GenGED vs AToM$^3$

Presented by Denis Dubé
Feb 28, 2005
Overview

- Introduction
- Generating visual languages
- Simulation & Animation
- Conclusion
Overview

- **Introduction**
  - Acronyms
  - Motivations
  - Philosophies
  - Implementations
- Generating visual languages
- Simulation & Animation
- Conclusion
Acronyms

- Generation of Graphical Environments for Design
- A Tool for Multi-formalism and Meta-Modeling
Motivations

- Visual modeling and specification techniques are extremely useful for a host of domain specific applications.
- Visual modeling environments are expensive to hand-code.
  - Therefore it is highly desirable to automatically generate the environment from a meta-model.
Philosophies

- **Visual** definition of visual languages and VL model manipulation

- Everything is a model
- Model everything explicitly
Philosophies Realization

- Emphasis on visuals results in integrated **graphical constraints** handler, PARCON package
- All model manipulation done using graph grammars, AGG package

- Explicit meta-model (ie. Entity Relationship) to create VL environments
- Graph grammars used to lesser extent, **not as visual**
Implementation

- **Java** but the PARCON constraints handler is in **Objective C**, thus GenGED works only on **Linux & Solaris**

- **Python 2.3** and **Tcl/Tk 8.3** (or better), completely **platform independent** (in theory)
Overview

- Introduction
- **Generating visual languages**
  - The AToM³ way
  - Alphabet editor
  - Alphabet rules
  - Visual language rules
  - Syntax and Parse Grammars
- Simulation & Animation
- Conclusion
Generating VL’s

- Entity Relationship

<table>
<thead>
<tr>
<th>Entity3</th>
</tr>
</thead>
<tbody>
<tr>
<td>name type=String init.value=Entity_</td>
</tr>
<tr>
<td>Graphical_Appearance type=Appearance init.value=graph_class0.py</td>
</tr>
<tr>
<td>cardinality type=List init.value=</td>
</tr>
<tr>
<td>attributes type=List init.value=</td>
</tr>
<tr>
<td>Constraints type=List init.value=</td>
</tr>
<tr>
<td>Actions type=List init.value=</td>
</tr>
</tbody>
</table>

Relationship3

See board
Generating VL’s

- Class diagrams (Entity Relationship model)

<table>
<thead>
<tr>
<th>AtomClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName type=String init.value=MyClass</td>
</tr>
<tr>
<td>ClassCardinality type=List init.value=</td>
</tr>
<tr>
<td>ClassAttributes type=List init.value=</td>
</tr>
<tr>
<td>ClassConstraints type=List init.value=</td>
</tr>
<tr>
<td>ClassAppearance type=Appearance init.value=graph_class0.py</td>
</tr>
<tr>
<td>Abstract type=Boolean init.value=False</td>
</tr>
</tbody>
</table>

[Diagram of class diagrams]
Generating VL’s

- Class diagrams (Class diagram model)
Overview

- Introduction
- Generating visual languages
  - The AToM$^3$ way
  - Alphabet editor
  - Alphabet rules
  - Visual language rules
  - Syntax and Parse Grammars
- Simulation & Animation
- Conclusion
Running Example

- Class diagrams VL

- Elements:
  - Class diagrams
  - Classes
  - Associations between classes
  - Association classes
Alphabet editor

- **Graphical Object Editor** (draw visual icons)
- **TypiEditor** (map icons to semantic objects)
- **ConEditor** (connect semantic objects)
Alphabet editor: GOE

- Primitive objects: rectangles, circles, arrows, etc.
- Composite of primitive objects linked via graphical constraints
Alphabet editor: TypiEditor

- Mapping to graph nodes/edges of:
  - Graphical Objects
  - Place holders (non-visual)

- Creation of attribute data types by instantiating built-in data types
Alphabet editor: ConEditor

- **Attribution mode**: map nodes/edges with one or more data types
- **Link mode**: source and target definition for edges
Overview

- Introduction
- Generating visual languages
  - The AToM$^3$ way
  - Alphabet editor
  - **Alphabet rules**
  - Visual language rules
  - Syntax and Parse Grammars
- Simulation & Animation
- Conclusion
Alphabet rules

- Automatically generated for insertion & deletion
  - Node insertion: $\text{LHS} = \text{empty} \rightarrow \text{RHS} = \text{new node}$
  - Edge insertion: $\text{LHS} = 1+ \text{nodes} \rightarrow \text{RHS} = \text{new edge}$

Example: Edge Insertion
Alphabet rules

- Automatically generated for insertion & deletion
  - Data types:
    - LHS = Node/edge → RHS = Attributed Node/edge

Example: String attribute insertion
VL Rule Editor

- Idea: use the basic alphabet rules to create more powerful ‘VL Rules’
  - Example: insertion of a class

Alphabet rule: Class diagram insertion

VL rule (not finished): Class insertion
VL Rule Editor

- End result: VL Rule replaces the automatically generated alphabet rule
  - Example: insertion of a class

VL rule (not finished): Class insertion

VL rule (finished): Class insertion
VL Rule Editor

- More VL rules examples:

  - **VL rule:** Insert association
  - **VL rule:** Insert association class
VL Rule Application

- How are these rules applied?
  - Example: automatically generated alphabet rule
  - Example: Edge Insertion
VL Rule Application

- Illustration of one match morphism for the previous rule
Overview

- Introduction
- Generating visual languages
  - The AToM³ way
  - Alphabet editor
  - Alphabet rules
  - Visual language rules
  - Syntax and Parse Grammars
- Simulation & Animation
- Conclusion
Syntax Grammar

- Of what benefit are the VL rules?

- The VL rules form a syntax grammar that ensure that a diagram being constructed or modified is always correct with respect to the VL model.

- Definition: A VL model is the set of all possible diagrams in a given visual language.
Syntax Grammar

- AToM³ emulates a syntax grammar (in some sense) with preconditions and postconditions

- Caveat: it is nonetheless possible to construct incorrect diagrams
Parse Grammar

- What if the syntax grammar is too restrictive for interactive diagram editing?
  
  - Create a set of rules that work from a simple start diagram and tries to build the current working diagram
  
  Or
  
  - Create a set of rules that removes components of the current working diagram until it reaches a simple end diagram

See board
GenGED Overview
Overview

- Introduction
- Generating visual languages
- **Simulation & Animation**
  - Motivation
  - The AToM³ way
  - Simulation grammar
  - Simulation VS Animation
  - Animation & View transformation
- Conclusion
Motivation for simulation

- Simulation rules give the operational semantics of the underlying system represented by the visual model

  - Example: Petri-nets for the Traffic model
Motivation for animation

- Intuitive understanding of system behavior (especially for *non-experts*) cannot be expected in a (semi-) formal modeling language (ie: Petri-nets, Automatons)

- Desirable to visualize model & behavior in the application domain (ie: want to work with Traffic models not Petri-nets)
Simulation & Animation

- AToM³ handles model **simulation** by:
  - Graph grammars (lack of negative application conditions means some coding is required)
  - Hard-coded simulator

- AToM³ handles model **animation** by:
  - Graph grammars (currently broken in version 0.3)
  - Hard-coded animation
Running example

- Producer Consumer VL

Legend:
- Edges/Nodes
- Data types

Alphabet for producer consumer VL
Running example

- Producer Consumer VL

Example visual model

Legend:
- Edges/Nodes
- Data types
Simulation

- Describe behavior of the VL model using graph grammars (aka: a simulation grammar)
  - Rules represent model modification steps

- Rules = \(!NAC + LHS \rightarrow RHS\)
  - Definition: a NAC is a negative application condition, if an LHS of a rule matches, but the NAC also matches, the rule is not applied
Simulation

Simulation Rule 1:
- Production of a good at a ‘Producer’ component

Note: Data types not shown explicitly in the abstract layer
Simulation

Simulation Rule 2:
- Delivery of a good from a Producer to a Buffer

Note: Data types not shown explicitly in the abstract layer.
Simulation

- Simulation Rule 3:
  - Removal of a good from the Buffer to the Consumer

Note: Data types not shown explicitly in the abstract layer
Simulation

- Simulation Rule 4:
  - Consumption of a good by the Consumer

Note: Data types not shown explicitly in the abstract layer
Simulation

- Each rule application/derivation is a simulation step.

Note: Data types not shown explicitly in the abstract layer.
GenGED Overview
Overview

■ Introduction
■ Generating visual languages
■ Simulation & Animation
  □ Motivation
  □ The AToM³ way
  □ Simulation grammar
  □ Simulation VS Animation
  □ Animation & View transformation
■ Conclusion
Simulation VS Animation

- **Simulation** visualizes *discrete* state changes within the VL model itself

- **Animation** visualizes *continuous* state changes in a domain-oriented layout
  - Example: A traffic system with cars that move along a road and traffic lights that change colors
Animation

- Transformation from VL model and the associated simulation rules to an animation view must be done with care
  - Must avoid deviations between the two or worse, contradictions!
  - In particular: we want to preserve the precision of the (semi-) formal model in the animation view
- Therefore: generate the animation view systematically from the VL model with a formal view transformation grammar
View Transformation

- The view transformation grammar:
  - Transforms the VL model to a domain specific layout
  - Transforms the simulation grammar into an animation grammar
  - Permits the addition of attributes to the simulation grammar that allow for continuously changing objects (i.e., position, size, color, of objects can change continuously between specified time intervals)
View Transformation

- Producer consumer model & two animation views
Transformation Grammar

- Idle Producer transformation
Transformation Grammar

- Busy Producer transformation
Transformation Grammar

- Empty Buffer transformation
Transformation Grammar

- Full Buffer transformation
Transformation Grammar

- Empty Consumer transformation
Transformation Grammar

- Full Consumer transformation
Animation Grammar

- Automatic transformation of *Simulation rule* to Animation rule
Overview

- Introduction
- Generating visual languages
- Simulation & Animation
- Conclusion
Conclusion

- GenGED and AToM³ are similar
  - Generate visual language environments
  - Allow simulation & animation
  - Rely on graph grammar transformations extensively

- The visual emphasis of GenGED, at least on the surface, makes it a far more accessible tool
  - No/less hand-coding
  - Systematic animation system
Conclusion

- Graphical Constraints

- GenGED provides **high level** constraints
  - Example: rectangle1 sameBorderwidth rectangle2

- These constraints are mapped to one or more **low level** constraints that PARCON understands
Conclusion

- Graphical Constraints

- Graphical constraints are a key component in GenGED since they are used to:
  - Create composite graphical objects with multiple primitives
  - Anchor arrow points at object borders
  - Enforce insideness relations between objects
Sources

- Sencario Views for Visual Behavior Models in GenGED
  - Authors: C. Ermel and R. Bardohl
  - Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02), Satellite Event of First Int. Conference on Graph Transformation (ICGT'02), Barcelona, Spain, Oct. 2002, pages 71-83

- A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars
  - Author: Roswitha Bardohl

- GenGED - A visual definition tool for visual modeling environments
  - Authors: Bardohl,R., Ermel,C., and Weinhold,I.
  - http://www.tfs.cs.tu-berlin.de/~rosi/publications/BEW03_AGTIVE03.ps.gz
Sources

- Conceptual Model of the Generic Graphical Editor GenGED for the Visual Definition of Visual Languages
  - Authors: Bardohl, R. and Ehrig, H.
  - Lecture Notes in Computer Science (LNCS) **1764**: Theory and Application of Graph Transformation (TAGT'98), Springer 1999, pages 252-266

  - Authors: Bardohl, R. and Ermel, C.

- Specifying Visual Languages with GenGED
  - Authors: Bardohl, R., Ehrig, K., Ermel, C., Qemali, A. and Weinhold, I.
  - Proc. APPLIGRAPH Workshop on Applied Graph Transformation (AGT'02), Satellite Event of ETAPS 2002, Grenoble, France, April 12-13, 2002, pages 71-82