MDE Assignment 3:
(Visual) Concrete Syntax

Vanessa Flügel
contact: vanessa.flugel@uantwerpen.be

October 15, 2025

1 Introduction

Up until this point, in the assignments, we haven't been very explicit about the distinction between (textual/visual) concrete syntax and abstract syntax. Nevertheless, we have already encountered many different concrete syntaxes, as summarized in Table 1.

When creating a (meta-)model, so far, we have always used some textual syntax. It doesn't have to be this way: in model-driven engineering, many tools, such as AToMPM, Itemis Create, MetaEdit+, StateMate, TAPAAL, . . . let the user create models using a visual syntax. Meta-modeling tools, such as AToMPM, further allow the user to define their own visual syntax (choosing icons, colors, styles, etc.).

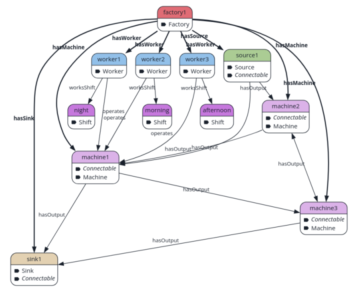
However, not all Visual Concrete Syntax is created equally: Moody gives explicit principles to design a good visual syntax [1].

- **Semiotic Clarity** 1:1 correspondence between semantics constructs and graphical symbols
- **Perceptual Discriminability** visual symbols should be easily and accurately distinguishable from each other (e.g., circle and rectangle: easy vs. rectangle and square: hard)
- **Semantic Transparency** use visual symbols, colors, . . . that suggest their meaning (e.g., green = good/safe, red = bad/danger)
- **Manageable Complexity** use mechanisms to keep complexity of diagrams manageable (e.g., use visual containment for hierarchical relationships, and edges for hierarchy-crossing or cyclical relationships)
- **Cognitive Integration** mechanisms to help the reader assemble information from separate diagrams into a coherent mental representation of the system
- **Visual Expressiveness** use the full range (extremes) and capacities of visual variables (e.g., black and white is better than different shades of gray). Concepts that have (sufficiently) different meaning, should look sufficiently different.
- **Dual Coding** use text to complement graphics, but not to distinguish graphical symbols
- **Graphic Economy** the number of different graphical symbols should be cognitively manageable
- **Cognitive Fit** use different visual dialects for different tasks and audiences (e.g., intuitive symbols for novice vs. more abstract symbols for expert)

<table>
<tr><th>Formalism</th><th>Textual Concrete Syntax</th><th>Visual Concrete Syntax</th></tr>
<tr><td>Refinery</td><td>class Factory {
    contains Machine[1..*] hasMachine
    contains Worker[1..*] hasWorker

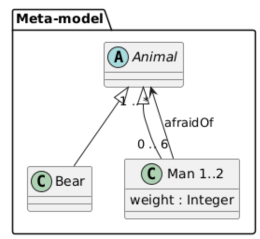
    contains Source[1] hasSource
    contains Sink[1] hasSink
}

abstract class Connectable {
    Connectable[0..2] hasOutput opposite hasInput
    Connectable[0..2] hasInput opposite hasOutput
}

class Source extends Connectable.
class Sink extends Connectable.</td><td>
(non-editable)</td></tr>
<tr><td>PlantUML</td><td>package "Meta-model" {
class "Bear " as 00000000_0000_0000_0000_00000000048a {
}
abstract class "Animal " as 00000000_0000_0000_0000_00000000046d {
}
class "Man 1..2" as 00000000_0000_0000_0000_000000000498 {
  weight : Integer
}

00000000_0000_0000_0000_00000000046d <|--
00000000_0000_0000_0000_00000000048a
00000000_0000_0000_0000_00000000046d <|--
00000000_0000_0000_0000_000000000498

00000000_0000_0000_0000_000000000498 "0 .. 6" --> "1 .. *"
00000000_0000_0000_0000_00000000046d : afraidOf
}

(in our case: auto-generated)</td><td>
(non-editable)</td></tr>
<tr><td>muMLE:
Class
Diagrams
(CD)</td><td>meta-model
CD-syntax
abstract class Animal

class Bear (Animal)

class Man [1..2] (Animal)

meta-model
OD-syntax
Animal:Class {
    abstract = True;
}

Bear:Class
:Inheritance (Bear -> Animal)

Man:Class {
    lower_cardinality = 1;
    upper_cardinality = 2;
}
:Inheritance (Man -> Animal)

two concrete syntaxes, one model</td><td>N/A</td></tr>
<tr><td>muMLE:
Object
Diagrams
(OD)</td><td>meta-model
OD-syntax
Animal:Class {
    abstract = True;
}

Bear:Class
:Inheritance (Bear -> Animal)

Man:Class {
    lower_cardinality = 1;
    upper_cardinality = 2;
}
:Inheritance (Man -> Animal)

model
OD-syntax
george:Man {
    weight = 15;
}
billy:Man {
    weight = 100;
}
bear1:Bear
bear2:Bear
:afraidOf (george -> bear1)
:afraidOf (george -> bear2)

one concrete syntax, two models</td><td>N/A</td></tr>
</table>

Table 1: Concrete syntaxes seen so far

2 Overview of Assignment

2.1 Tasks

The main goal of this assignment is to apply Moody's principles in two distinct ways. You will judge existing examples for Visual Concrete Syntax and create your own.

1. Observe the examples given in section 3. Judge each of them based on Moody's principles and explain your answer. *(Note: Not all principles will be applicable in the same way; Choose the ones that the language either does well or violates.)*

2. Create your own Concrete Visual Syntax for the role-playing game (RPG) from last week's assignment. Motivate your choices using Moody's principles. You are free to choose how to create the syntax, for example drawing on a piece of paper or using digital tools such as `https://app.diagrams.net`. In Table 2, you can find an overview of the "visual expressiveness" permitted by diagrams.net.

Write a short report including your answers and your Visual Concrete Syntax. Remember to justify your answers and explain how you arrived at your solution!

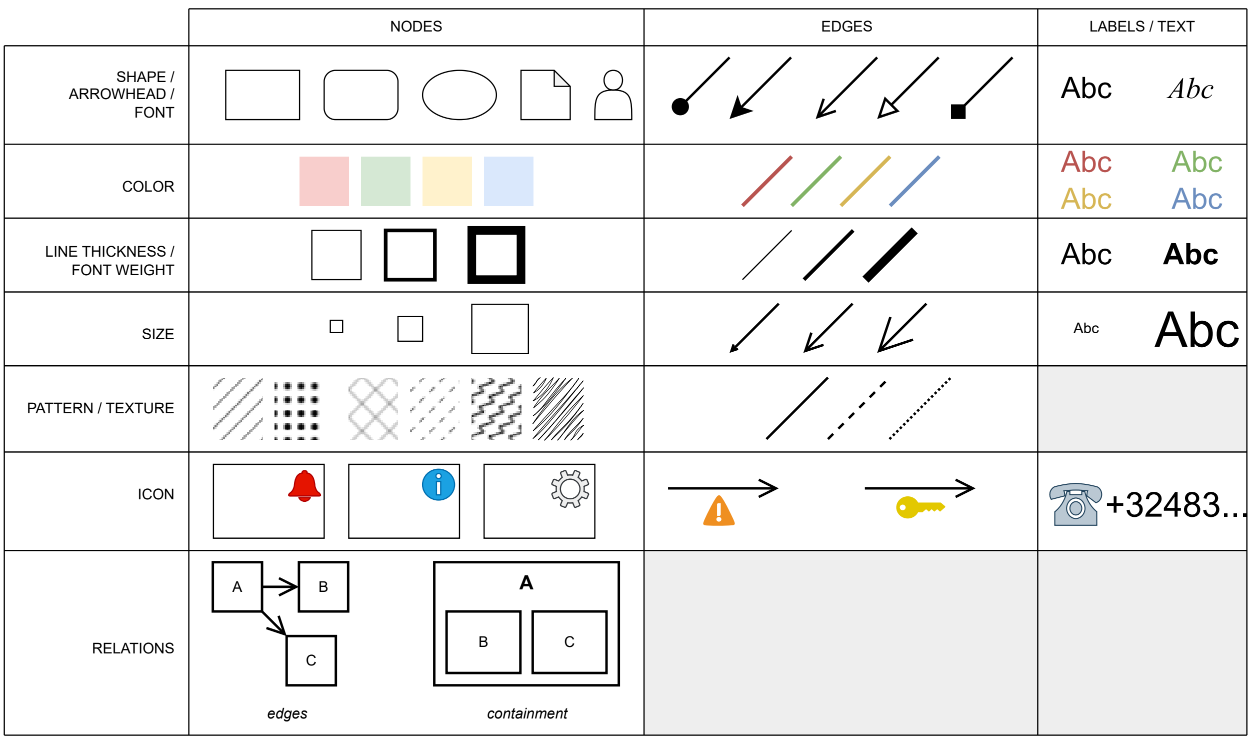


Table 2: Dimensions of visual variability

2.2 Practical

- Students work in pairs.
- One team member submits a ZIP file containing your report and Visual Concrete Syntax (Screenshots or pictures/scans).
- Deadline: 22 October 2025, 23:59.

3 Visual Concrete Syntaxes

(1) Insurance Specification Figure 1. A DSL to define insurance products.

(2) Scratch Figure 2. A DSL to define small, executable computer programs. See `https://scratch.mit.edu/projects/editor/` to see more examples.

(3) Unreal Engine Blueprint Figure 3. A DSL for visual scripting. See `https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine#generalscripting` for more information.

(4) PureData Figure 4. A DSL to define electronic music. See `https://puredata.info/docs/StartHere/` for more examples.

(5) Max9 Figure 5. Another DSL to define electronic music. See `https://cycling74.com/products/max` for more examples.

(6) Param-O Figure 6. A DSL to create 3D-objects. See `https://graphisoft.vn/en/news/what-is-param-o/` for more examples.

(7) Houdini Figure 7. Another DSL for 3D-objects and animation. See `https://www.sidefx.com/products/houdini/` for more information (though not many more examples of the syntax are shown).

(8) Simulink Figure 8. A DSL to define simulations. See `https://nl.mathworks.com/products/simulink.html` for more examples.

References

[1] Daniel L. Moody. The "physics" of notations: a scientific approach to designing visual notations in software engineering. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 485–486. ACM, 2010.

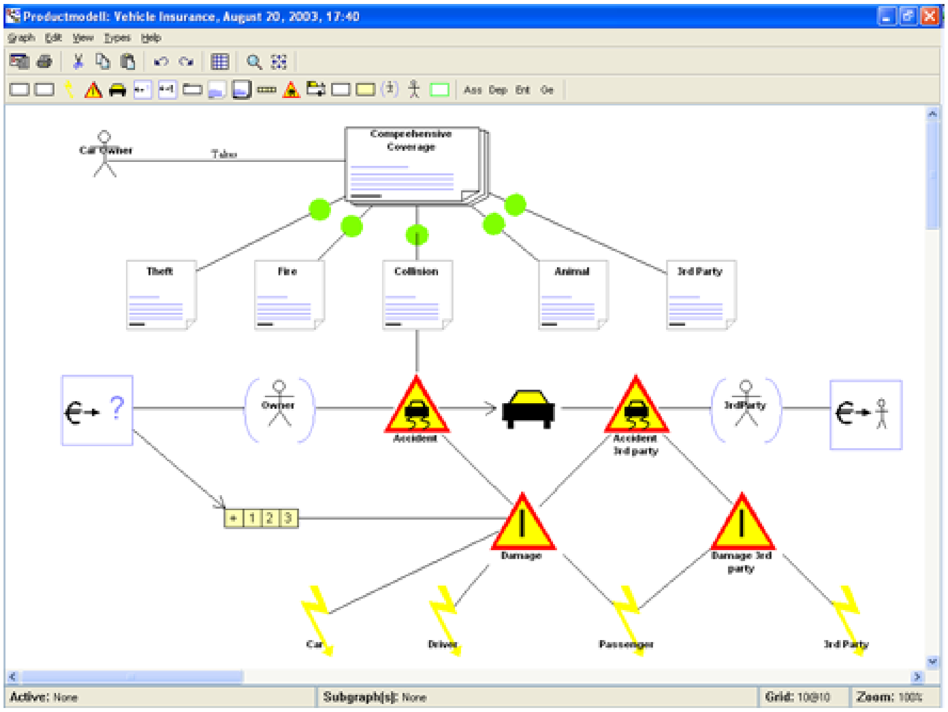


Figure 1: Insurance Products

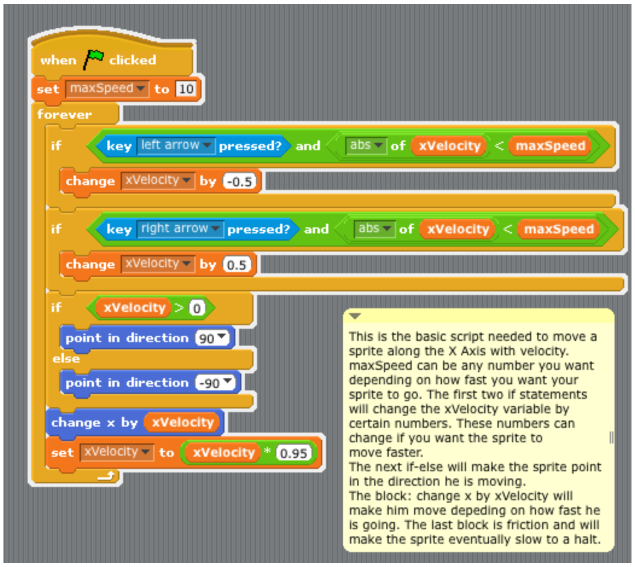


Figure 2: Scratch

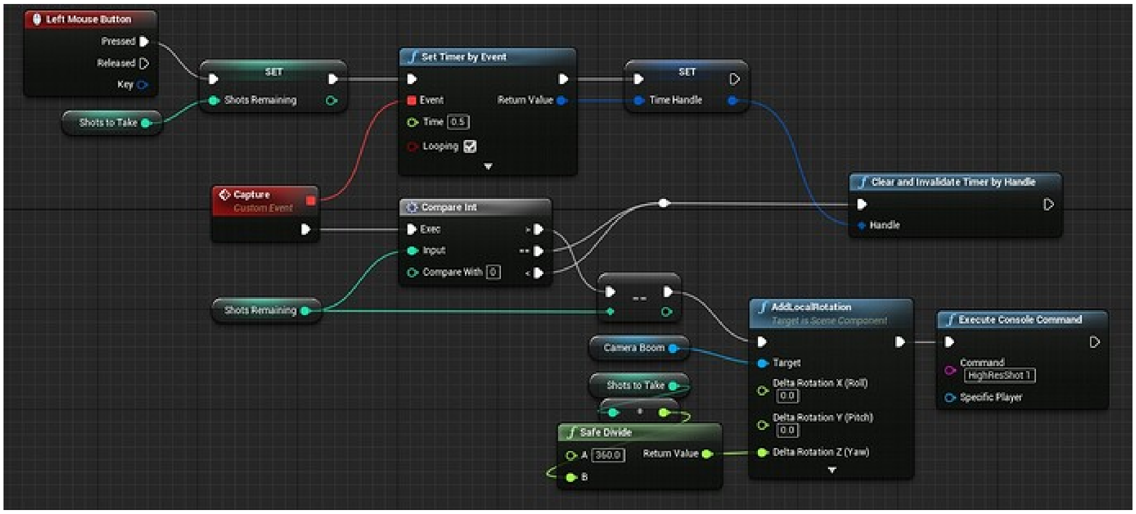


Figure 3: Unreal Engine Blueprint

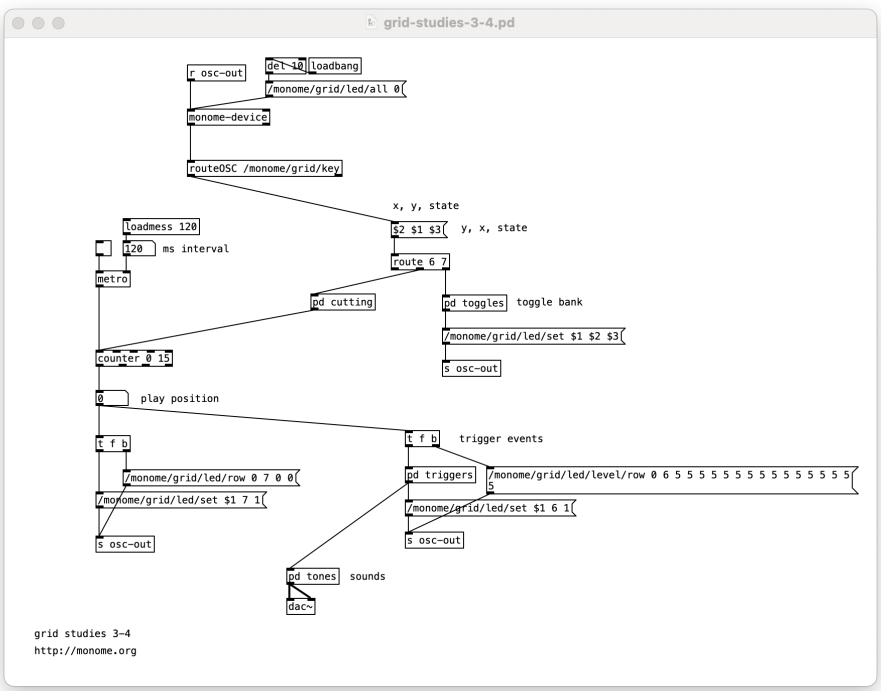


Figure 4: PureData

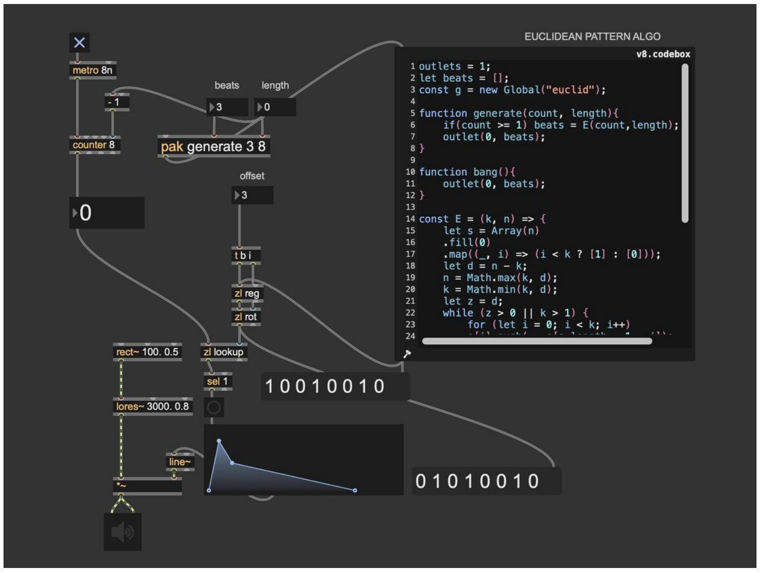


Figure 5: Max9

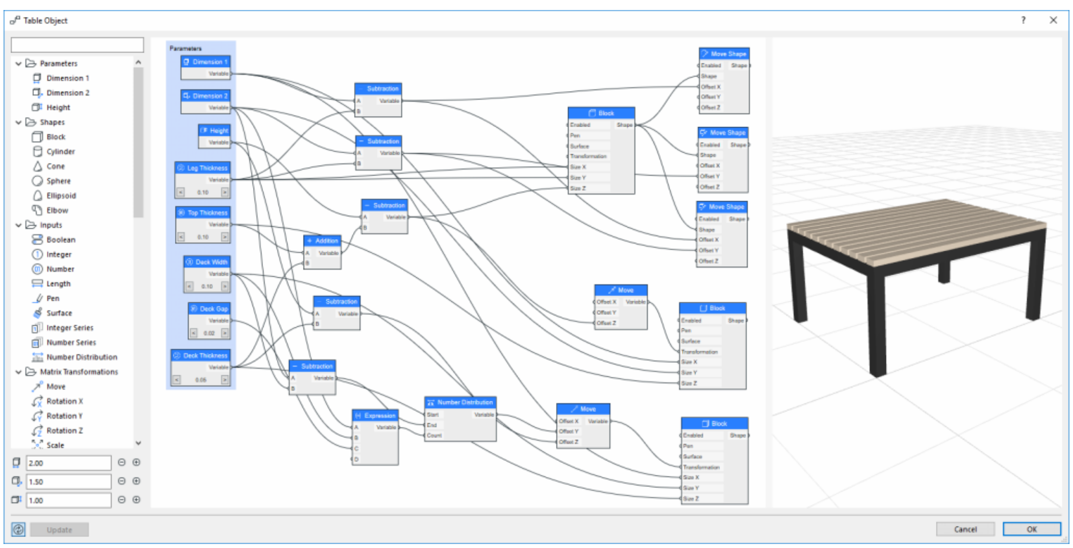


Figure 6: Param-O

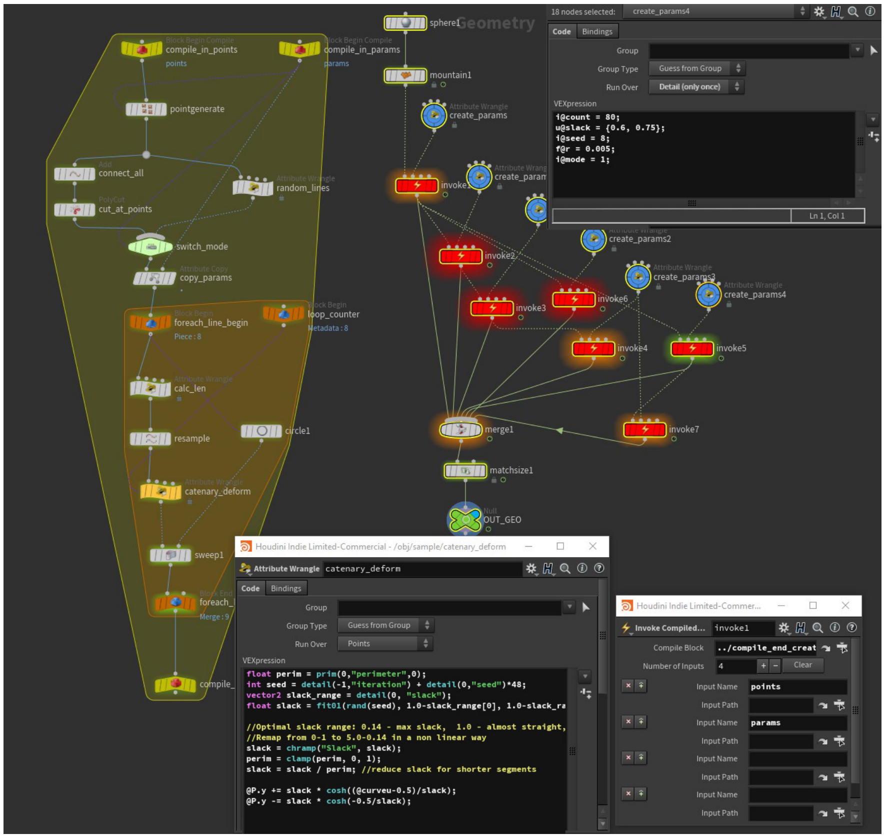


Figure 7: Houdini

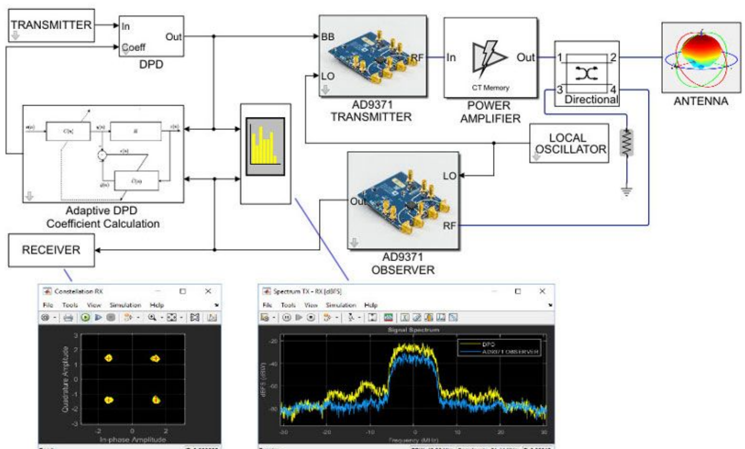


Figure 8: Simulink