2007 Bellairs CAMPaM Workshop
24 April 2006

Aspects of CAMPaM

Hans Vangheluwe

McGill

Modelling, Simulation and Design Lab (MSDL)
School of Computer Science, McGill University, Montréal, Canada

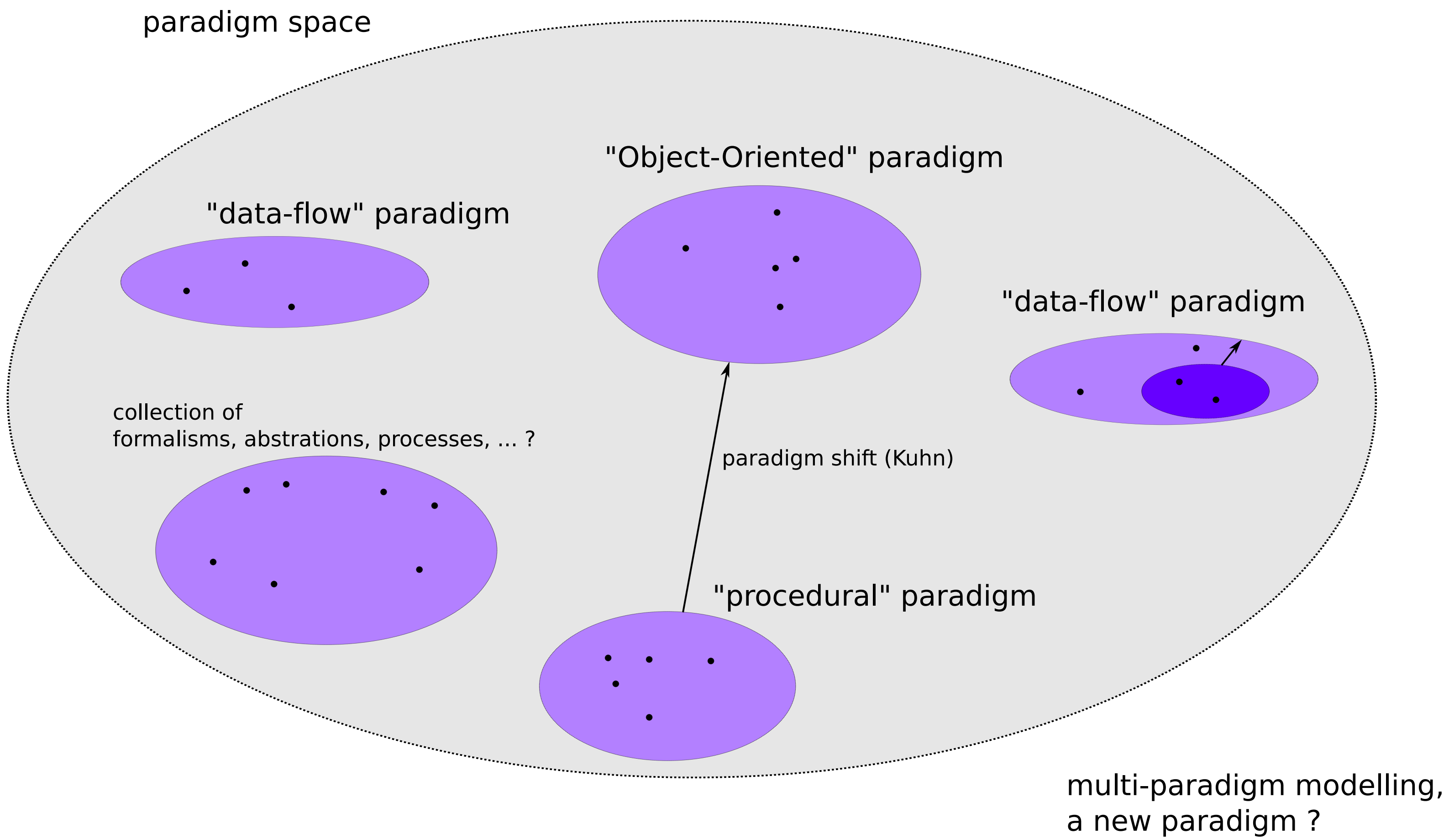
Overview

1. Multi-Paradigm Modelling, by example
2. Domain-Specific Modelling
3. Dissecting (and modelling) Modelling Languages
4. Building CAMPaM tools effectively
5. Challenges
6. Conclusions

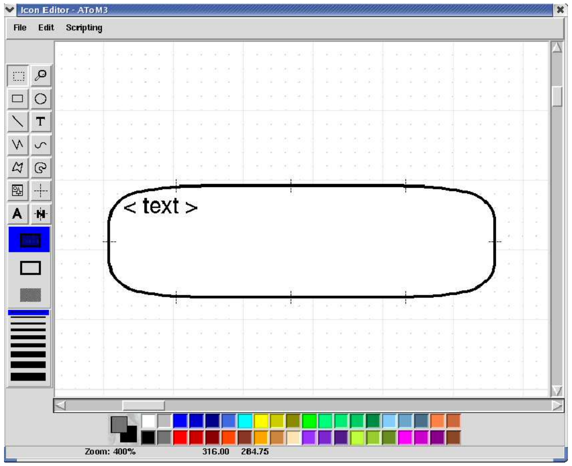
Multi-Paradigm Modelling . . .

- Intuitive notion (relative): OO paradigm, dataflow paradigm, . . .
- Thomas Kuhn, *The Structure of Scientific Revolutions*, 1962.
- Need to define more precisely . . .

Multi-Paradigm Modelling . . .



Modelling Variety of Complex Systems . . .

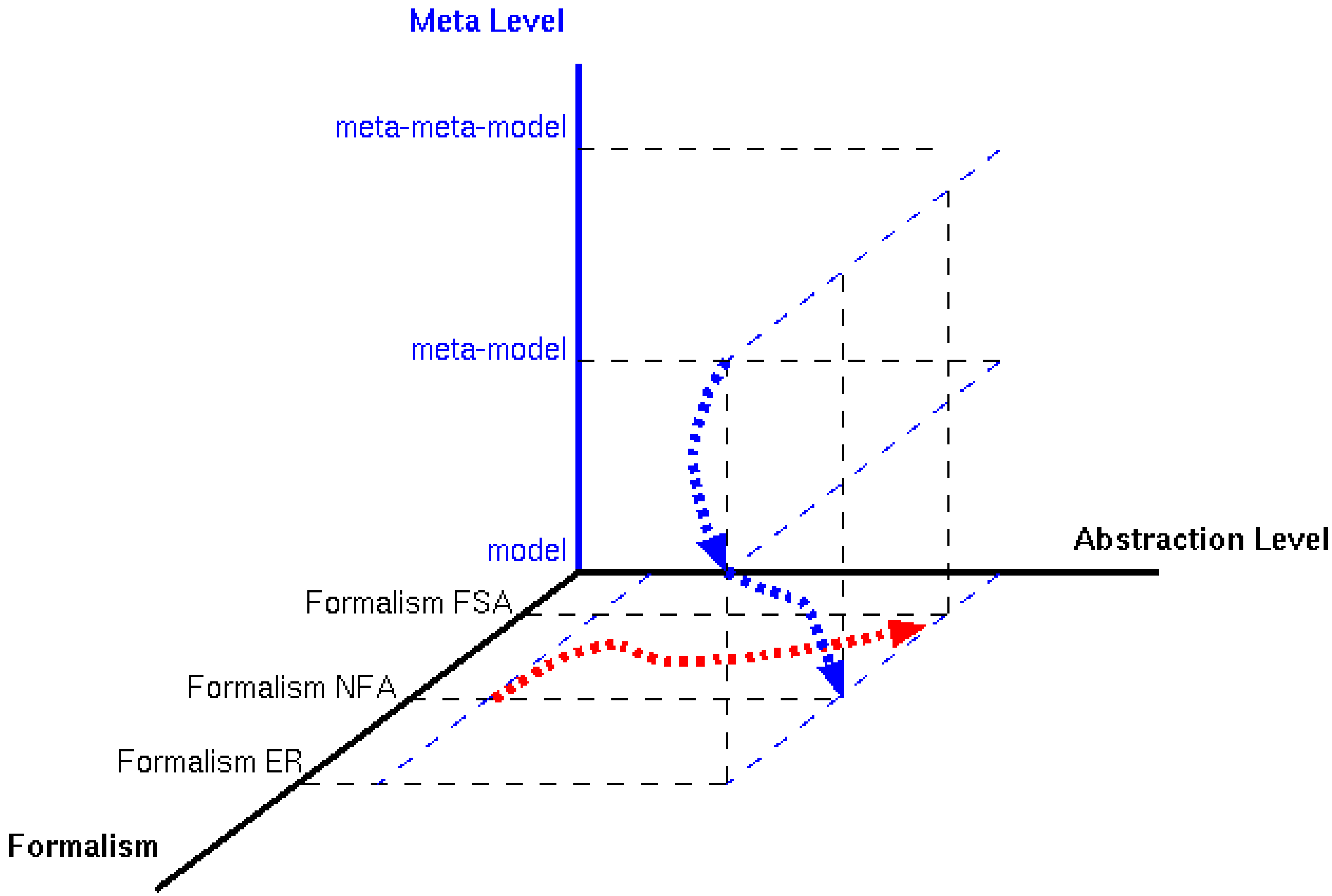


Need to be **modelled**

- at most appropriate **level of abstraction**
- in most appropriate **formalism(s)**
- with **transformations** as first-class models

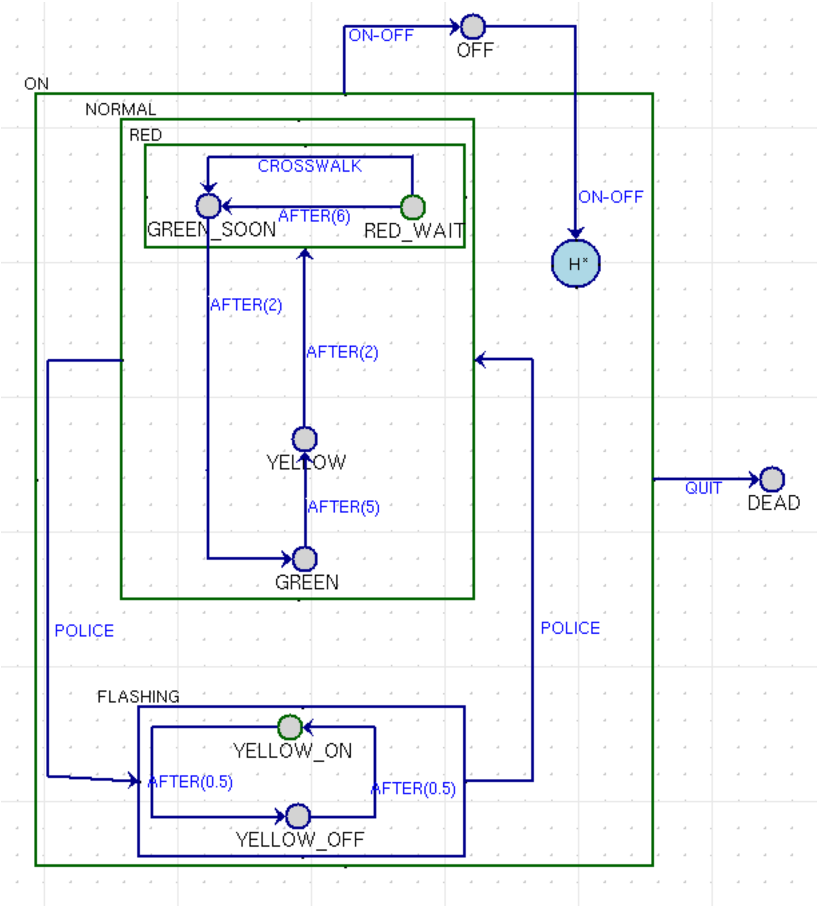
Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation: Transactions of the Society for Modeling and Simulation International*, 80(9):433-450, September 2004. Special Issue on Grand Challenges for Modeling and Simulation.

Multi-Paradigm Dimensions

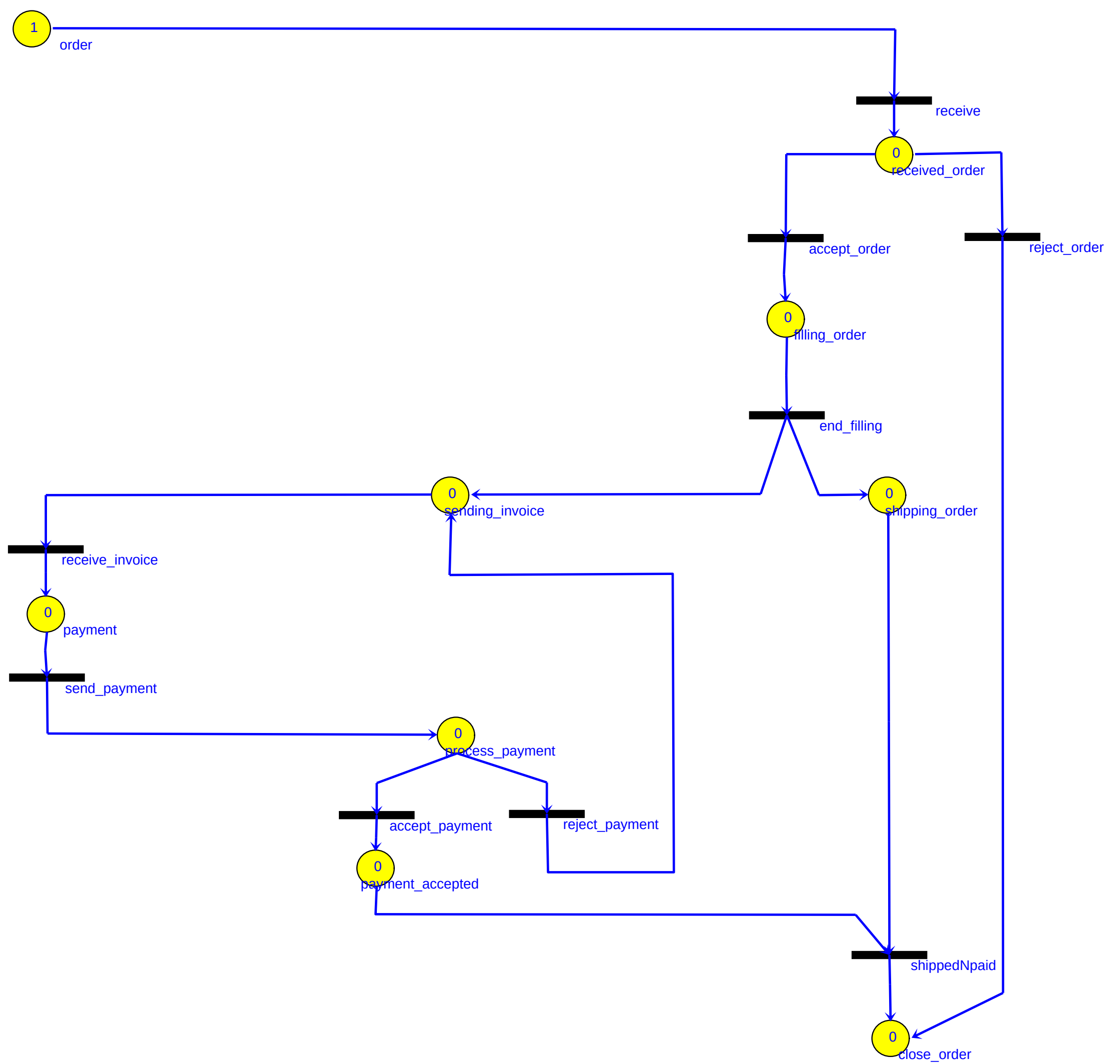


(note: dimensions are not totally ordered)

Available Information, Questions to be Answered, . . .
⇒ choice of Abstraction Level/Formalism

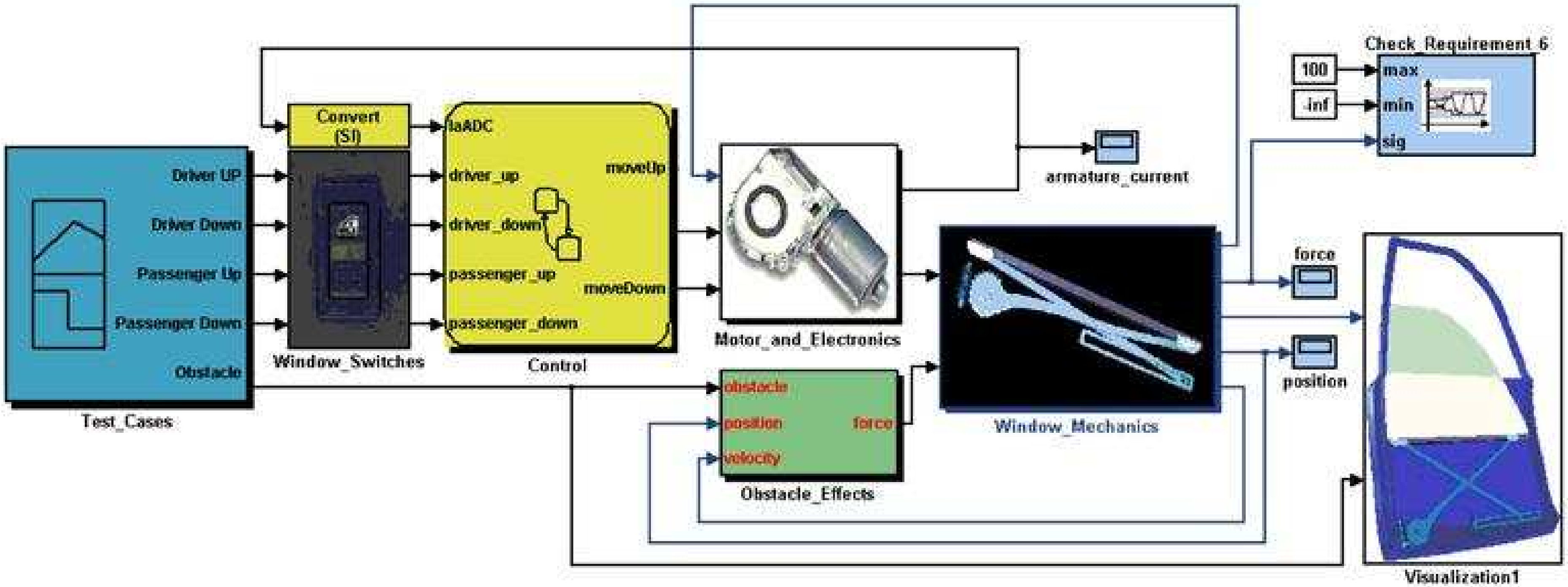


Choice of Abstraction Level/Formalism
Order Processing



Need Multiple Formalisms: Power Window

The Model Couples different Formalisms

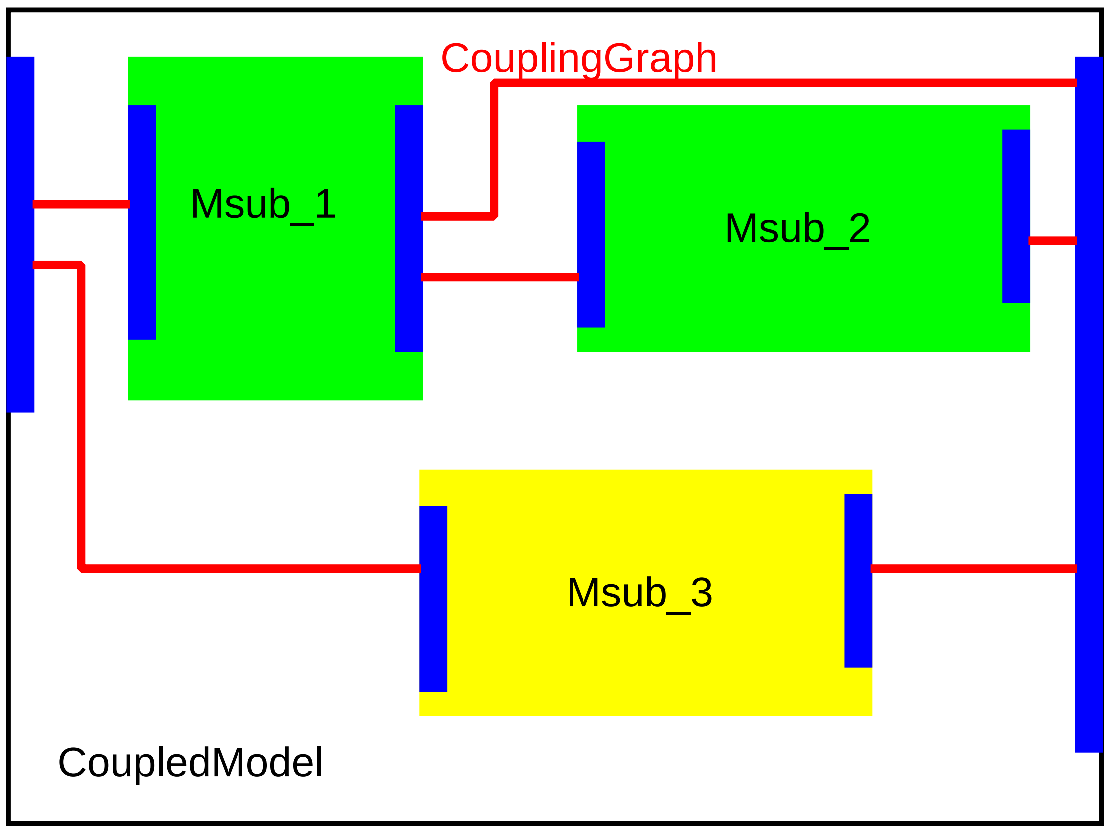


www.mathworks.com/products/demos/simulink/PowerWindow/html/PowerWindow1.html

Semantics of Coupled Models

- Super-formalism subsumes all formalisms
- Co-simulation (coupling resolved at trajectory level)
- Transform to common formalism

Multi-formalism coupled model: co-simulation



Co-simulation of multi-formalism coupled models

- Sub-models simulated with formalism-specific simulators.
- Interaction due to coupling is resolved at trajectory level.

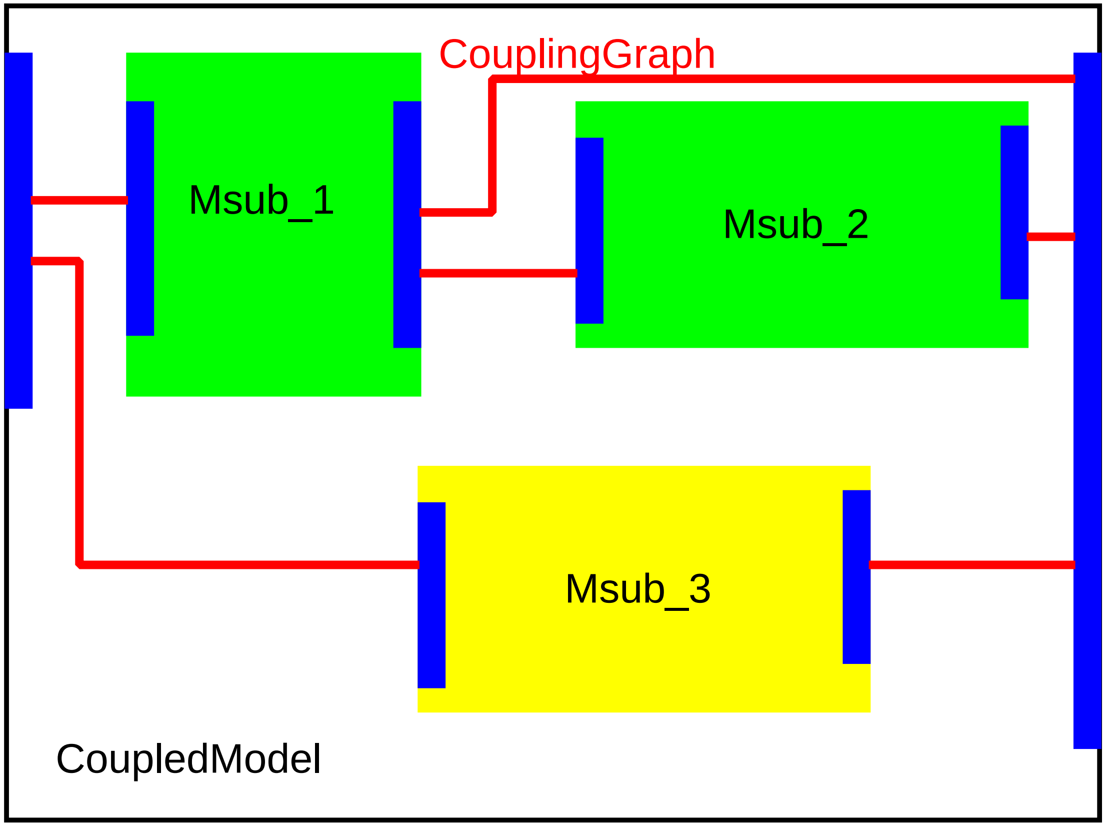
→ Loss of information.

→ Questions can *only* be answered at trajectory level.

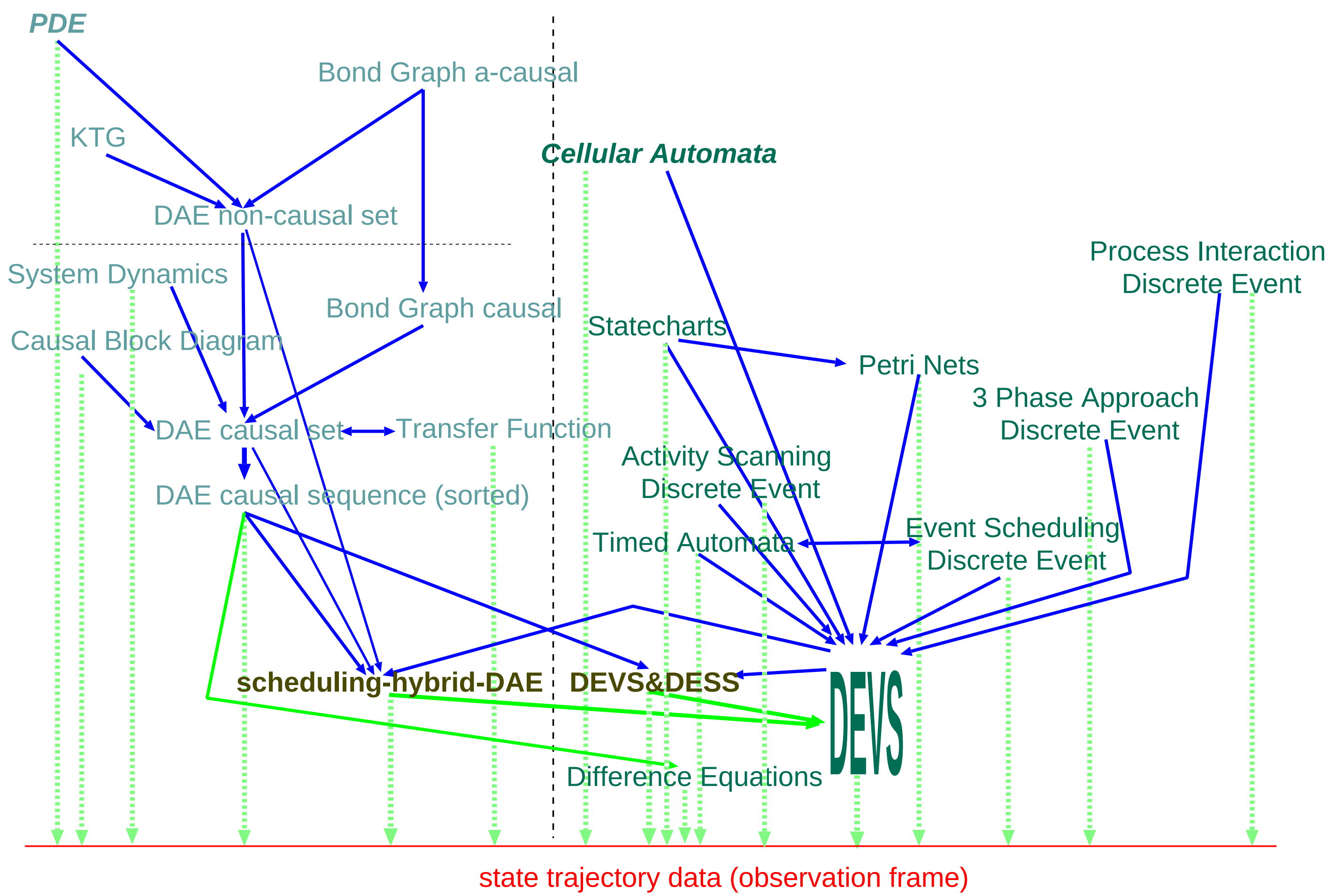
→ Speed and numerical accuracy problems
for continuous formalisms.

→ Meaningful for discrete-event formalisms (beware of legitimacy !).
Basis of the DoD High Level Architecture (HLA)
for simulator interoperability.

Multi-formalism coupled model: multi-formalism modelling



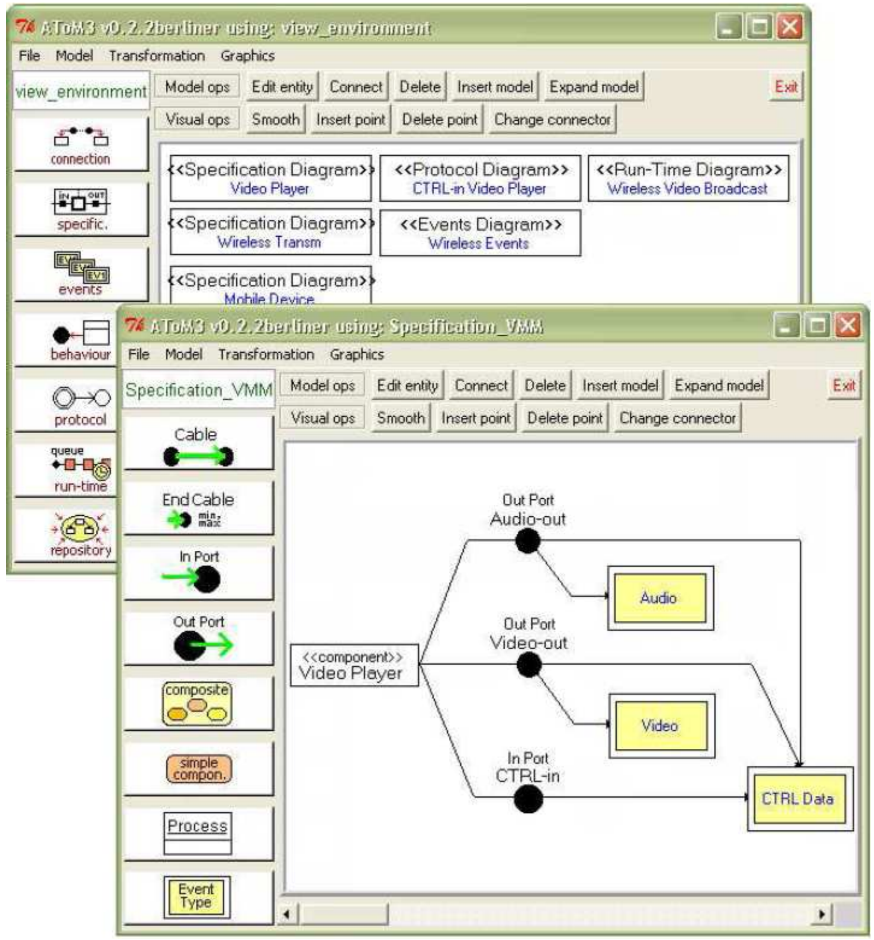
Formalism Transformation Graph



Multi-formalism modelling $\neq$ co-simulation

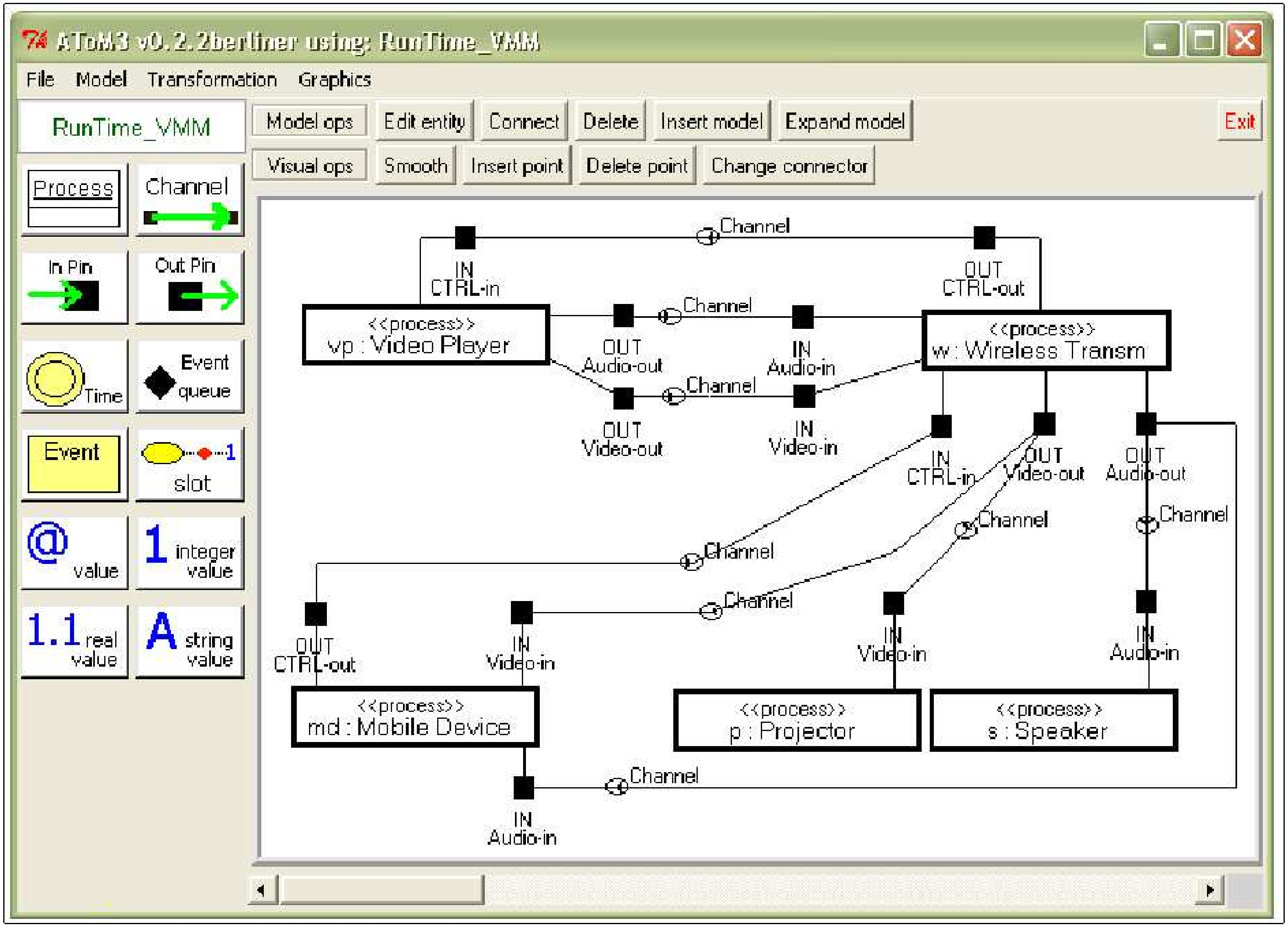
1. Start from a coupled multi-formalism model. Check consistency of this model (*e.g.*, whether causalites and types of connected ports match).
2. Cluster all formalisms described in the same formalism.
3. For each cluster, implement closure under coupling.
4. Look for the best common formalism in the Formalism Transformation Graph all the remaining different formalisms can be transformed to. Worst case: trajectory level (fallback to co-simulation).
5. Transform all the sub-models to the common formalism.
6. Implement closure under coupling of the common formalism.

Multiple (consistent !) Views (possibly in $\neq$ Formalisms)

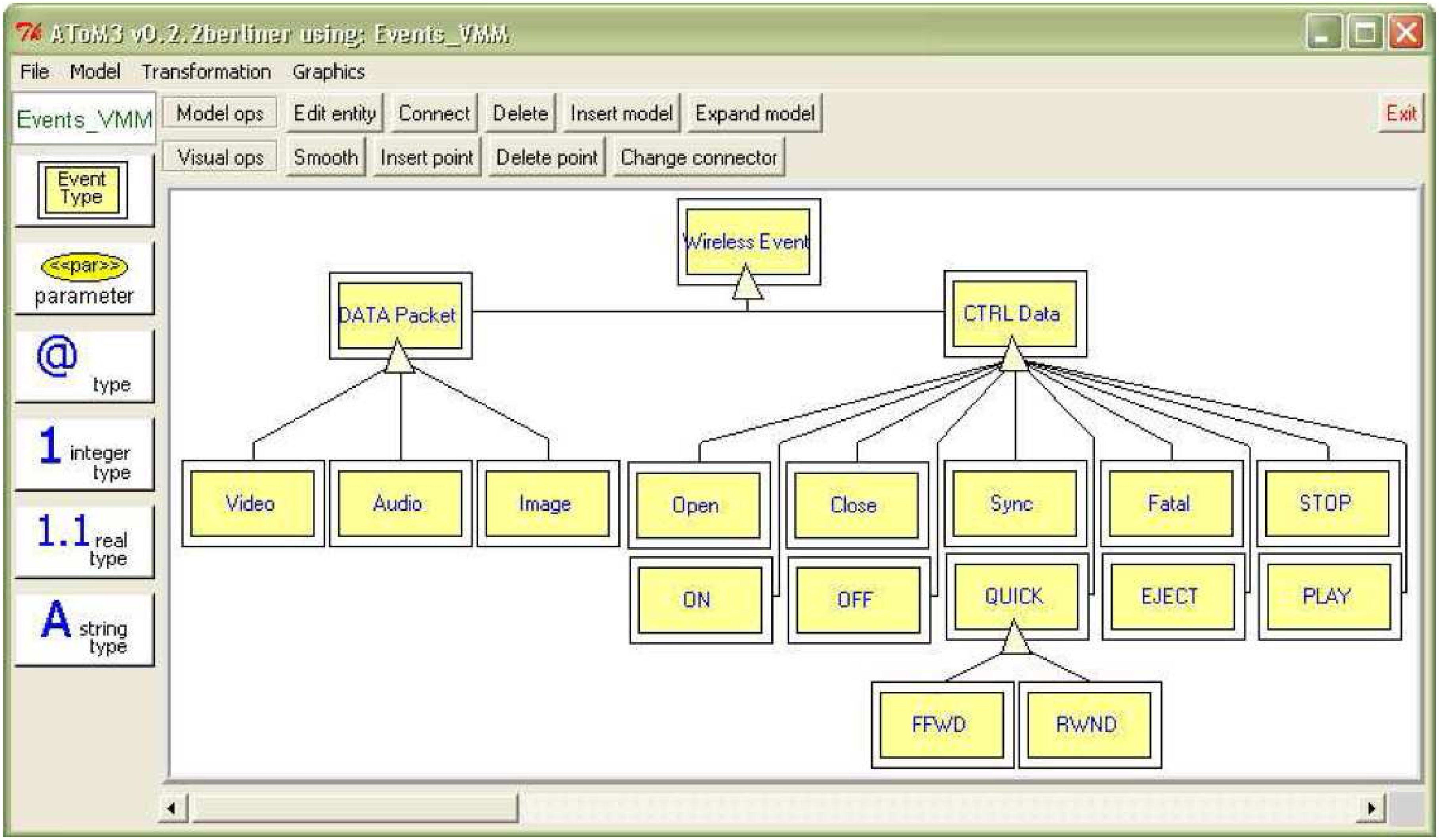


(work by Esther Guerra and Juan de Lara using projections of a "repository")

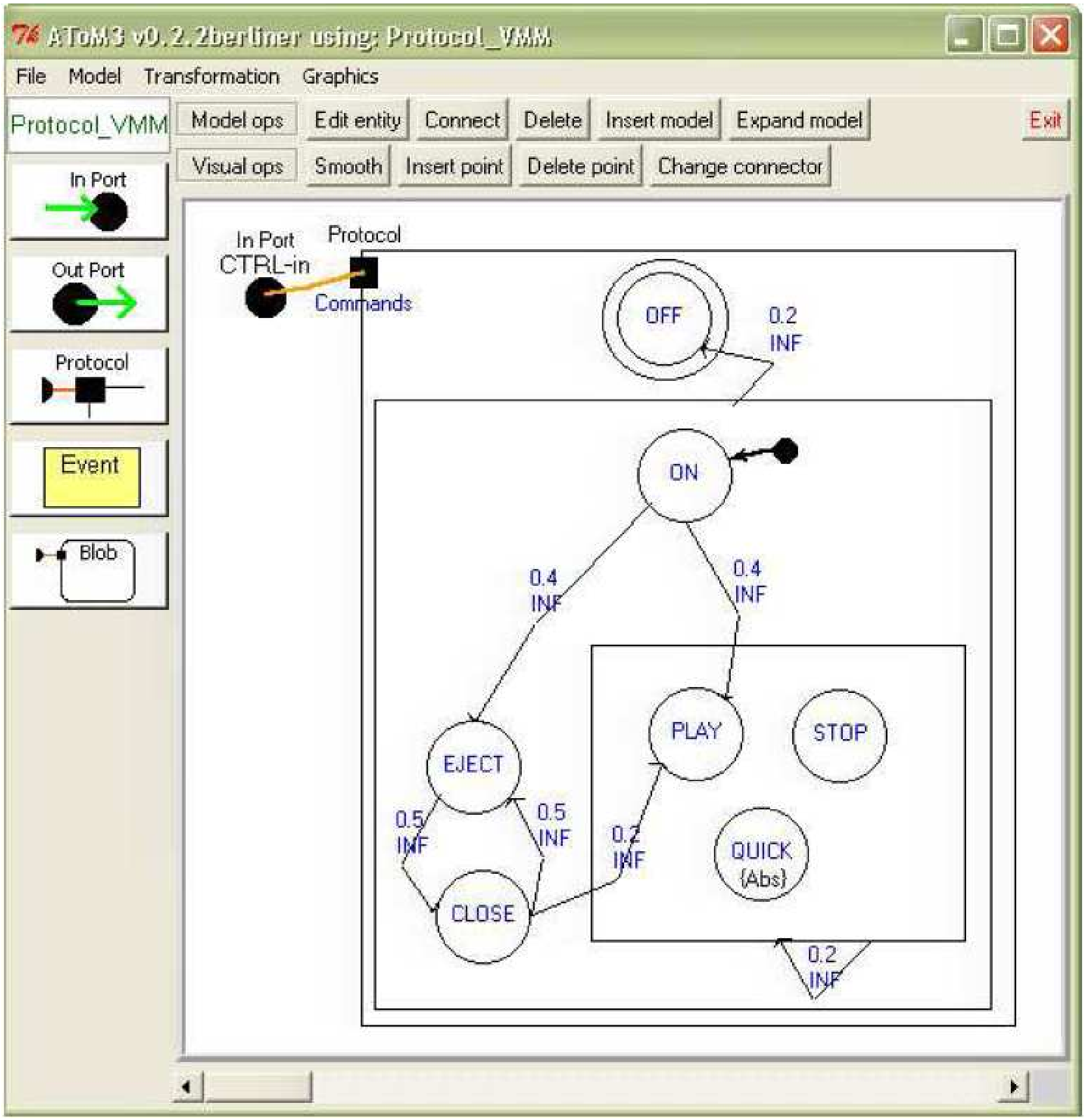
View: Runtime Diagram



View: Events Diagram

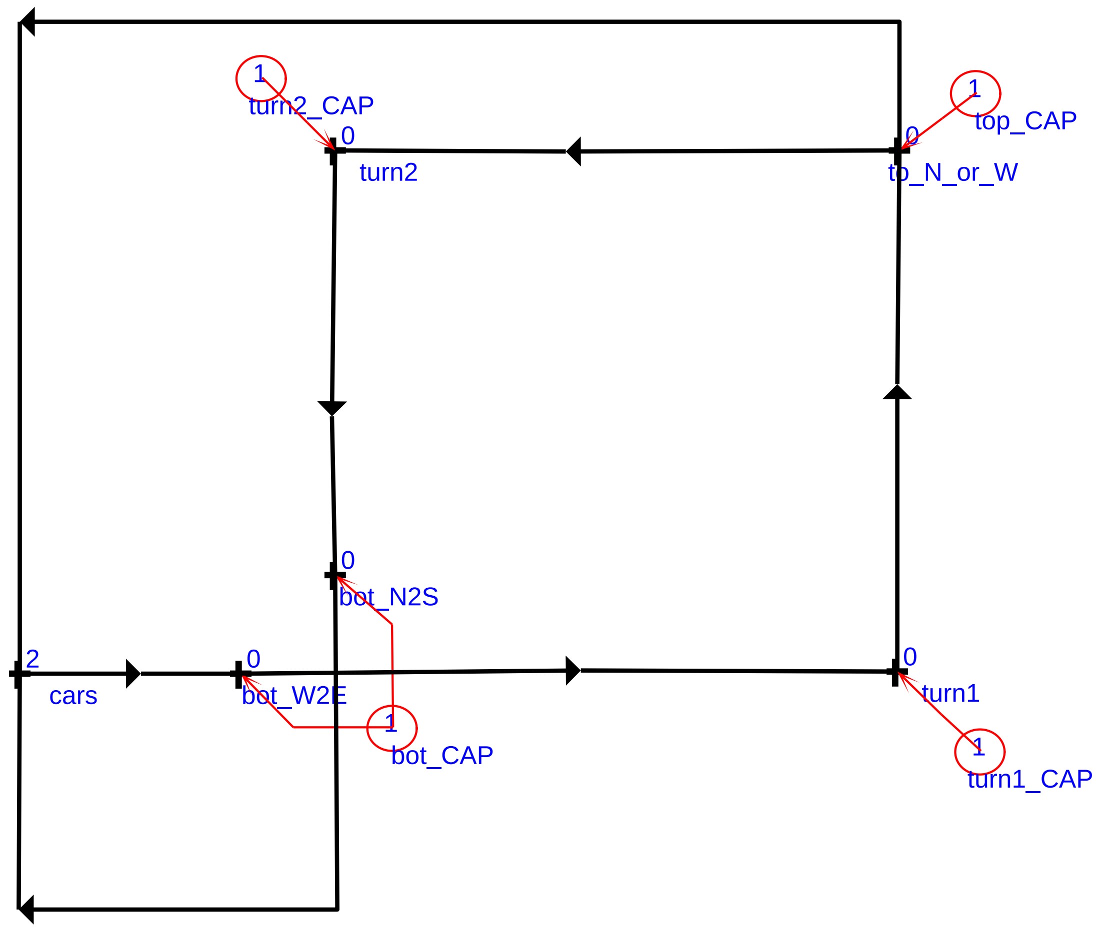


View: Protocol Statechart



The need for (modelled) Transformations Model/Analyze/Simulate Traffic Networks

An un-timed Traffic model



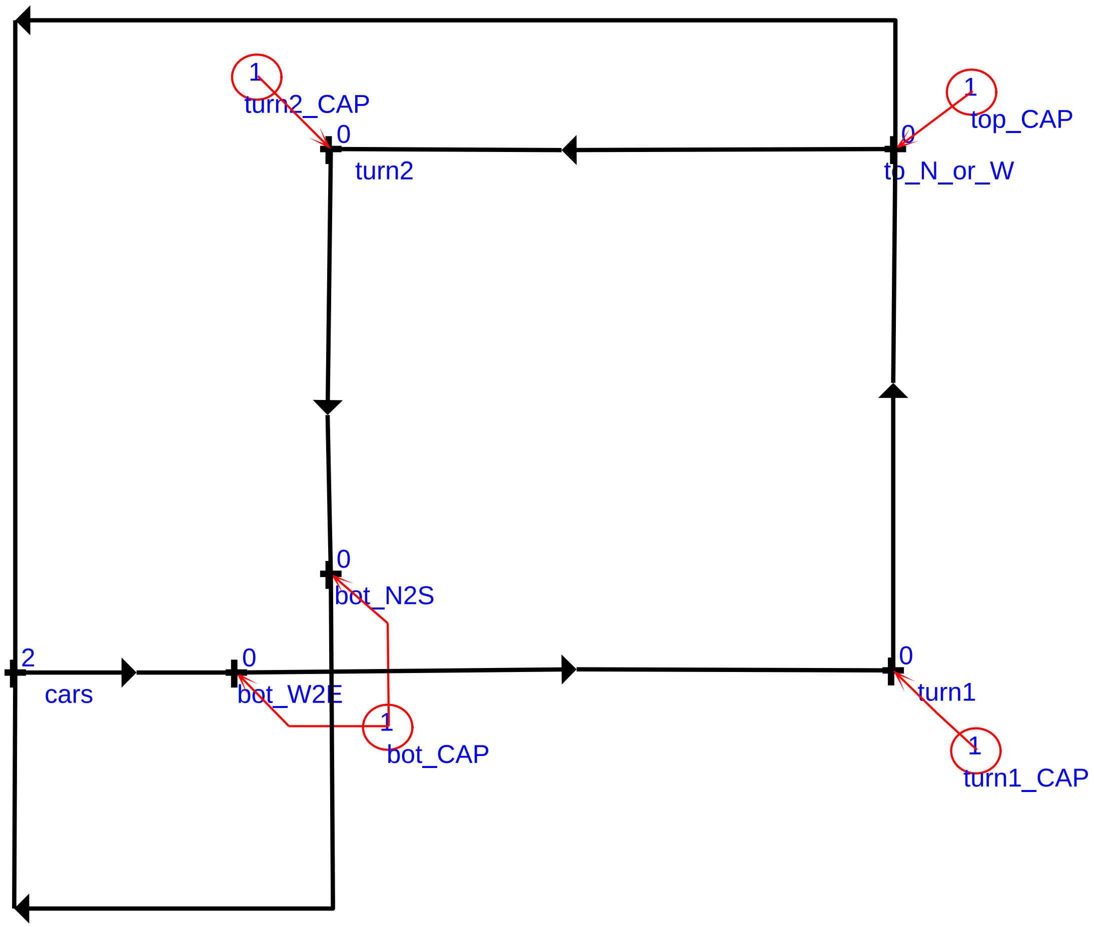
Modelling Traffic's Semantics

- choices: timed, un-timed, ... (level of abstraction)
- **denotational**: map onto known formalism (TTPN, PN)
  ... good for analysis purposes
- **operational**: procedure to execute/simulate model
  ... may act as a reference implementation
- note: need to **prove** consistency between denotational and operational semantics if both are given !

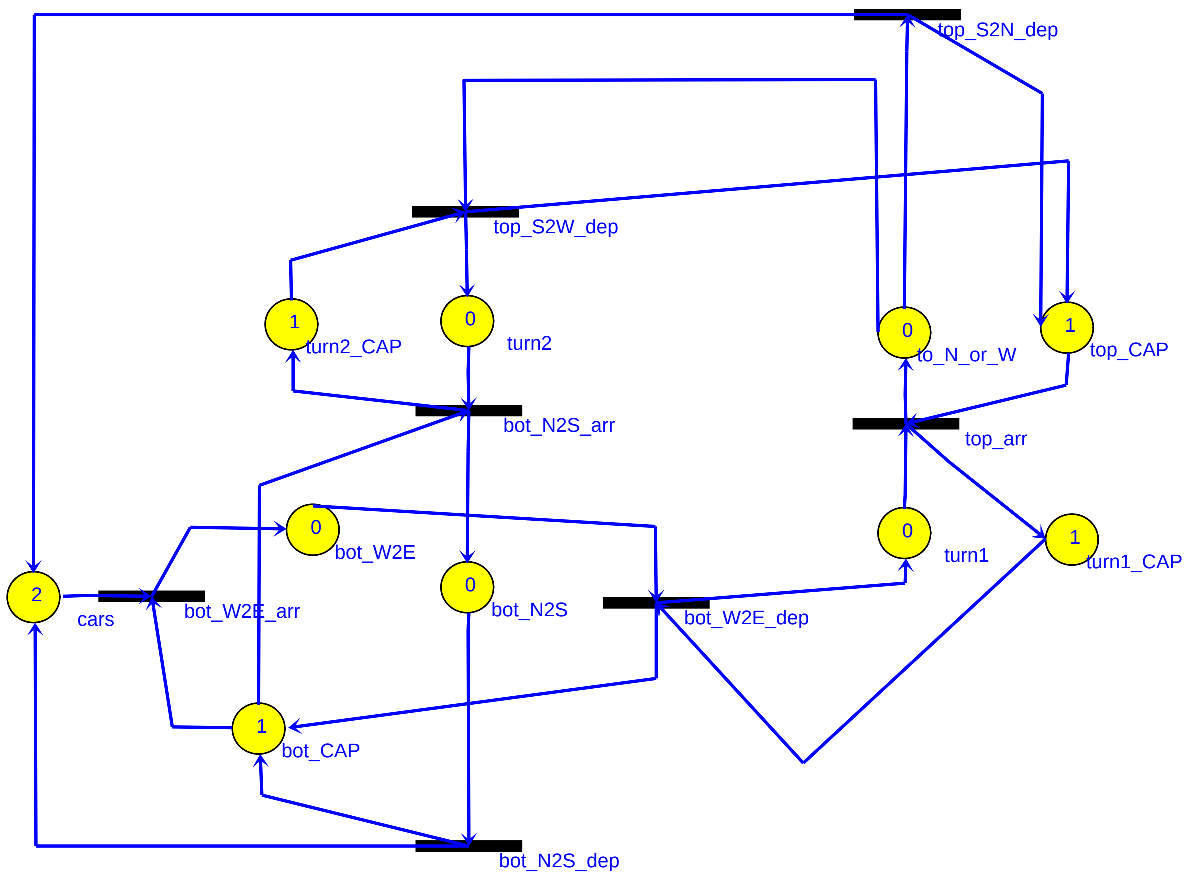
Modelling Traffic's (un-timed) semantics in terms of Petri Nets

- need a (meta-)model of **Traffic**
- need a (meta-)model of **Petri Net**s
- need a model of the mapping: **Traffic** $\Rightarrow$ **Petri Net**

Input to semantic mapping transformation



The Petri Net describing its behaviour obtained by automatic transformation



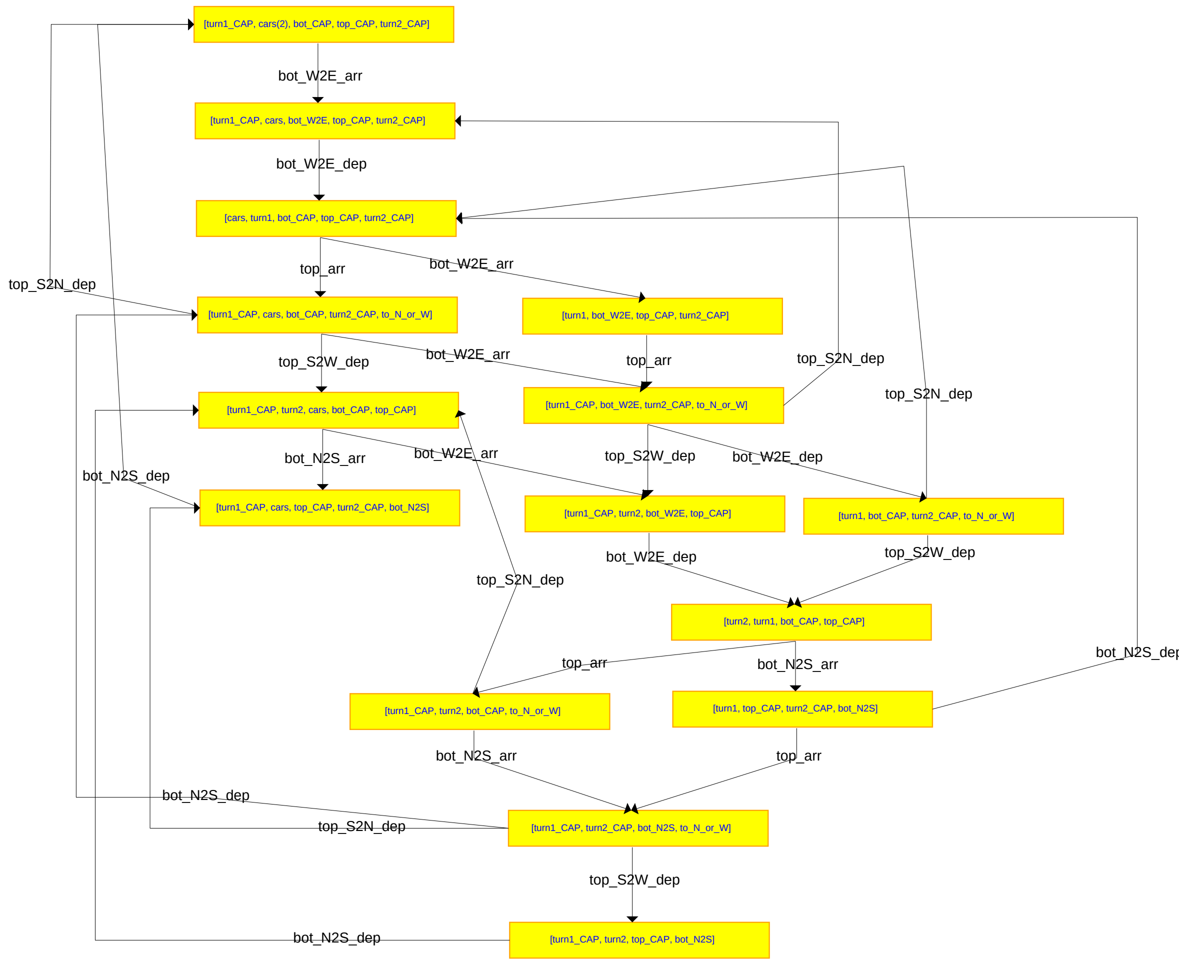
Static Analysis of the Transformation Model

The transformation (specified by a Graph Grammar) model must satisfy the following requirements (of semantic mapping):

- **Termination:**
  the transformation process is *finite*
- **Convergence/Uniqueness:**
  the transformation results in a *single* target model
- **Syntactic Consistency:**
  the target model must be *exclusively* in the target formalism

These properties can often (but not always)
be **statically** checked/proved.

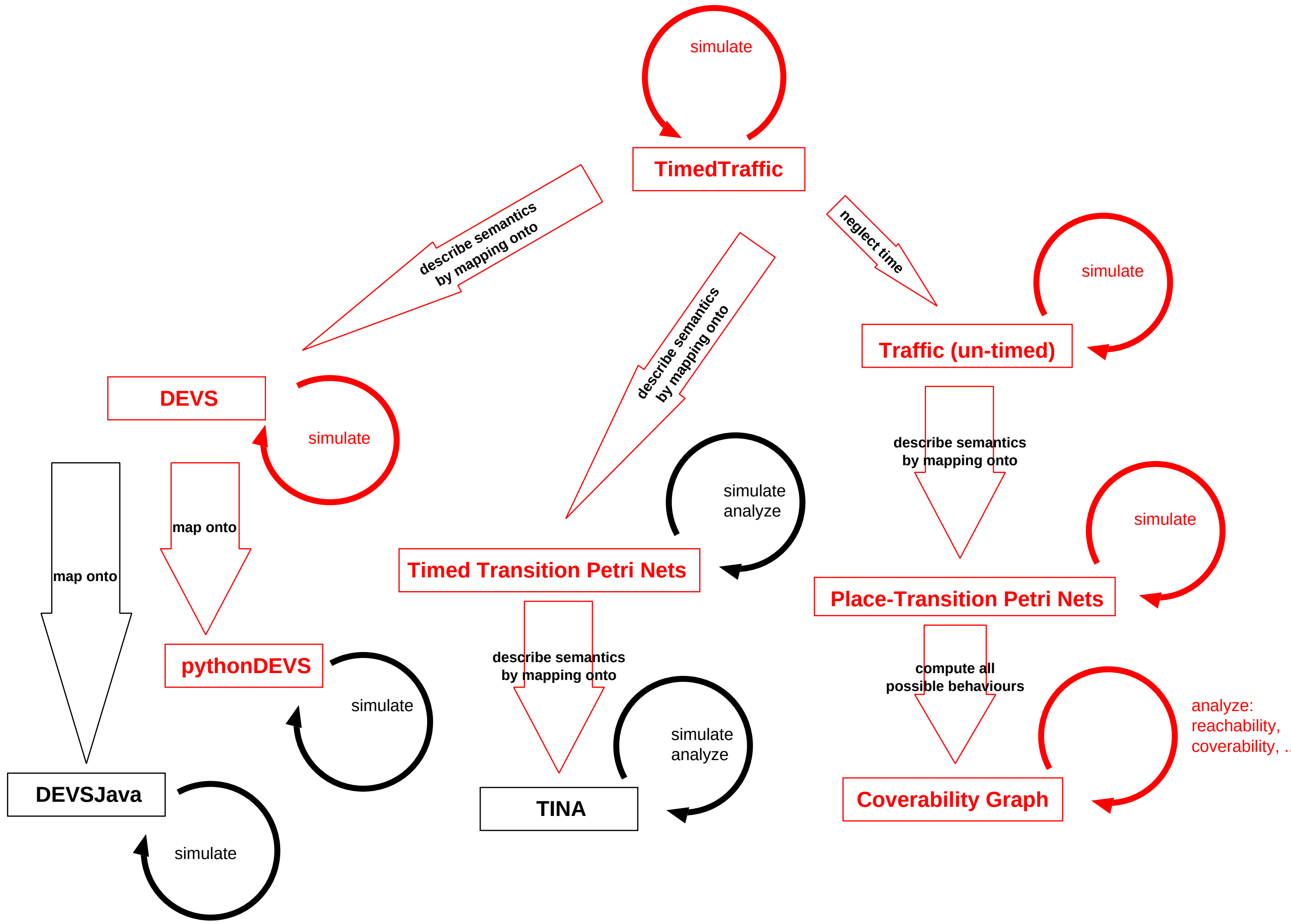
More transformations: Liveness Analysis on Coverability Graph



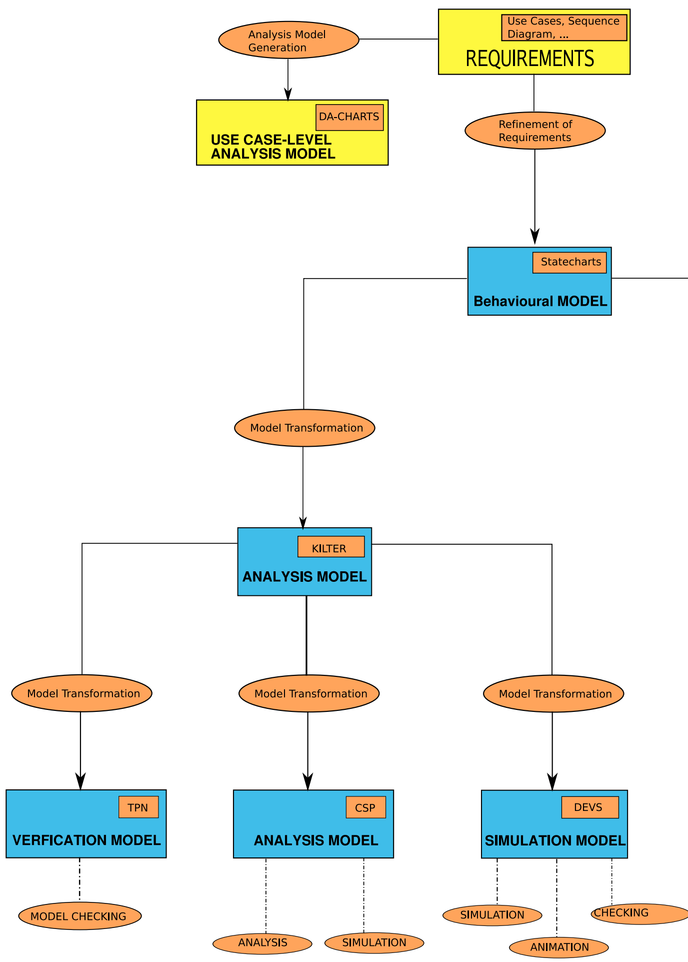
Conservation Analysis

```
1.0 x[turn1_CAP] + 1.0 x[turn1] = 1.0

1.0 x[cars] + 1.0 x[bot_W2E] + 1.0 x[turn1] +
1.0 x[to_N_or_W] + 1.0 x[turn2] + 1.0 x[bot_N2S] = 2.0

1.0 x[top_CAP] + 1.0 x[to_N_or_W] = 1.0

1.0 x[turn2_CAP] + 1.0 x[turn2] = 1.0

1.0 x[bot_CAP] + 1.0 x[bot_W2E] + 1.0 x[bot_N2S] = 1.0
```

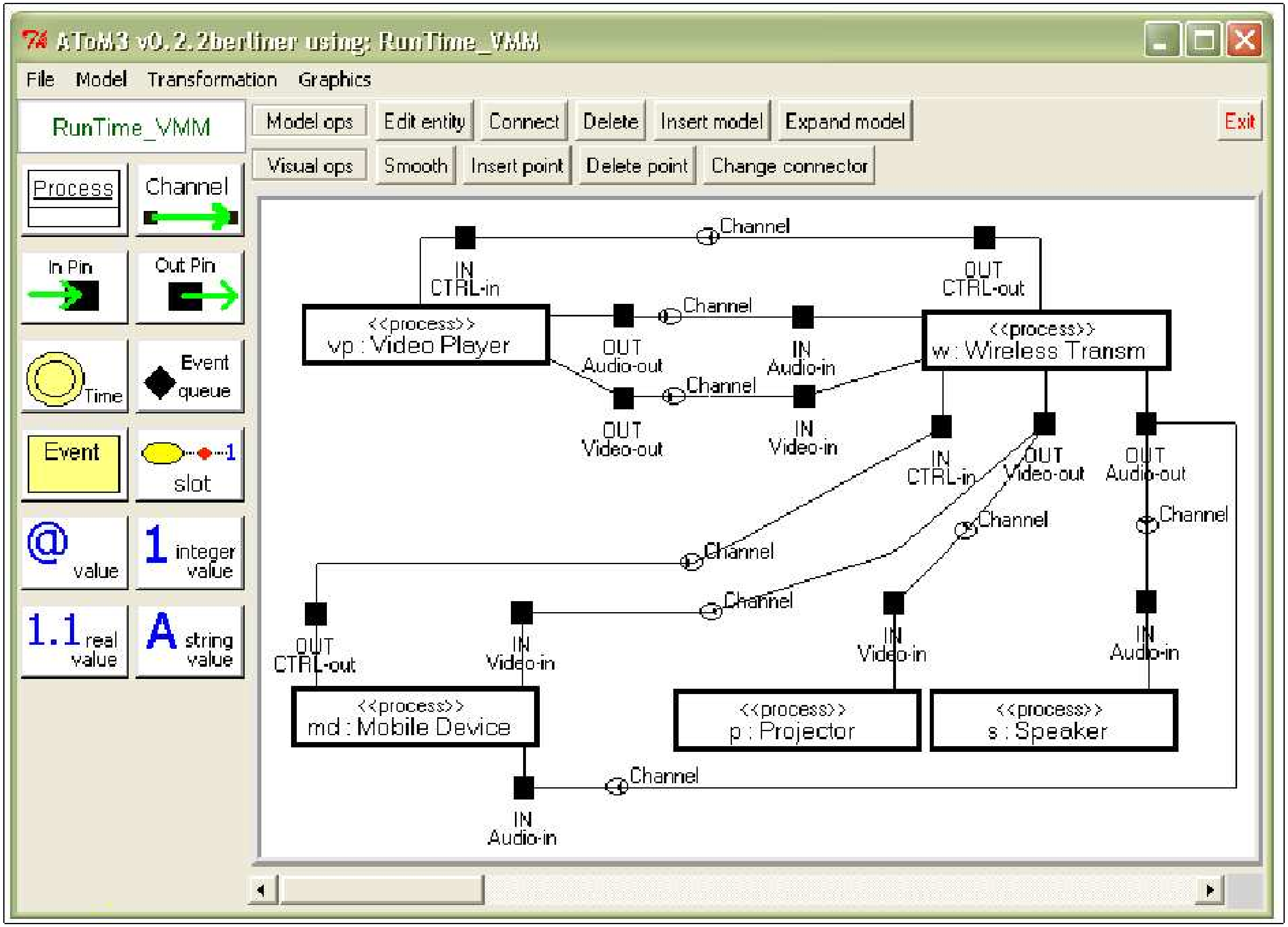
The Big Picture: Transform Everything!



Software Development Process: Transformations (refinement vs. translation)



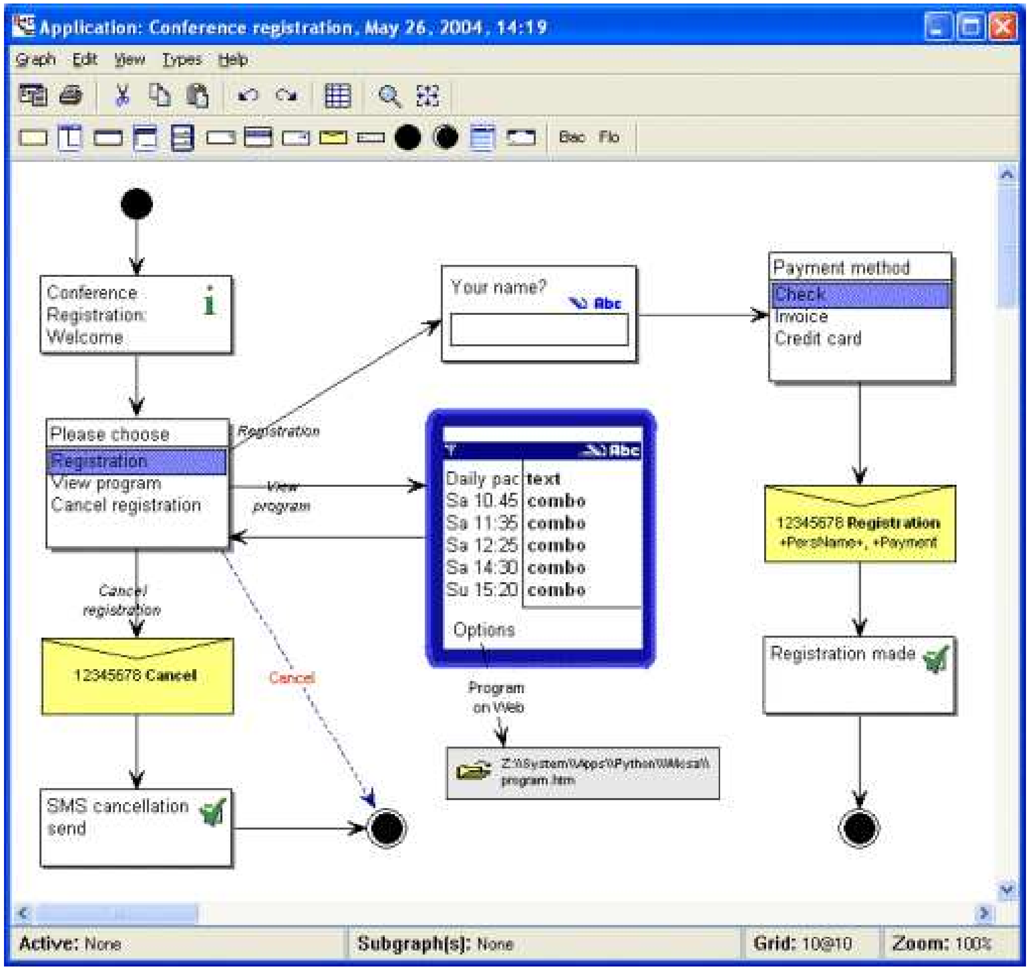
Domain-Specific (Visual) Modelling



DS(V)M Example: smart phones, the application

MetaEdit+ (`www.metacase.com`)

DS(V)M Example: smart phones, the Domain-Specific model

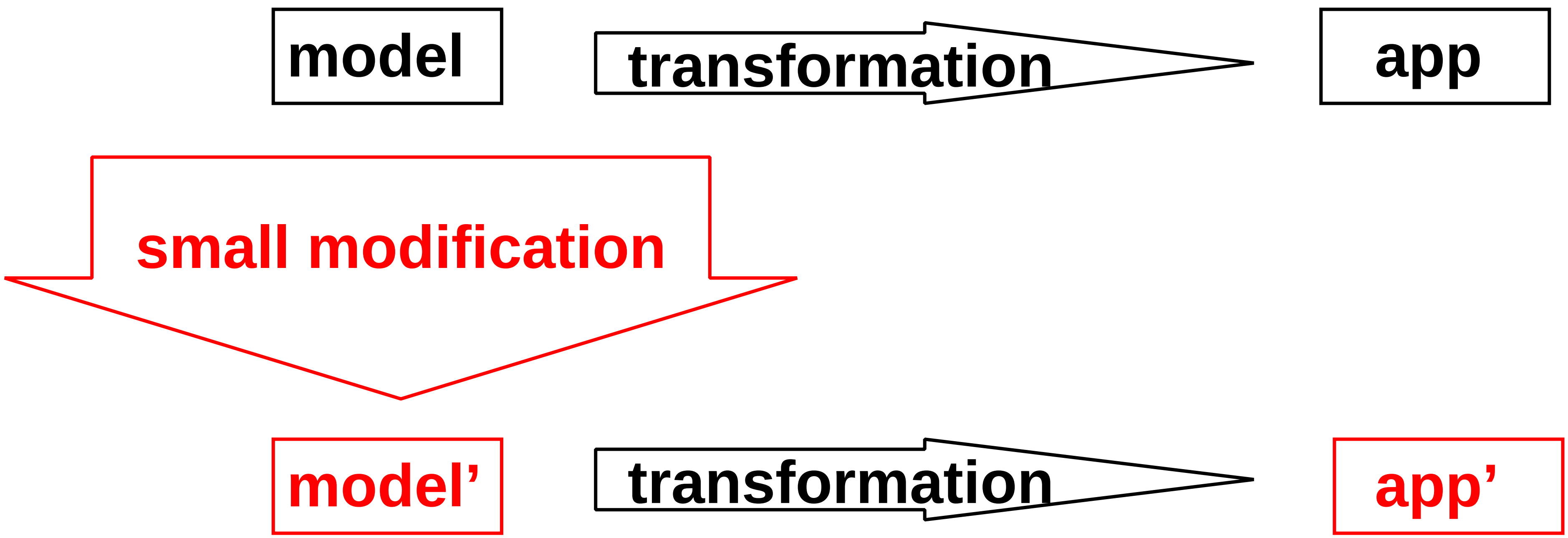


Why DS(V)M ?
(as opposed to General Purpose modelling)

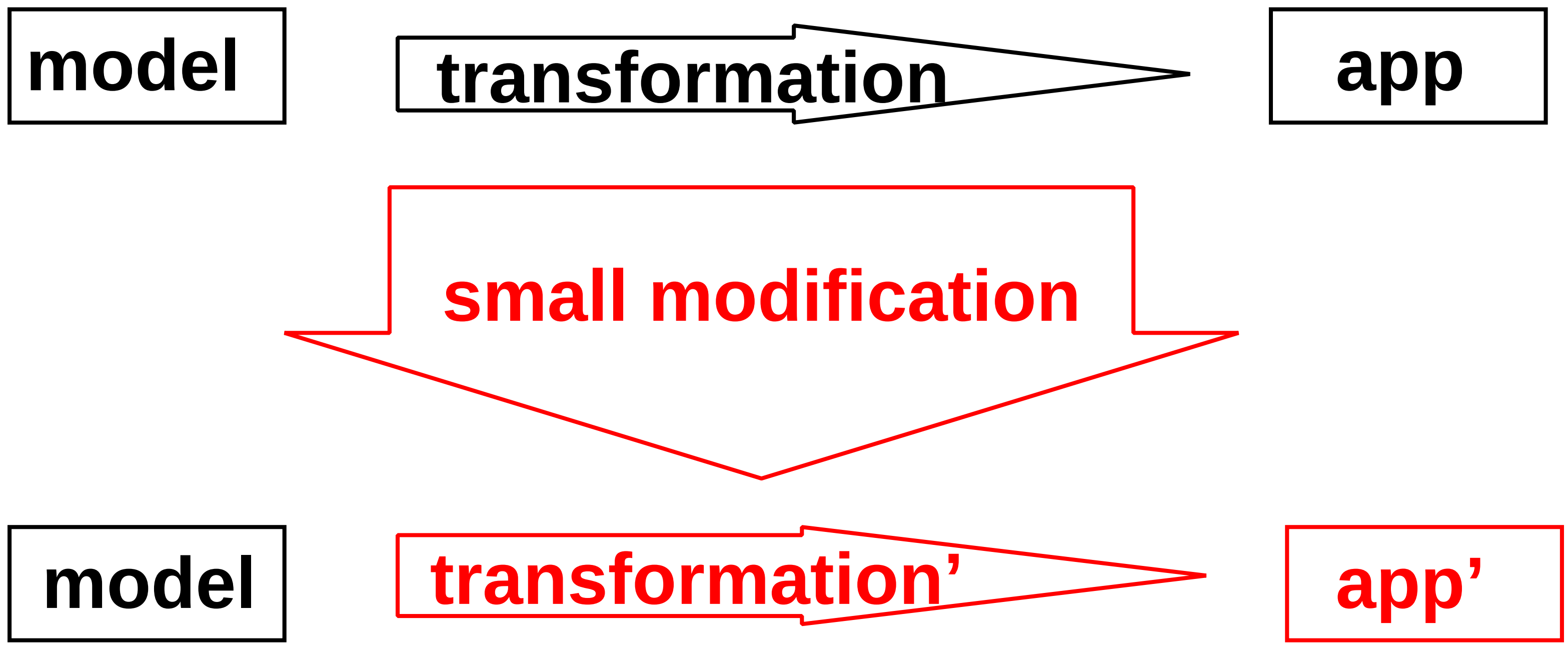
- **match the user's mental model** of the problem domain
- **maximally constrain** the user (to the problem at hand)
  $\Rightarrow$ easier to learn
  $\Rightarrow$ avoid errors
- **separate** domain-expert's work
  from analysis/transformation expert's work

Anecdotal evidence of 5 to 10 times speedup

Model-Based Development: Modify the Model



Model-Based Development: Modify the Transformation (model)



Transformation may be multi-step

- divide-and-conquer, modularity, re-use, . . . ;
- re-use existing transformations;
- potential for optimization at every level;
- multi-formalism modelling by transforming onto a common formalism;
- in case of Domain-Specific formalisms: usually small transformation onto known (syntax & semantics) formalism.

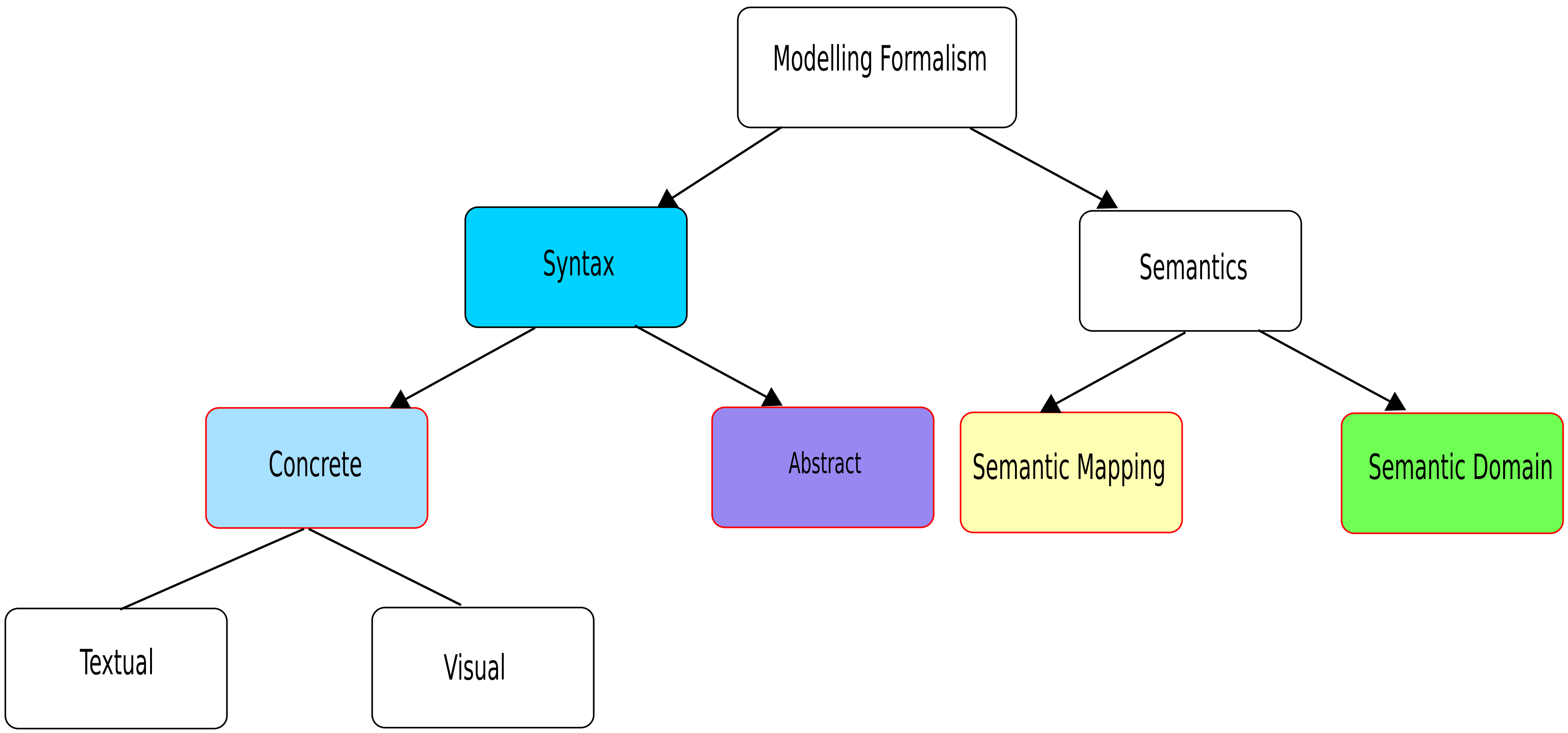
Building DS(V)M Tools Effectively ...

- **development cost** of DS(V)M Tools may be prohibitive !
- we want to effectively (rapidly, correctly, re-usably, ...)
  1. Specify DS(V)L **syntax**:
     - **abstract** $\Rightarrow$ **meta-modelling**
     - **concrete** (textual/visual)
  2. Specify DS(V)L **semantics**:
     transformation
  3. Model (and analyze and execute) model **transformations**:
     $\Rightarrow$ **graph rewriting**

$\Rightarrow$ **model everything**

(in the most appropriate formalism,
at the appropriate level of abstraction)

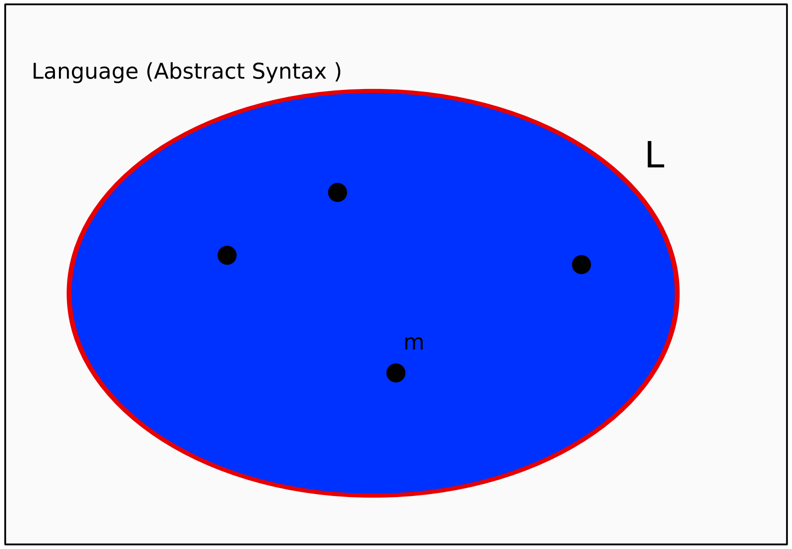
Dissecting a Modelling Language (tool builder's view)



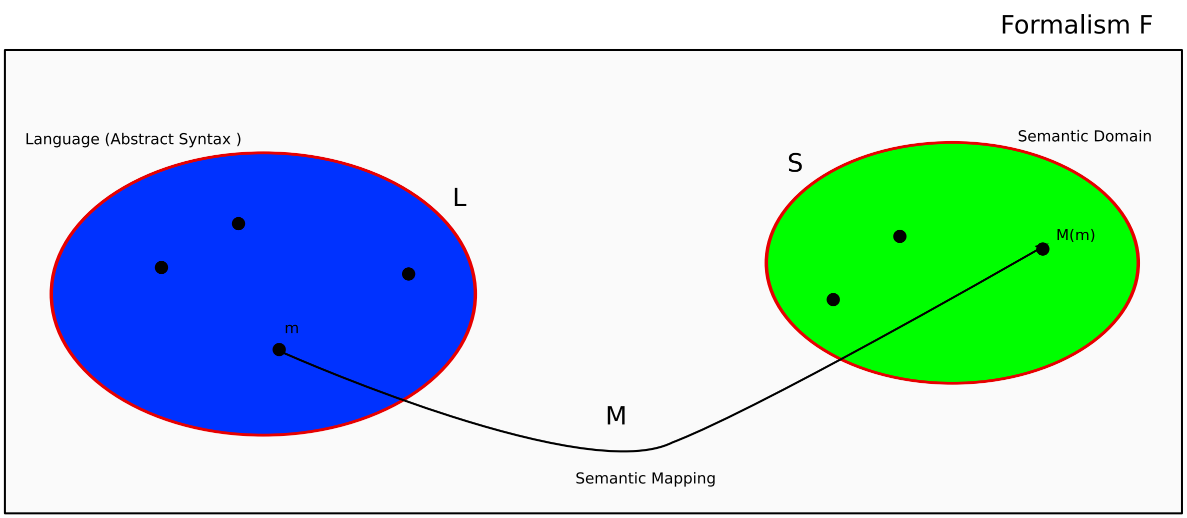
Deciding on terminology

$$\Sigma$$

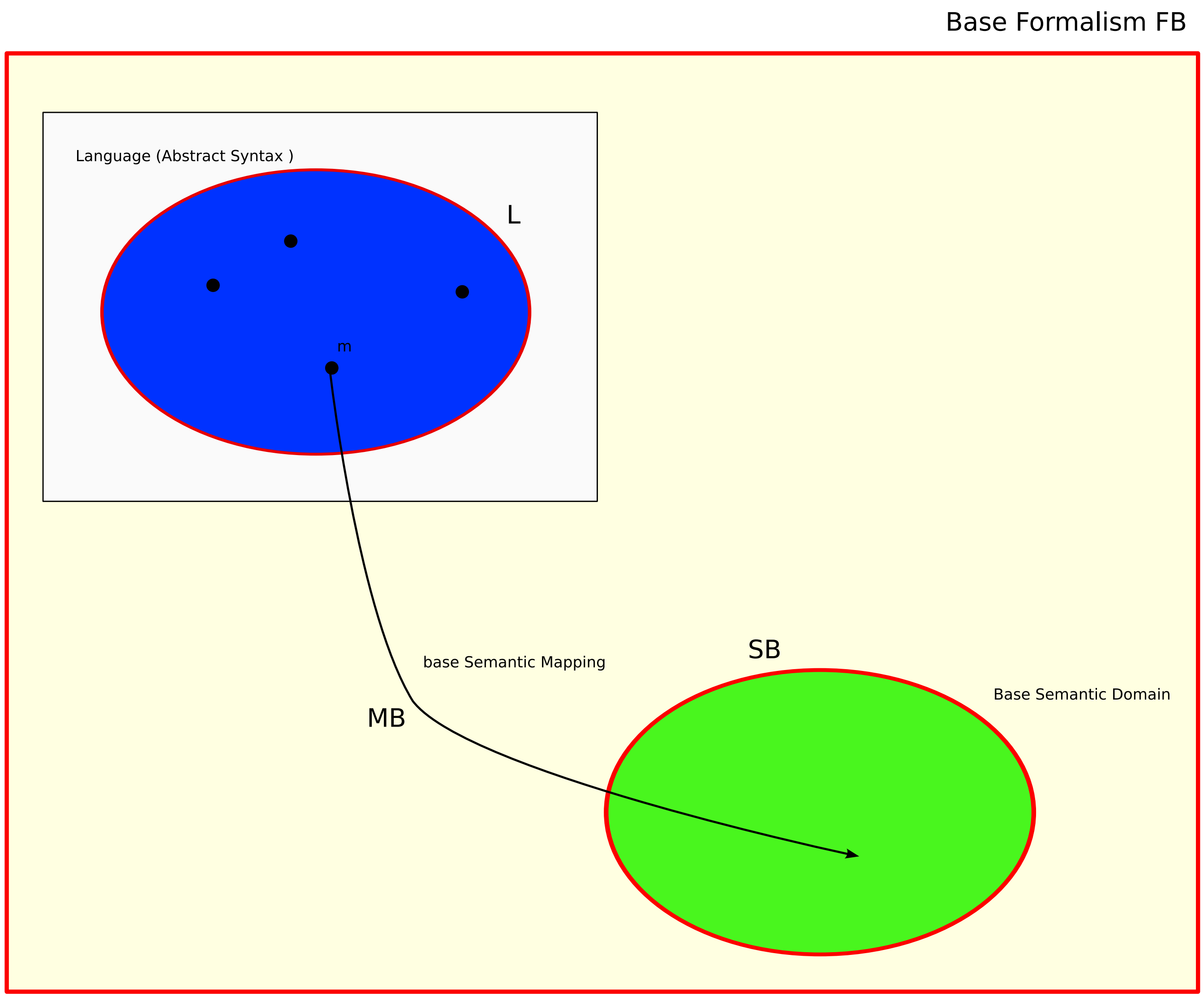
What's in a name ? Language



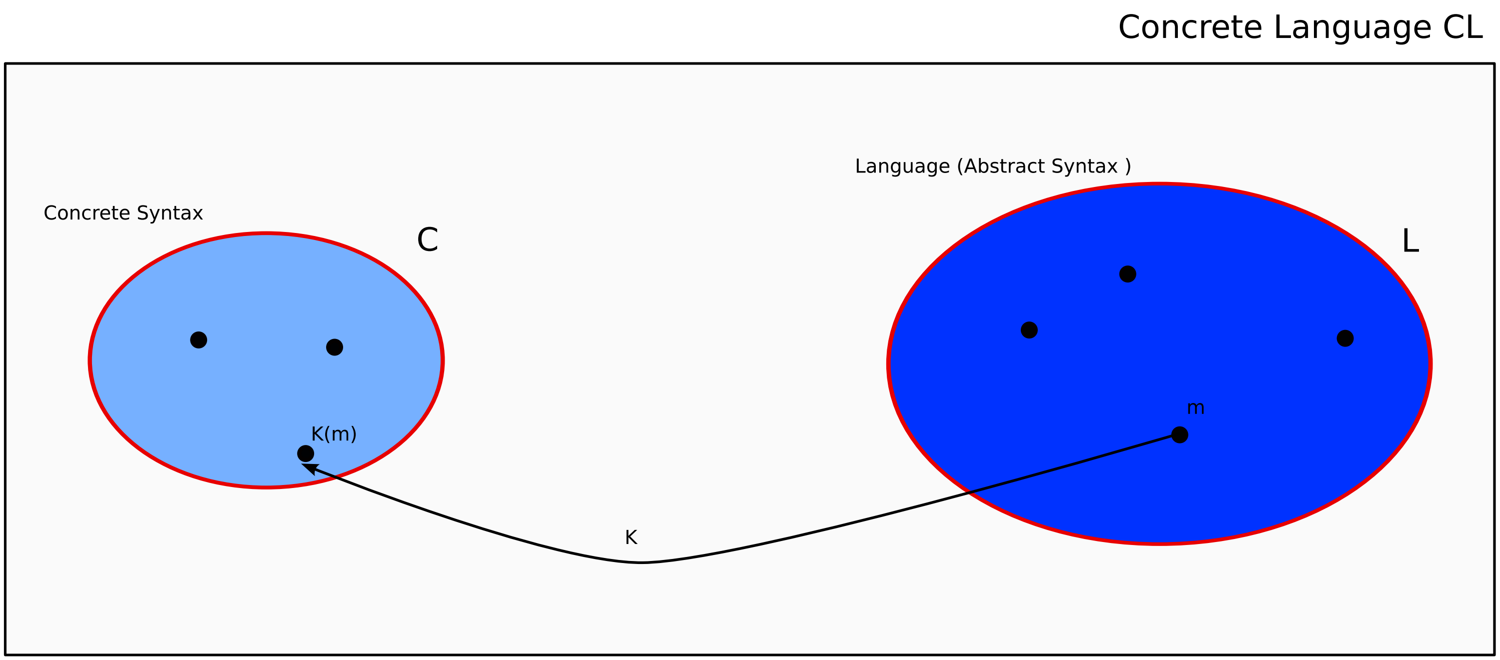
What's in a name ? Formalism



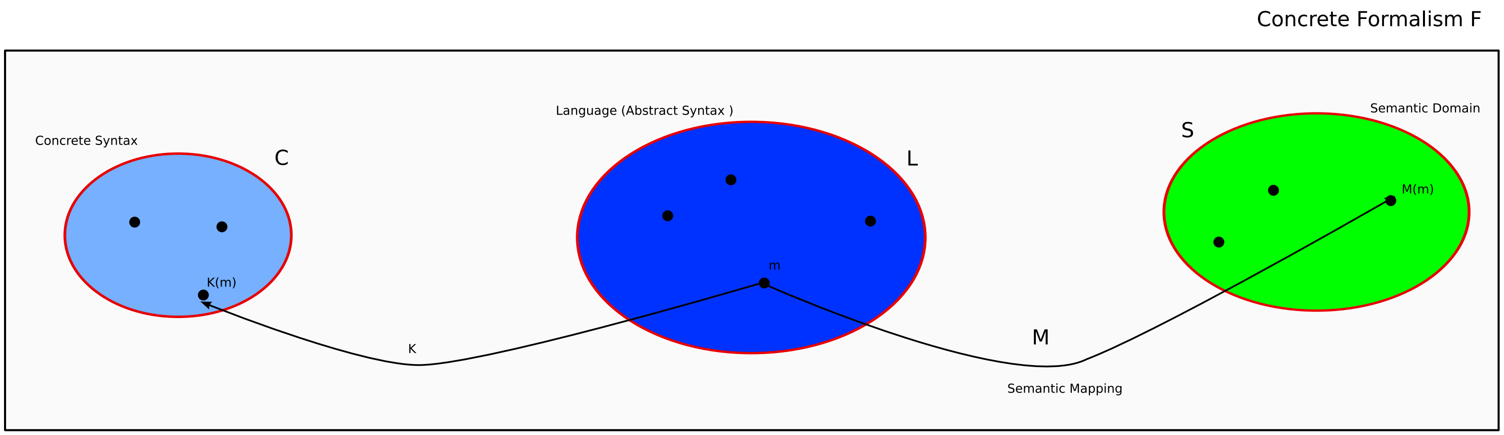
What's in a name ? Base Formalism



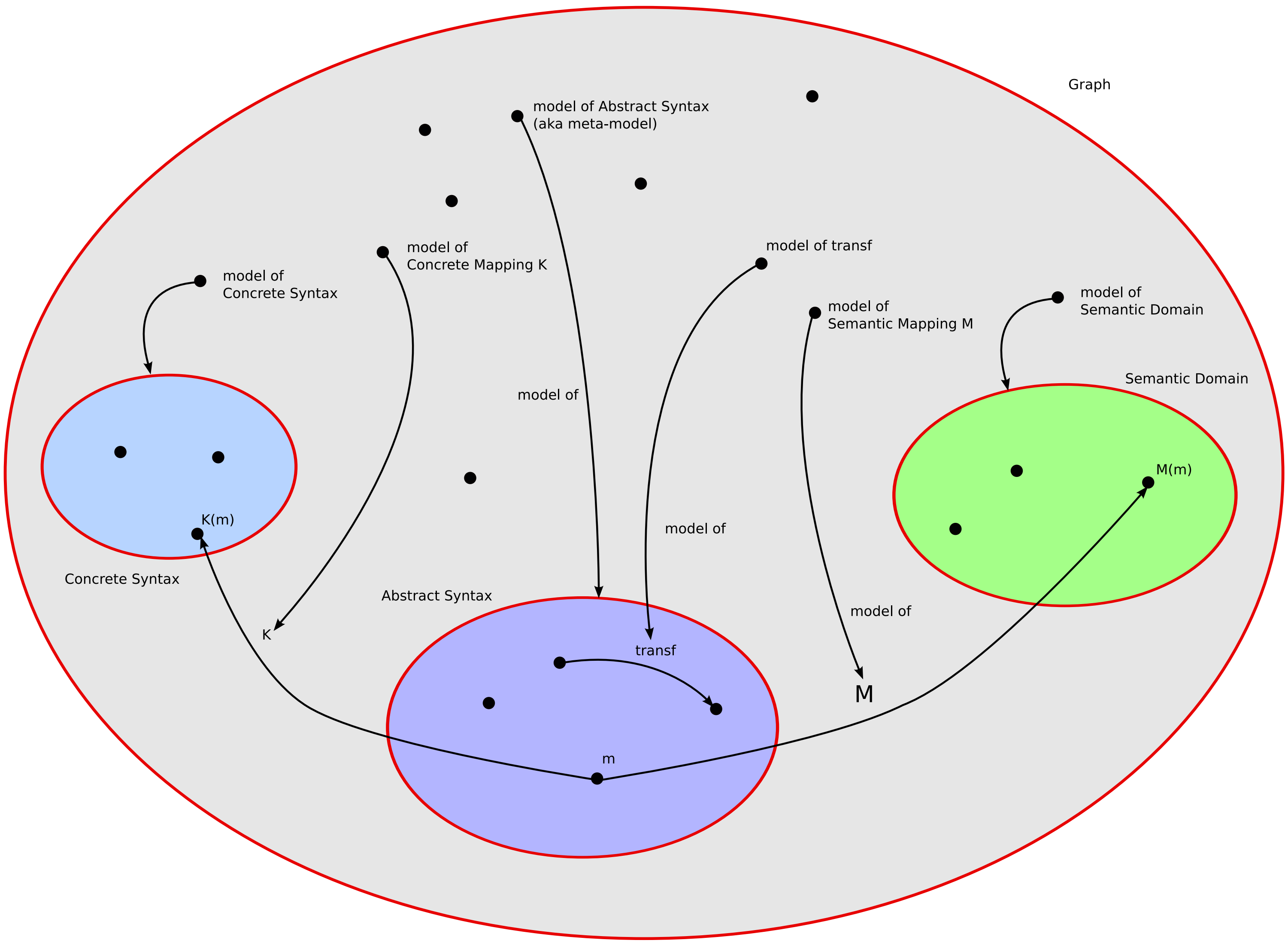
What's in a name ? Concrete Language



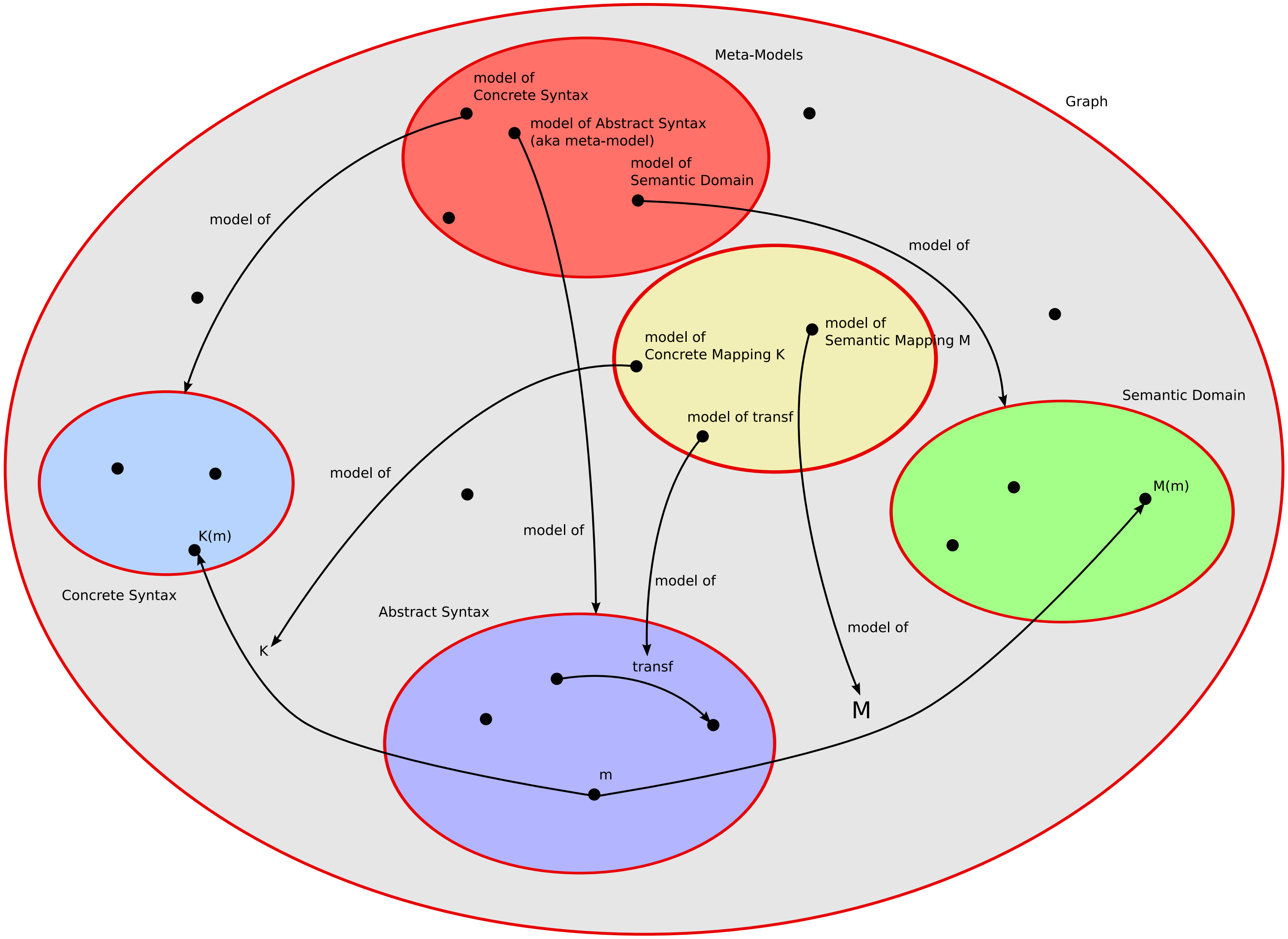
What's in a name ? Concrete Formalism



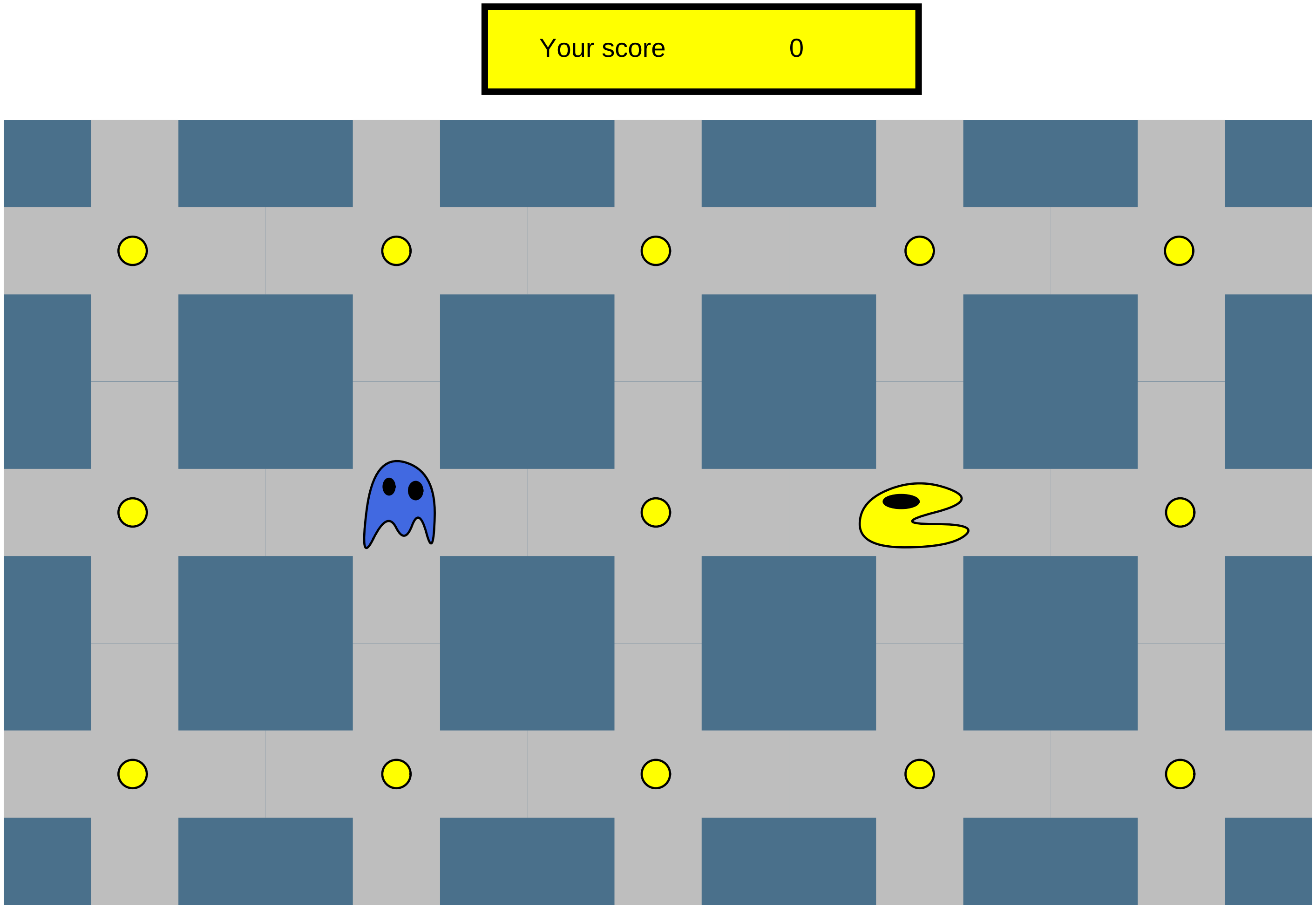
Modelling a Modelling Language/Formalism



Modelling a Modelling Language/Formalism

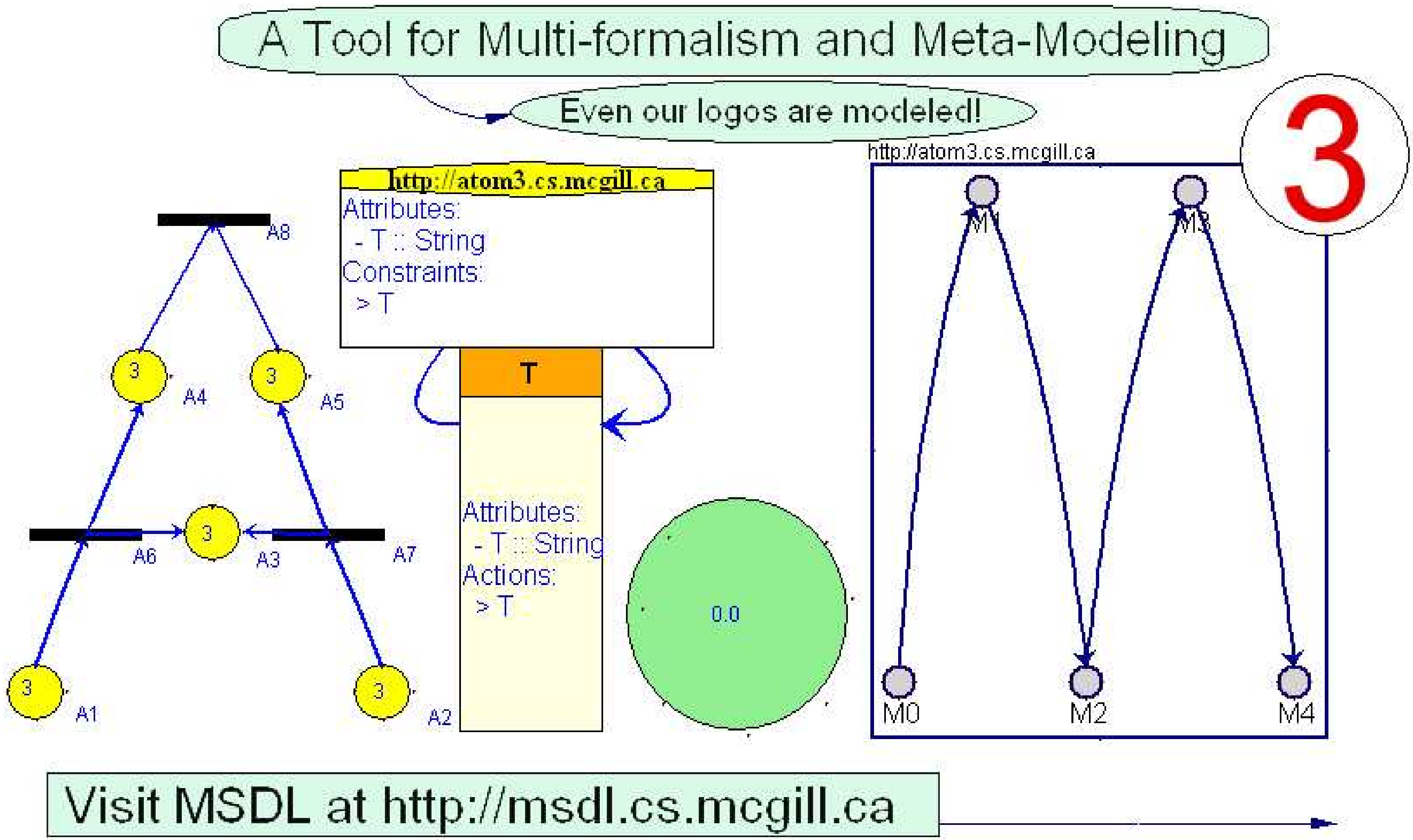


Example: the PacMan Formalism

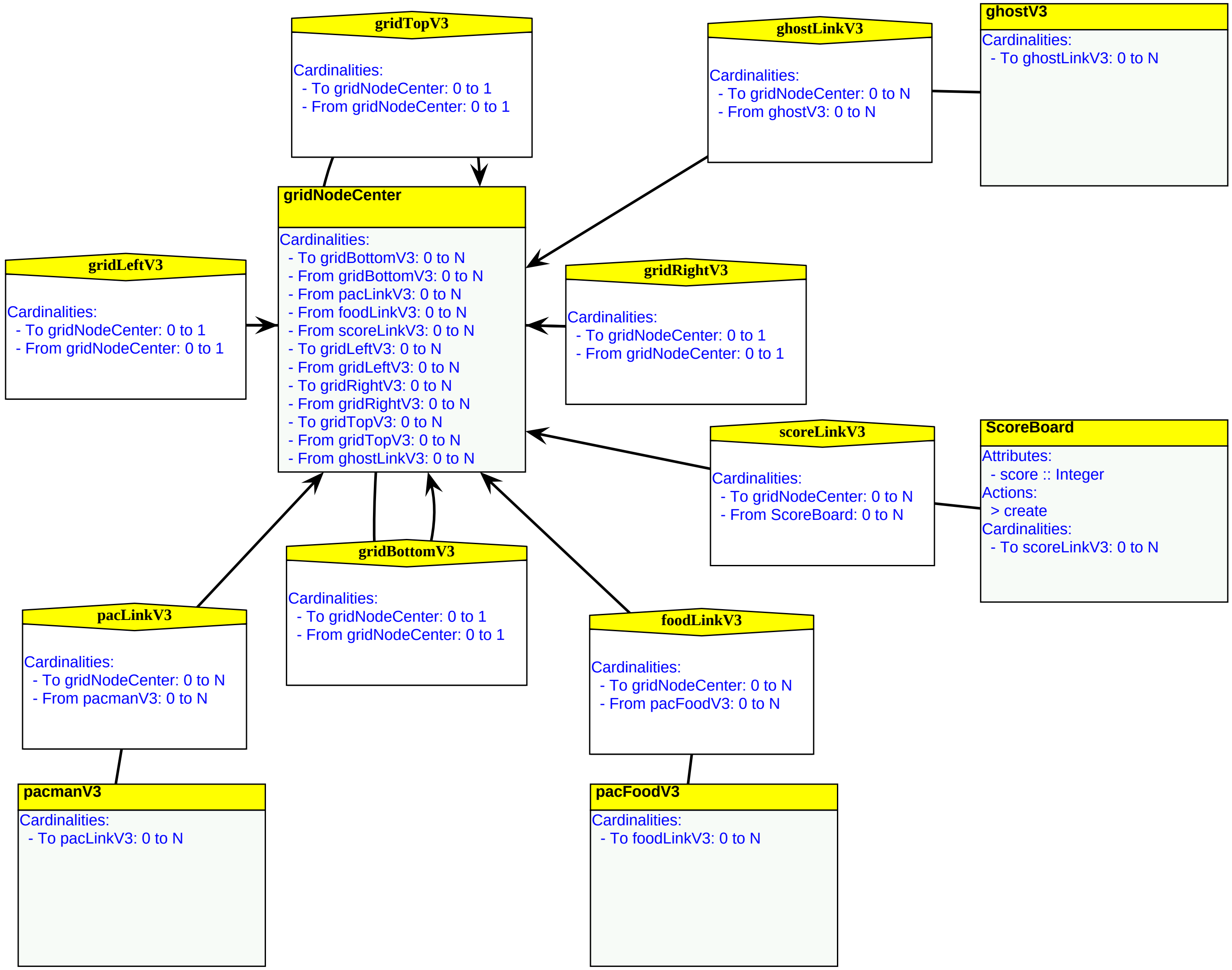


(thanks to Reiko Heckel)

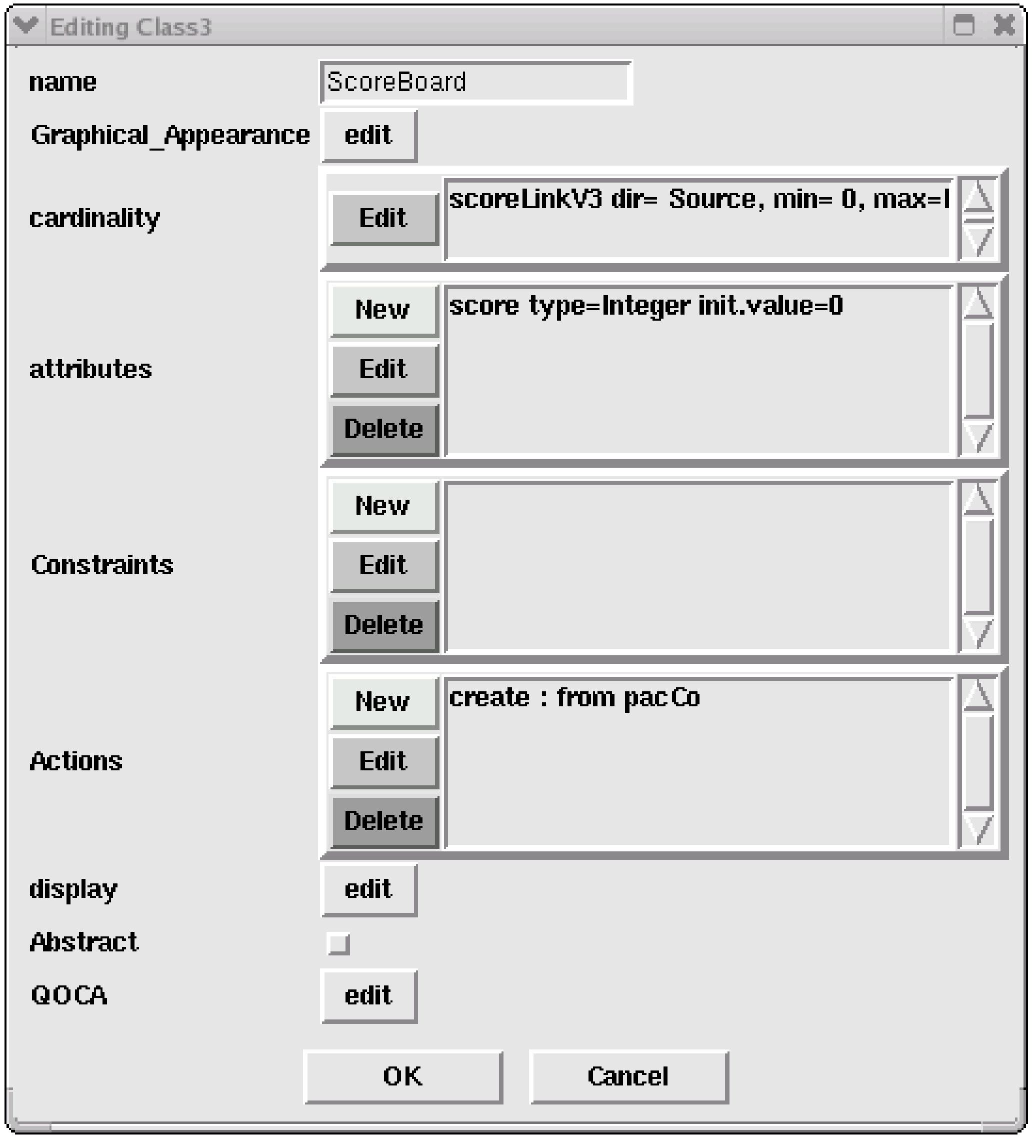
From now on: use AToM^3



Modelling Abstract Syntax (meta-model)



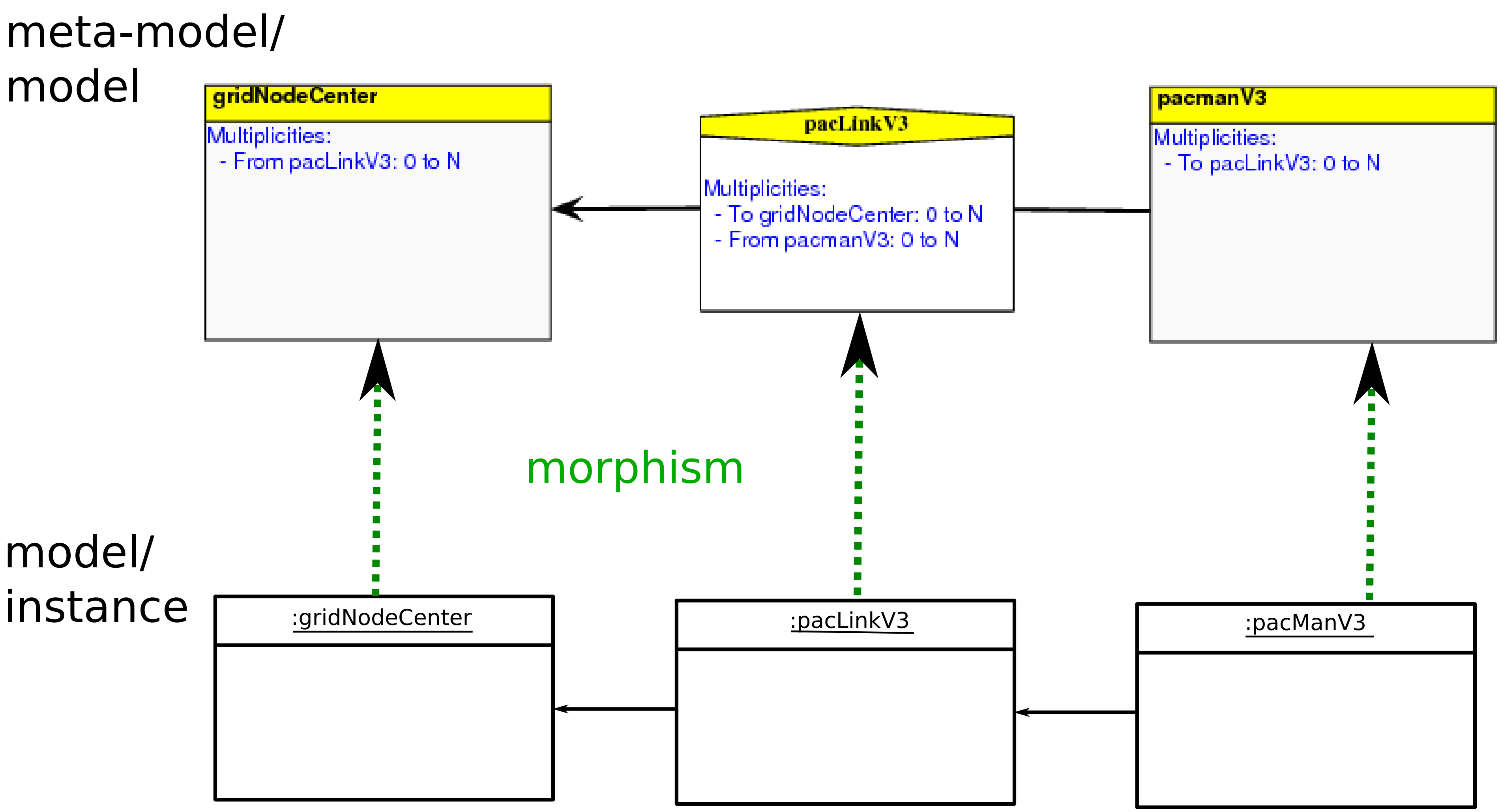
Modelling the Scoreboard Entity



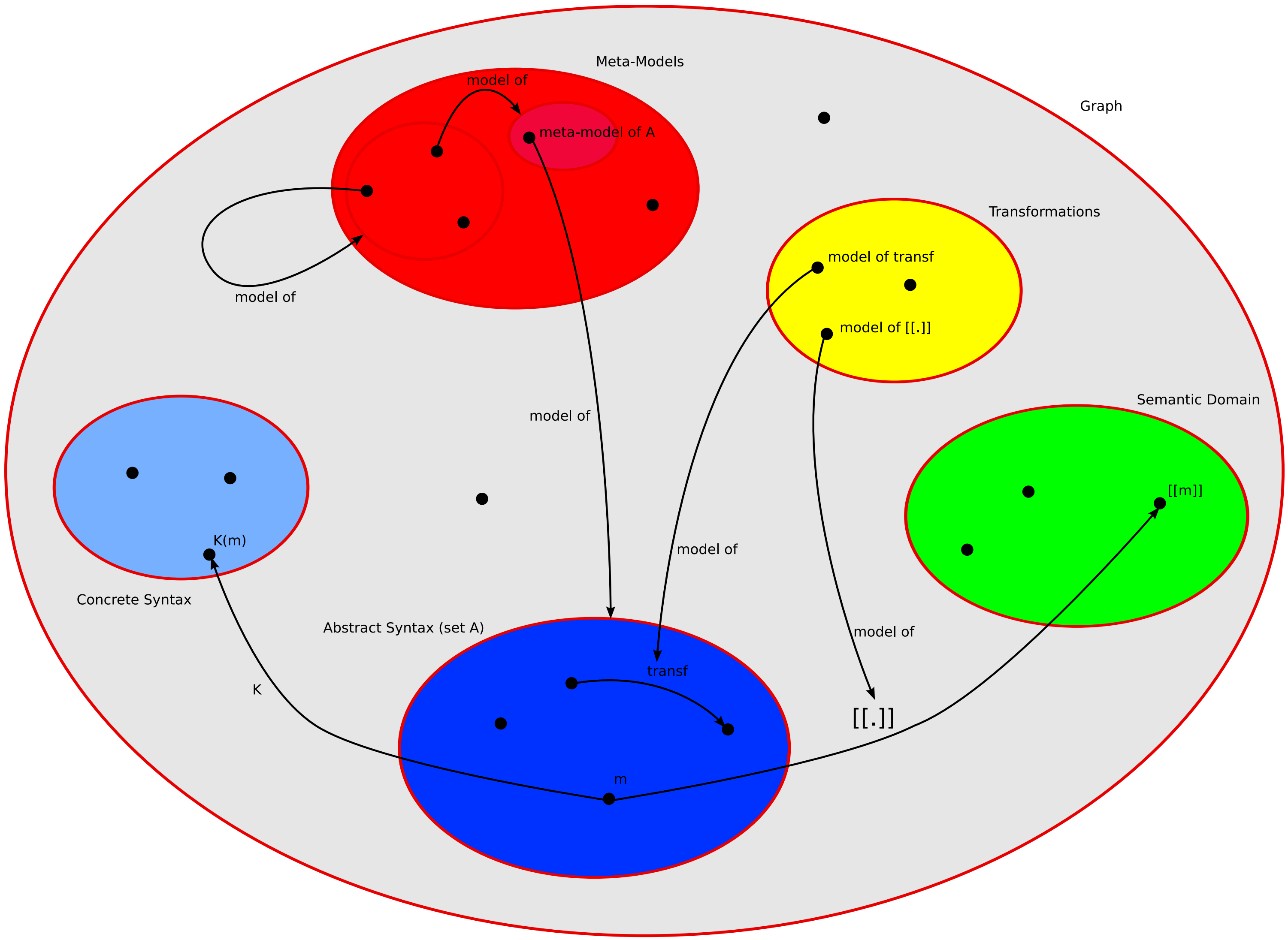
Synthesis of Code from this Design model

```
class ScoreBoard(ASGNode, ATOM3Type):

  def __init__(self, parent = None):
     ASGNode.__init__(self)
     ATOM3Type.__init__(self)
     self.graphClass_ = graph_ScoreBoard
     self.isGraphObjectVisual = True
     self.parent = parent
     self.score=ATOM3Integer(0)
     self.generatedAttributes = {'score': ('ATOM3Integer' ) }
     self.directEditing = [1]

  def clone(self):
     cloneObject = ScoreBoard( self.parent )
     for atr in self.realOrder:
        cloneObject.setAttrValue(atr, self.getAttrValue(atr).clone() )
     ASGNode.cloneActions(self, cloneObject)
     return cloneObject
```

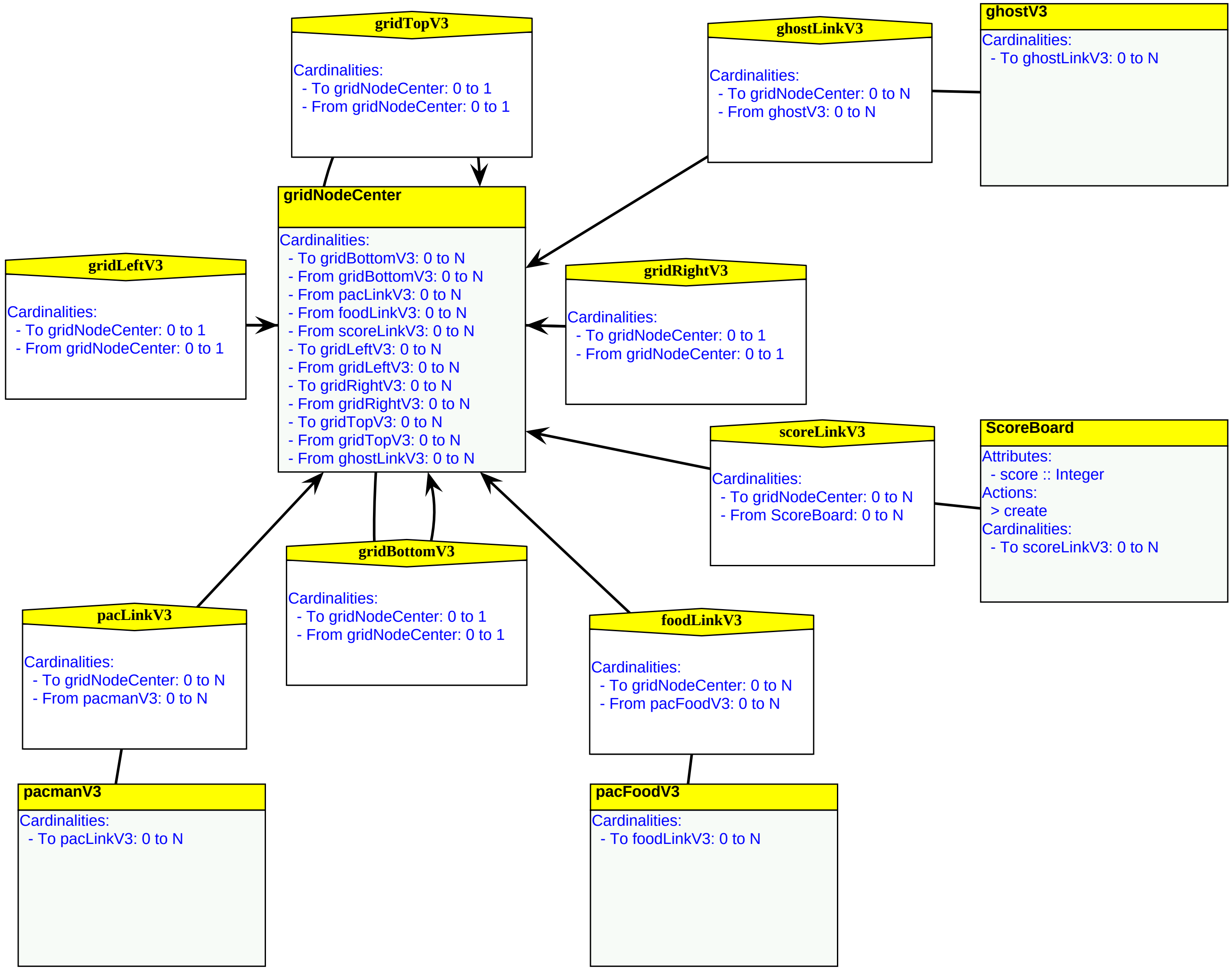
Meta-modelling: model-instance morphism



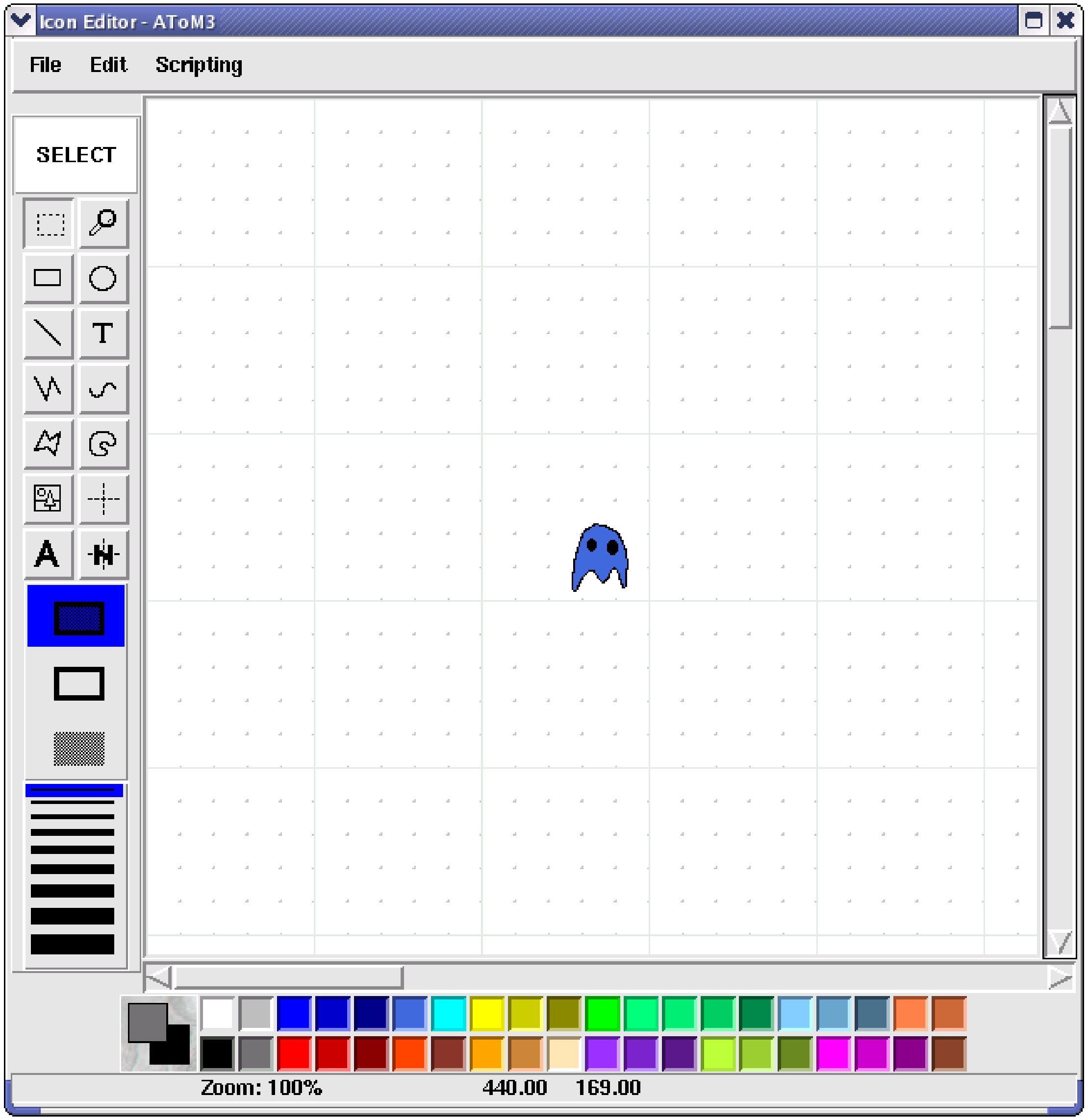
Meta-meta-. . . : Meta-circularity



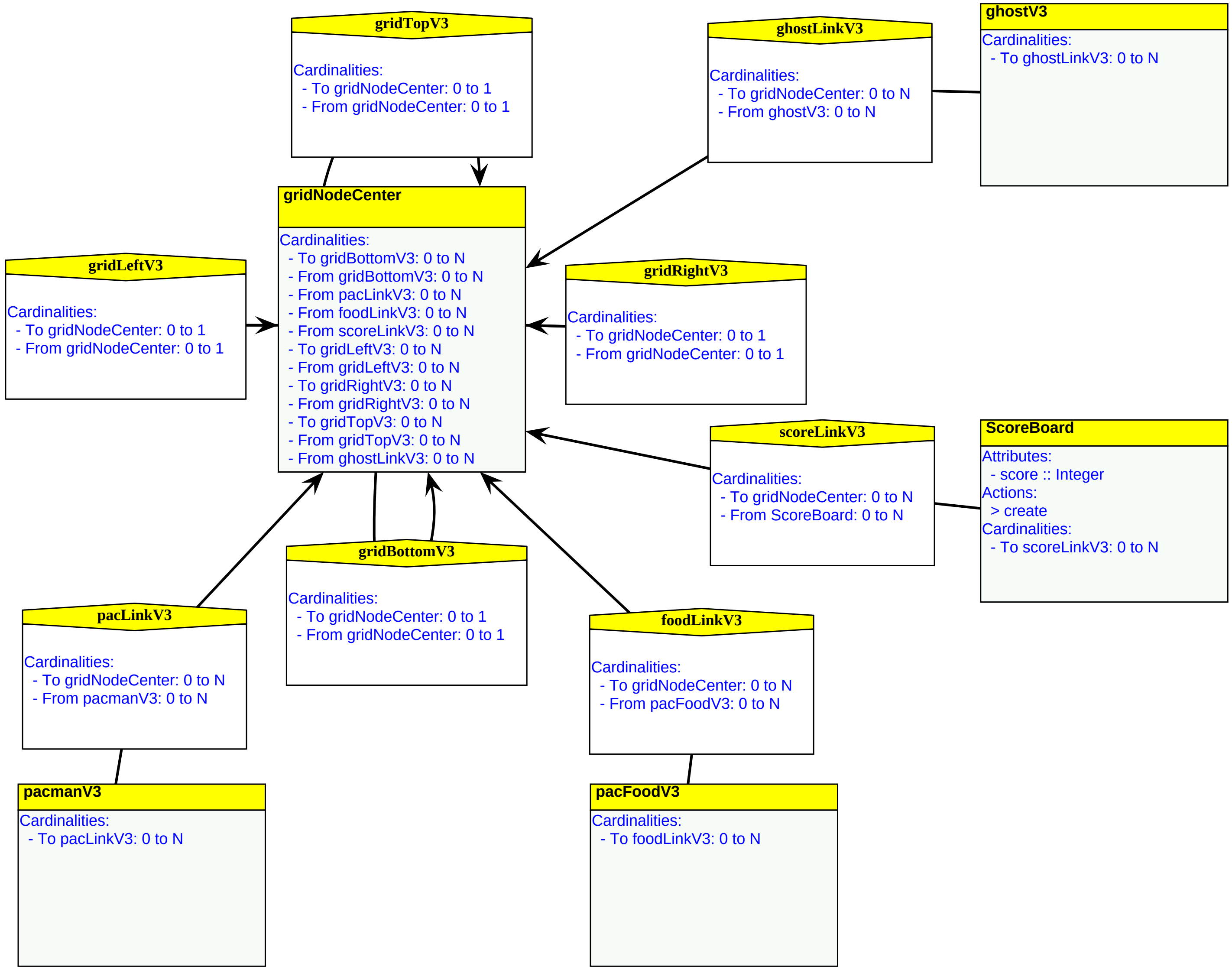
Add Concrete Visual Syntax to Classes



Modelling Ghost Concrete Visual Syntax



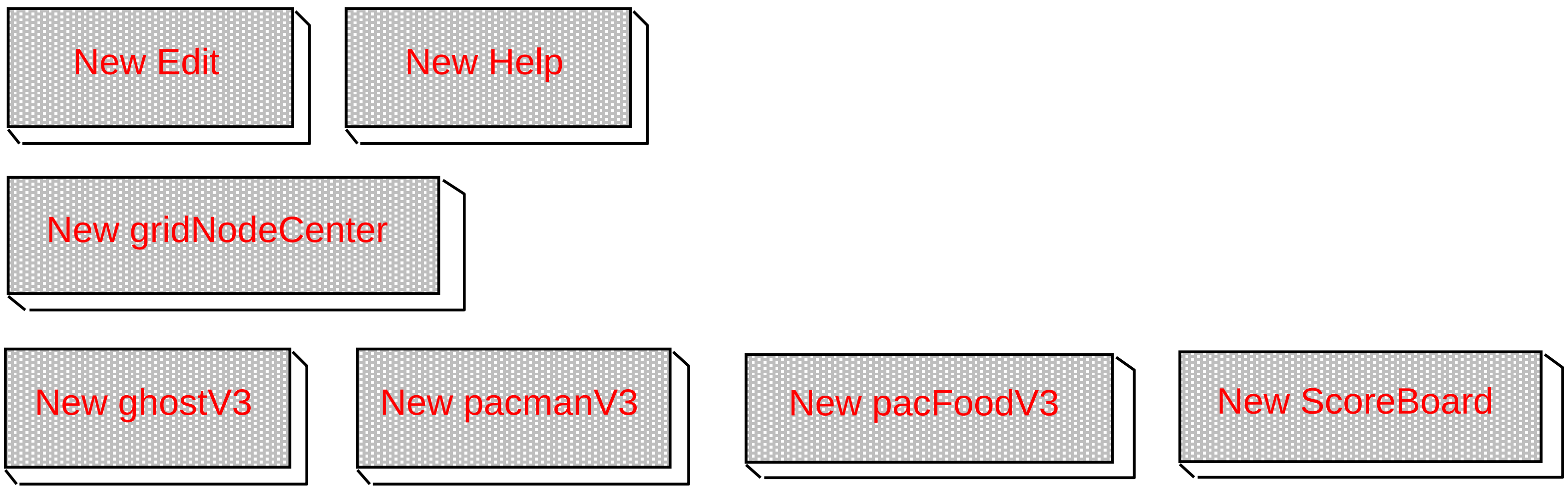
Add Concrete Visual Syntax to Associations



GhostLink Concrete Visual Syntax

```
# Get n1, n2 end-points of the link
n1 = self.in_connections_[0]
n2 = self.out_connections_[0]

# g1 and g2 are the graphEntity visual objects
g0 = self.graphObject_    # the link
g1 = n1.graphObject_      # first end-point
g2 = n2.graphObject_      # second end-poing

# Get the high level constraint helper and solver
from Qoca.atom3constraints.OffsetConstraints import OffsetConstraints
oc = OffsetConstraints(self.parent.qocaSolver)

# The constraints
oc.CenterX((g1, g2, g0))
oc.CenterY((g1, g2, g0))
oc.resolve()
```

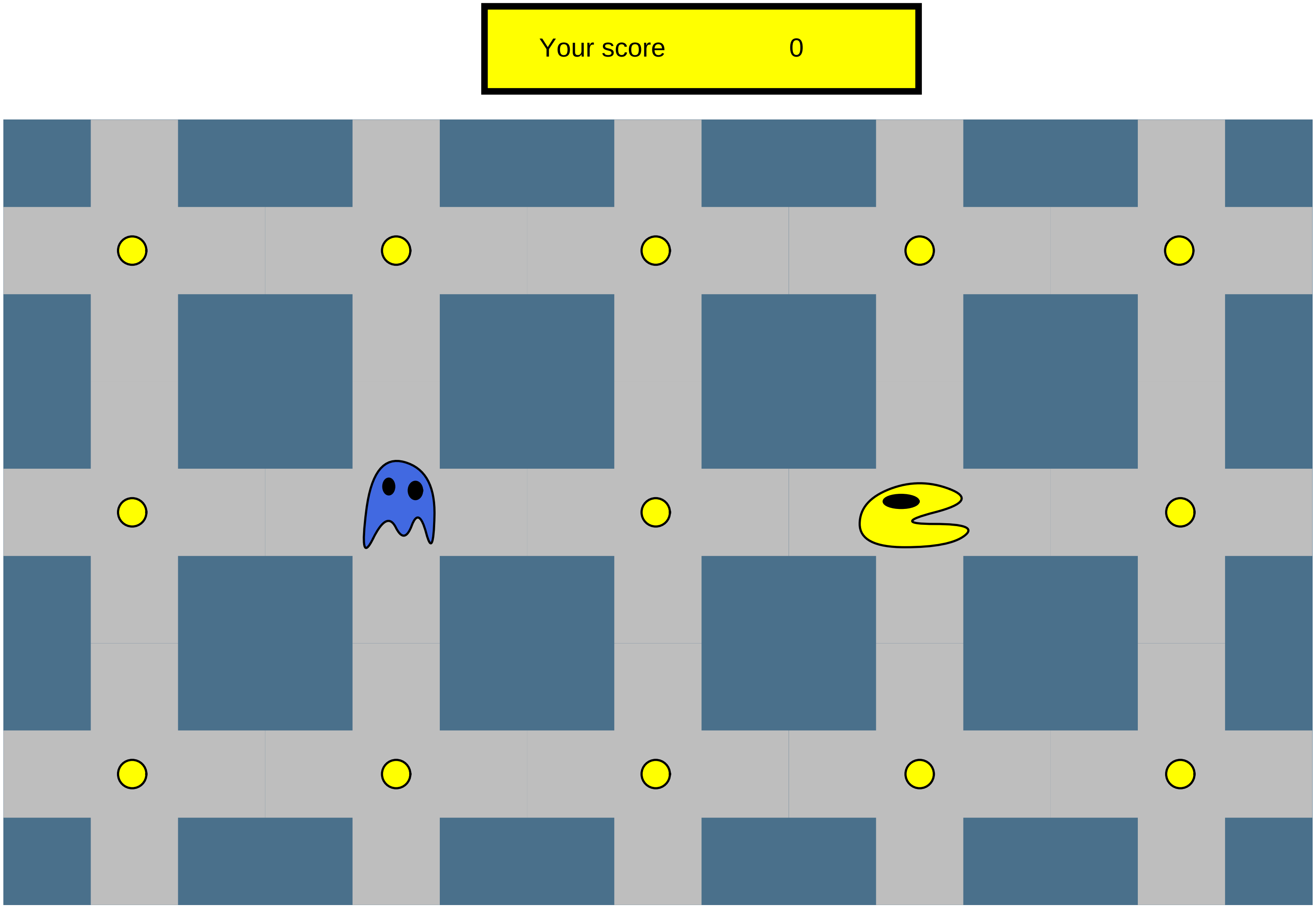
Synthesize + Customize Buttons model



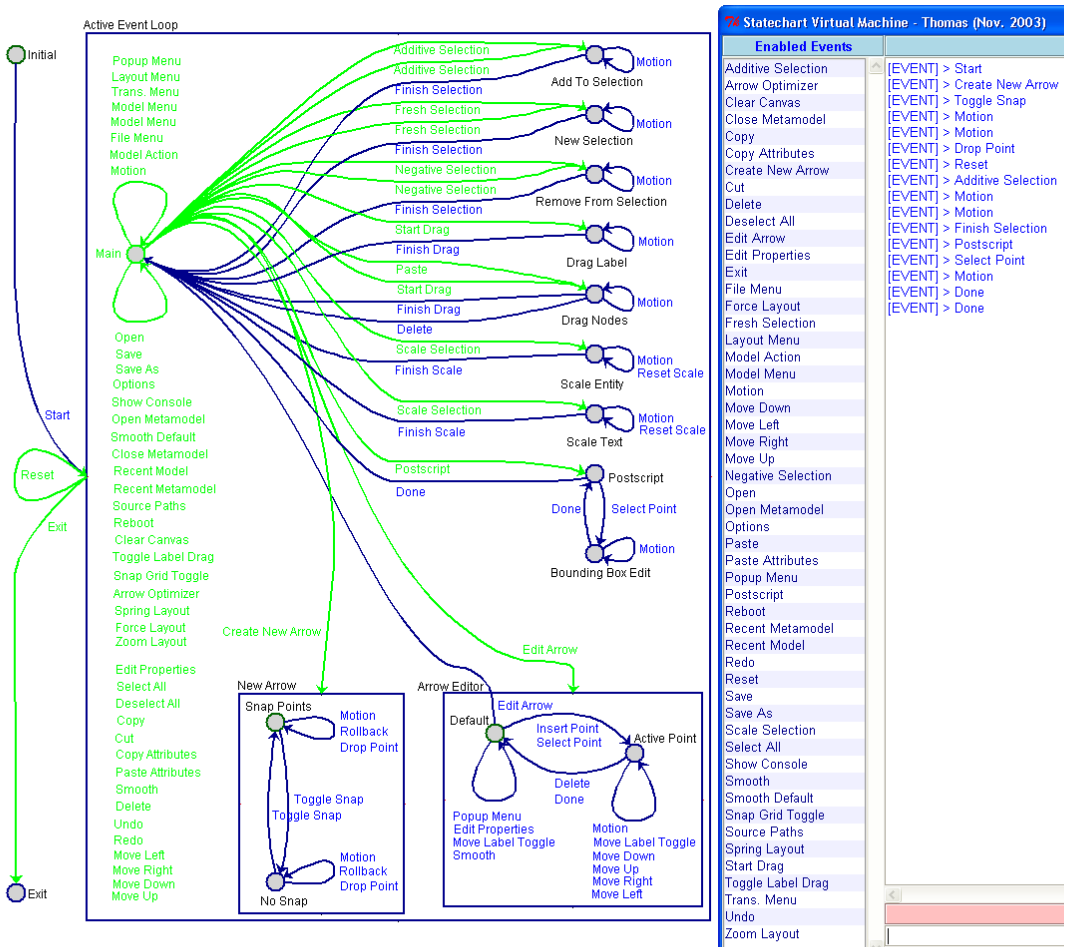
Default generated Buttons code for ghostV3

```
# This method has as parameters:
#   - wherex : X Position in window coordinates where the user clicked.
#   - wherey : Y Position in window coordinates where the user clicked.
newPlace = self.createNewghostV3 (self, wherex, wherey)\n'))
```

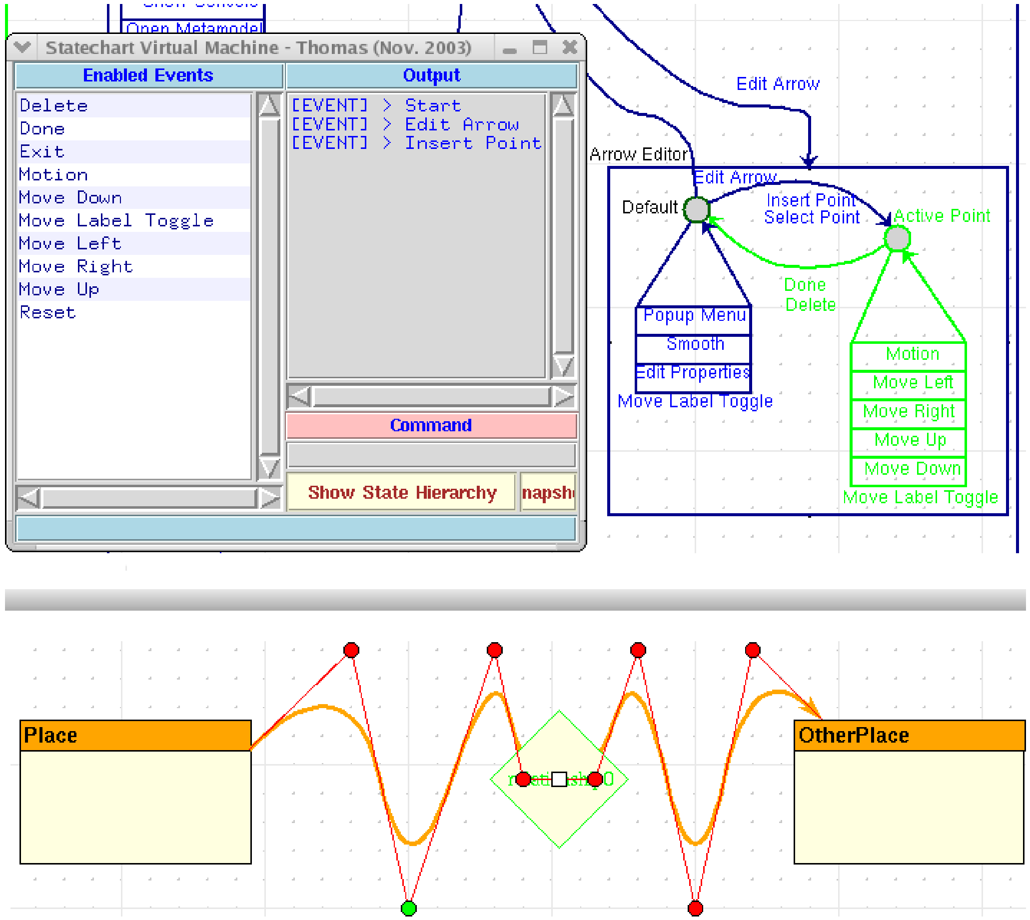
Can now do syntax-directed editing of PacMan models ?



Model the GUI's Reactive Behaviour ! in the Statechart formalism

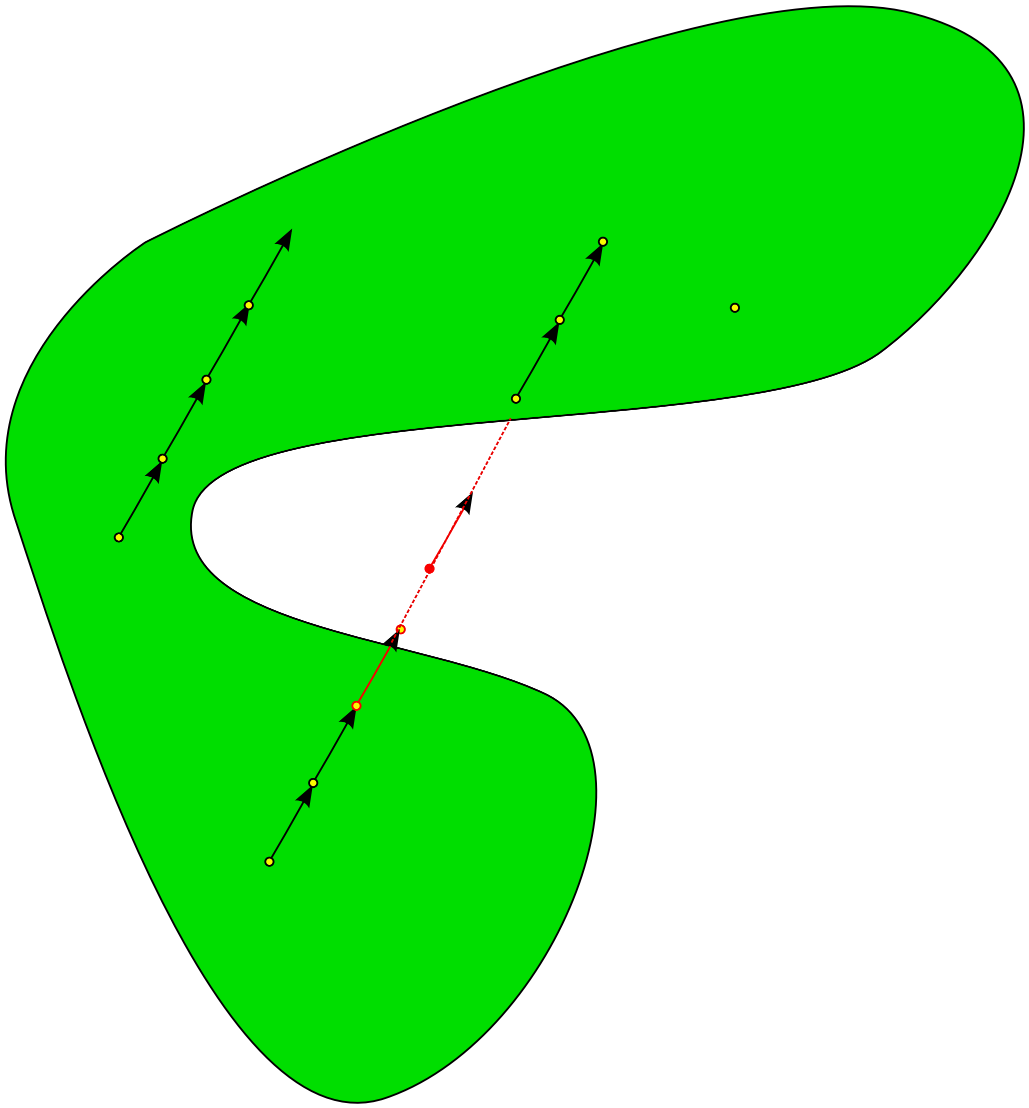


The GUI's reactive behaviour in action



challenge: what is the *optimal* formalism to specify GUI reactive behaviour ?

Syntax Directed Editing (vs. Freehand)



Modellling PacMan Operational Semantics

note: for Denotational Semantics: map for example onto Petri Net

Specifying Model Transformations

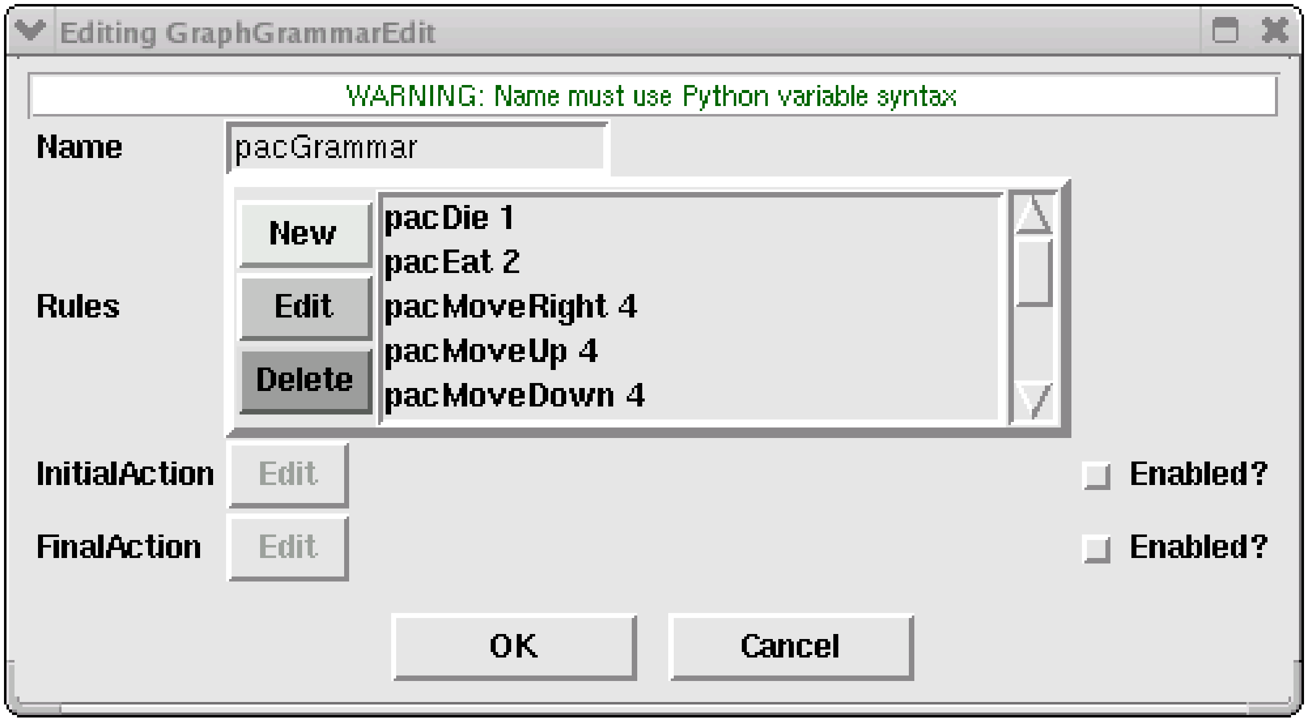
What is the "optimal" formalism ?
Models are often graph-like $\Rightarrow$ natural to express model transformation by means of **graph transformation** models.

Ehrig, H., G. Engels, H.-J. Kreowski, and G. Rozenberg.
Handbook of graph grammars and computing by graph transformation.
1999. World Scientific.

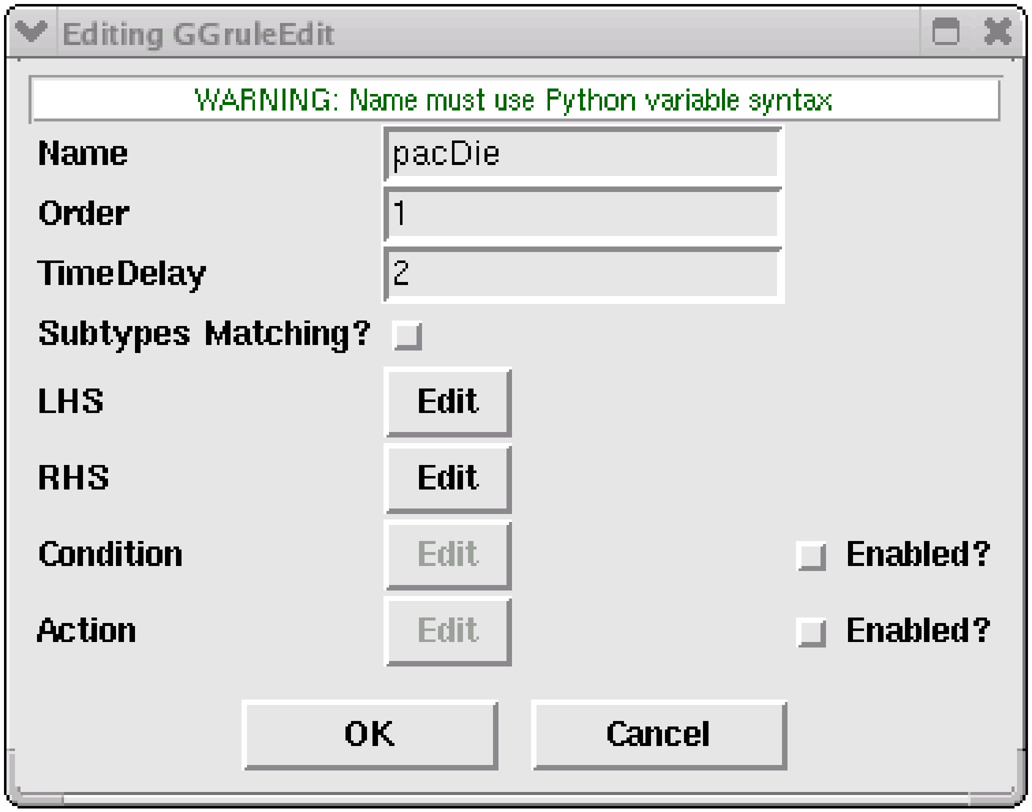
Tools:

GME/GReAT, PROGRES, AGG, AToM^3, Fujaba, GROOVE, ...
First two (and Fujaba) used in large industrial applications.

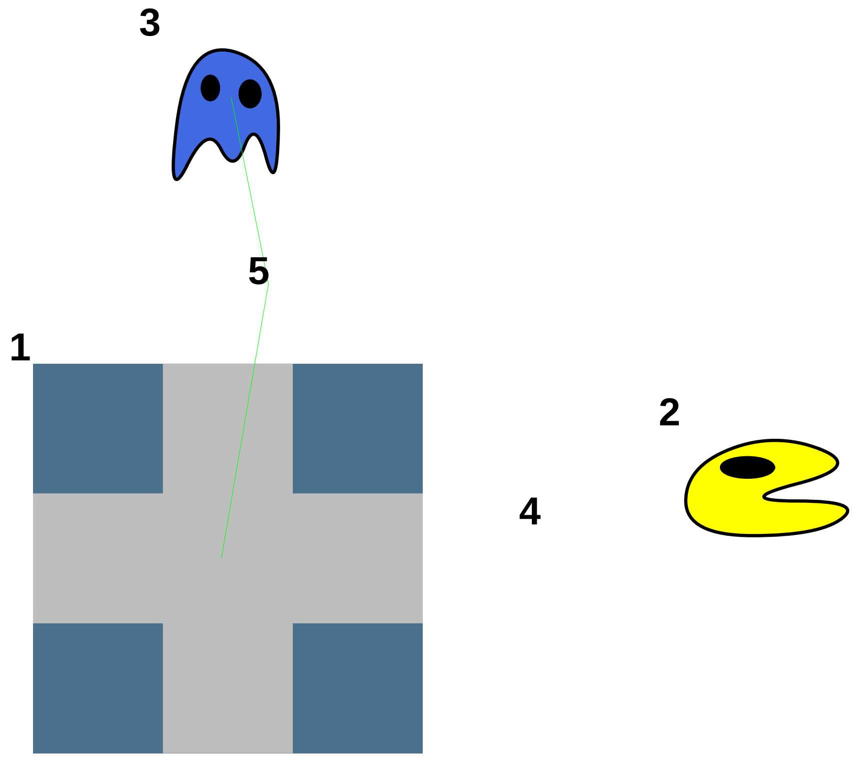
Model Operational Semantics using GG



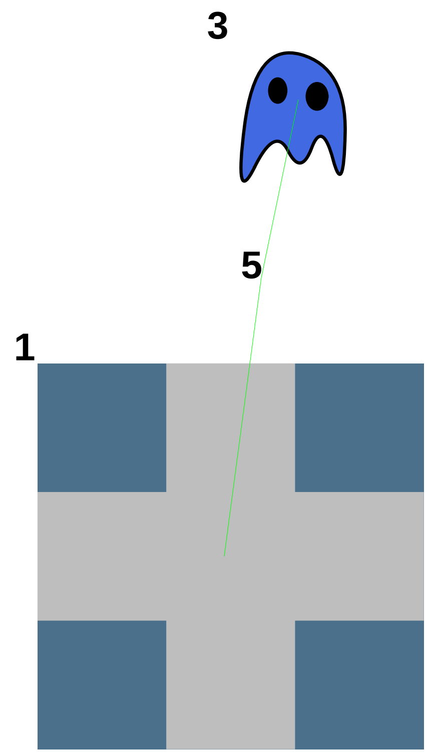
PacMan Die rule



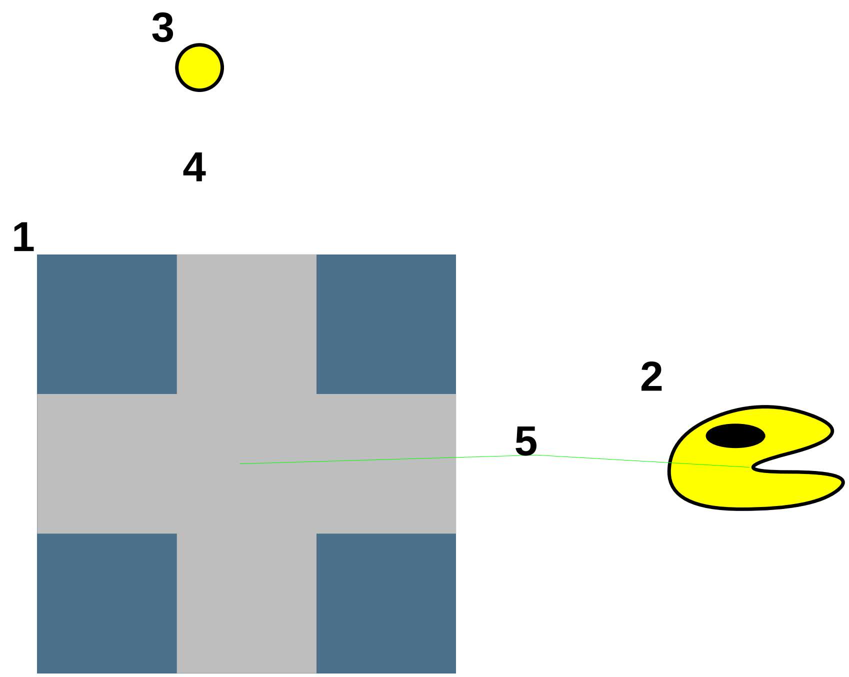
PacMan Die rule LHS



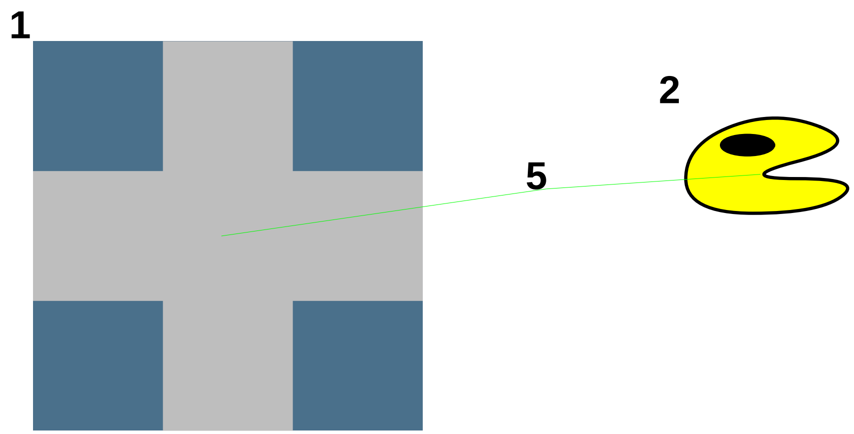
PacMan Die rule RHS



PacMan Eat rule LHS

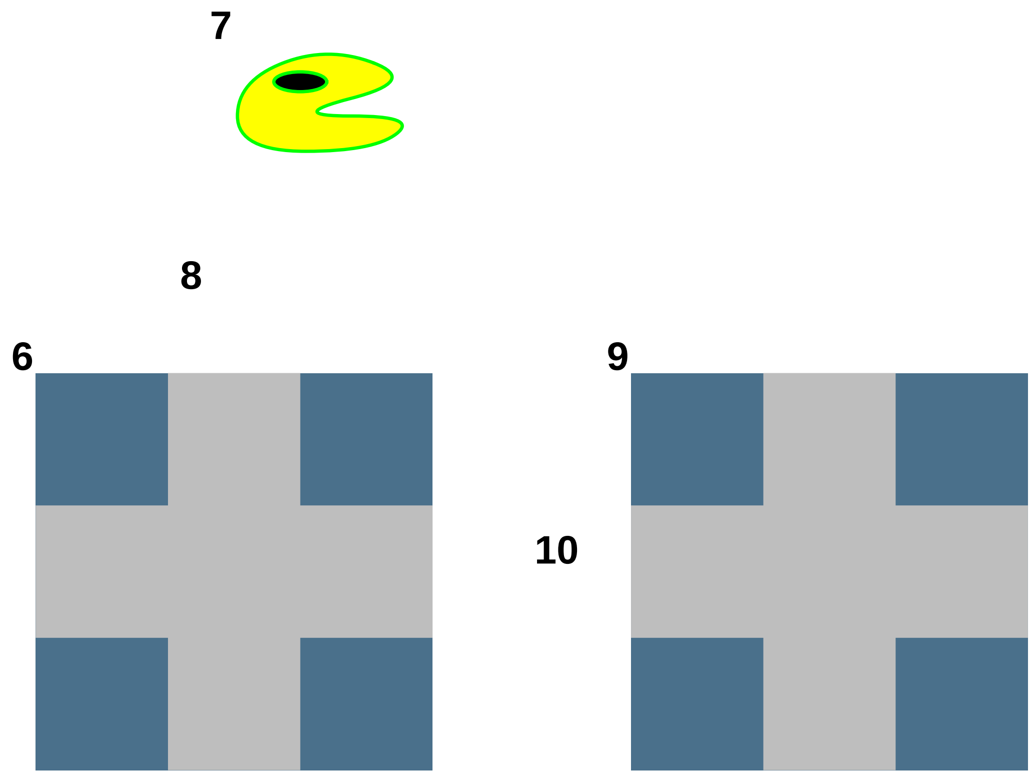


PacMan Eat rule RHS

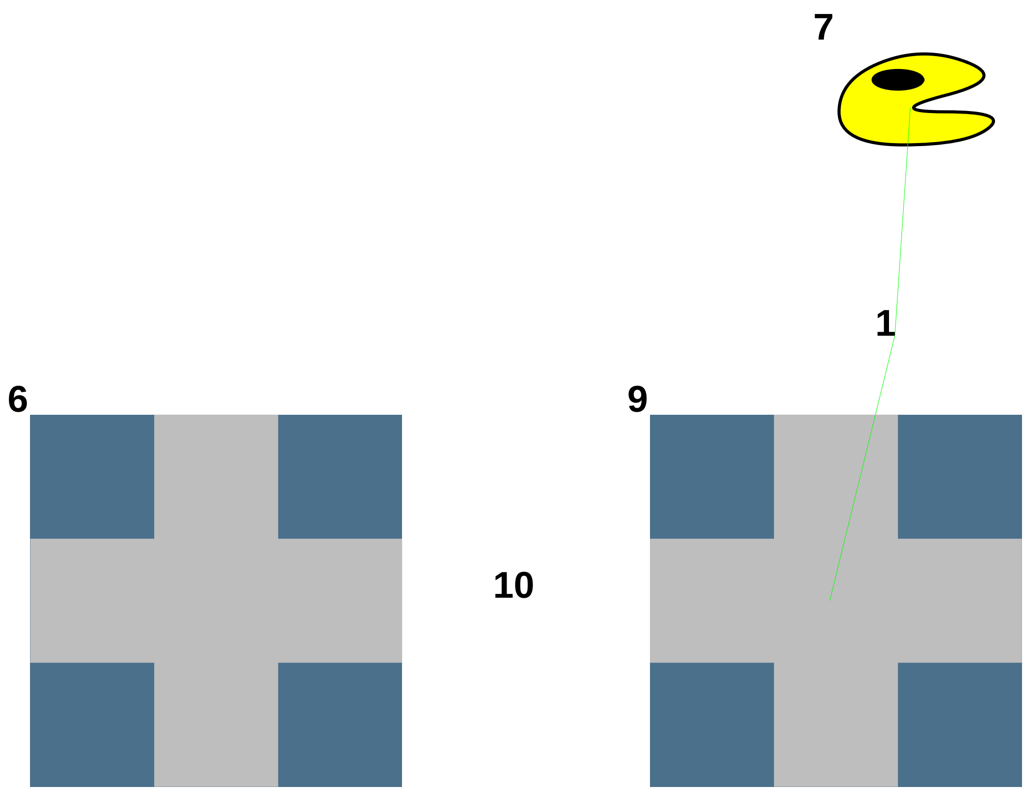



```
scoreBoard = None
scoreBoards = atom3i.ASGroot.listNodes['ScoreBoard']
if (not scoreBoards):
  return
else:
  scoreBoard = scoreBoards[0]
  scoreVal = scoreBoard.score.getValue()
  scoreBoard.score.setValue(scoreVal+1)
  scoreBoard.graphObject_.ModifyAttribute('score',scoreVal+1)
```

PacMan Move rule LHS



PacMan Move rule RHS



Specifying/Executing Trsf. with GGs

(+) Models are often Graph-like

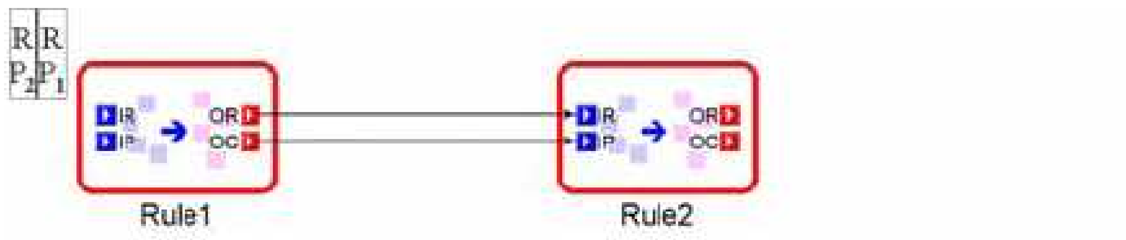
(+) Visual specification (documentation)
For insight/debugging: execution + visual display
For performance: execution on data structures in memory

(+) Little or no programming knowledge required (allows understanding/modification by domain-experts)

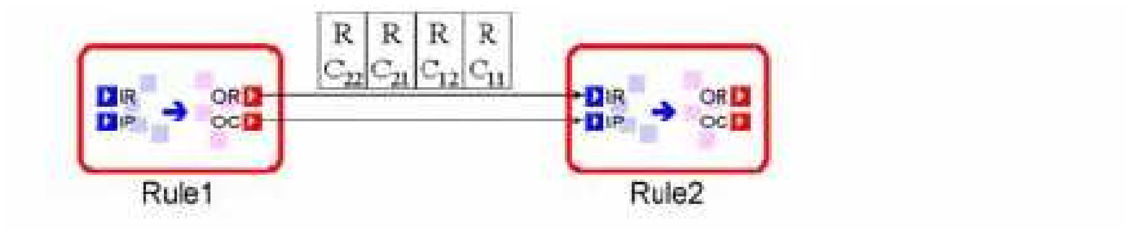
(-) Does it scale up ?
Yes, need to use modular GGs (*e.g.*, GReAT, PROGRES)

(-) Performance is bad ! (due to sub-graph matching)
But sometimes no alternative
– model transformation for graph-like models
– don't want to code matching yourself
But give (domain-specific) hints to kernel (or compile)
But use as specification for manual implementation
– executable specification = reference implementation
– automatic generation of unit tests
(including expected correct result)

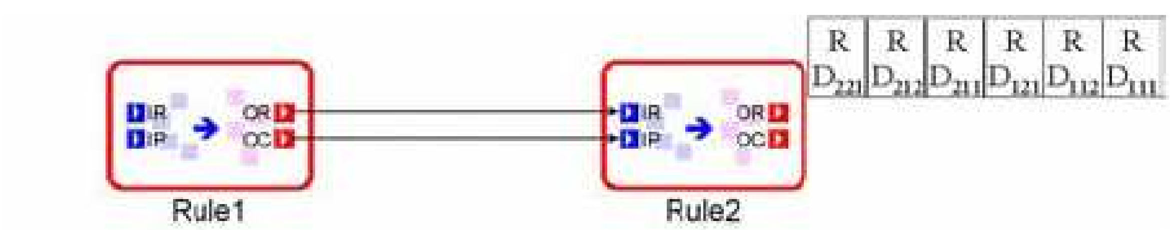
Modular Graph Rewriting: GReAT Control Structures: Sequence



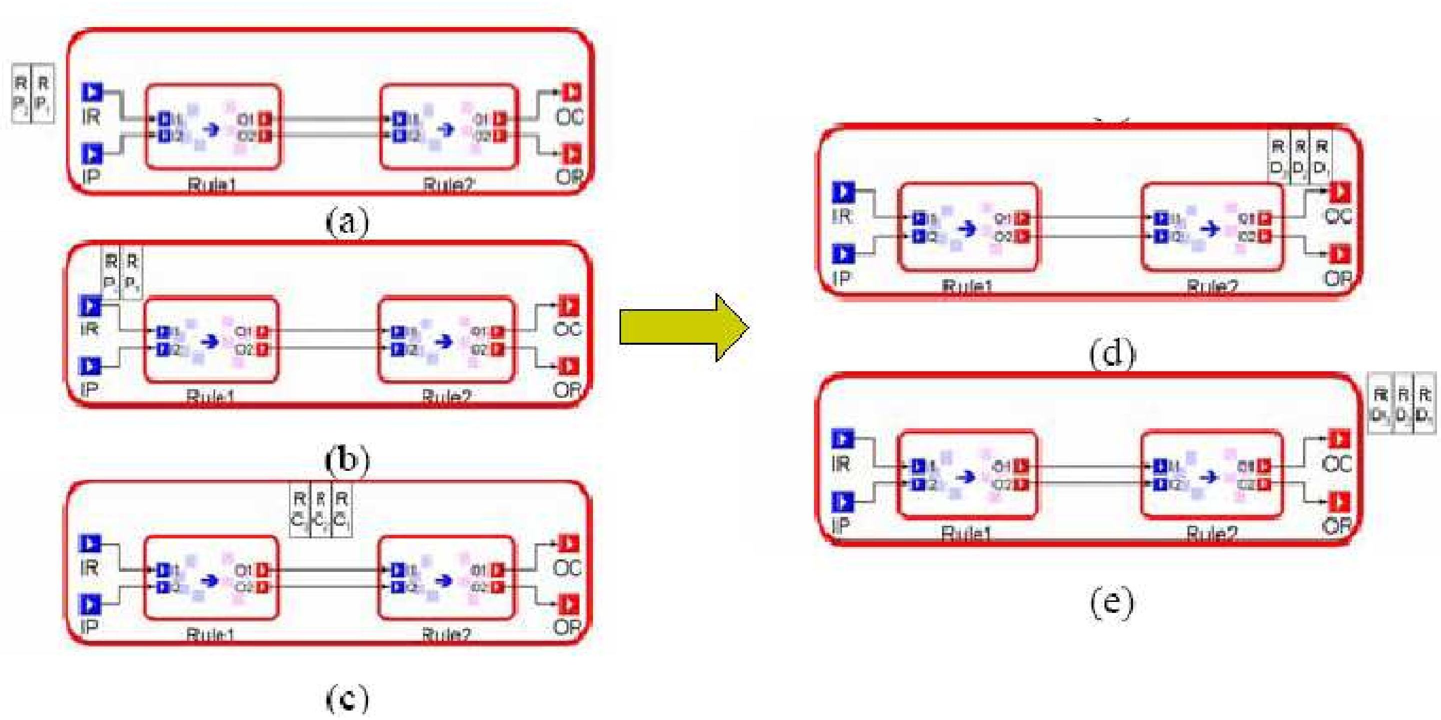
(a)



(b)



Modular Graph Rewriting: GReAT Control Structures: Nesting



Model Based Development: some Open Problems

1. deal with concrete syntax (textual, visual) in a unified manner
2. deal with legacy models (code)
3. trace-ability (backward links, use TGGs ?)
4. (meta-) model evolution
5. multi-formalism modelling
6. multi-abstraction modelling
7. multi-view modelling, (semantic) consistency
8. model refinement (cfr. GUI example)
9. automated testing (of models)
10. modularize transformations, mix trsf. formalisms, graft on existing formalism

11. transformation models are first-class models $\Rightarrow$ higher-order transformation
12. quantify “accidental complexity”, leads to choice of “most appropriate formalism”
13. mega-modelling

Conclusions

1. Through anecdotal evidence, demonstrated the usefulness of **(Computer Automated) (Domain-Specific) Multi-Paradigm Modelling**.
2. Demonstrated **feasibility** of rapidly and re-usably building Domain-Specific Visual Modelling, Analysis, Simulation tools using **meta-modelling** and **graph rewriting**.
3. Many problems have been solved, but . . .
4. Still many **open research problems** (good news for researchers, challenge for industry) !

model **everything** !