Multi-Paradigm Modelling,
and the quest for tool support

Hans Vangheluwe

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

1. **Model** Everything !
2. Multi-Paradigm Modelling by example
3. Building CAMPaM tools effectively
4. Challenges
5. Conclusions
Jean Bézivin

Everything is a model!

Jean-Marie Favre

Nothing is a model!

Hans Vangheluwe

Model everything!

Hans Vangheluwe
hv@cs.mcgill.ca

model

system

model

system

sus

CAMPaM
Modelling Variety of Complex Systems . . .

Need to be (Multi-Paradigm) modelled

• at most appropriate level of abstraction
• in most appropriate formalism(s)
• with transformations as first-class models
Available Information, Questions to be Answered, ... ⇒ choice of Abstraction Level/Formalism
Need Multiple Formalisms: Power Window
The Model Couples different Formalisms

Multiple (consistent !) Views (in ≠ Formalisms)
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!

• recent work (submitted to FASE): automatically generate denotational from operational semantics definition
Modelling Traffic’s (un-timed) semantics in terms of Petri Nets

- need a (meta-)model of Traffic
- need a (meta-)model of Petri Nets
- 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 \times [\text{turn1\_CAP}] + 1.0 \times [\text{turn1}] = 1.0

1.0 \times [\text{cars}] + 1.0 \times [\text{bot\_W2E}] + 1.0 \times [\text{turn1}] + 1.0 \times [\text{to\_N\_or\_W}] + 1.0 \times [\text{turn2}] + 1.0 \times [\text{bot\_N2S}] = 2.0

1.0 \times [\text{top\_CAP}] + 1.0 \times [\text{to\_N\_or\_W}] = 1.0

1.0 \times [\text{turn2\_CAP}] + 1.0 \times [\text{turn2}] = 1.0

1.0 \times [\text{bot\_CAP}] + 1.0 \times [\text{bot\_W2E}] + 1.0 \times [\text{bot\_N2S}] = 1.0
The Big Picture: Transform Everything!

- **Timed Traffic**
  - describe semantics by mapping onto
  - simulate

- **Traffic (un-timed)**
  - describe semantics by mapping onto
  - simulate

- **Timed Transition Petri Nets**
  - analyze: reachability, coverability, ...
  - compute all possible behaviours
  - simulate

- **Place-Transition Petri Nets**
  - simulate

- **Coverability Graph**
  - analyze

- **TINA**
  - simulate

- **pythonDEVS**
  - simulate

- **DEVS**
  - simulate

- **DEVSJava**
  - simulate

- **Timed Transition Petri Nets**
  - describe semantics by mapping onto
  - simulate

- **DEVS**
  - simulate

- **PythonDEVS**
  - simulate

- **Coverability Graph**
  - analyze: reachability, coverability, ...

- **TINA**
  - simulate

- **TimedTraffic**
  - simulate

- **Traffic (un-timed)**
  - describe semantics by mapping onto
  - simulate

- **Timed Traffic**
  - describe semantics by mapping onto
  - simulate
Domain-Specific (Visual) Modelling
Domain-Specific Modelling Example

NATO’s Sarajevo WWTP

www.nato.int/sfor/cimic/env-pro/waterpla.htm
DS(V)M Environment

www.hemmis.com/products/west/
DS(V)M Example: smart phones, the application

MetaEdit+ (www.metacase.com)
DS(V)M Example: smart phones, the Domain-Specific model

[Diagram of a smart phone registration application]
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)
  ⇒ easier to learn
  ⇒ 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 **existing** transformations, tools, . . .
- 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.
Formalism Transformation Graph
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 ⇒ meta-modelling
   - concrete (textual/visual)

2. Specify DS(V)L semantics:
   transformation

3. Model (and analyze and execute) model transformations:
   ⇒ graph rewriting

⇒ model everything
   (in the most appropriate formalism, at the appropriate level of abstraction)
Dissecting a Modelling Language (tool builder’s view)
Deciding on terminology
What’s in a name? Language
What’s in a name? Formalism

Formalism F

Language (Abstract Syntax) $L$

Semantic Domain $S$

Semantic Mapping $M$

$m$
What’s in a name? Base Formalism

Language (Abstract Syntax)

Base Semantic Mapping

Base Semantic Domain
What’s in a name? Concrete Language

Concrete Syntax

Language (Abstract Syntax)

Concrete Language CL
What’s in a name? Concrete Formalism

Concrete Formalism F

Concrete Syntax

Language (Abstract Syntax)

Semantic Domain

Semantic Mapping
From now on: use AToM$^3$

A Tool for Multi-formalism and Meta-Modeling

Even our logos are modeled!

A model in the PacMan Formalism

(thanks to Reiko Heckel)
Modelling Abstract Syntax (meta-model)
Modelling the Scoreboard Entity

![Image of Scoreboard configuration interface]

- **name**: ScoreBoard
- **Graphical Appearance**: Edit
- **cardinality**: scoreLinkV3 dir= Source, min= 0, max=1
- **attributes**: score type=integer init.value=0
- **Constraints**: New create : from pacCo
- **Actions**: Edit
- **display**: Edit
- **Abstract**: Edit
- **QOCA**: Edit
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
Syntax Directed Editing (vs. Freehand)
Meta-modelling: model-instance morphism

meta-model/model

model-instance

 META- MODELLING: MODELINSTANCE MORPHISM

Model Everything!
Meta-meta-...: Meta-circularity
Adding to Abstract Syntax ...

Cardinalities:
- To gridBottomV3: 0 to N
- From gridBottomV3: 0 to N
- From pacLinkV3: 0 to N
- From foodLinkV3: 0 to N
- From gridRightV3: 0 to N
- To gridTopV3: 0 to N
- From gridTopV3: 0 to N
- From ghostLinkV3: 0 to N

Cardinalities:
- To pacLinkV3: 0 to N
- From pacmanV3: 0 to N
- To foodLinkV3: 0 to N
- From pacFoodV3: 0 to N

Attributes:
- score :: Integer

Actions:
- > create

Cardinalities:
- To gridNodeCenter: 0 to 1
- From gridNodeCenter: 0 to 1
- To gridLeftV3: 0 to 1
- From gridLeftV3: 0 to 1
- To gridRightV3: 0 to 1
- From gridRightV3: 0 to 1
- To gridTopV3: 0 to 1
- From gridTopV3: 0 to 1
- To gridBottomV3: 0 to 1
- From gridBottomV3: 0 to 1
- To ghostLinkV3: 0 to N
- From ghostV3: 0 to N
- To scoreLinkV3: 0 to N
- From ScoreBoard: 0 to N
- To scoreLinkV3: 0 to N

Cardinalities:
- To scoreLinkV3: 0 to N
- From ScoreBoard: 0 to N
- To ghostLinkV3: 0 to N
- From ghostV3: 0 to N

Cardinalities:
- To gridNodeCenter: 0 to 1
- From gridNodeCenter: 0 to 1
Modelling Ghost Concrete Visual Syntax
Adding to Abstract Syntax . . .
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-point

# 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

Note: create vs. execute
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)
Can now build valid PacMan models?
Model the GUI’s Reactive Behaviour in the Statechart formalism
The GUI’s reactive behaviour in action

current work: find *optimal* formalism to specify GUI reactive behaviour
Specifying Model Transformations

What is the “optimal” formalism?
Models are often graph-like ⇒ natural to express model transformation by means of graph transformation models.


Tools:

GME/GReAT, PROGRES, Fujaba, AGG, AToM³, GROOVE, ... First three used in large industrial applications.
Modelling PacMan Operational Semantics using Graph Grammar models

note: for Denotational Semantics: map for example onto Petri Net
Model Operational Semantics using GG
PacMan Die rule

WARNING: Name must use Python variable syntax

Name: pacDie
Order: 1
TimeDelay: 2
Subtypes Matching: 

LHS: Edit
RHS: Edit
Condition: Edit
Action: Edit

Enabled?

OK  Cancel
PacMan Die rule LHS

Hans Vangheluwe
hv@cs.mcgill.ca
Model Everything!
PacMan Die rule RHS
PacMan Eat rule LHS
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

6

9

10

Hans Vangheluwe
hv@cs.mcgill.ca
Model Everything!
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, graft on DEVS (AGTIVE)
Modular Graph Rewriting: emulate priorities
Model Based Development: some Open Problems

1. deal with concrete syntax (arbitrary mix of textual, visual) in a unified manner
2. deal with legacy models (code)
3. trace-ability (backward links)
4. consistency (TGGs + modularity)
5. (meta-) model evolution
6. multi-formalism modelling
7. multi-view modelling, (semantic) consistency
8. model refinement
9. design space exploration
10. automated testing (of models and model transformations)
11. transformation models are first-class models ⇒ higher-order transformation
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!