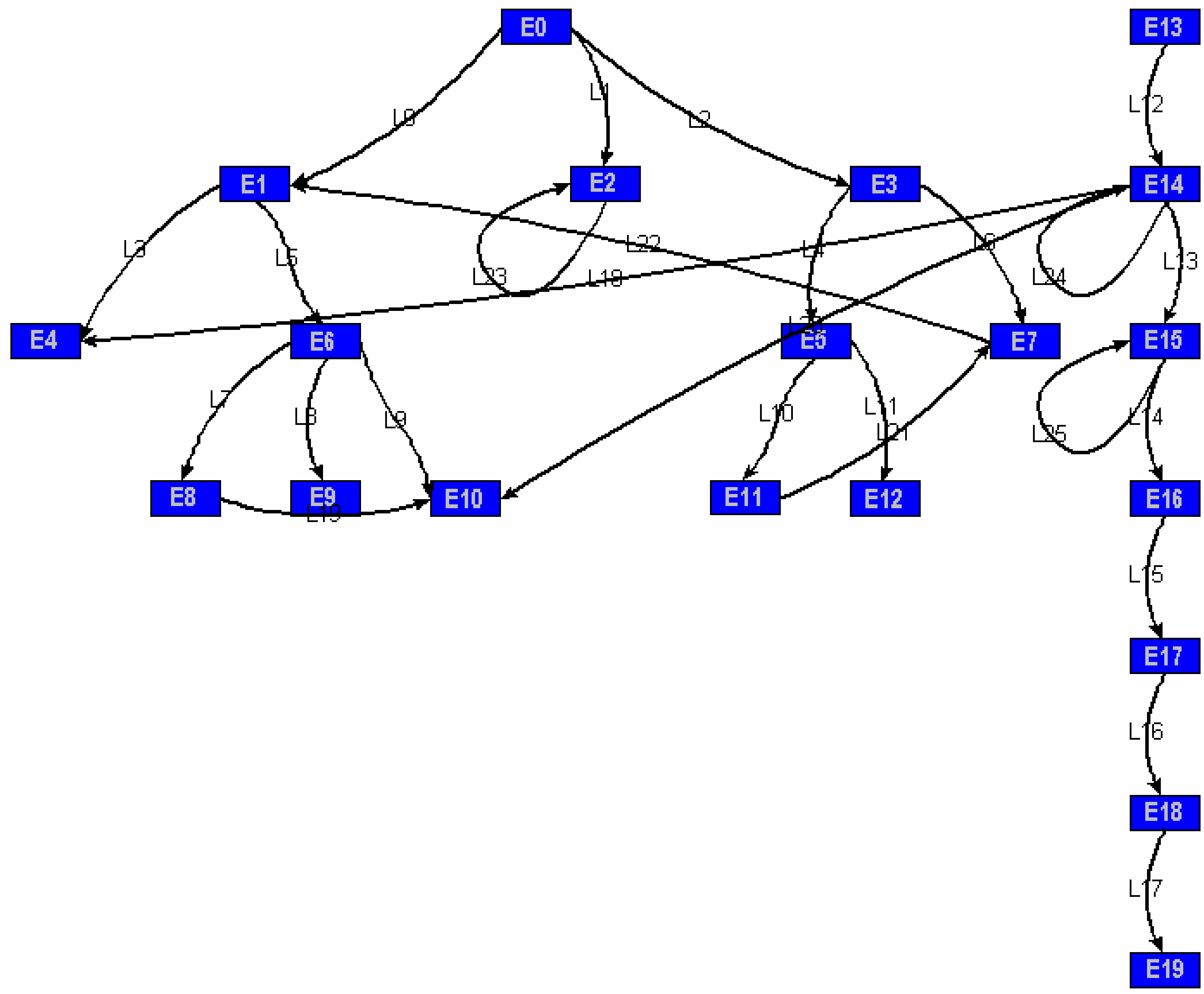
Force Transfer Algorithm: Devs model



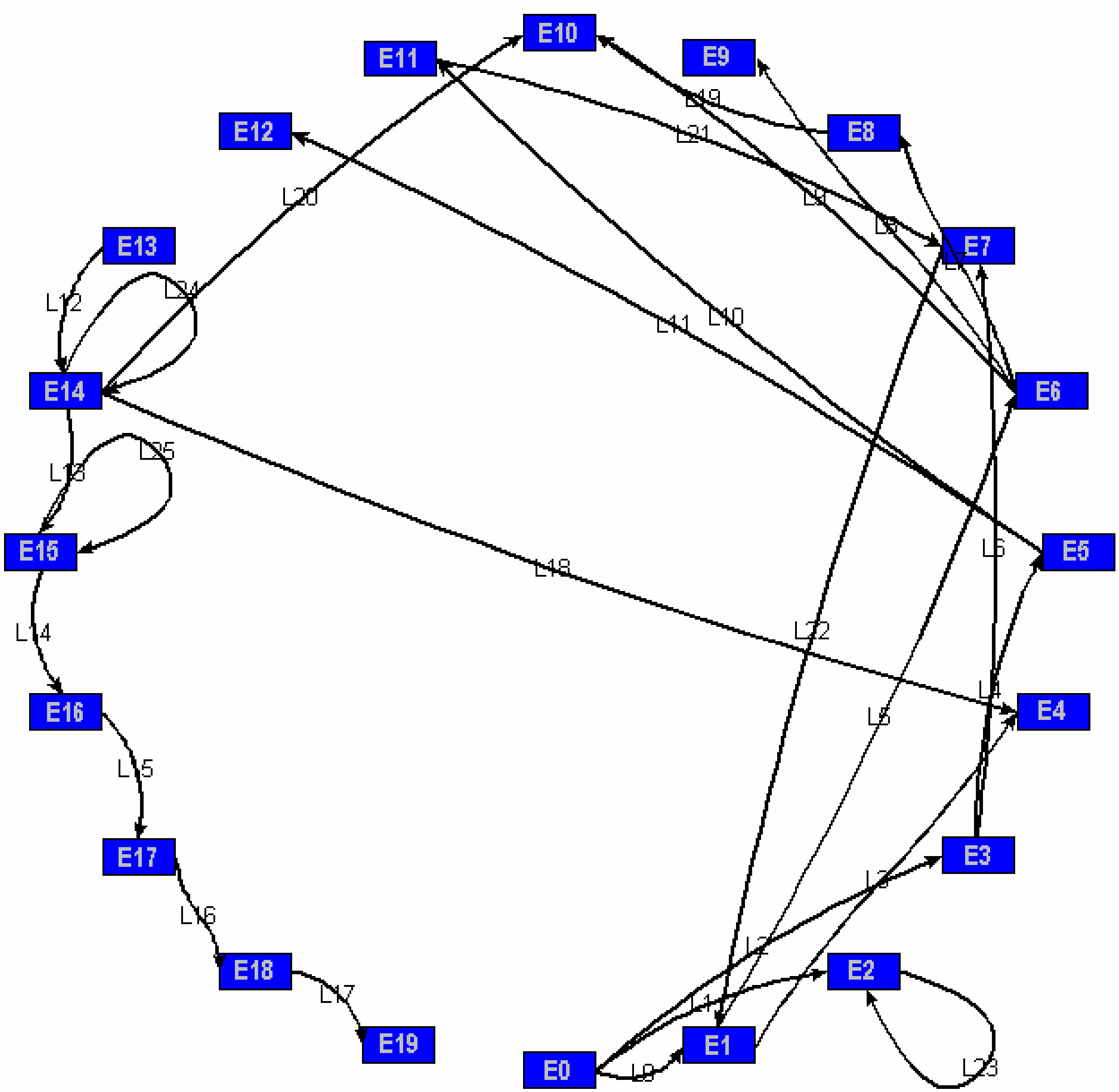
Spring-Electrical layout: Random test graph



Tree-like layout: Random test graph



Circle layout: Random test graph



Export/Import capabilities

- AToM^3 can export graphs to the following formats:
  - GML (Graph Modeling Language) , GXL (Graph Exchange Language) , and DOT
  - Can be imported by: yED, JGraphpad, and GraphViz

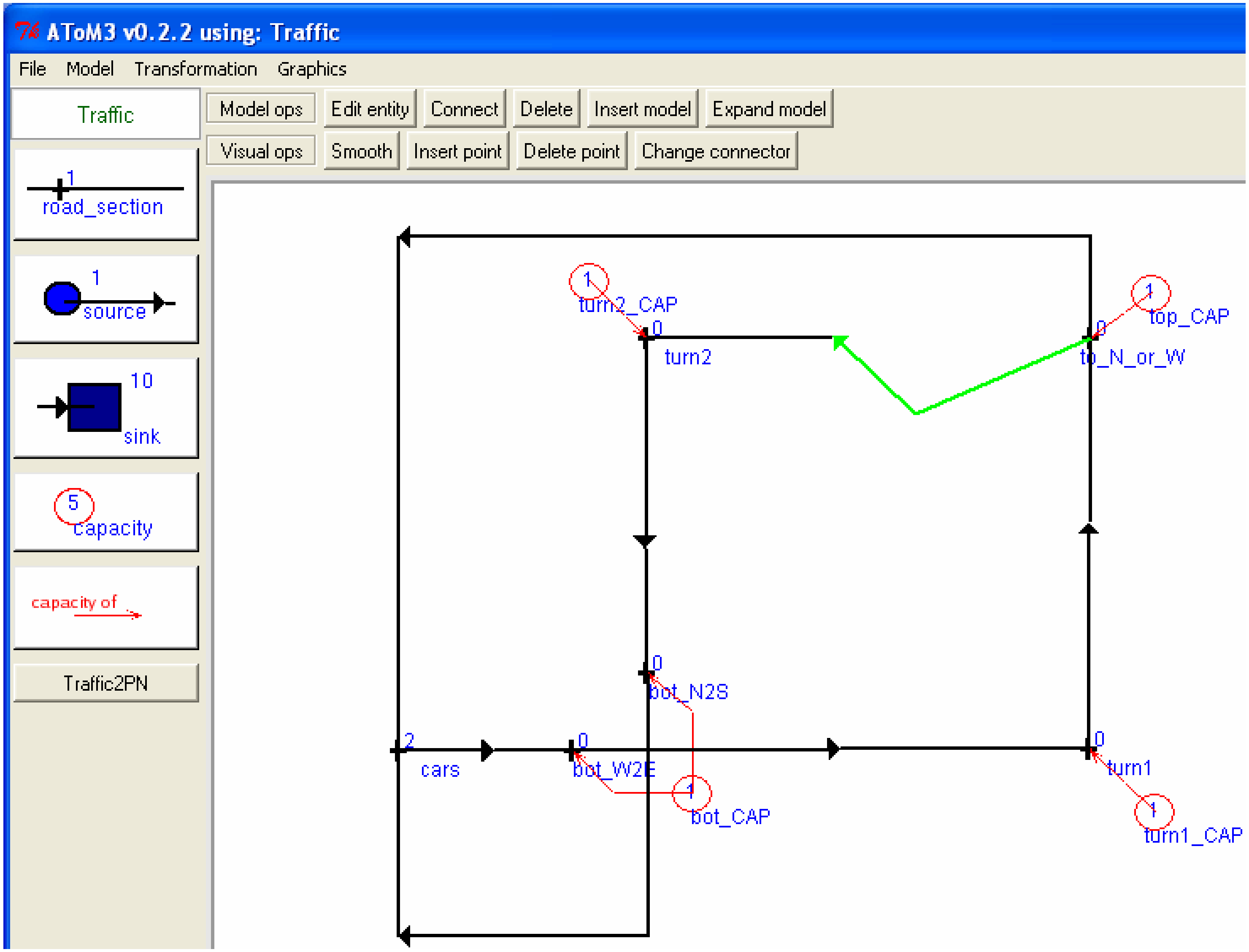
- In particular, yED is very powerful, and AToM^3 can re-import yED output, thus preserving AToM^3 model graphics

Questions

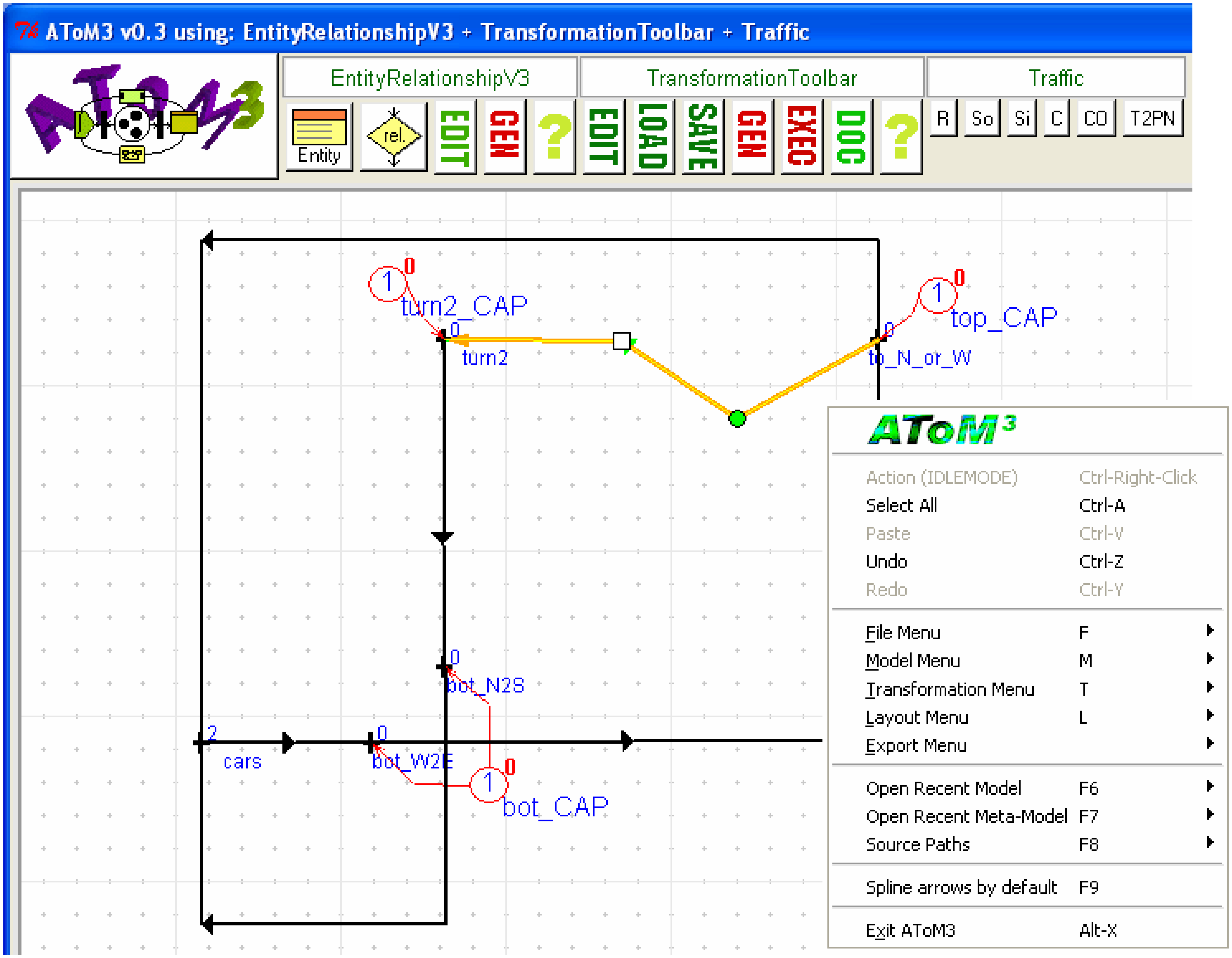
- Graph grammar documentation
- QOCA linear constraints
- Hierarchical layout
- Force transfer layout
- Sprint-Electrical layout
- Circle layout
- Tree-like layout
- Export/import tool support

- Next: GUI and statecharts

Graphical user interface (Before)



Graphical user interface (After)



GUI: List of improvements

- Context sensitive popup menus
- Help dialogs
- Uncaught exception handler (GUI + Logging)
- Combined option dialog and option file database
- Ability to select/manipulate more than one node/edge at a time
- Ability to scale nodes and edge drawings
- Cut/copy/paste nodes or just the semantic attributes
- Undo/redo

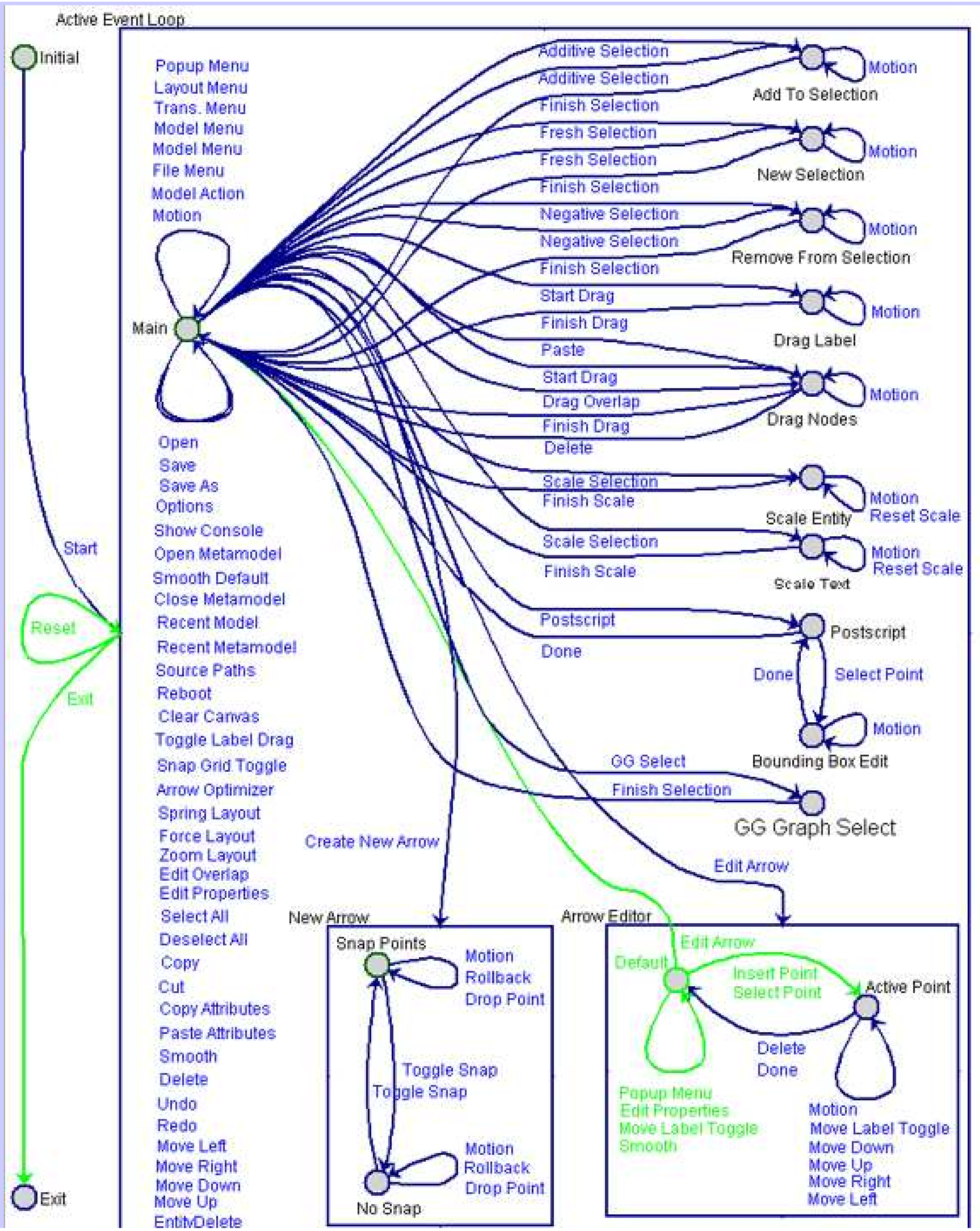
GUI: Under the hood

- Old method: if-statements and dictionary

```
self.UMLmodel.bind("<Button-1>", self.buttonPressed )
def buttonPressed (self, event):
        for action in self.userActionsMap.keys():
                if self.mode == action:
                   self.userActionsMap[action](self, event.x, event.y)
                   return
```

- DChart method: send event to statechart

```
def handler(event):
     self.UI_Statechart.event("Fresh Selection", event)
     self.UI_Statechart.event("Select Point", event)
     self.UI_Statechart.event("Drop Point", event)
     self.UI_Statechart.event("Start Drag", event)
canvas.bind("<ButtonPress-1>", handler)
```



Reactive behavior of the user interface described by a DChart

Currently simulating state reached after double-clicking on an arrow to edit the control points

DChart formalism, simulator, and code generator by Thomas Feng

GUI: Scoped User Interface

- Why have only one DChart for the entire application?

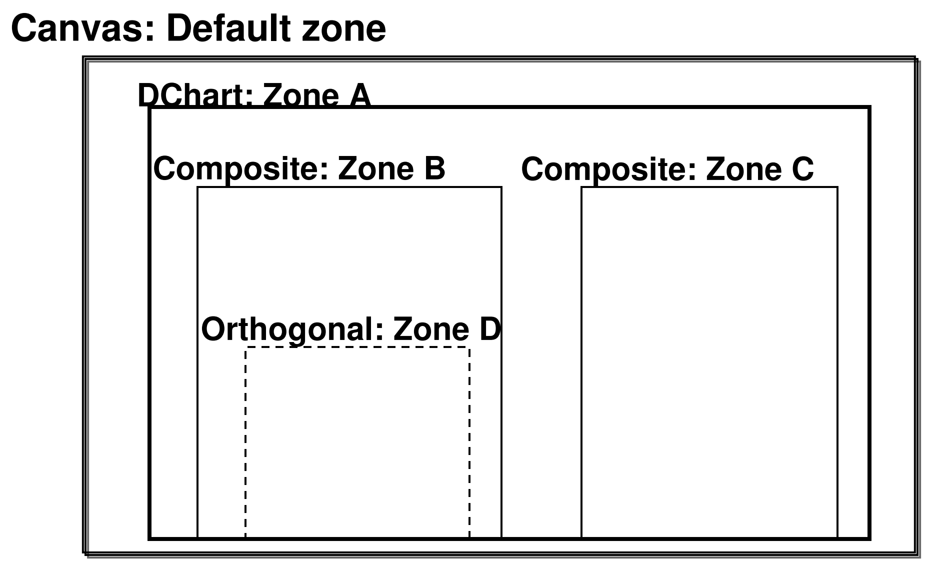
- New idea:
  - Divide the canvas into scoped UI zones
  - If input occurs inside a scoped UI zone, send input to all the DCharts defined for that zone

GUI: Scoped User Interface

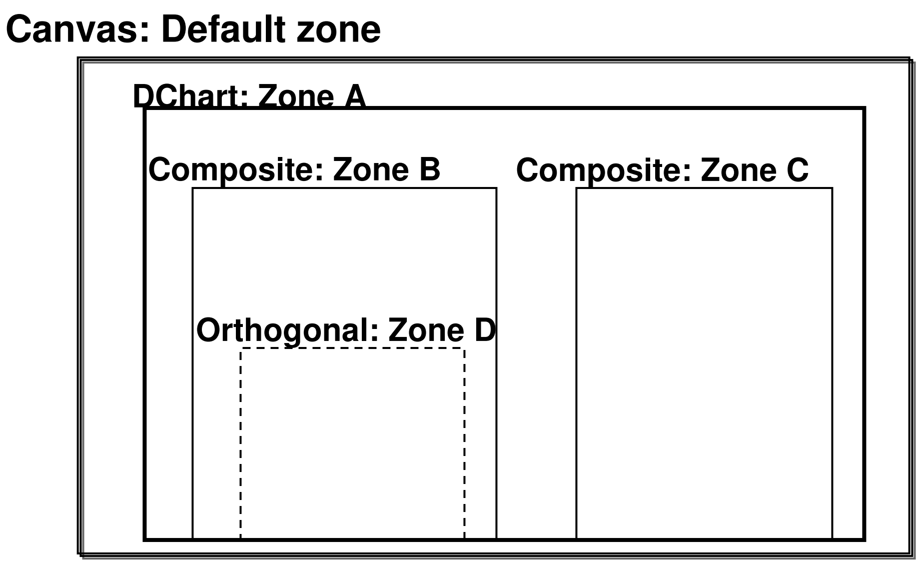
- Scoped UI bindings:

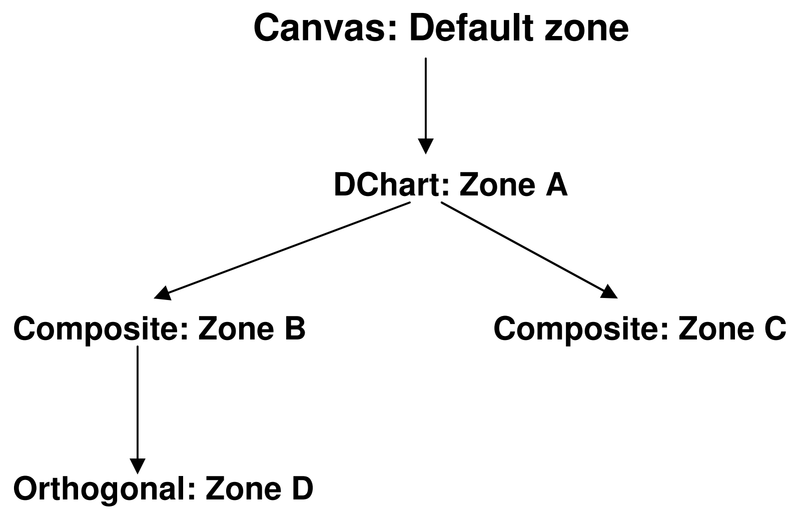
canvas.bind("<ButtonPress-1>", lambda event, scopedUI=self.UI_zone: scopedUI ('<ButtonPress-1>', event))

- Scoped UI example on a DChart like model

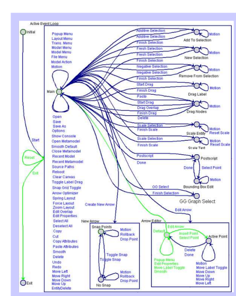


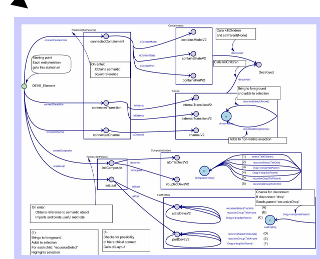
GUI: Scoped User Interface





<User Input> → Use event coordinates to find the deepest UI zone in the tree → Send event to each UI statechart defined for that zone

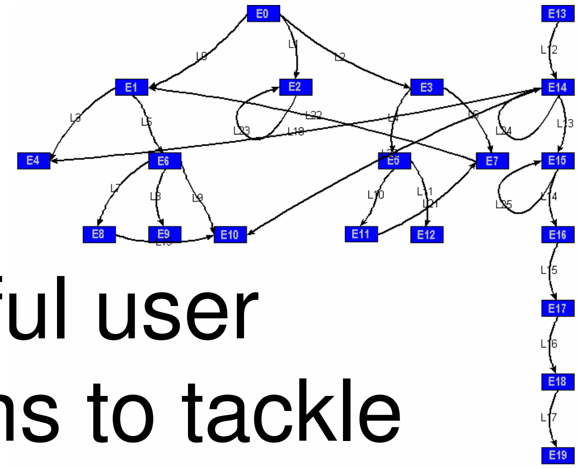




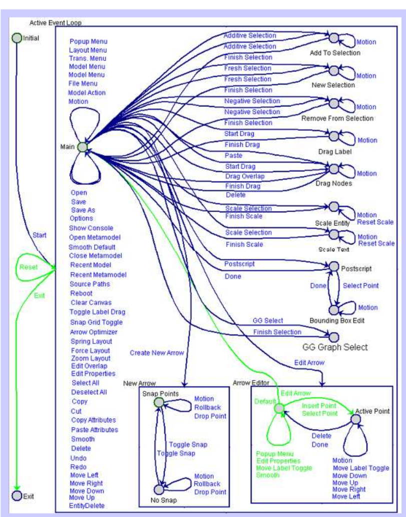
GUI: Scoped User Interface

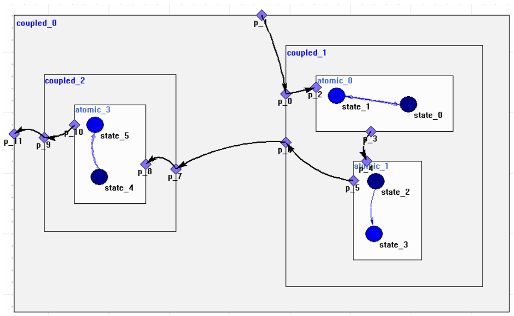
- Advantages:
  - Ability to create domain-specific user interfaces
  - Possibility of multiple domains co-existing with different behaviors
  - Ability to assign more than one UI behavior statechart to a given scope level
    - Although this duplicates the functionality of orthogonal states, it might be desirable to address different concerns in different statecharts
    - Example: reactive behavior versus automated layout triggers

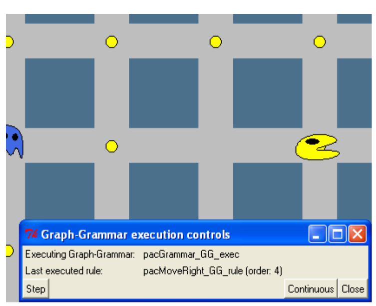
Conclusion

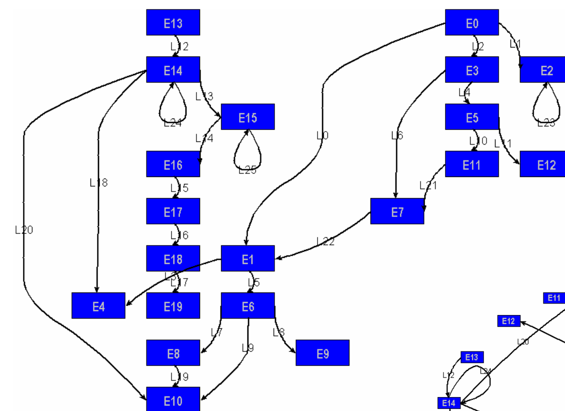


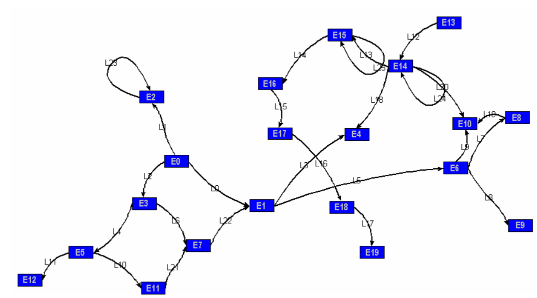
- AToM^3 now posses a powerful user interface and many algorithms to tackle graph layout

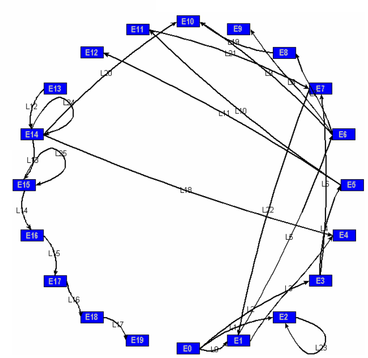












Future work

- Finish the new and improved DChart formalism featuring scoped UI
- Implement orthogonal layout for formalisms like Causal Block Diagrams
- Improve edge routing, perhaps taking inspiration from yED
- Extend QOCA integration in AToM^3 (on request)

Questions

1. Automatic graph layout
   - Graph grammars and QOCA linear constraints
   - General layout algorithms
     - Hierarchical, Force transfer, Spring-electrical, Circle, Tree-like, Import/Export
2. Graphical user interfaces and statecharts
   - GUI improvements
   - DCharts GUI behavior
   - Scoped UI