March 2007 Meeting

D.H.Akehurst

iemn Recherche C.N.R.S.

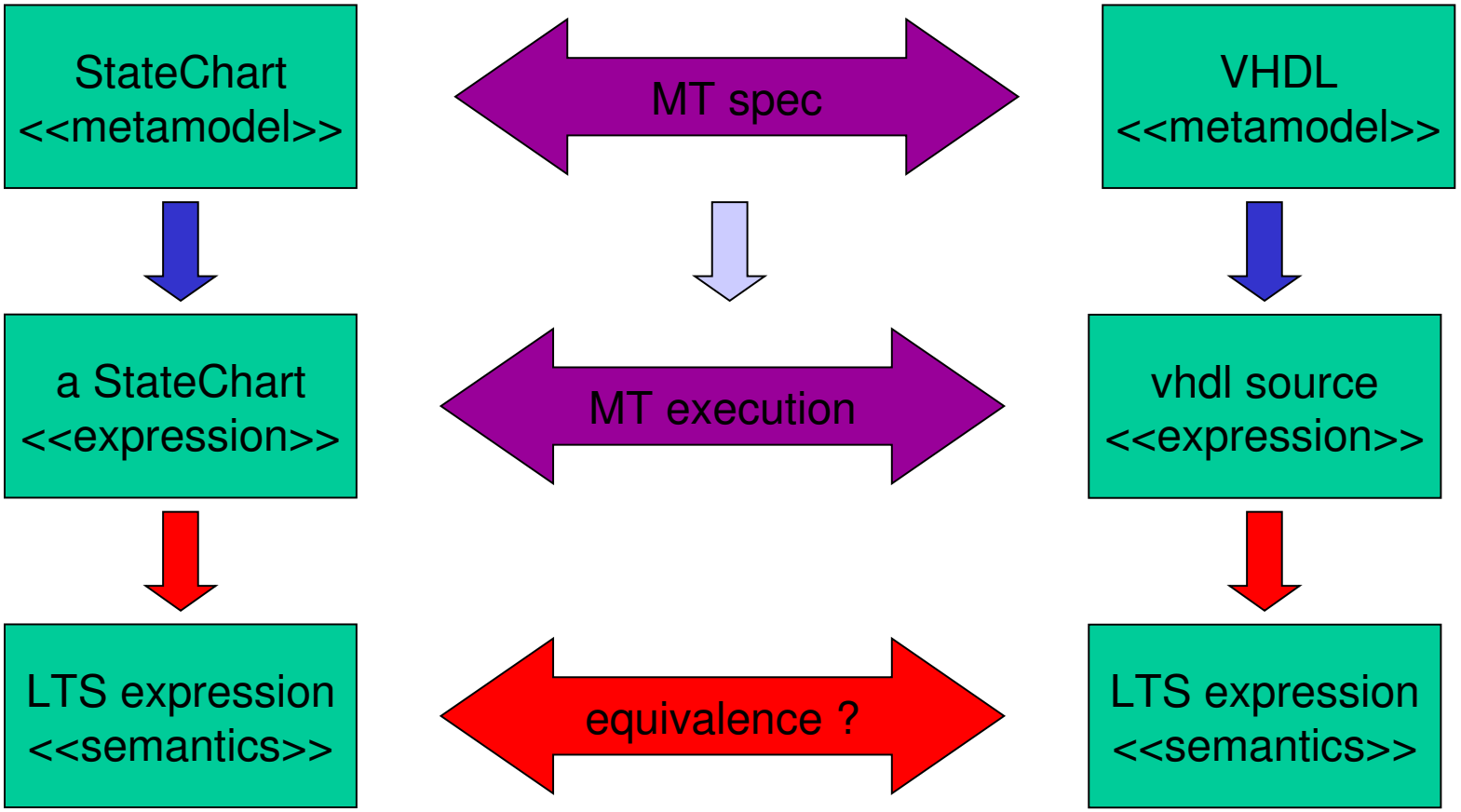
University of Kent

INRIA FUTURS

Introduction

- Verifying Model Transformations
- Verifying Embedded Systems
- Domain Specific Languages

Verifying Model Transformations



Verifying Model Transformations

- Solving this seems to end up as trying to solve the issue of whether two programs are equivalent
  - This is not solvable!
    - … in the general case.
  - Never give up hope...
    - … with faith anything is possible ☺
  - We will try another, more specific, approach.

Verification of Embedded Systems

- Often verification associated with simulation
  - Supported by many synthesis tools
  - Not so much scope for research,
    - much done in commercial tools.
- Formal Verification
  - Does system meet a precise requirement!
  - Proving or disproving the correctness of a system with respect to a certain property, using formal methods of mathematics
  - Much scope for research

Two alternatives for FV

- Transform input specification into another language for which verification tools/algorithms exist.
  - Can be difficult/messy to express some concepts in another language.
  - Problem converting results back into user language
- Re-implement verification algorithms/tools for existing specification language.
  - Have to ‘trust’ new implementations/algorithms
- I have done the first before, and think it is worth looking at the second.
  - TA tools have state space issues
  - I will look at Timed Binary Decision Diagrams

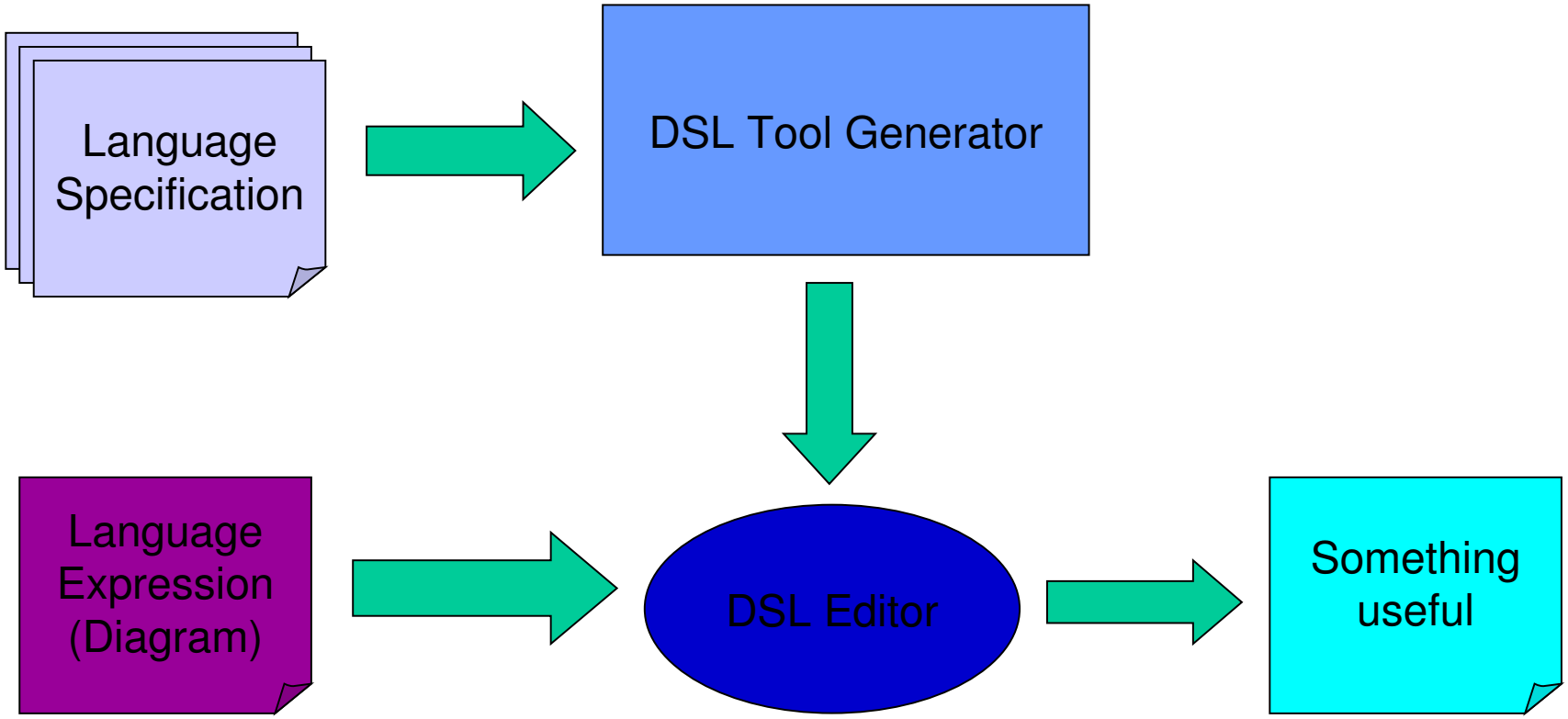
Progress in this area

- We have a good understanding of the underlying issues
- Choosing preferred approach critical
- any input from the project team would be welcomed?

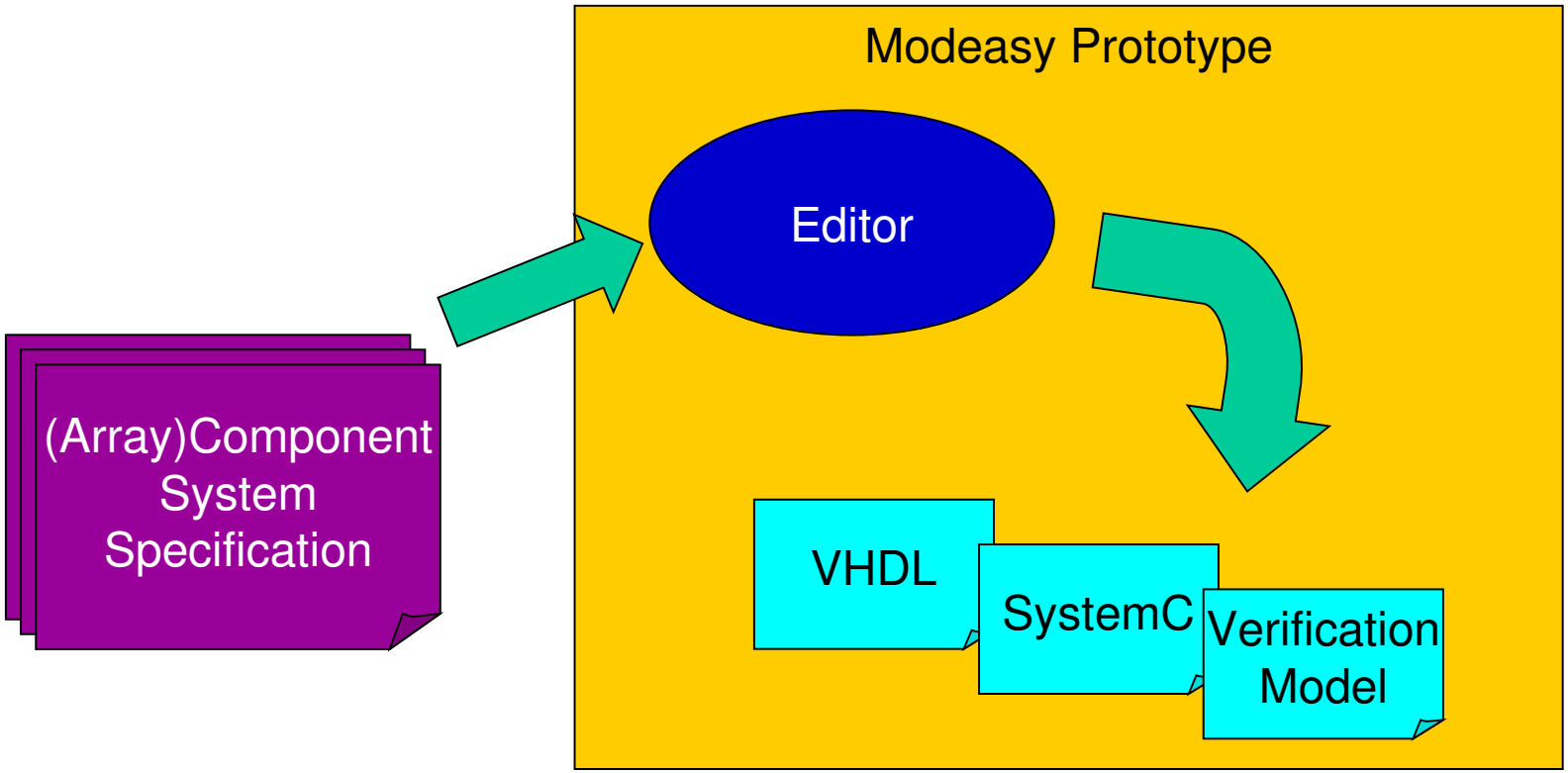
Domain Specific Languages

- Why are DSLs relevant to Modeasy?
  - Because I find them interesting ☺ !
  - Actually, mainly due to Array-OL
  - There is no good tool support for Array-OL
    - not in an integrated way with UML.
  - Even VHDL and SystemC are a little lacking in comparison with Java or .Net languages.
    - certainly in Open source community
  - To build prototype tools, quickly and efficiently, for new specification languages
    - we need some support from DSL tool generators

DSL Tool Generators



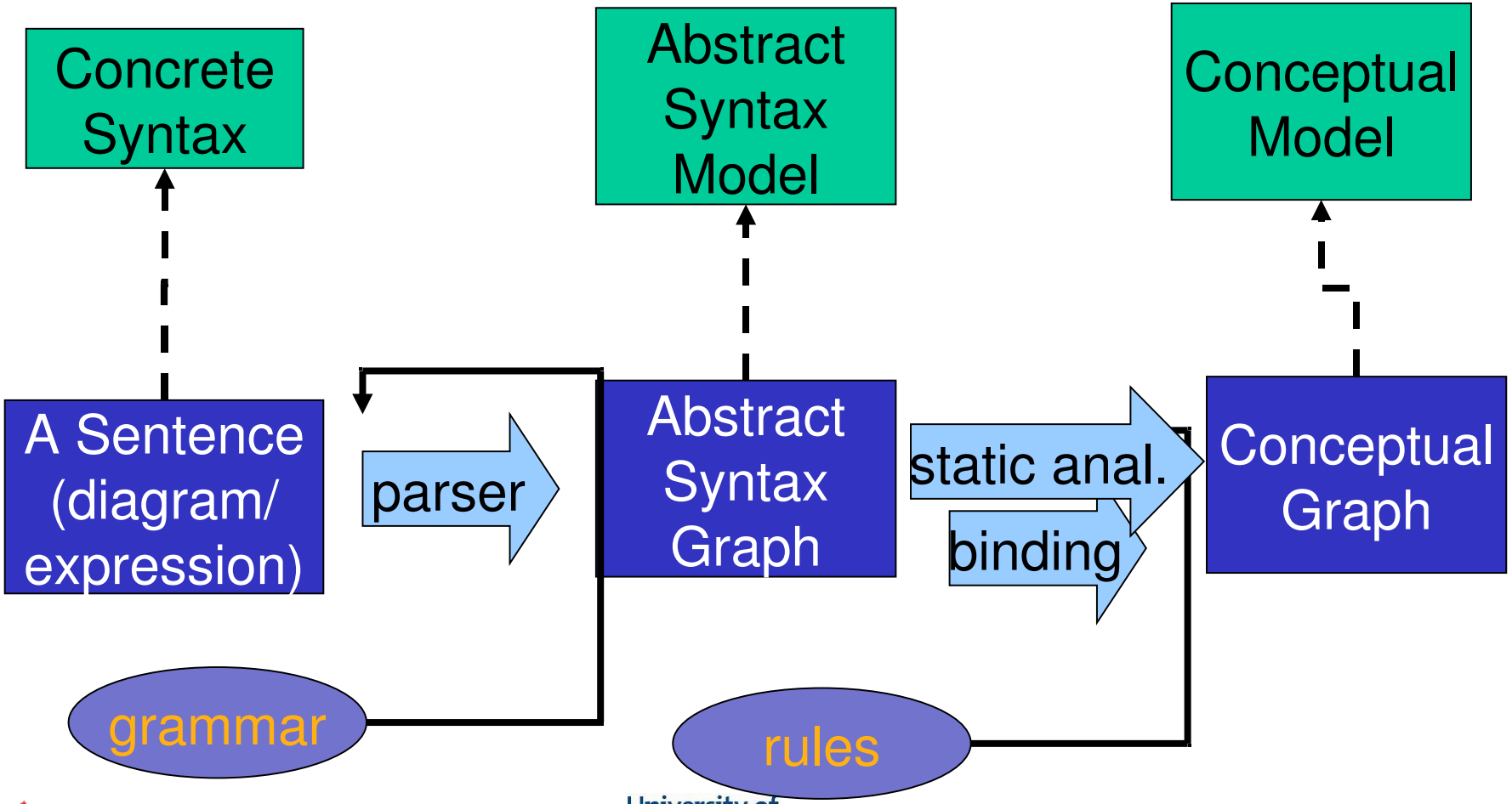
For Modeasy


Modeasy Prototype
Editor
(Array)Component
System
Specification
VHDL
SystemC
Verification
Model

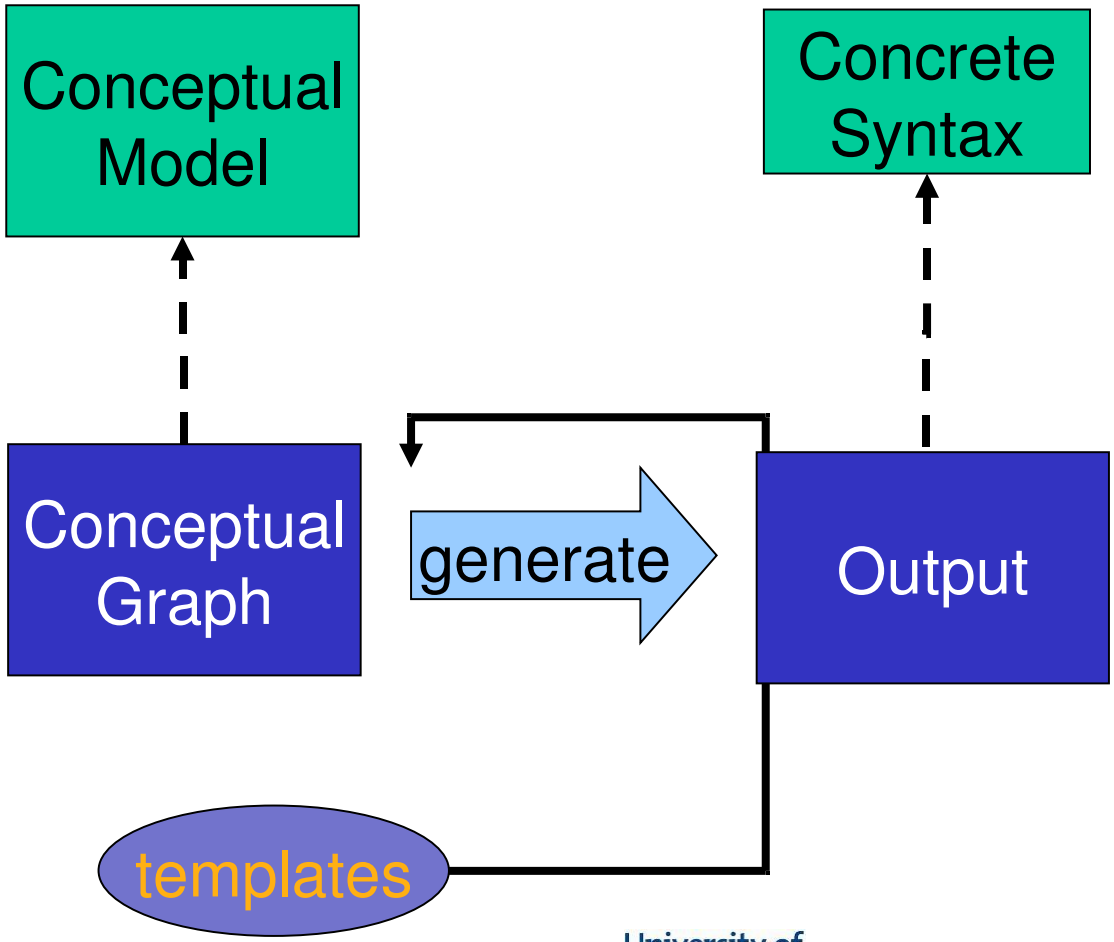
Designing Domain Specific Languages

- Is a skill
- Simple things are (normally) simple
- Complex languages take skill, engineering, design, iterations to get them “correct”
- However, basically consist of three parts
  - Concrete Syntax (text or graphics)
  - Conceptual Model (concepts in the language)
  - Target – what do you want to do with the language
    - Execute, synthesis (to other language), ….etc.

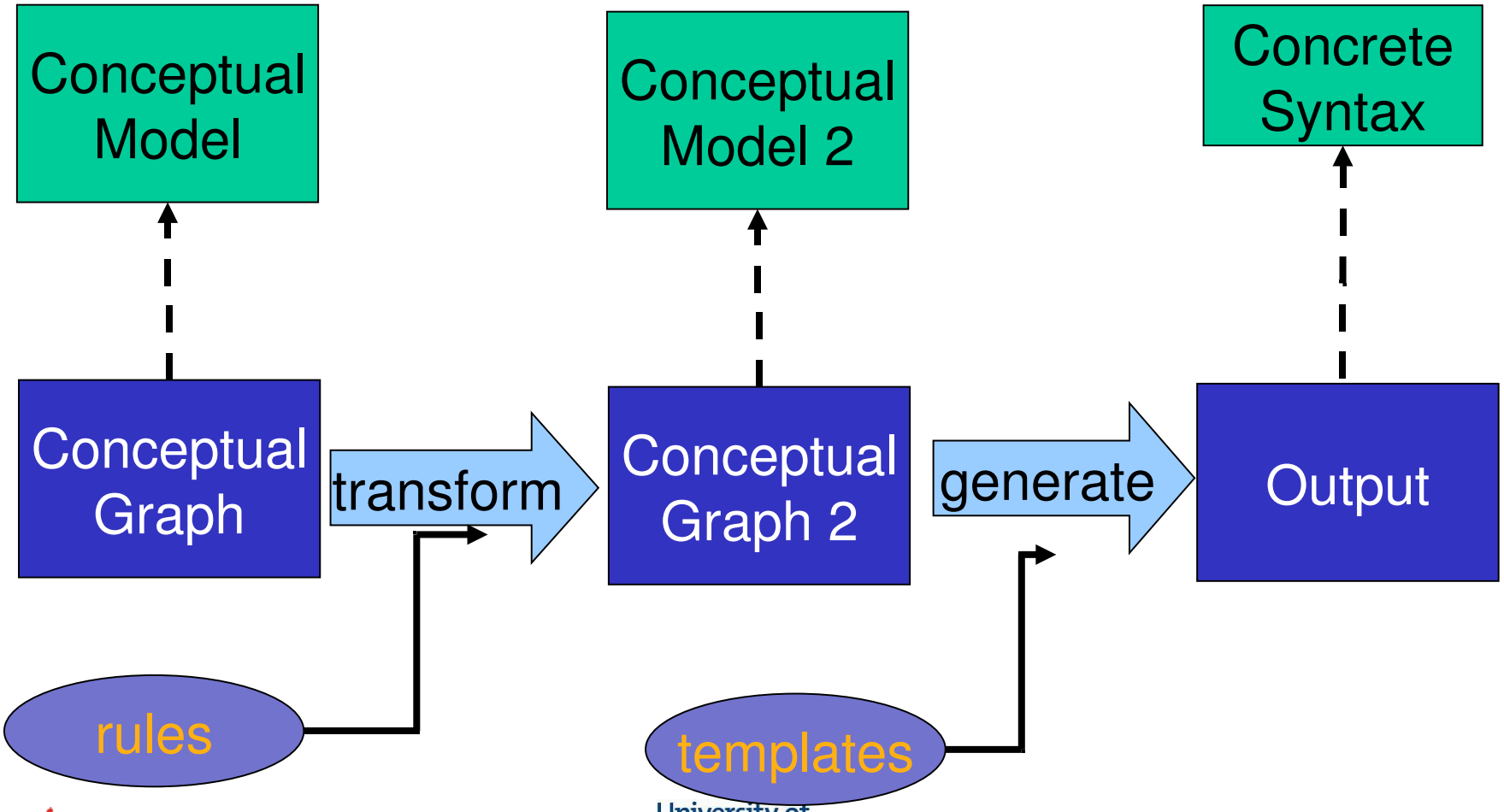
What is a Language Specification?



Code Generation



Code Generation 2



Text vs Graphical Languages

- Text languages
  - Well understood, well used
  - Compiler theory well documented
  - OCR, Lexical analysis, Parsing, Abstraction, Binding, Synthesis/Code Generation/Execution
- Graphical Languages
  - Less well understood, often used
  - No fully accepted theory for definition

Pixels -> Symbols

- Optical Vector Graphics Recognition
  - Convert pixels in to basic vector graphic symbols
  - Rectangles, lines, circles, polygons, etc
  - Requires definition of symbols to recognise
- Parallel with Text
  - OCR
  - Text editors provide characters (ASCII) as output
- Editors
  - Easy to build editors that produce symbols as output.
  - Vector Graphic Diagram Editors / Text Editors (notepad!)

Symbols -> Tokens

- Lexical Analysis
  - Output: Token Graph (spatial relationship graph)
  - Construct more complex symbols from simpler ones
  - relationships between tokens
    - touches, surrounds, near, above, below, toLeft, toRight
  - Requires Lexical rules
- Parallel with Text
  - Lex, and other tokenisers.
  - Output: Sequence of Tokens with assumed spatial relationships of previous and next.
- Editors
  - Mobile phone – predictive text

Tokens -> Nodes

- Parsing
  - Output: Parse Graph
  - constructs more complex grouping (relationships) between certain tokens
  - Requires grammar rules
- Parallel with Text
  - Output: Parse Tree
  - Yacc, Antlr, Javacc, etc
- Editors
  - Syntax directed editors – not accepted for text editing

Nodes -> abstract elements

- Abstraction
  - Output: Abstract Syntax Graph
  - abstracts away from the (concrete) symbols used in the input of the expression
  - Requires some abstraction rules!
- Parallel with Text
  - normally done by parser when using modern parser generators
- Editors
  - Code completion – very useful, widely accepted

Abstract Elements -> Model Elements

- Binding
  - Output: Instance of language metamodel
  - no relationship to syntax
    - multiple syntaxes map to same metamodel
  - Type checking typically done at binding stage
- Parallel with Text
  - not often addressed by traditional compiler theory
  - Synthesis can derive from abstract syntax tree
- Editors
  - CASE tool editors often provide operate at this level
    - model elements are added to diagrams
    - syntax elements created for the model element added

To Be published

- This is ongoing work.
- In collaboration with Anneke Kleppe.
- Hope to be submitted for publication next month.

Other In progress Publications

- Tools Europe 2007
  - Java: Hiding the loops
    - OCL like collections library
- Models 2007
  - OCL: Modularising the language
    - redesign of the standard to allow OCL to be more easily extended.
- SoSyM special Issue
  - Maths vs Metamodelling
    - Does Metamodelling bring us anything new?