

Domain-Specific Languages (DSLs)

Juan de Lara, Elena Gómez, Esther Guerra

{Juan.deLara, MariaElena.Gomez, Esther.Guerra}@uam.es

Computer Science Department
Universidad Autónoma de Madrid

Index

- **Introduction.**

- **Syntax.**

- **Semantics.**

- **Examples.**

- **Types of modelling environments.**

- **Technologies to build DSLs.**

- **Bibliography.**



Domain-Specific Languages (DSLs)

- Languages oriented to a particular application domain or problem (in contrast to general-purpose languages).
- They capture the knowledge and experience in a specific application area.
- High-level, expressive, powerful primitives.
- Premise: DSLs enhance productivity compared to using general-purpose languages.
- DSLs are extensively created/used in MDE solutions.

Problem domain vs Solution domain

Domain-Specific Languages

- oriented to users way of thinking
- smaller semantic gap to problem



**(semi)automatic
transformation**

General-Purpose Languages

- oriented to developers way of thinking
- need to transform into technical domain



Types of Domain-Specific Languages

- **Internal or embedded:** they use the infrastructure of an existing host language (e.g., Ruby, UML profiles).
 - Shorter development time
 - Same concrete syntax as the host language
- **External:** they are built from scratch.
 - Flexibility on the concrete syntax of the language
 - Costly implementation (requires implementing parser, syntactic analyzer, interpreter or compiler, editing environment, etc.)
 - ...but there are frameworks that facilitate their development, like Sirius (for graphical DSLs) or Xtext (for textual DSLs)

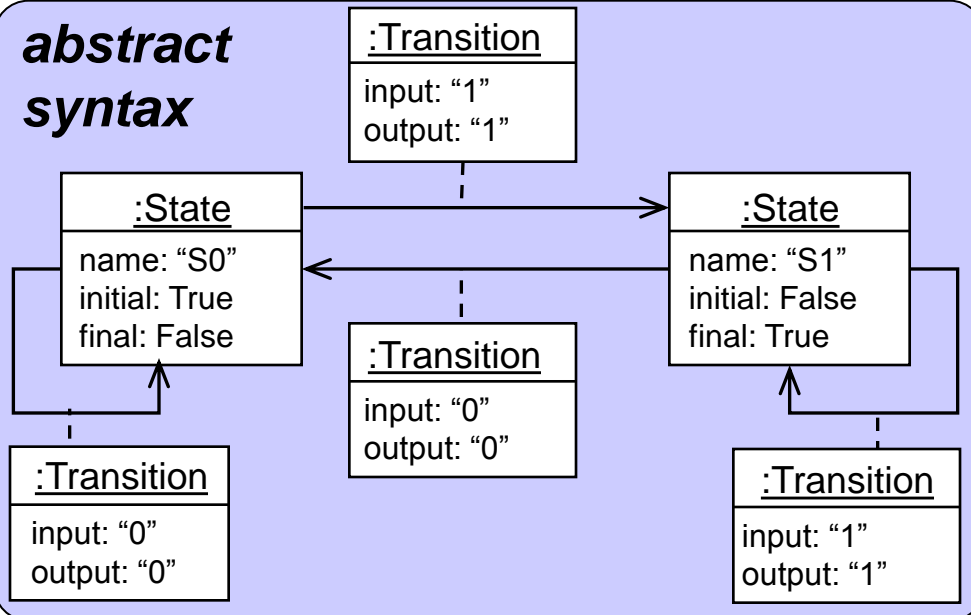
External Domain-Specific Languages

- DSLs can be graphical, textual, or a combination of text and graphics such as:
 - OCL + UML
 - Action language of UML
 - Languages including mathematical expressions
- Multi-view language: set of diagrams describing different aspects of a system.
- Combined with code generators and simulators.

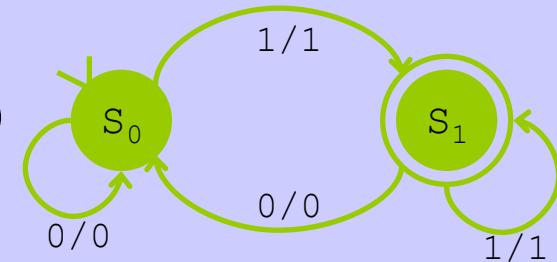
Syntax

- **Abstract syntax:** language concepts, relations and attributes. It can be defined using a meta-model or a creation graph grammar.
- **Concrete syntax:** visualization of the abstract syntax elements.
 - Not necessarily a 1-to-1 mapping
 - Spatial relationships (e.g. containment, adjacency)
 - Spatial constraint languages (e.g. QOCA, <https://www.swmath.org/software/756>)

abstract syntax



concrete syntax (graphical)



concrete syntax (textual)

States:

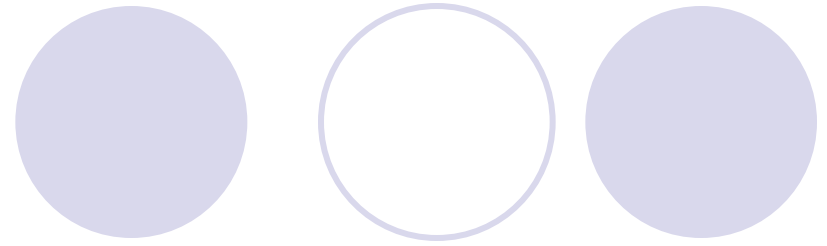
S0 {initial},
S1 {final}

Transitions:

S0 = 0/0 => S0
S0 = 1/1 => S1
S1 = 0/0 => S0
S1 = 1/1 => S1

Concrete syntax

Meta-modelling

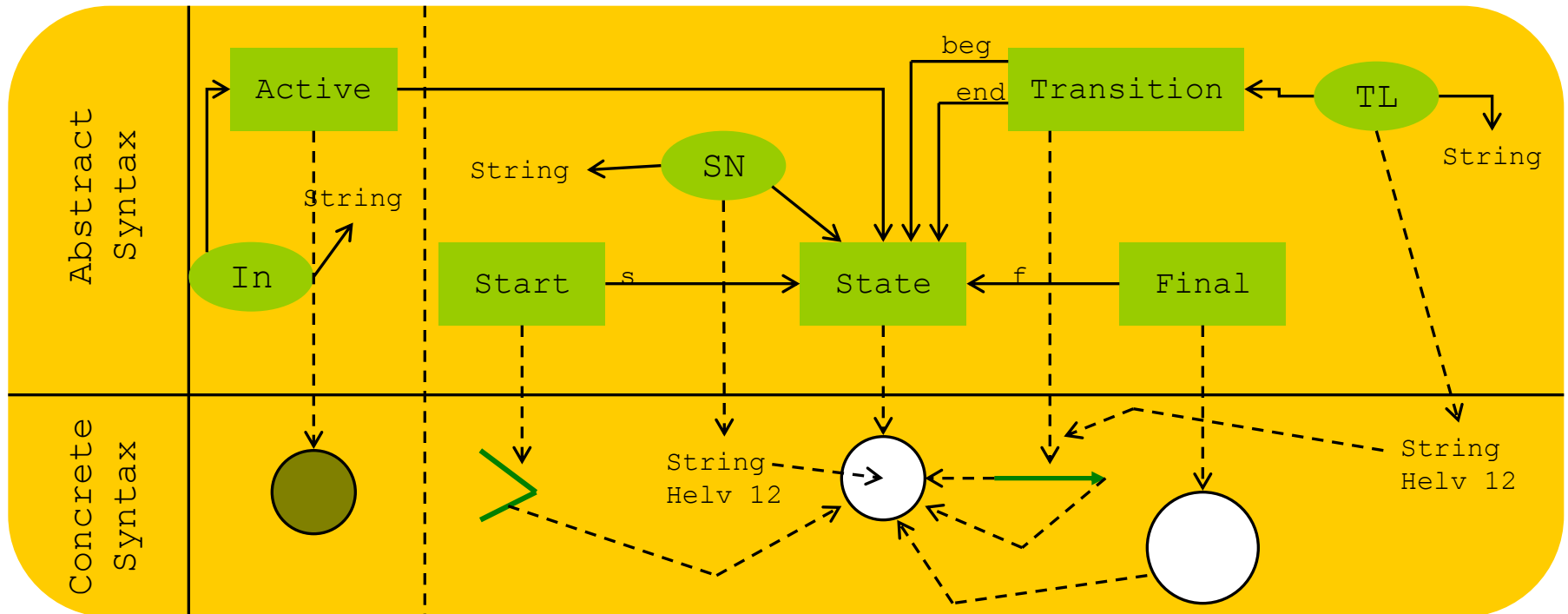


- The concrete syntax can be given as graphical attributes of the classes and associations.
- For relations *n-to-m*, this can be very restrictive:
 - A meta-model for the concrete syntax and a meta-model for the abstract syntax. Transformations between them.
- Spatial relations, e.g. “contained”, “aligned with”, “touches”, etc.

Concrete syntax

Creation grammars

- Rules can use symbols of the alphabet of the concrete syntax.
- GenGED: it uses editor of symbols + constraint satisfaction system.



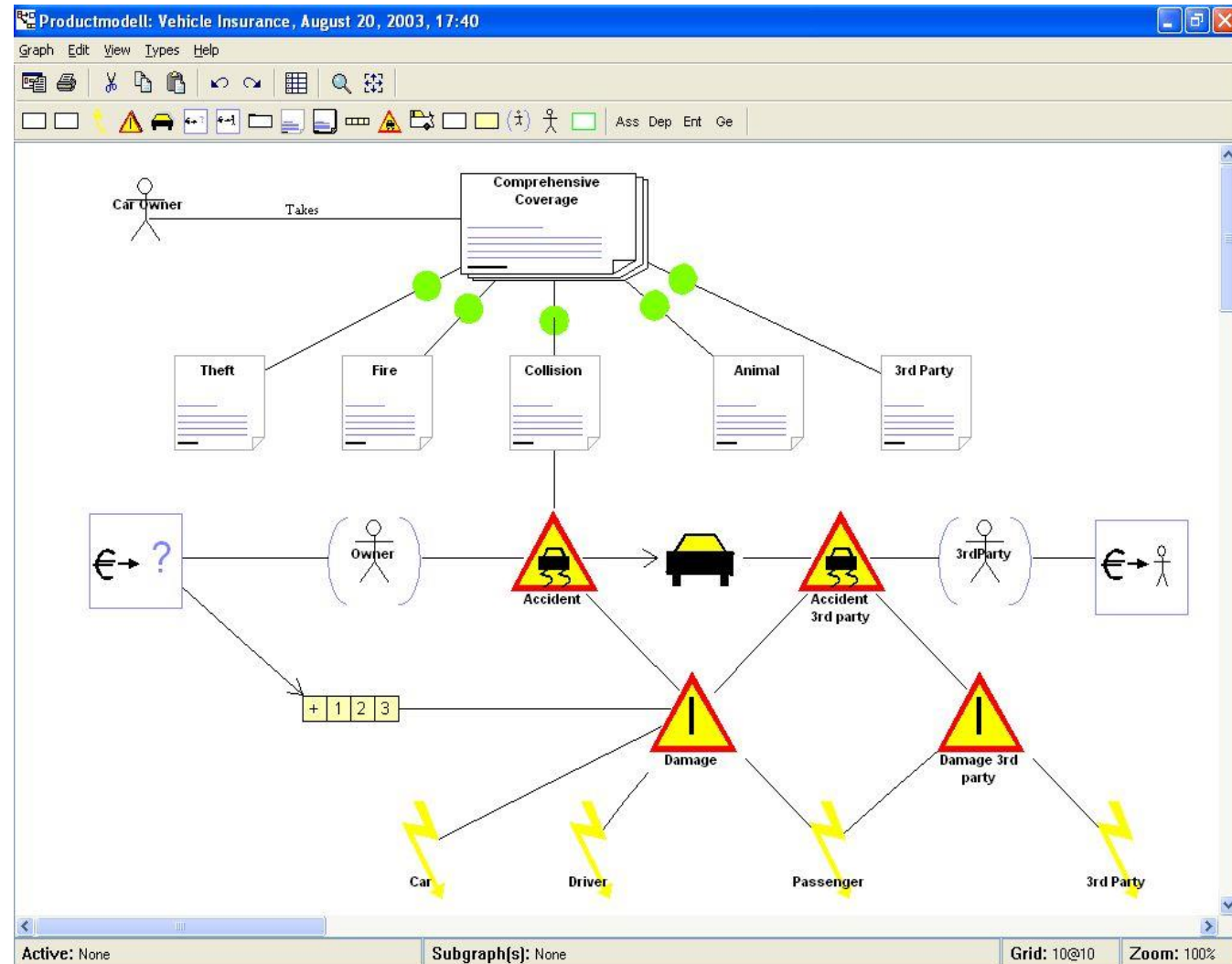
Semantics



- **Static semantics:** Additional constraints.
 - Usually described using a constraint language like OCL
 - Is it semantics or syntax?
- **Operational semantics:** How to execute the model (simulator or “virtual machine” for the language).
 - Graph transformation, in-place model transformation techniques
 - A programming language
- **Denotational semantics:** Meaning of each construction in terms of a different formalism.
 - Model-to-model transformation
 - Code generation

Examples

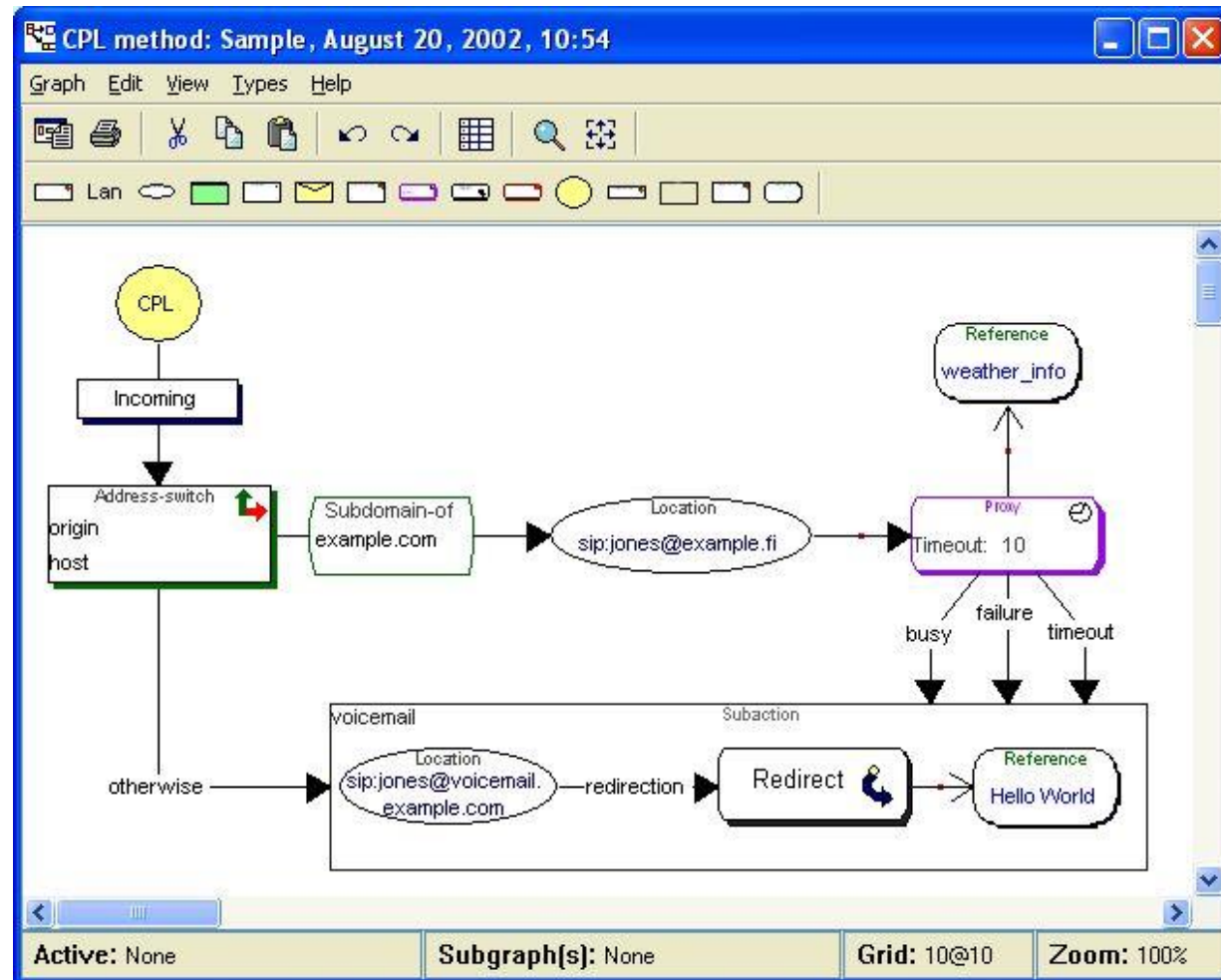
- Expert domain concepts.
- Simple code generation.
- Valid in well-known domains.
- Usable by non-programmers.



Ensurance company / J2EE

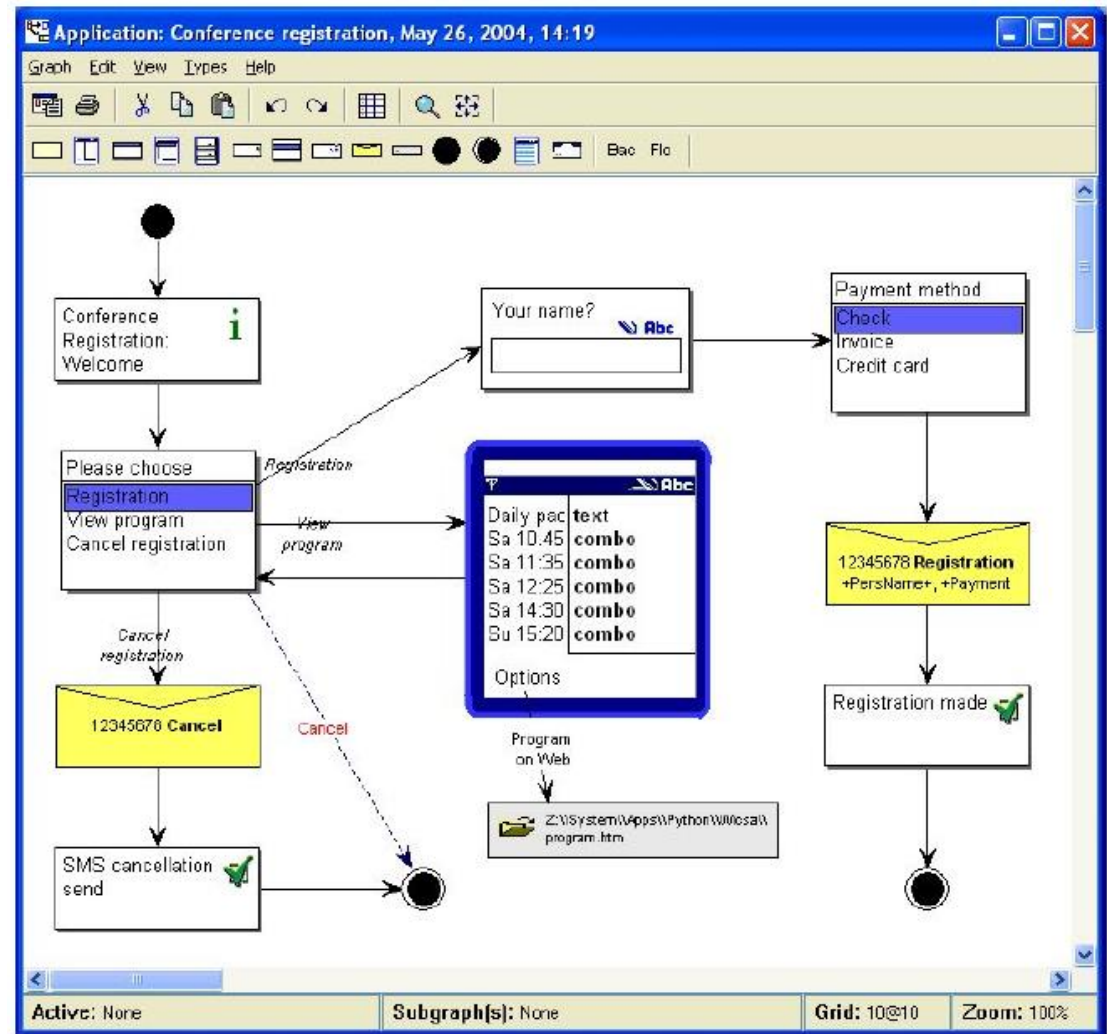
Examples

- Programming concepts.
- Static part is easy (data structures).
- In the limit, visual notation for programming language.
- Danger of low level of abstraction, small increase in productivity.



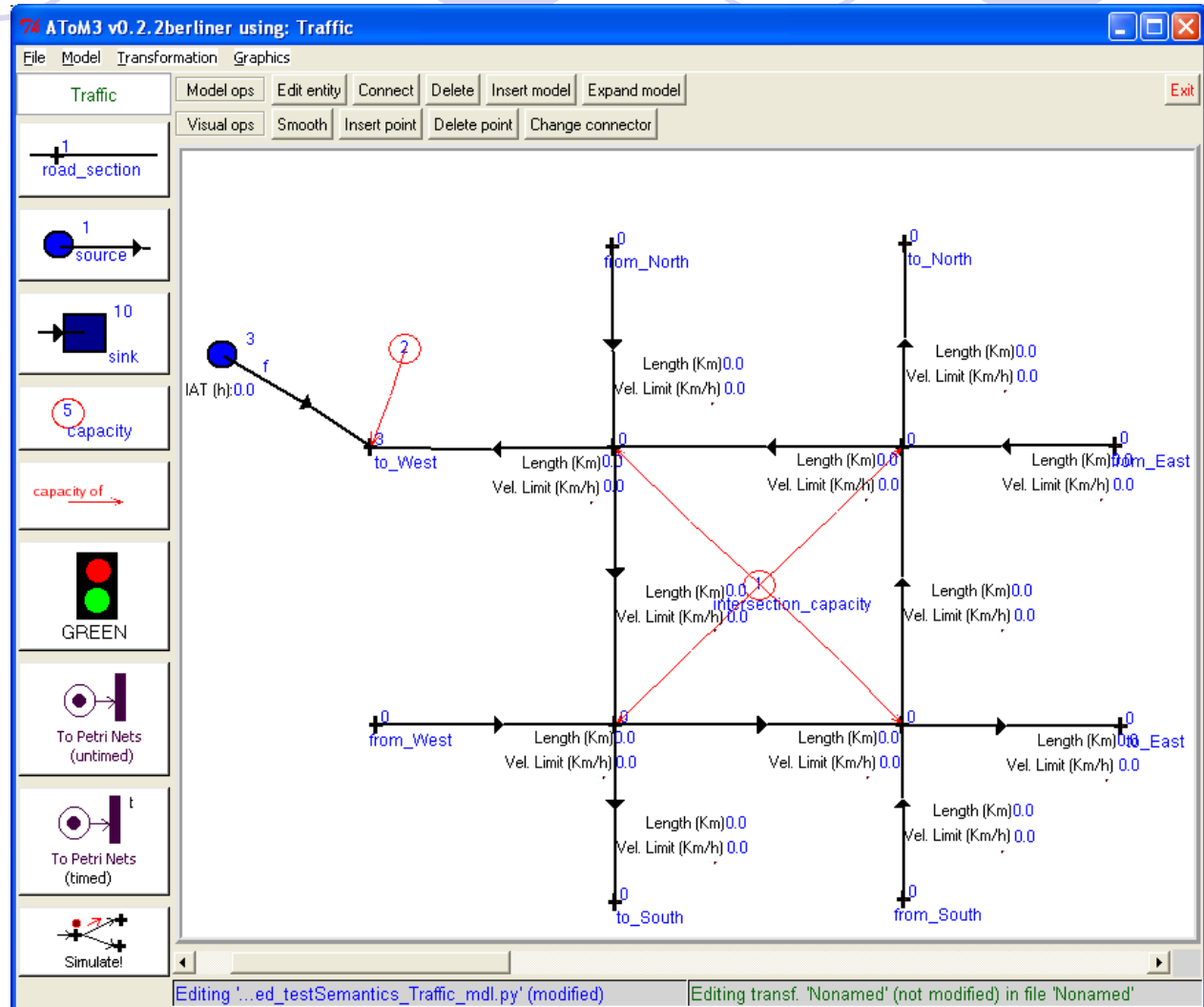
Examples

- Constructions that handle the user interface.
- Similar to state machines.
- Concepts are easy to identify.



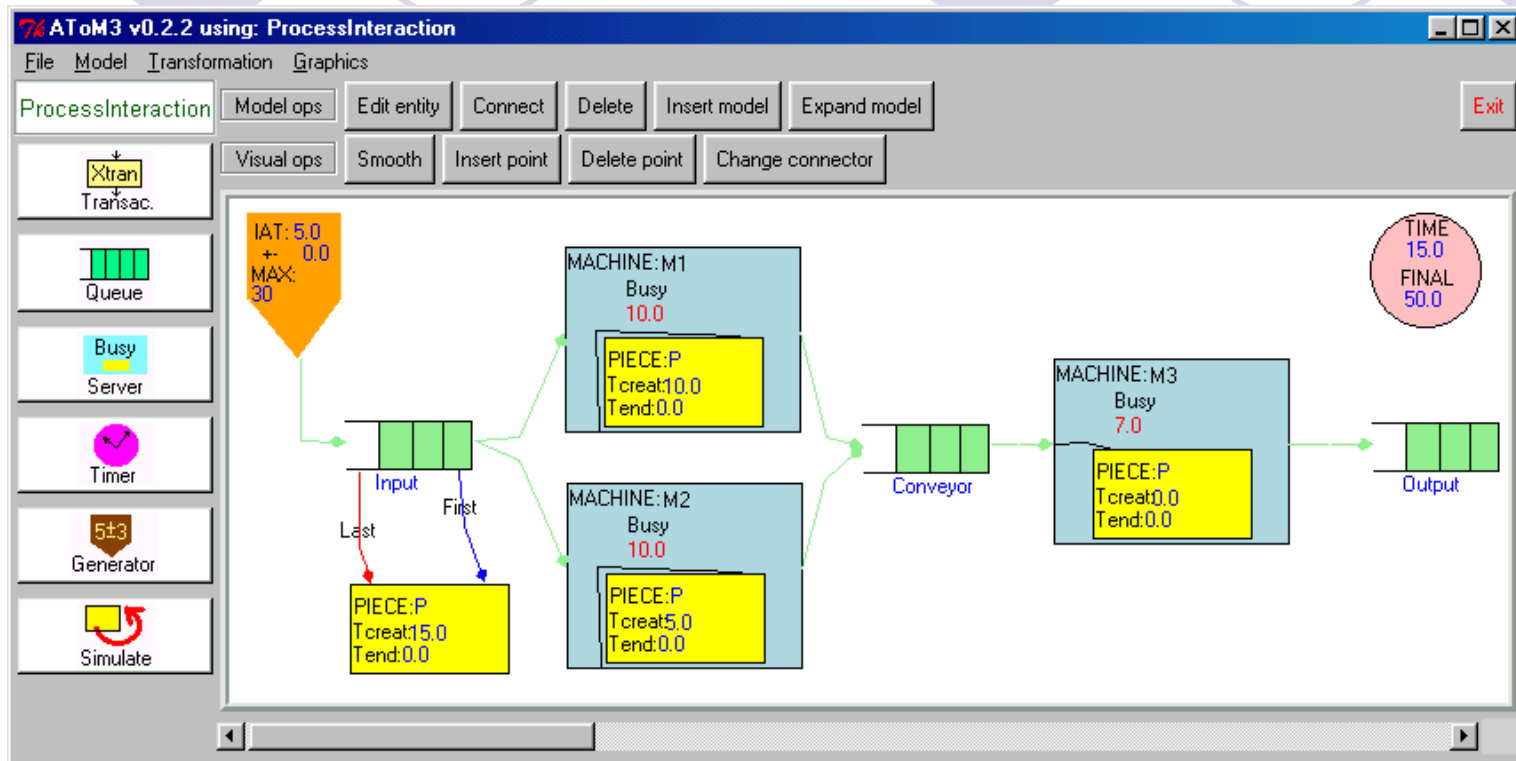
Examples

- Description of physical systems (nets of roads).
- Operational semantics (simulator).
- Denotational semantics (transformation into Petri nets).



Road nets / Petri nets

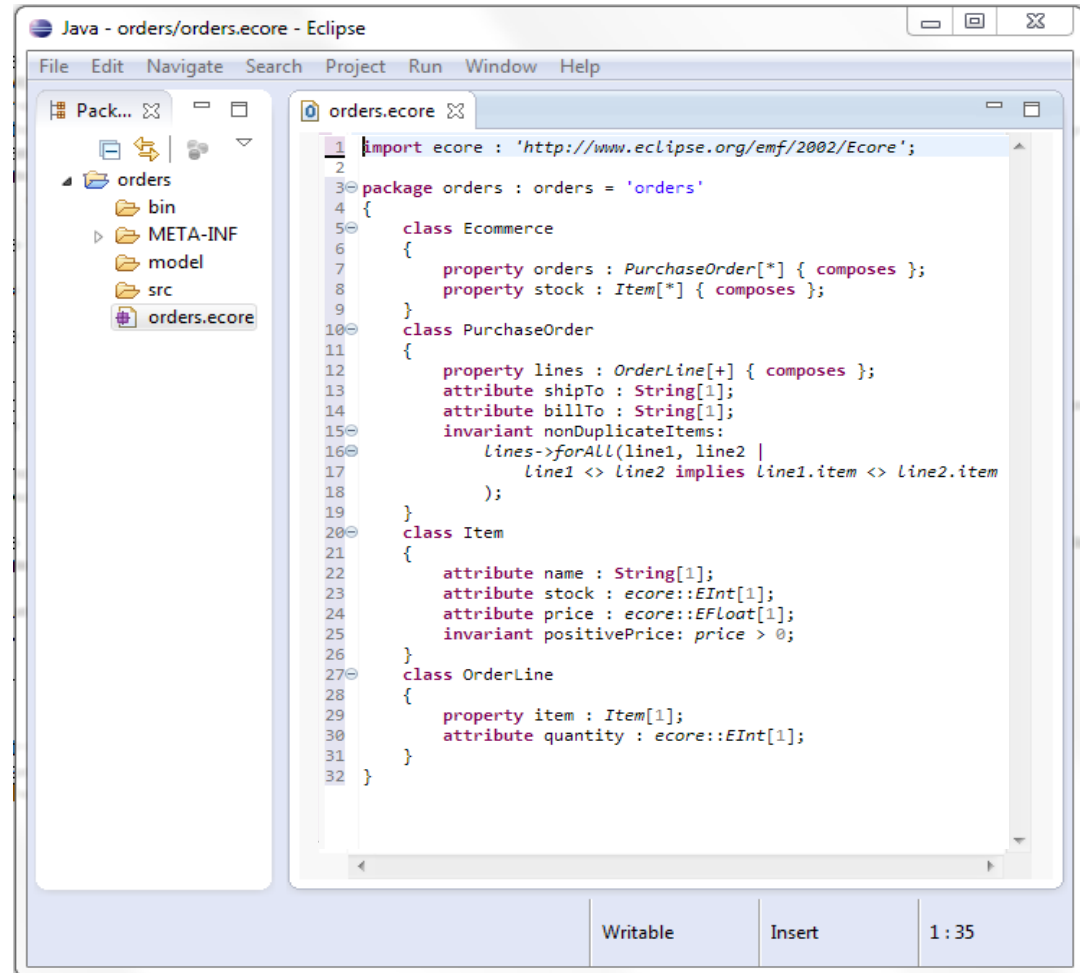
Examples



- DSVL to describe manufacture systems (discrete simulation).
- Educational purpose.

Examples

- OclInEcore: textual DSL to specify meta-models.
- Java code generation.



The screenshot shows the Eclipse IDE with a project named 'orders'. The 'orders.ecore' file is open in the editor. The code defines an Ecore meta-model for an orders system. It includes an import for the Ecore package, a package declaration for 'orders', and three classes: 'Ecommerce', 'PurchaseOrder', and 'Item'. 'Ecommerce' has two properties: 'orders' of type 'PurchaseOrder[*]' and 'stock' of type 'Item[*]', both with 'composes' relationships. 'PurchaseOrder' has a property 'lines' of type 'OrderLine[+]' with a 'composes' relationship, and an invariant 'nonDuplicateItems' that ensures no two lines share the same item. 'Item' has three attributes: 'name' of type 'String[1]', 'stock' of type 'ecore::EInt[1]', and 'price' of type 'ecore::EFloat[1]', with an invariant 'positivePrice' that ensures the price is greater than 0. 'OrderLine' has a property 'item' of type 'Item[1]' and an attribute 'quantity' of type 'ecore::EInt[1]'.

```
1 import ecore : 'http://www.eclipse.org/emf/2002/Ecore';
2
3 package orders : orders = 'orders'
4 {
5     class Ecommerce
6     {
7         property orders : PurchaseOrder[*] { composes };
8         property stock : Item[*] { composes };
9     }
10    class PurchaseOrder
11    {
12        property lines : OrderLine[+] { composes };
13        attribute shipTo : String[1];
14        attribute billTo : String[1];
15        invariant nonDuplicateItems:
16            lines->forAll(line1, line2 |
17                line1 <> line2 implies line1.item <> line2.item
18            );
19    }
20    class Item
21    {
22        attribute name : String[1];
23        attribute stock : ecore::EInt[1];
24        attribute price : ecore::EFloat[1];
25        invariant positivePrice: price > 0;
26    }
27    class OrderLine
28    {
29        property item : Item[1];
30        attribute quantity : ecore::EInt[1];
31    }
32 }
```

meta-model + constraints / Java code

Examples

- LilyPond: textual DSL to specify music sheets.
- Graphical music sheet generation.

The image displays the LilyPond workflow for generating a musical score. On the left, a snippet of the LilyPond source code is shown, featuring musical notation in a textual DSL (e.g., `fis2. fis4 e4. e8 a2.~ a4 r r|`) and dynamic markings like `\pp`, `\p`, and `\f`. On the right, the corresponding graphical music sheet is presented, titled "from The Crucifixion (1887)" by Sir John Stain. The score is for a Quartet or Chorus, with parts for Soprano (S), Alto (A), Tenor (T), and Bass (B). The tempo is marked "Andante ma non lento." with a metronome marking of 90. The lyrics are: "God so loved the world, God so loved the world, gave His on-ly be-got-ten Son, that who-so be-liev-eth, be-Him should not per-ish, should not per-ish, but have ev-er-last-For God sent not His Son into the world to con-demn the world, God s". The graphical output shows the notes, rests, and dynamic markings as they would appear in a printed score.

(textual) music sheet / (graphical) music sheet

Index

- Introduction.
- **Types of modelling environments.**
 - Free-hand environments.
 - Syntax-directed environments.
- Technologies to build DSLs.
- Bibliography.



Free-hand environments



- “Low-level” editors which allow users to manipulate directly the diagram.
- Parser to recognise the syntactic structure and correctness of the diagram.
- Freedom in the way diagrams are edited.
- This can be a disadvantage, as users have no guidance on how to build their models.

Syntax-directed environments

- Editing actions are modelled as graph grammar rules.
- It requires both creation rules and deleting rules.
- Interesting technique for complex editing actions (e.g. creating or connecting many elements).
- Having many different rules can make this approach difficult to manage.

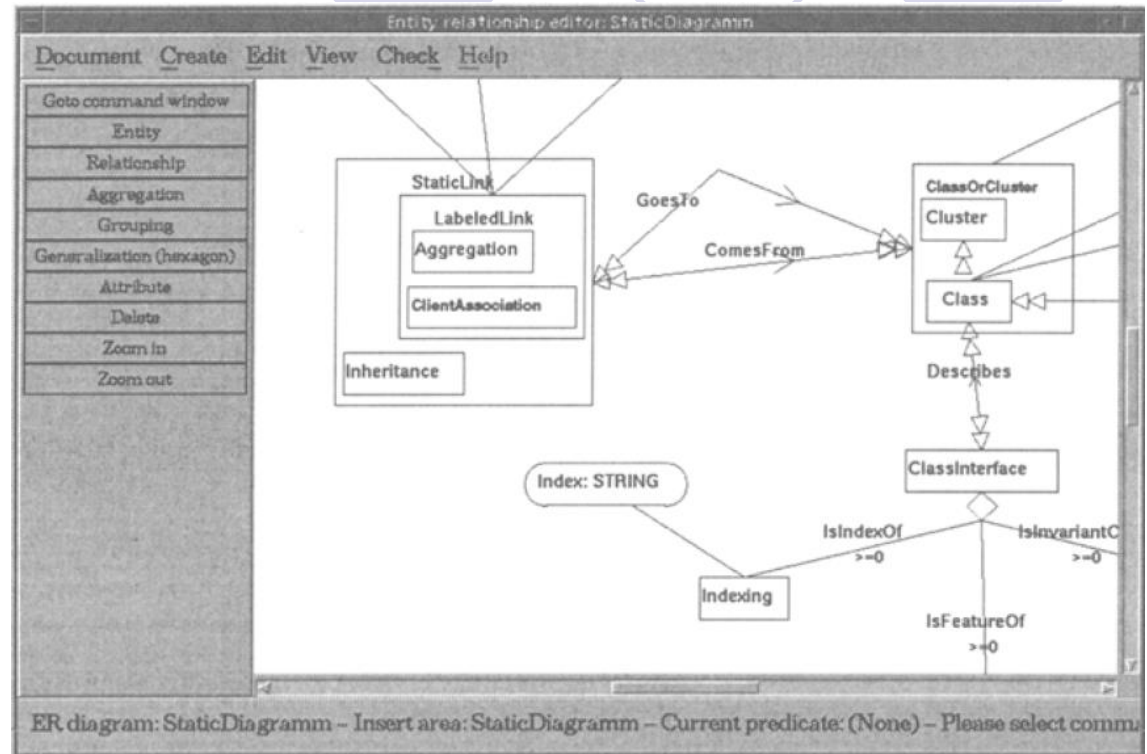
Index

- Introduction.
- Types of modelling environments.
- **Technologies to build DSLs.**
- Bibliography.



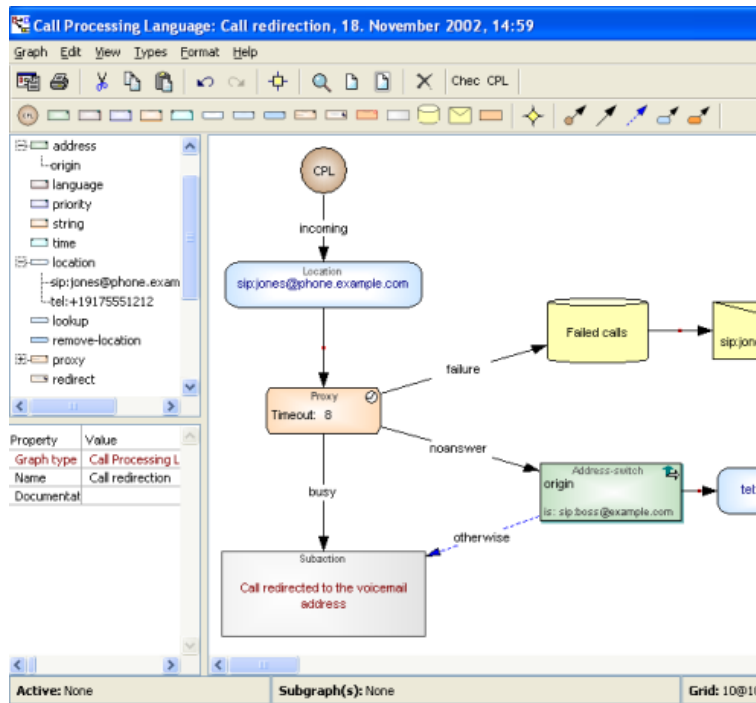
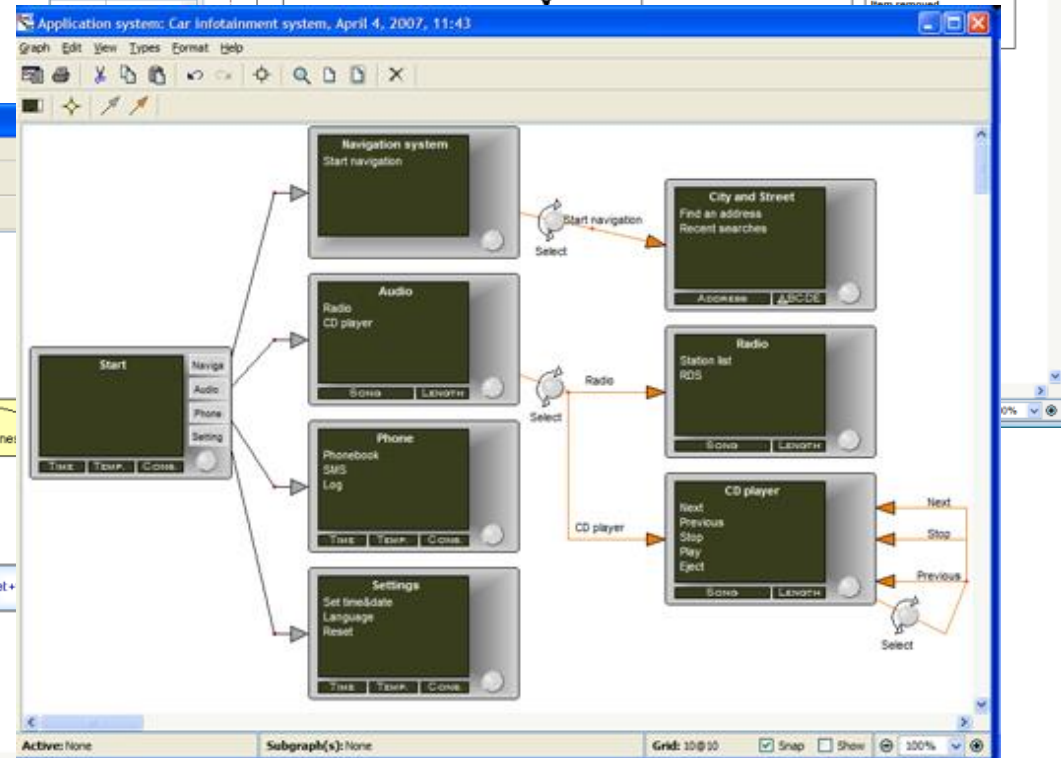
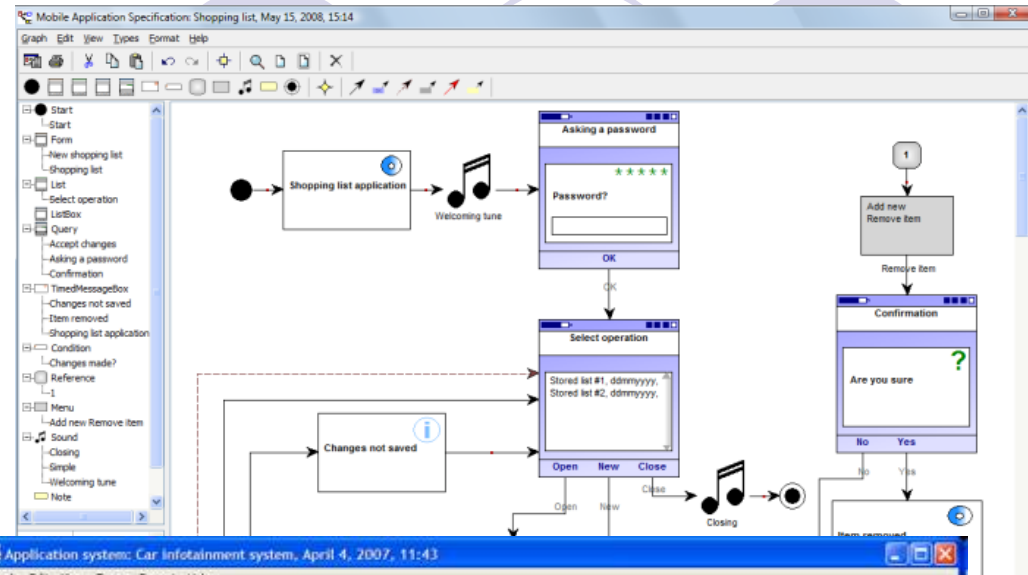
KOGGE

- 1997.
- Ebbert, Süttenbach, Uhe (Loblenz).
- Meta-CASE tools, to build CASE tools.



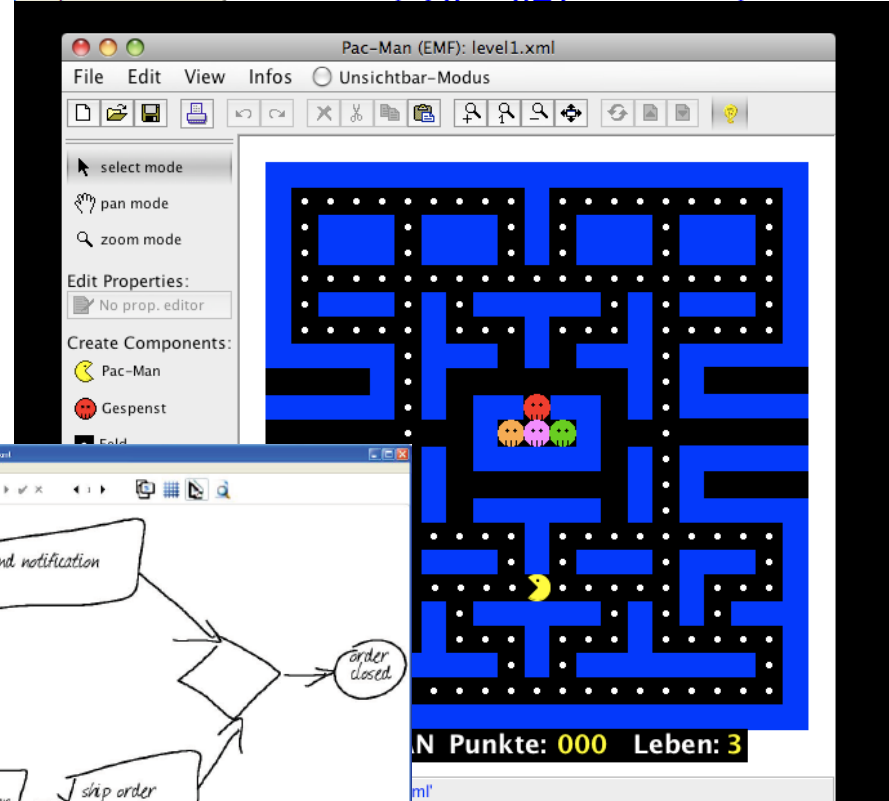
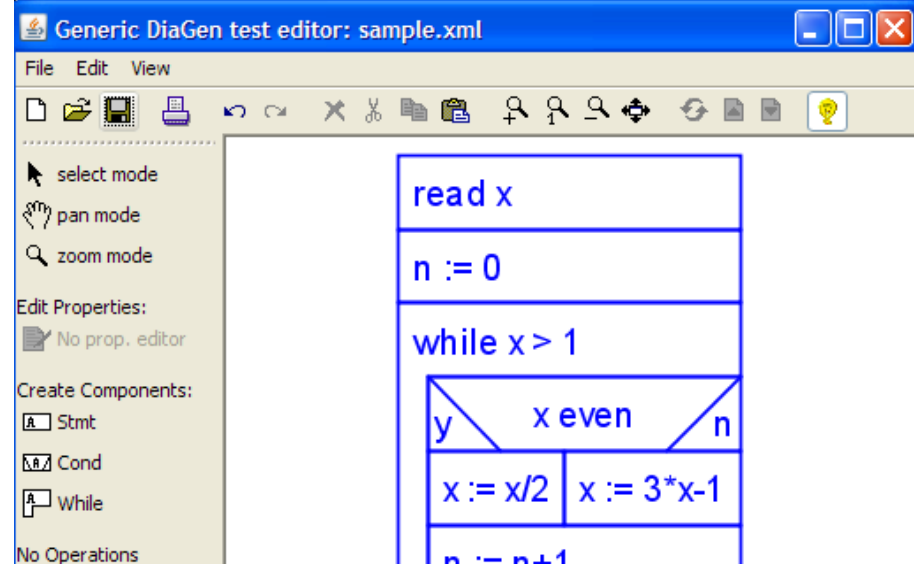
MetaEdit+

- First version in 1995 (<http://www.metacase.com>).
- Commercial, multi-user.



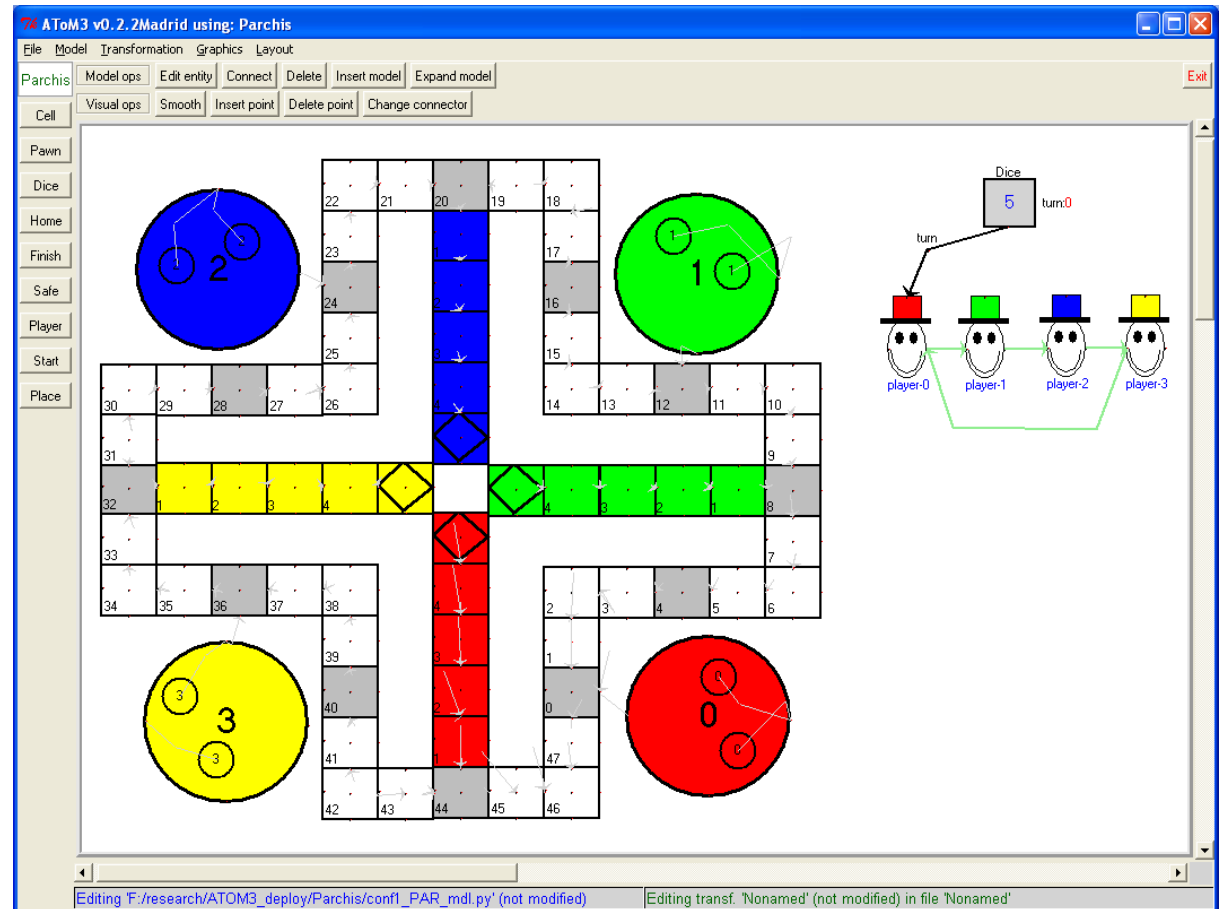
DiaGen/DiaMeta

- First version in 1993 (<http://www.unibw.de/inf2/DiaGen/>).
- Based on hypergraph grammars.
- Sketching.
- Mark Minas (Munich).



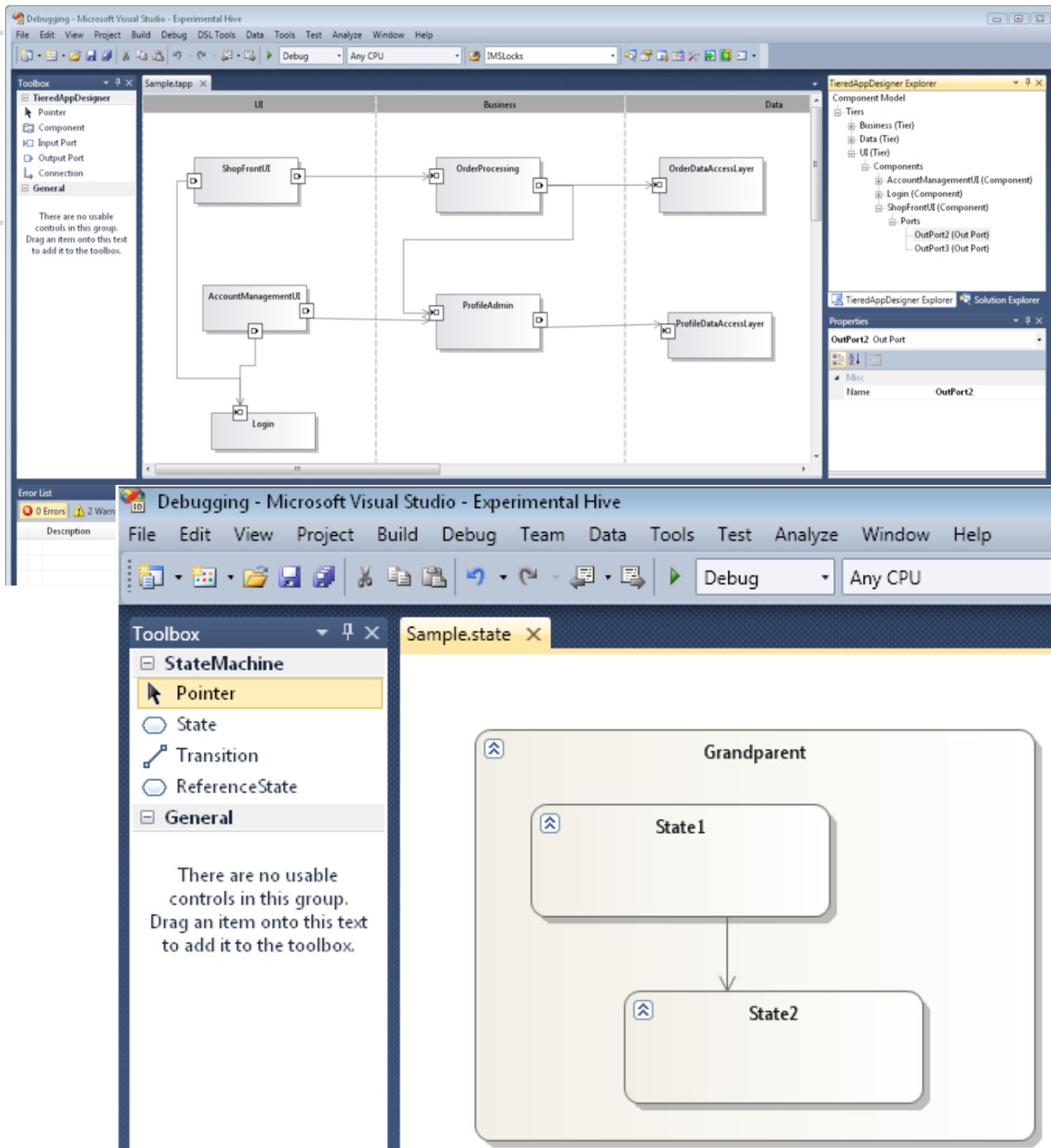
AToM³

- 2002.
- Model manipulation can be graphically defined using graph transformation.
- Simulation.



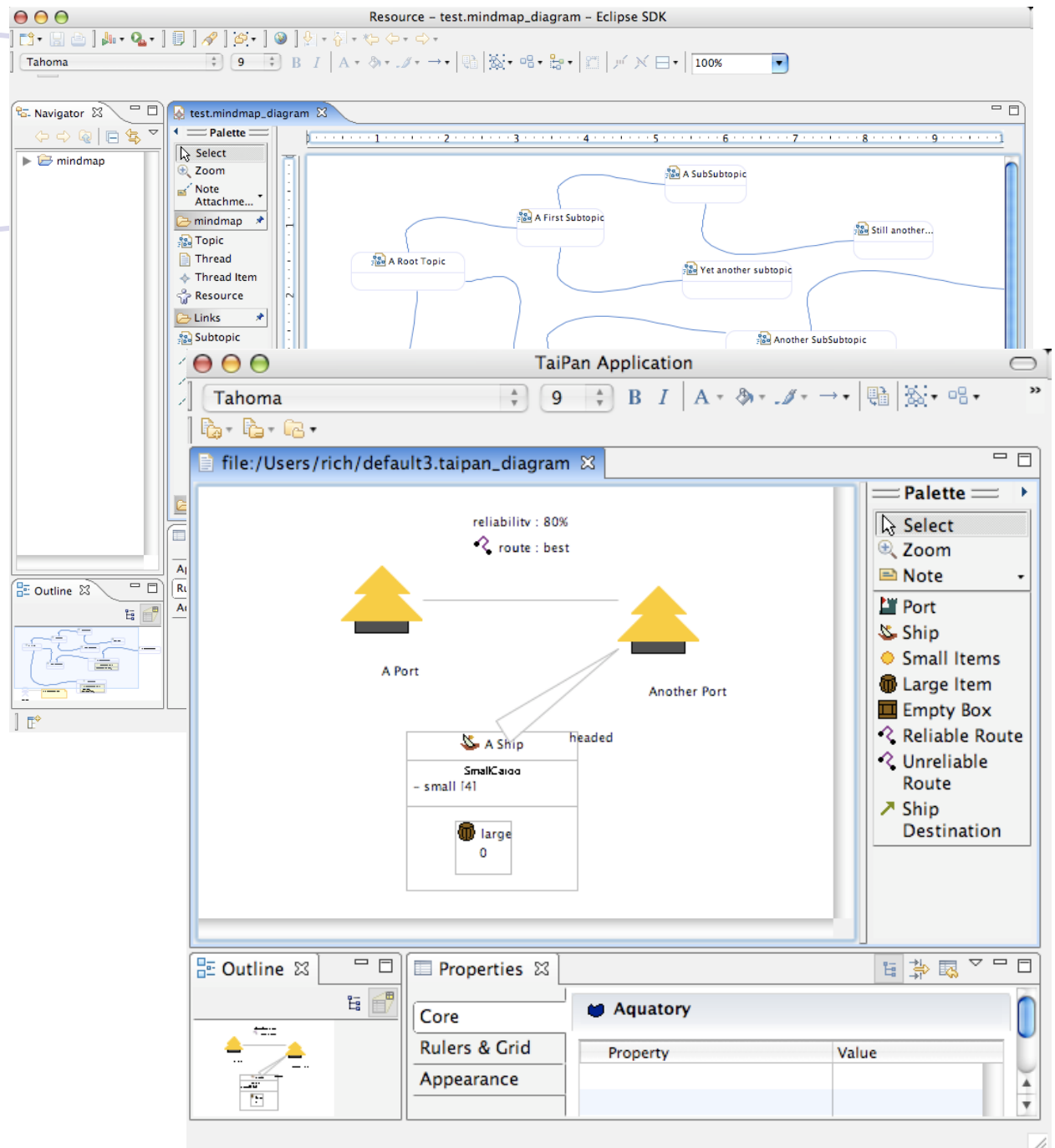
DSL Tools

- Microsoft/Visual Studio.



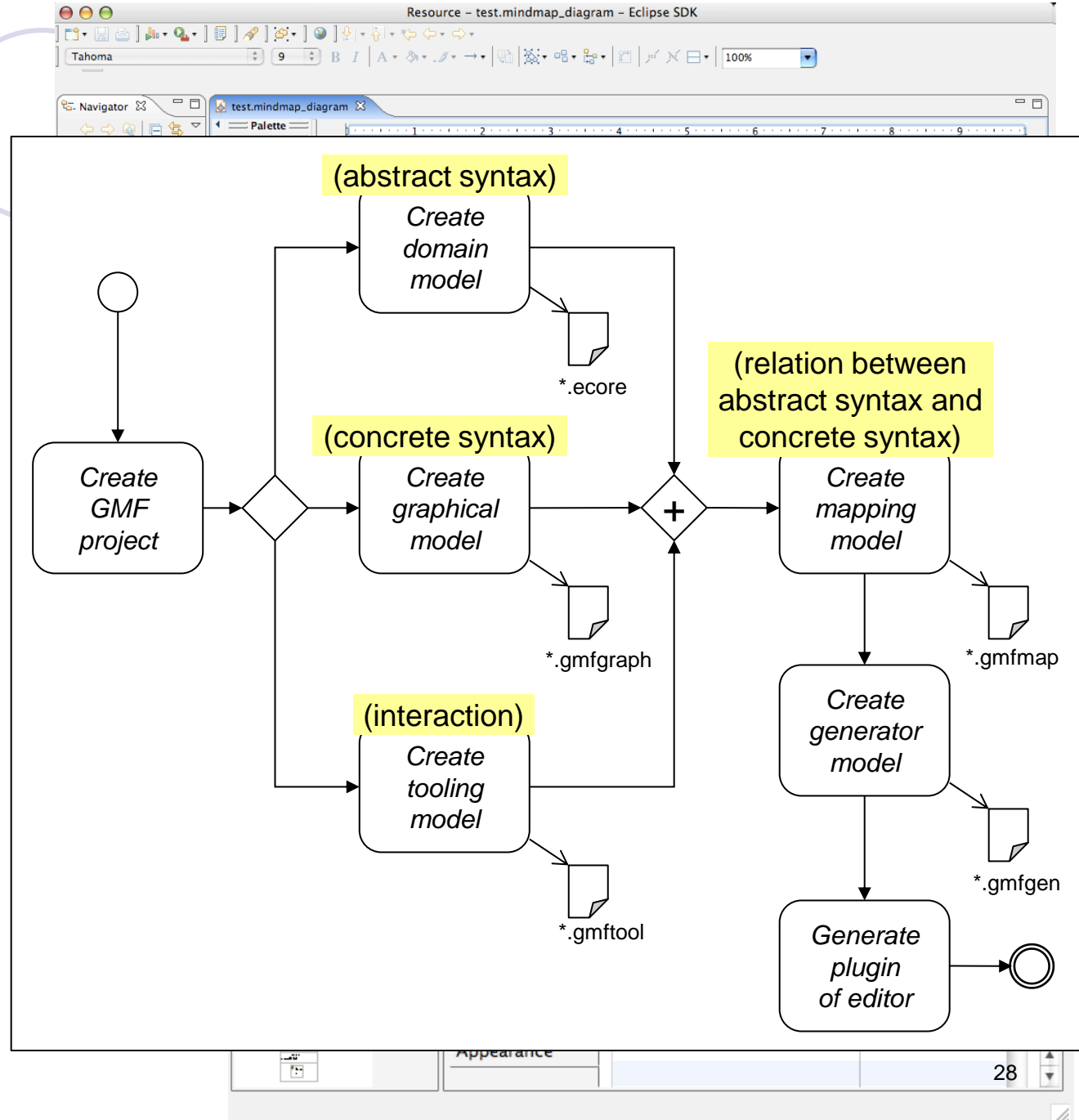
GMF

- EMF/Eclipse.



GMF

- EMF/Eclipse.
- Complex!



Eugenia

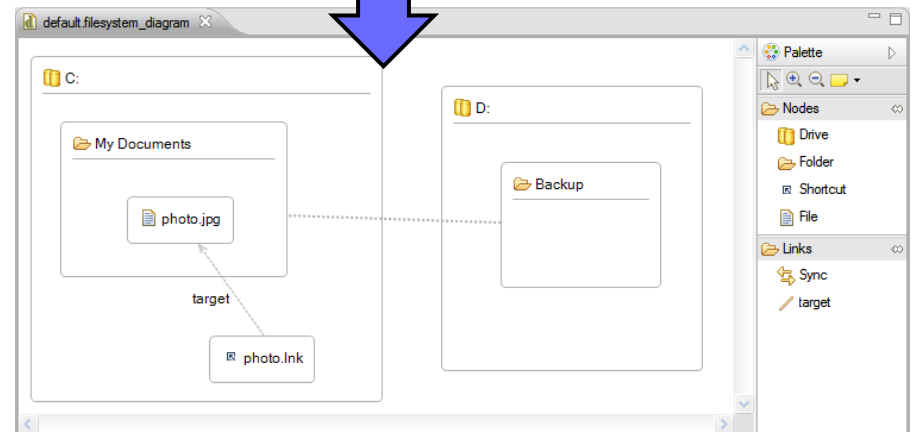
- EMF/Eclipse.
- <http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/>.
- It generates GMF editors from annotated ecore meta-models.
- The generated GMF editor must be maintained by hand.

```
class Folder extends File {  
    @gmf.compartment  
    val File[*] contents;  
}
```

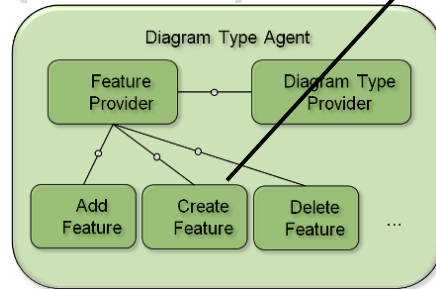
```
class Shortcut extends File {  
    @gmf.link(target.decoration="arrow", style="dash")  
    ref File target;  
}
```

```
@gmf.link(source="source", target="target", style="dot", width="2")  
class Sync {  
    ref File source;  
    ref File target;  
}
```

```
@gmf.node(label = "name")  
class File {  
    attr String name;  
}
```



Graphiti



- EMF/Eclipse.
- <http://www.eclipse.org/graphiti/>.
- Flat learning curve (Java API + Graphiti objects), high flexibility, common look and feel with sensible defaults.
- Spray
(<https://code.google.com/a/eclipselabs.org/p/spray/>): DSL to describe Graphiti editors.

```

public class CreatePurchaseOrderFeature
    extends AbstractCreateFeature
    implements ICreateFeature {

    public CreatePurchaseOrderFeature(IFeatureProvider fp) {
        super(fp, "PurchaseOrder", "Creates a new PurchaseOrder");
    }

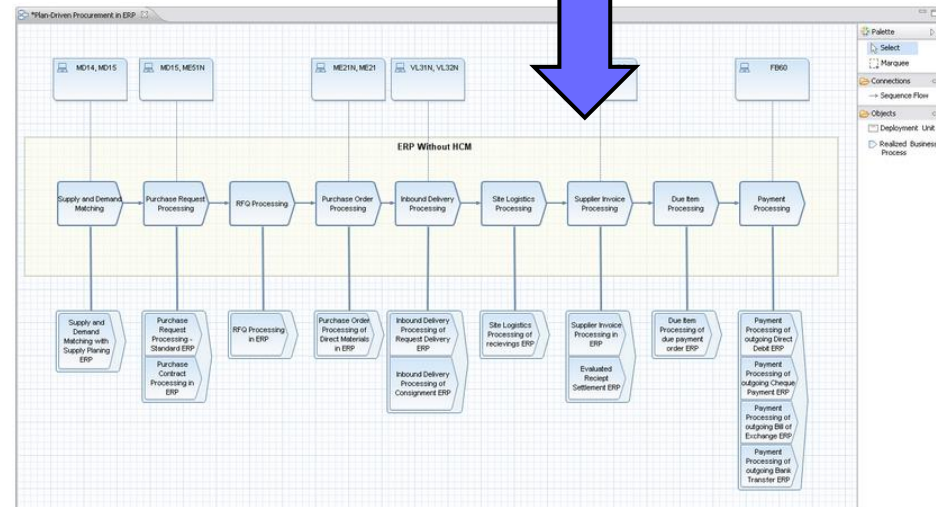
    @Override
    public boolean canCreate(ICreateContext context) {
        // check appropriate context
        return context.getTargetContainer().instanceof Diagram;
    }

    @Override
    public Object[] create(ICreateContext context) {
        // create the domain object
        PurchaseOrder newPurchaseOrder =
            OrdersFactory.eINSTANCE.createPurchaseOrder();

        // attribute values
        String shipTo = (String) JOptionPane.showInputDialog(
            (new JFrame(), "Ship to");
        newPurchaseOrder.setShipTo(shipTo);

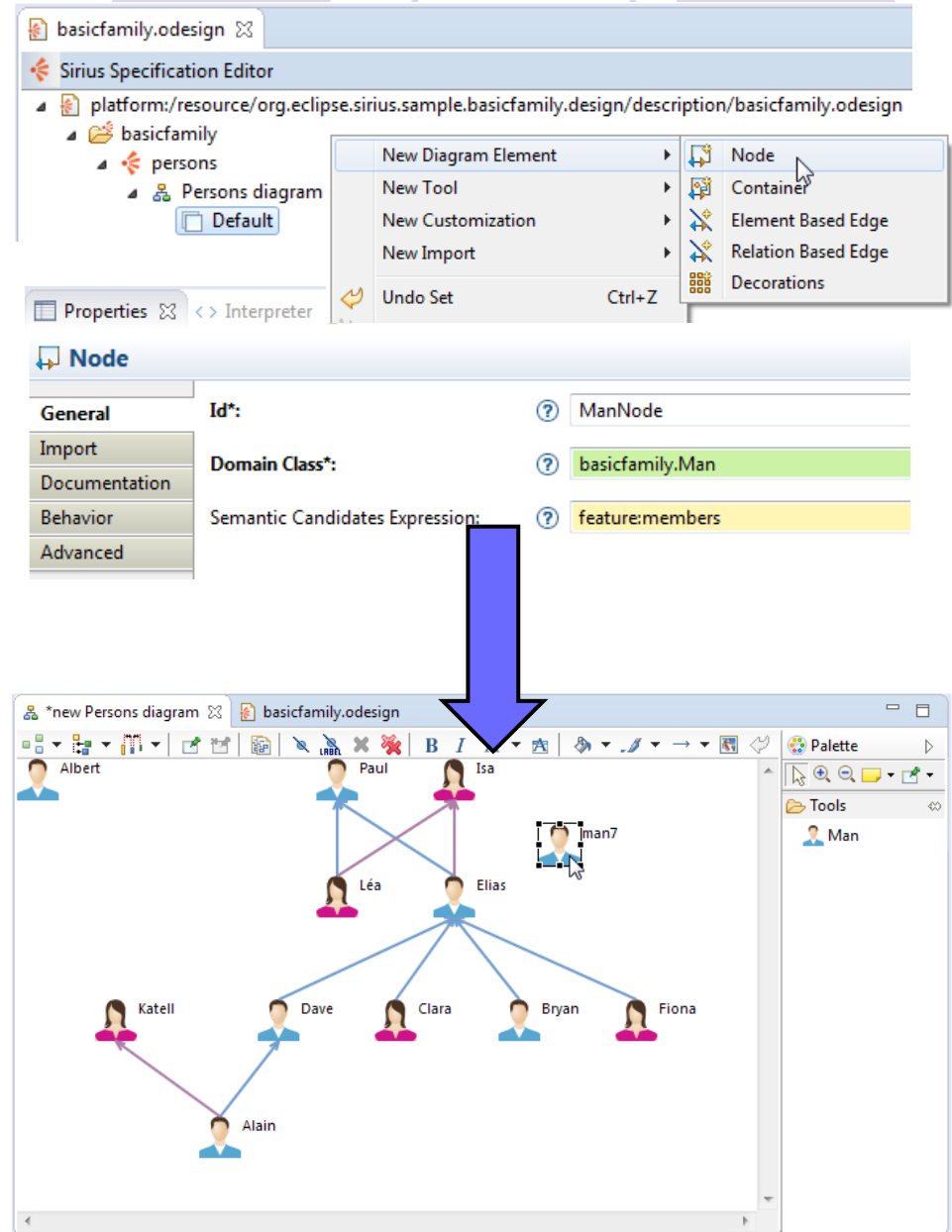
        // add object to diagram
        getDiagram().eResource().getContents().add(newPurchaseOrder);

        // add graphical representation of obj
        addGraphicalRepresentation(context, newPurchaseOrder);
    }
  }
  
```



Sirius

- EMF/Eclipse.
- <http://www.eclipse.org/sirius/>.
- Tutorials:
<http://www.eclipse.org/sirius/getstarted.html>.
- Easy to use; interpreted at runtime; definition is a model describing syntax, editing tools and validation rules.



Index

- Introduction.
- Types of modelling environments.
- Technologies to build DSLs.
- **Bibliography.**



Bibliography

- Domain-specific languages:
 - OOPSLA workshops on Domain Specific Languages.
 - “*Defining domain-specific modeling languages: Collected experience*”. 2004. J. Luoma, S. Kelly, J.-P. Tolvanen. OOPSLA Workshop on Domain Specific Languages.
- Visual languages:
 - Conference GT-VMT “Graph Transformation Visual Modelling Techniques”.
 - Conference IEEE VL/HCC “Visual Language / Human Centric Computing”.