Grammarware, Semantics and Modelware

Barrett R. Bryant

Department of Computer Science and Engineering

University of North Texas

Denton, Texas, USA

Introduction

- "Grammarware comprises grammars and all grammar-dependent software." [Klint, Lämmel and Verhoef, 2005]
- Semantics are concerned with the meaning of the syntax described by grammars.
- Modelware concerns the building of models and associated modeling tools for software systems (e.g., UML)

Outline

1. Component-Based Language Implementation
2. Grammar Inference of Domain Specific Languages and Domain Specific Metamodels
3. Formalization of Modeling Language Semantics

Component-Based Language Implementation

- Compiler construction vs. cooking a wedding cake
- Cooking facilities: YACC, JavaCC, CUP, ...
- Cooking complexity
  - Compiler design is known as a “dragon” task
  - Good modularity enables you to divide-and-conquer the complexity
  - As long as the pieces can be assembled together

No Decomposition of Language Definitions

Most parser generators don't support modular grammar definitions at all

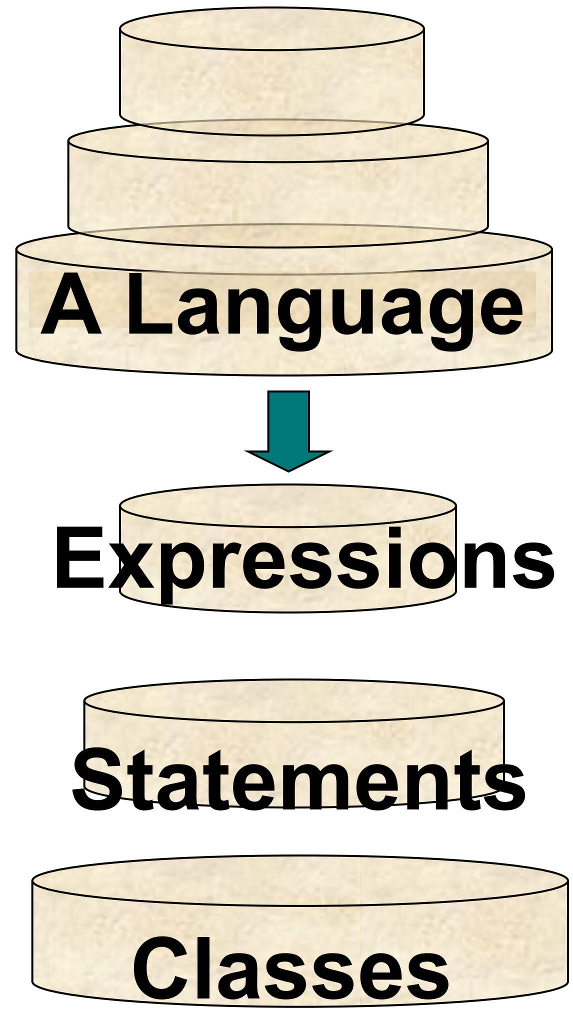
Cobol 85 is 2500 lines of specification, more than 1000 variables

Comprehensibility

Changeability

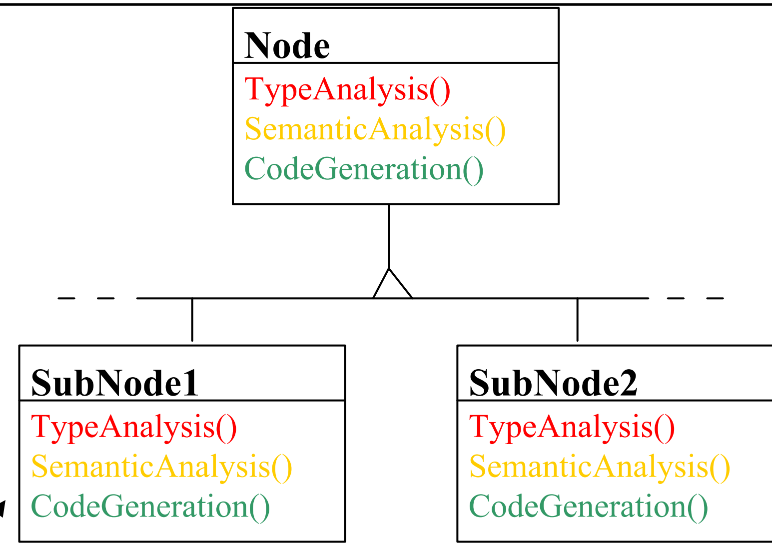
Reusability

Independent development



No Clear Separation of Compiler Construction Phases

- Syntax and semantics
  - Syntax analysis -- formal specification
  - Semantic analysis -- programming languages
  - The communication between syntax and semantics makes the specification and code tangled together
- Among different semantic phases
  - Pure object-oriented design, code scattered all over the syntax tree class hierarchy



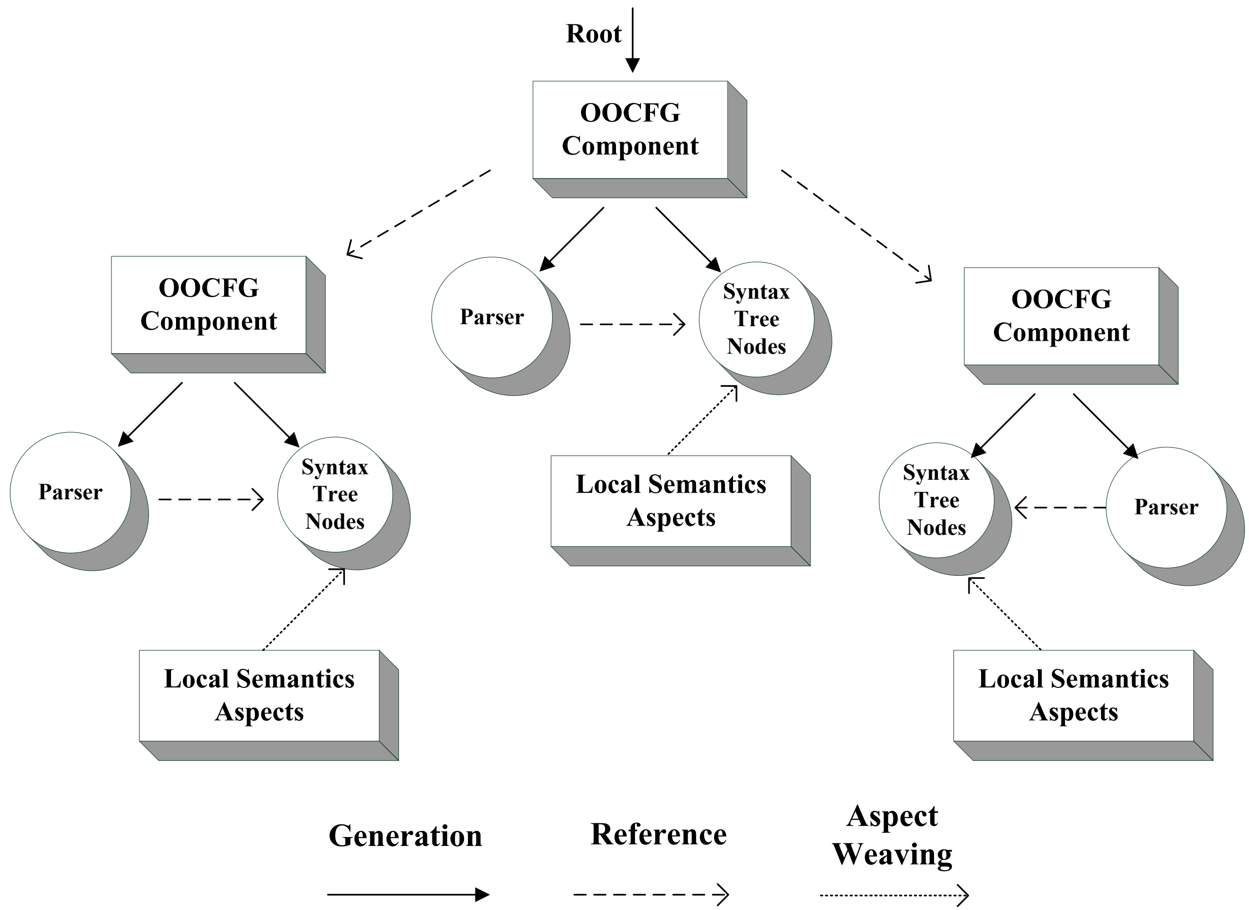
Hard to maintain and evolve!!

Framework Overview

Structure

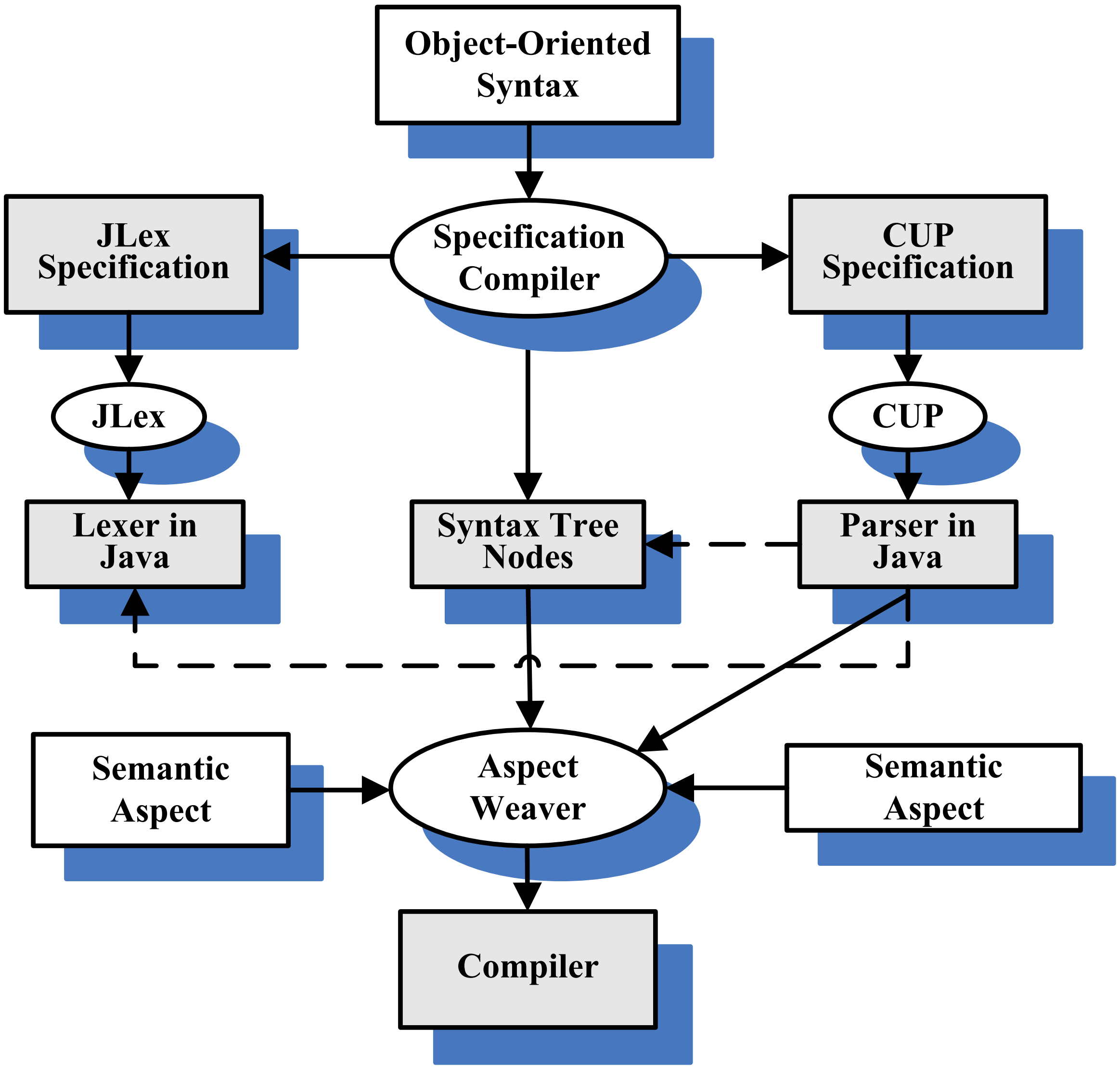
Function

Component-based LR (CLR) parsing decomposes a large language into a set of smaller languages



Object-Oriented Syntax (OOS) and Aspect-Oriented Semantics (AOS) facilitate separation of different phases

OOS + AOS Implementation

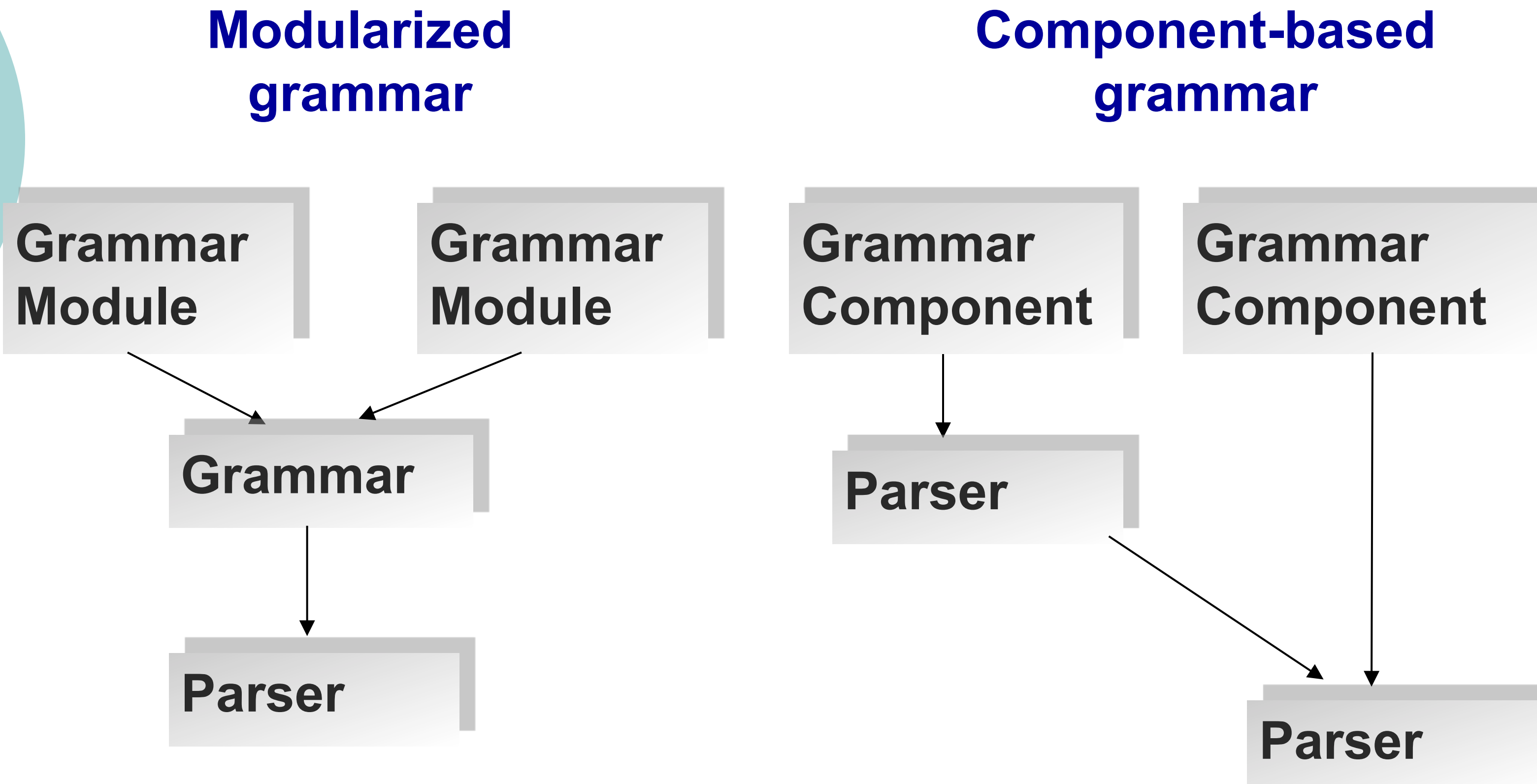


Contribution

- CLR decreases the development complexity by reducing the granularity of a language
  - Syntax composition at the parser level ➔ reduced coupling between grammar modules
  - More expressive than regular LR parsing
- OOS + AOS isolates syntax and semantics as well as semantic phases themselves into different modules
  - Separation of declarative and imperative behavior
  - Separation of generated code and handwritten code
  - OOS - generation of both parser and syntax tree
  - AOS - transparent to node classes, flexible in tree walking and phase composition.

The overall paradigm increases the comprehensibility, reusability, changeability, extendibility and independent development ability of the syntax and semantic analysis with less development workload required from compiler designers.

Component-Based Grammar vs. Modularized Grammar



Code-level composition, less coupled definition, smaller parsing table, multiple lexers, etc ...

Software Engineering Benefits

Comprehensibility

Intertwined symbols and productions are reduced

Changeability

Changes are isolated inside individual components
Only local recompilation needed

Reusability

Components can be plugged and played

Independent development

Dependencies are handled at the code-level
instead of the grammar level

Aspect-Oriented Semantics Implementation

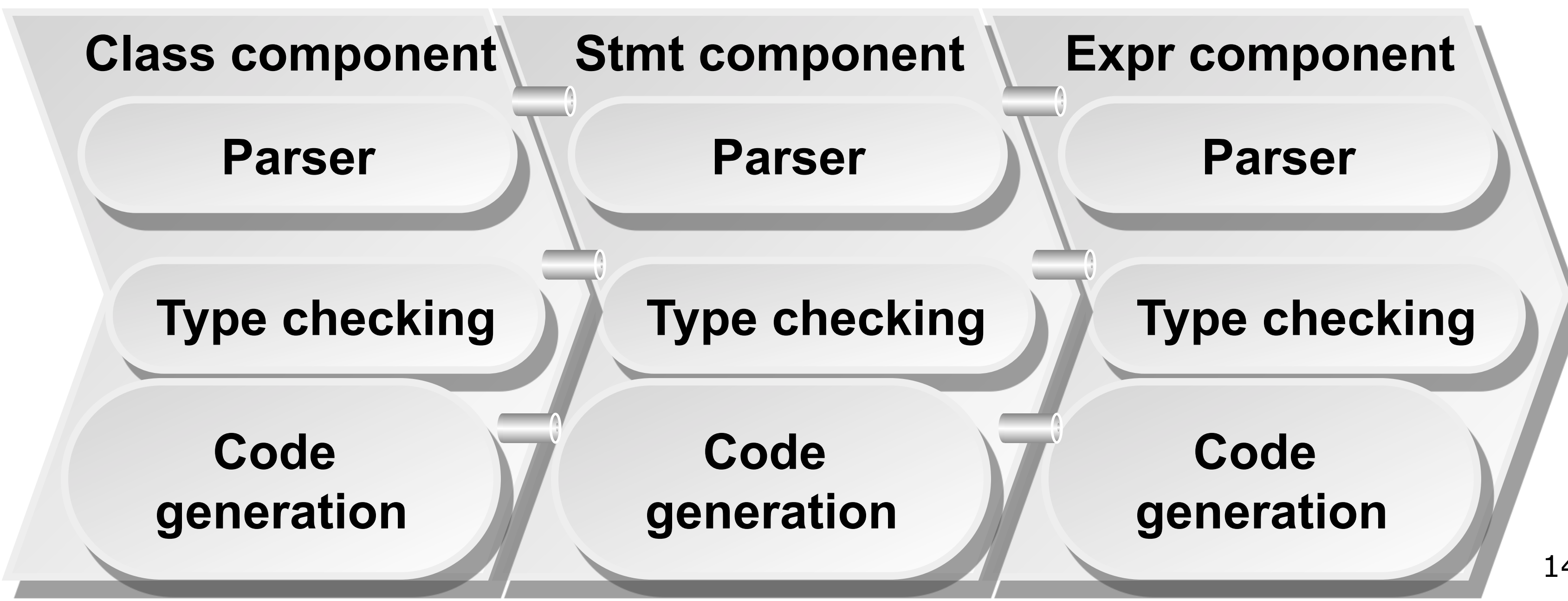
- Each semantic concern is modularized as an aspect
  - An independent semantic pass
  - A group of action codes
- Semantic pass
  - Implemented as introductions to the syntax tree classes
- Crosscutting actions applied to a group of nodes
  - Weaved into syntax tree classes as interceptions

AOS Advantages

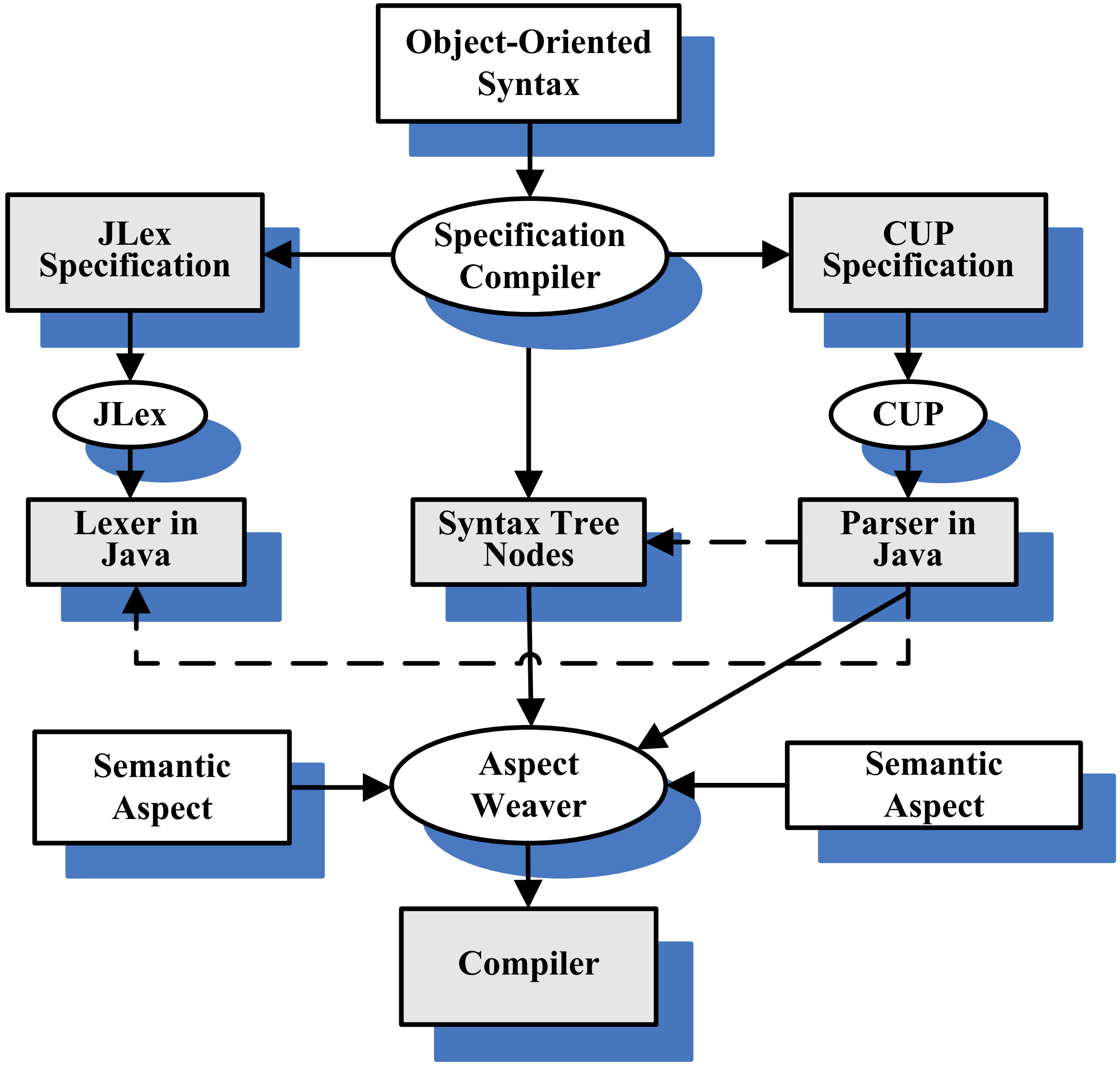
- Aspect-orientation can isolate crosscutting semantic behavior in an explicit way
  - Each semantic aspect can be freely attached to (generated) AST nodes without “polluting” the parser or AST node structure.
  - Different aspects can be selectively plugged in for different purposes at compile time.
  - Since each aspect is separated with other aspects, developers can always come back to the previous phase while developing a later phase.

Integration with CLR Parsing

- Syntax specification ➔ The restrictions of OOS can be applied to CCFG without generating any side-effects
- Syntax tree construction ➔ CLR's parse tree generation process is inlined with OOS tree generation
- Semantic analysis ➔ Semantic composition follows syntax tree composition



OOS + AOS Implementation



Outline

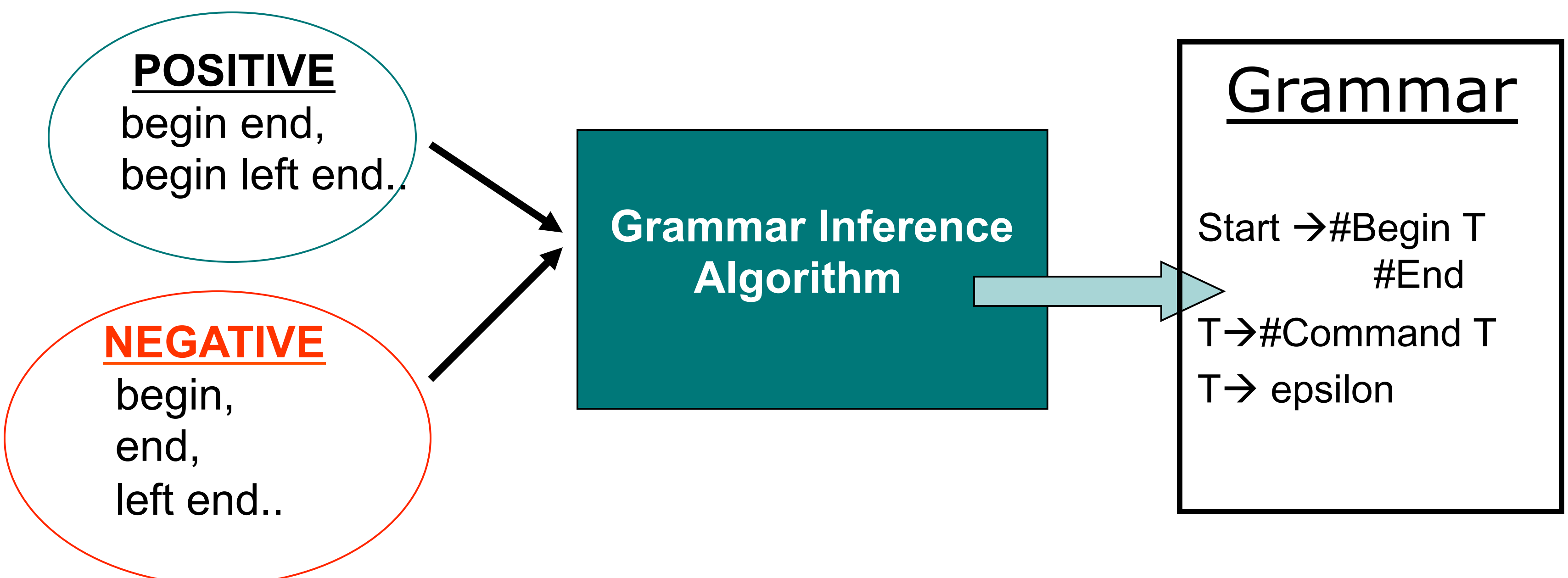
1. Component-Based Language Implementation
2. Grammar Inference of Domain Specific Languages and Domain Specific Metamodels
3. Formalization of Modeling Language Semantics

Grammar Inference of Domain Specific Languages and Domain Specific Metamodels

- Assist a Domain-Specific Language (DSL) developer in developing the DSL implementation from examples
- Recover metamodels from model instances which have become orphaned through software evolution, and facilitate evolution of those orphaned models

Grammar Inference : An Overview

- **Machine learning technique:** learning from examples.
- **Complete information:** Positive + Negative samples
- **Positive information:** Positive samples only
- **Characteristic sample set:** samples which exercise every rule of a grammar



Inference of Domain-Specific Language Specifications from DSL Programs

- **Open problem:** Expedite DSL development by incorporating the description-by-example paradigm of language development
- Domain experts are not knowledgeable about computer science and/or programming language development so they are not able to design and implement DSLs.
- However, they can write examples of the types of domain specific programs they would like to write.

MAGIc

- Memetic algorithms are evolutionary algorithms with local search operator
  - use of evolutionary concepts (population, evolutionary operators)
  - improves the search for solutions with local search

MAGIc

```
print id where
id=num
print num+id
where id=num
```

```
print a where
c=2
print 5+b
where b = 10
```

MAGIc

print id where id=num
print num+id where id=num

But where to change the grammar?

Apply diff command!
1a2,3
> num
> +

MAGIc

Configurations returned from the LR(1) parser:

$Nx \rightarrow \alpha_1 \bullet \alpha_2$

$Ny \rightarrow \beta \bullet$

$Nz \rightarrow \bullet \gamma$

Use information from LR(1) parsing on 2nd sample.

MAGIc

N1 ::= print N3 N2 where id = num
N2 ::= id
N3 ::= num +
N3 ::= ε

print a where c=2
print 5+b where b = 10

N1 → print • N2 where id = num

N1 ::= print N2 where id = num
N2 ::= id

Results - 12 Input Samples of DESK Language

1. print a
2. print 3
3. print b + 14
4. print a + b + c
5. print a where b = 14
6. print 10 where d = 15
7. print 9 + b where b = 16
8. print 1 + 2 where id = 1
9. print a where b = 5, c = 4
10. print 21 where a = 6, b = 5
11. print 5 + 6 where a = 3, c = 14
12. print a + b + c where a = 4, b = 3, c = 2

Results – DESK Language

Original grammar:

1. DESK ::= print E C
2. E ::= E + F
3. E ::= F
4. F ::= id
5. F ::= num
6. C ::= where Ds
7. C ::= ε
8. Ds ::= D
9. Ds ::= Ds , D
10. D ::= id = num

Inferred grammar:

1: NT1 -> print NT3 NT5
2: NT2 -> + NT3
3: NT2 -> ε
4: NT3 -> num NT2
5: NT3 -> id NT2
6: NT4 -> , id = num NT4
7: NT4 -> ε
8: NT5 -> where id = num NT4
9: NT5 -> ε

Results – abc Languages

Olgierd Unold, Marcin Jaworski. Learning context-free grammar using improved tabular representation. Applied Soft Computing 10 (2010) 44–52

CFG for two languages have been inferred after 900 (600) generations:
L1 = $\{a^n b^n c^m \mid n,m > 0\}$
L2 = $\{ac^n \cup bc^n \mid n > 0\}$

MAGIc infered following CFGs after 6(5) generations:

NT1 ::= a NT2 b NT3
NT2 ::= a NT4
NT2 ::= ε
NT3 ::= c NT3
NT3 ::= c
NT4 ::= NT2 b

NT1::= NT3 NT2
NT2 ::= c NT2
NT2 ::= c
NT3 ::= a
NT3 ::= b

Results - DSL for Hypertree Description

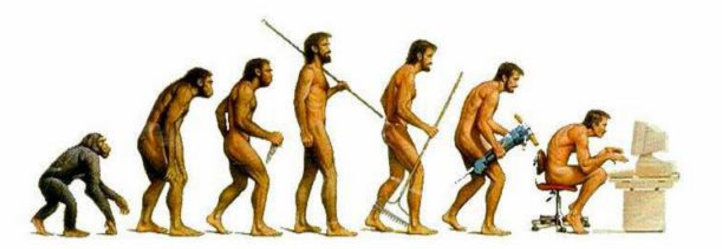
```
RESOLUTION 300 400 300
ITERATIONS 3000000
POINTINIT 0 0 0
TREEDEPTH 5
BRANCHDEPTH 1
HYPERVOLUME -0.6 0.6 -1 0.6 -0.6 0.6

DEPTHCOLOR 0-1 0.7+/-0.0 0.7+/-0.0 0.5+/-0.0
DEPTHCOLOR 2-5 0.25+/-0.25 0.75+/-0.25
     0.25+/-0.25
TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SHEAR (0,0,0) (0.5,0.5,0.5) (2,2,2) SHEAR_XZ
SCALE (0.3,0.3,0.3) (0.4,0.4,0.4) (0.3,0.3,0.3)
ROTATE (-80,-80,-80) (0,0,0) (0,0,0)
ROTATE (0,0,0) (45,45,45) (0,0,0)
TRANSLATE (0,0,0) (-0.72,-0.72,-0.72) (0,0,0)

TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.6,0.6,0.6) (0.6,0.6,0.6) (0.6,0.6,0.6)
ROTATE (0,0,0) (50,50,50) (0,0,0)
TRANSLATE (0,0,0) (-0.4,-0.4,-0.4) (0,0,0)

TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.8,0.8,0.8) (0.8,0.8,0.8) (0.8,0.8,0.8)
ROTATE (0,0,0) (150,150,150) (0,0,0)
TRANSLATE (0,0,0) (-0.8,-0.8,-0.8) (0,0,0)

CONDENSATION 1
CONE -1.0 0.5 0.02 0.0 CONE_Y
```

Results - Inferred Grammar for Hypertree Description DSL

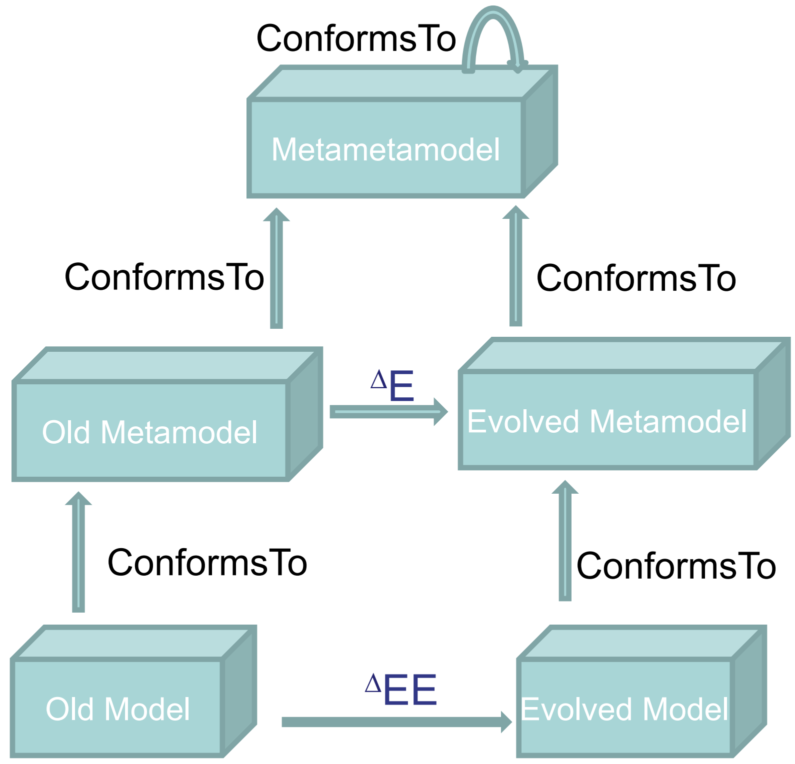
```
NT1 -> #resolution NT2 #iterations #num NT3 NT2 #treedepth #num #branchdepth #num
       #hypervolume NT2 NT2 #condensation #num #cone NT2 #num #coney
NT2 -> #num #num #num NT4
NT3 -> #pointinit
NT3 -> #lineinit #num #num #num #num
NT4 -> #depthcolor #range #bpp #bpp #bpp NT4
NT4 -> epsilon
NT4 -> #name #progname NT4
NT4 -> #scale #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #rotate #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #translate #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar NT4
NT4 -> #transform #num #num NT4
NT4 -> #shear #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #rpar #shearxz NT4
NT4 -> #perturb #lpar #num #comma #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #comma #num #rpar
       #lpar #num #comma #num #comma #num #comma #num #rpar NT4
```

Model Co-Evolution

- Metamodel Evolution (ΔE)
- Model Co-Evolution (ΔEE)



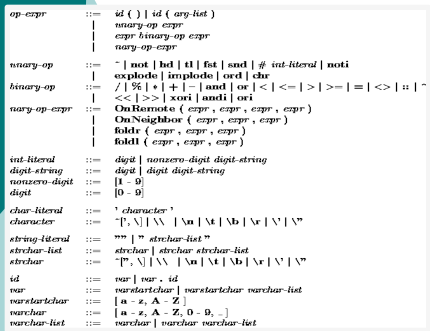
Evolution!



Metamodel Drift

- It has been observed in practice that as a metamodel undergoes frequent evolution, previous model instances may become orphaned.
- Solution: Infer metamodel from domain models.
- *A technique is developed to recover the metamodel schema definition from orphaned instances, which is semi-automated, grammar driven, and uses grammar inference concepts.*

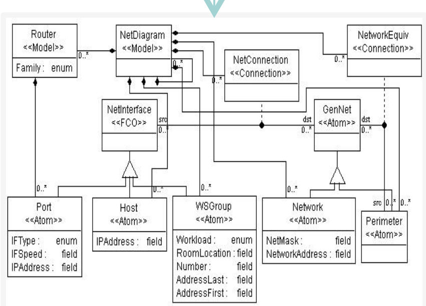
Grammar Inference Applied in Domain-Specific Modeling



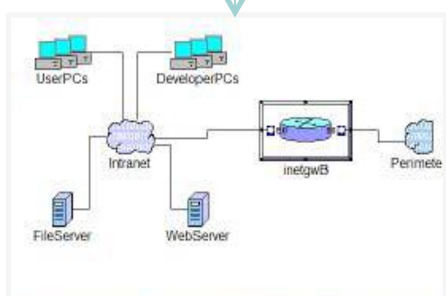
Grammar

Program

Compiler



Metamodel



Model

Interpreter

Challenges of Mining Domain Instance Models

- Idea: Apply grammar inference techniques to the metamodel drift problem.
- Problem: Modeling tools export XML files; mismatch in representation expected by grammar inference techniques.
- Solution: Translate XML to textual DSL (Domain-Specific Language) - MRL

Model Representation Language (MRL):

START ::= GME

GME ::= MODEL_OR_ATOM GME | MODEL_OR_ATOM

MODEL_OR_ATOM ::= MODEL | ATOM

MODEL ::= model #Id \{ M_BODY \}

M_BODY ::= MODELS FIELDS CONNECT

MODELS ::= #Id \; MODELS | epsilon

FIELDS ::= fields OTHER_FIELDS \;

OTHER_FIELDS ::= #Id \, OTHER_FIELDS | epsilon

CONNECT ::= connection CONNECTIONS | epsilon

CONNECTIONS ::= #Id \: #Id \-> #Id \; CONNECTIONS | epsilon

ATOM ::= atom #Id \{ FIELDS \}

Overview of MARS

Metamodel Inference Process

Generated DSL Textual Representation of Model

```
model NetDiagram {
Network ;
NetDiagram ;

connections
NetworkEquiv : Perimeter ->
Network ;
NetworkEquiv : Perimeter ->
Network ;
}
...
```

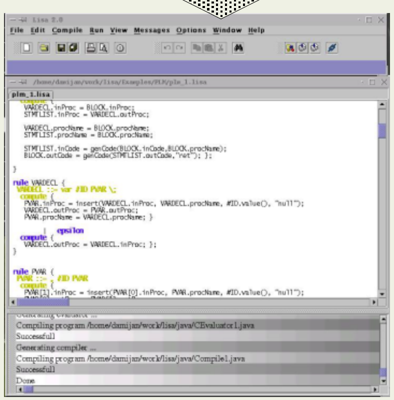
❶

XSLT Translator

Set of Instance Models in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "mga.dtd">

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "mga.dtd">

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "mga.dtd">

<project guid="{00000000-00000-00000}"
   metaguid="{6C3AD01E-DAD000000000}"
   metaname="networking">
  <name>mynetwork</name>
  <folder id="id-006a-01" kind="RootFolder">
    <name>mynetwork</name>
      <model id="id-22" kind="NetDiagram">
      ...
```

❷

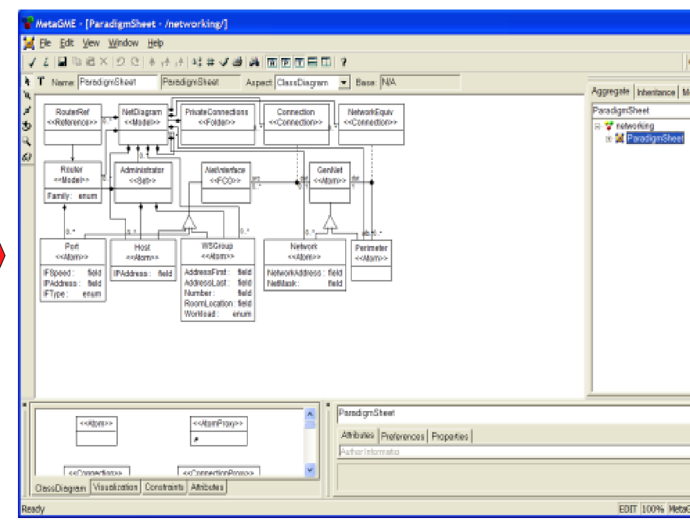


LISA Grammar Inference Engine

❸

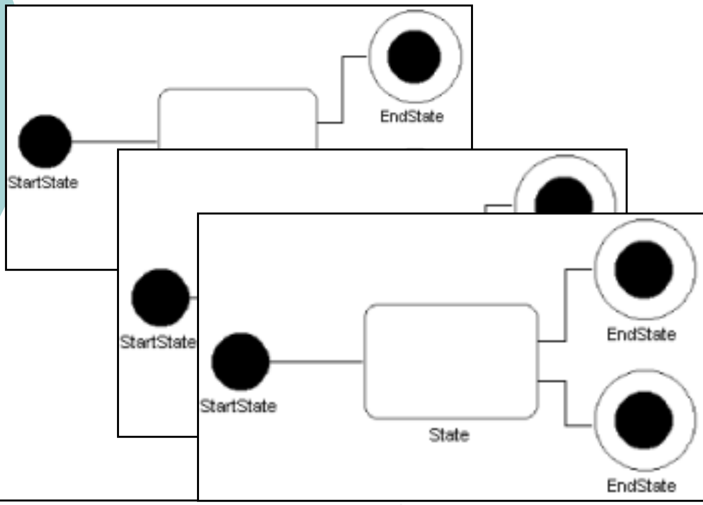
```
NETDIAGRAM -> <<MODEL>>
netdiagram { PARTS0 }
PARTS0 -> MODELATOM0 FIELDS0
MODELATOM0 -> NETWORKS
NETDIAGRAMS WSGROUPS
PERIMETERS HOSTS ROUTERS
NETWORKS -> NETWORK NETWORKS |
eps
HOSTS -> HOST HOSTS | eps
ROUTERS -> ROUTER | eps
...
```

Inferred Grammar



Inferred Metamodel

From GME Models to MRL

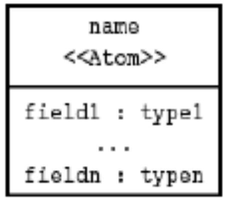
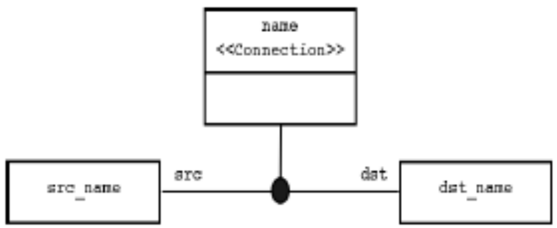
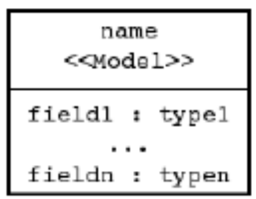
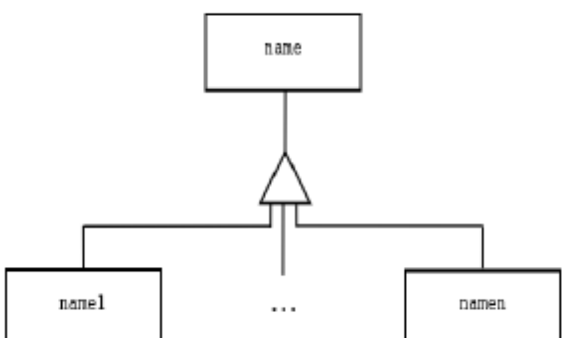



```
<connection id="id-0068-00000070" kind="Transition" role="Transition">
        <name>Transition</name>
        <connpoint role="dst" target="id-0066-00000080"/>
        <connpoint role="src" target="id-0066-00000086"/>
</connection>
<connection id="id-0068-00000071" kind="Transition" role="Transition">
        <name>Transition</name>
        <connpoint role="dst" target="id-0066-00000084"/>
        <connpoint role="src" target="id-0066-00000081"/>
</connection>
<connection id="id-0068-0000006c" kind="Transition" role="Transition">
        <name>Transition</name>
        <connpoint role="src" target="id-0066-00000083"/>
        <connpoint role="dst" target="id-0066-00000084"/>
</connection>
```

XSLT

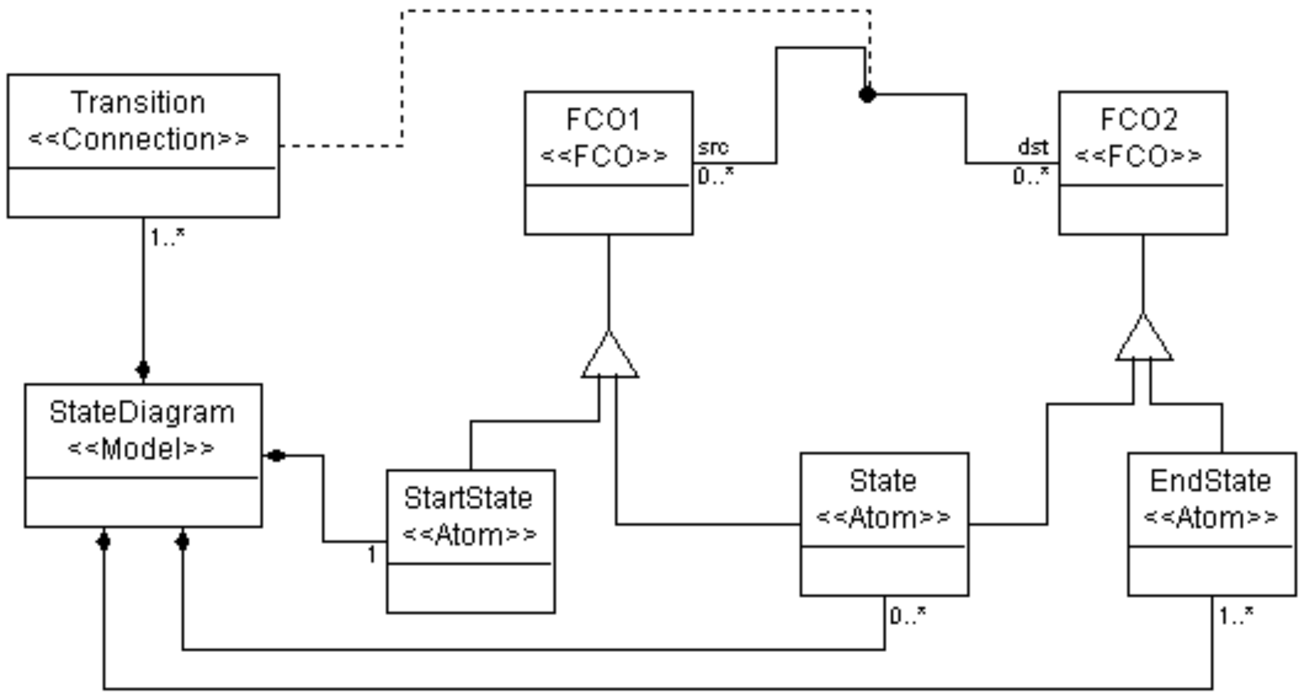
```
model StateDiagram {
    StartState;
    EndState;
    State;
    State;
    fields;
    connection
    Transition : StartState → State;
    Transition : State → State;
    Transition : State → EndState;
}
atom StartState {
    fields ;
}
atom EndState {
    fields ;
}......
```

Model to CFG Conversion

1.		NAME → 'atom' name {FIELDS} FIELDS → 'fields' field1 ... fieldn
2.		NAME → 'connection' name ':' SRC -> DST; SRC → SRC_NAME DST → DST_NAME
3.		NAME → 'model' name {PARTS} PARTS → MODELATOM FIELDS CONNECTIONS FIELDS → 'fields' field1 ... fieldn MODELATOM → ... CONNECTIONS → ... (see transformations 8 and 9)
4.		FCO → 'fco' NAME NAME → NAME1 | ... | NAMEn

From MRL to Inferred Metamodel

```
.......
model StateDiagram {
    StartState;
    EndState;
    State;
    State;
    fields;
    connection
    Transition :
    StartState → State;
    Transition : State →
    State;
    Transition : State →
    EndState;
}
atom StartState {
    fields ;
}
atom EndState {
    fields ;
}......
```



Metamodel Inference

Examples of Inference Rules

- A non-terminal that represents a model or an atom is optional in the model description

  MODELATOM → NAMEOPT1 NAME2 … NAMEk

  NAMEOPT1 → NAME1 | ε //option

- A non-terminal that represents a model or an atom can appear zero or more times in the model description

  MODELATOM → NAMES1 NAME2 … NAMEk

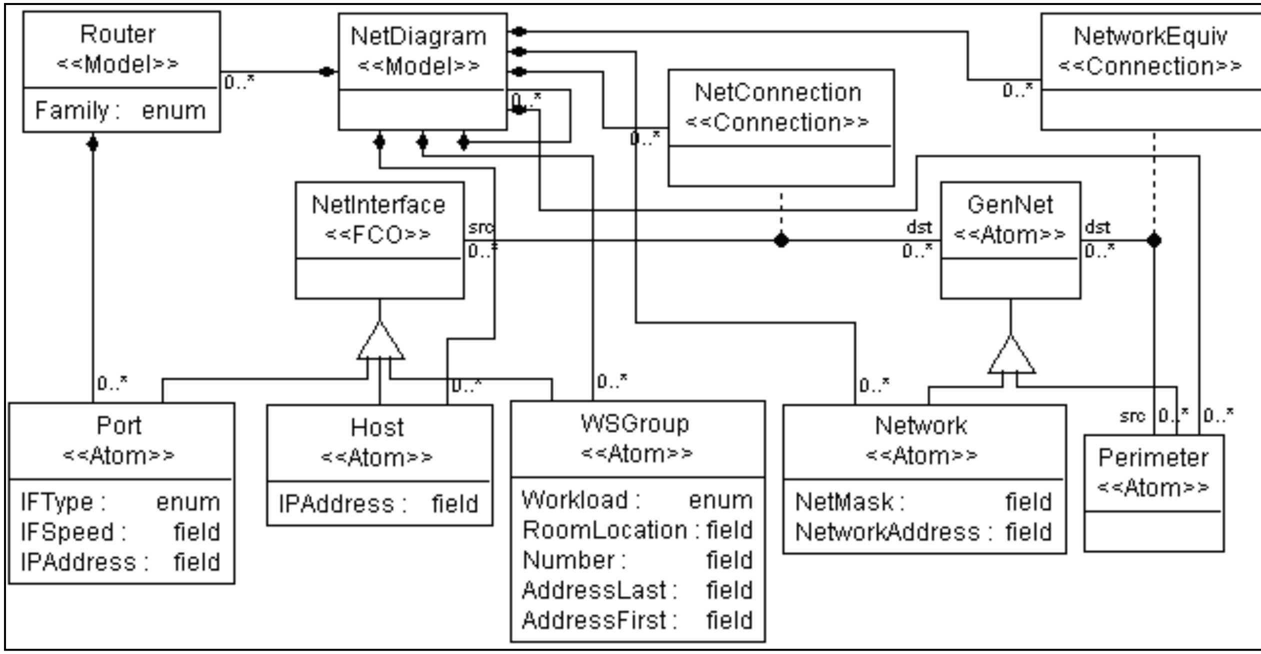
  NAMES1 → NAME1 NAMES1 | ε

  //repetition of zero or more

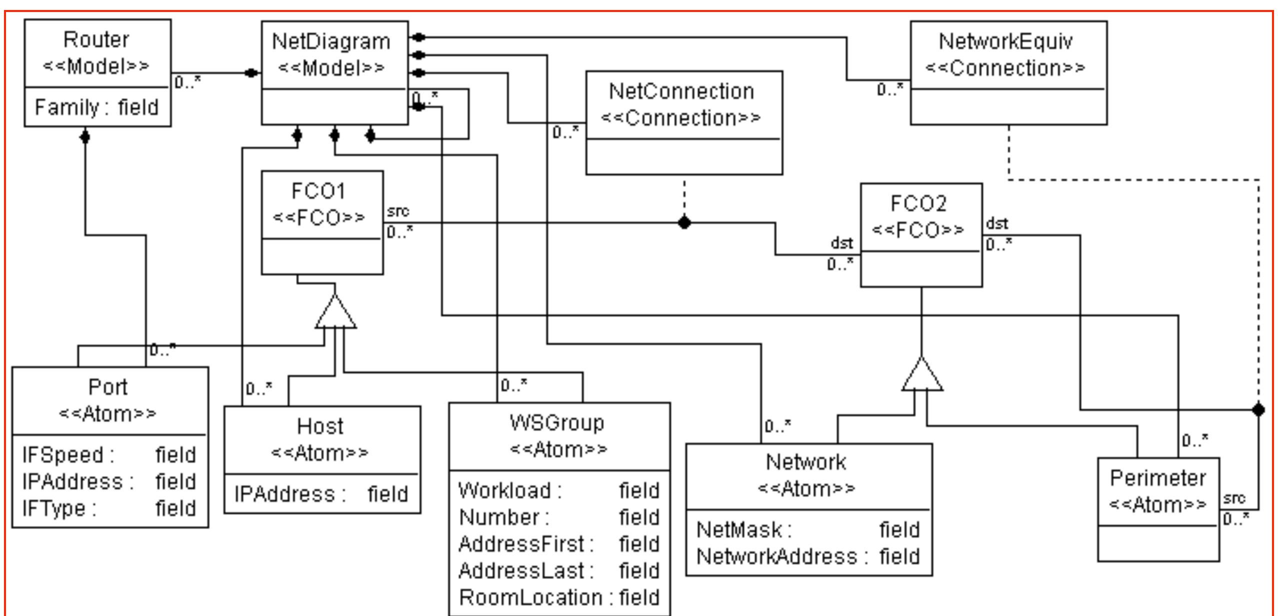
Inferred CFG for the FSM Metamodel

1. STATEDIAGRAM → 'model' StateDiagram { PARTS0 }
2. PARTS0 → MODELATOM0 FIELDS0 CONNECTIONS0
3. MODELATOM0 → STARTSTATES ENDSTATES STATES
4. STARTSTATES → STARTSTATE
5. ENDSTATES → ENDSTATE ENDSTATES | ENDSTATE
6. STATES → STATE STATES | ε
7. FIELDS0 → ε
8. CONNECTIONS0 → 'connection' TRANSITION
9. TRANSITION → transition : SRC0 → DST0 ; TRANSITION | transition : SRC0 → DST0 ;
10. SRC0 → 'fco' FCO1
11. FCO1 → STARTSTATE|STATE
12. DST0 → 'fco' FCO2
13. FCO2 → ENDSTATE|STATE
14. STARTSTATE → 'atom' StartState { FIELDS1 }
15. FIELDS1 → ε
16. ENDSTATE → 'atom' EndState { FIELDS2 }
17. FIELDS2 → ε
18. STATE → 'atom' State { FIELDS3 }
19. FIELDS3 → ε

Original vs. Inferred (Network Domain - name)

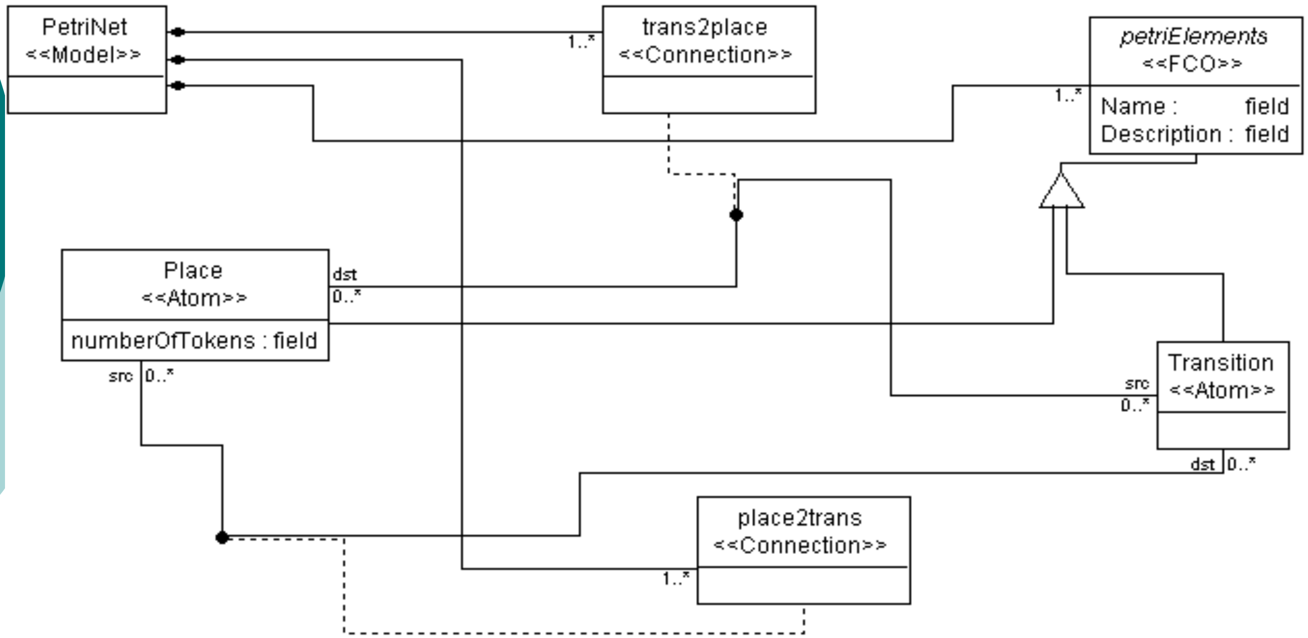


Original

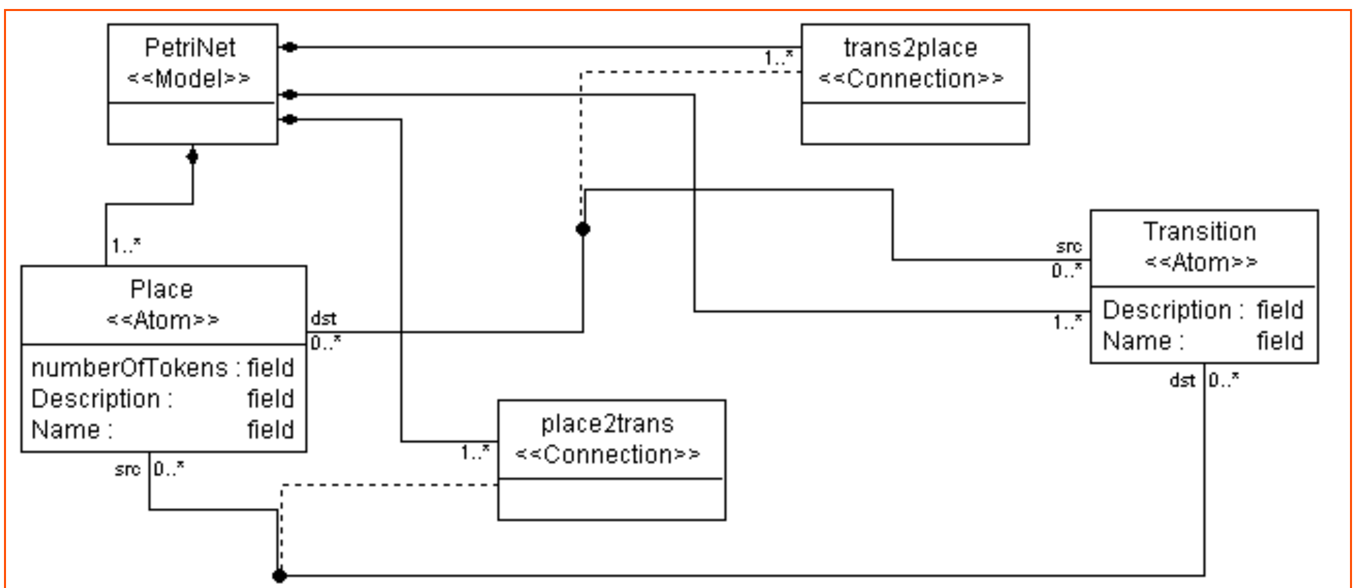


Inferred

Original vs. Inferred (Petri Net – generalization hierarchy)

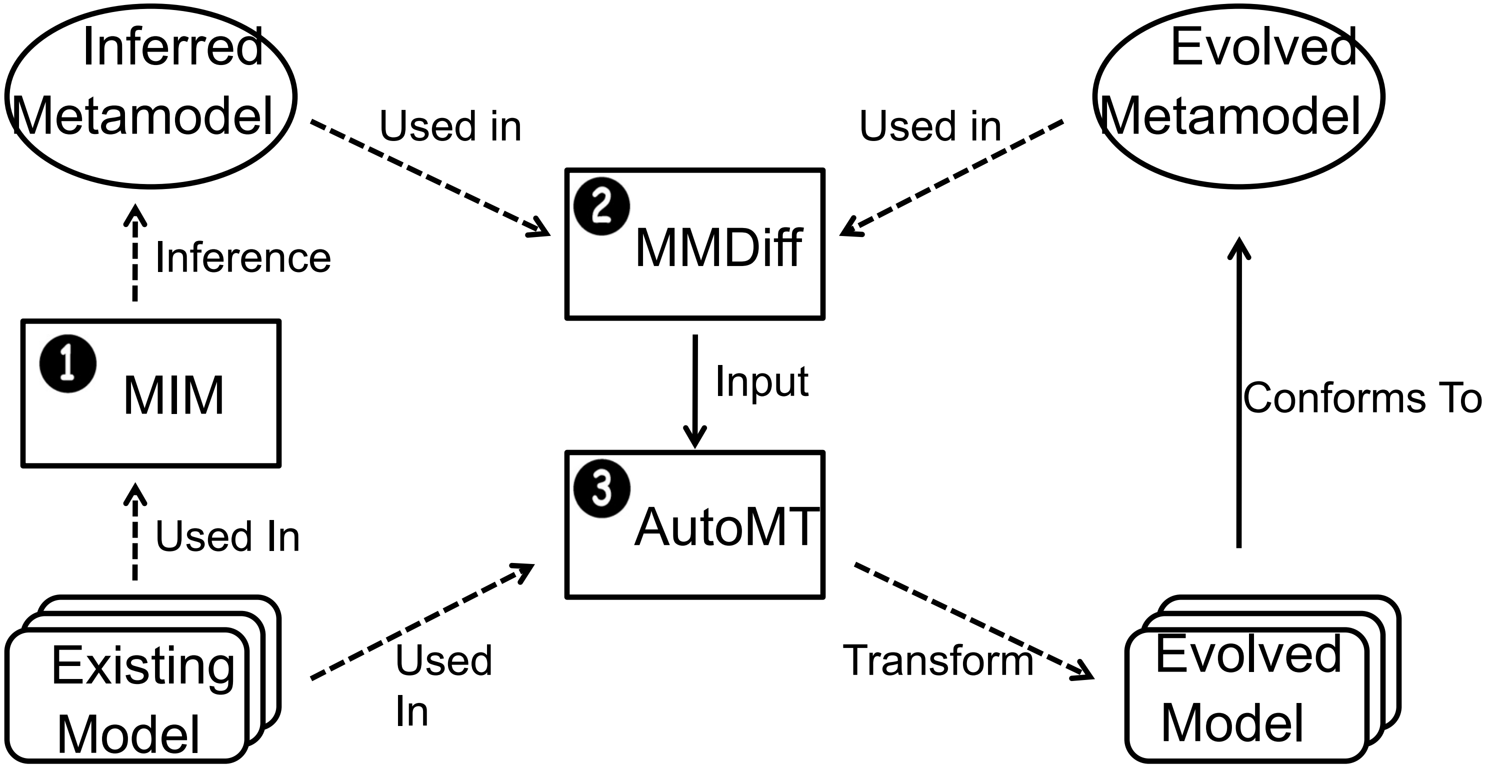


Original



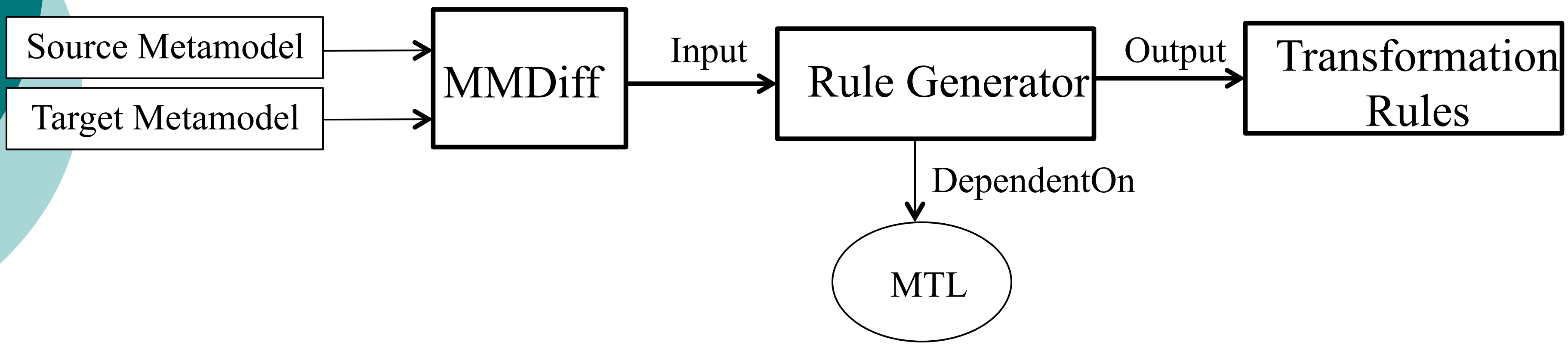
Inferred

Using Metamodel Inference for Co-Evolution



- MIM (Metamodel Inference from Models)
- MMDiff (Metamodel Differencing)
- AutoMT (Automated Model Transformation)

Automated Model Transformation



- Transformation rules are expressed using GReAT (Graph Rewriting and Transformation)
- A GReAT rule is designed as a 9-tuple: R = (pattern, action, input interface, output interface, guard, attribute mapping, match condition).

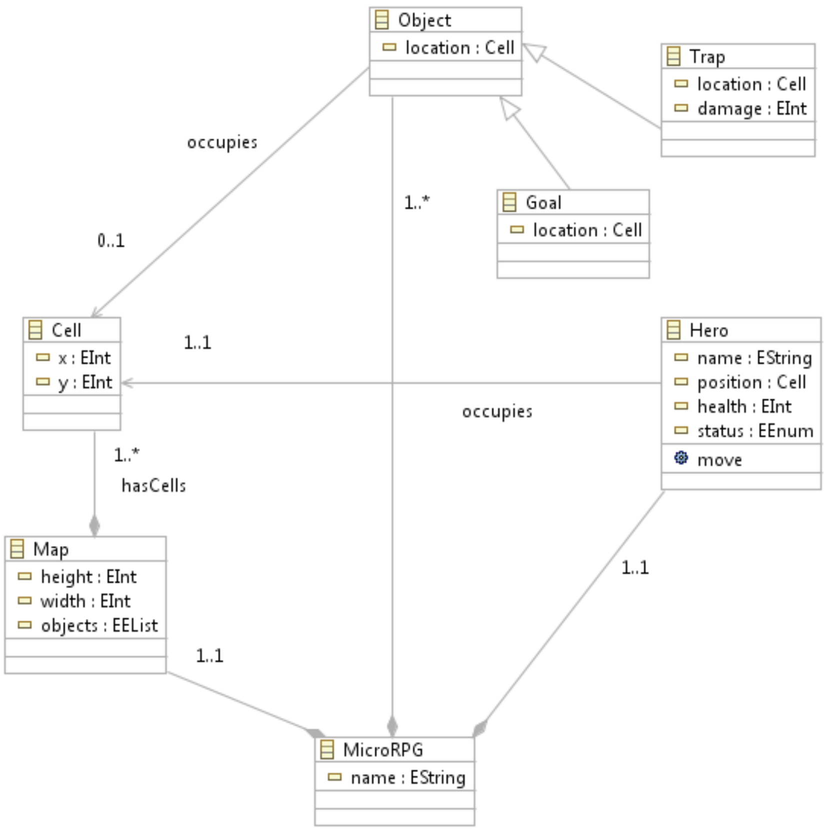
Outline

1. Component-Based Language Implementation
2. Grammar Inference of Domain Specific Languages and Domain Specific Metamodels
3. Formalization of Modeling Language Semantics

Formalizing the Semantics of Modeling Languages

- Semantics is a mapping from the abstract syntax of the DSML to some semantic domain.
- Abstract syntax defines the fundamental modeling concepts, their relationships, and attributes used in the DSML.
- The semantic domain is some mathematical framework whose meaning is well-defined.

A Proposed Approach



- Establish denotational semantics for metamodel
- Map semantics to Haskell functions
- Metamodel components lend themselves to functional programming

Tool Generation Challenges

- Proving properties about concepts and relationships in the domain is not possible without semantics.
- A model interpreter cannot be automatically generated in most cases.
- Various other model-based tools (e.g., debuggers, test engines, simulators, verifiers) also cannot be generated automatically.

Tool Analysis Challenges

- Verifying a model interpreter is a very difficult, if not impossible task, without semantics.
- Verification, optimization, and parallelization of models can be expressed only through general purpose programming languages.

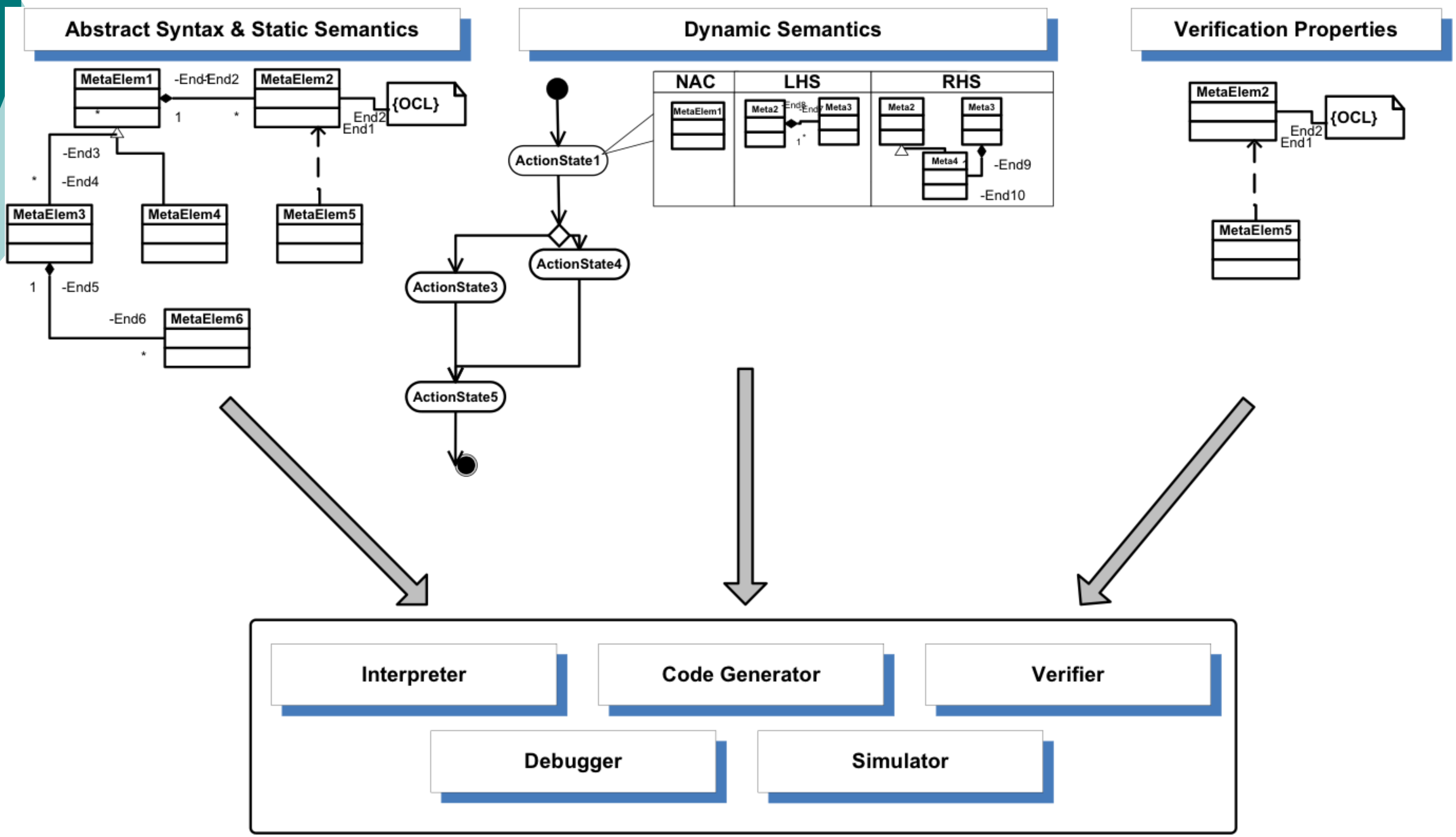
Modeling Language Composition

- Multiple domains might be involved to describe different perspectives of a modeled system.
- In such a case, there is a need for composing DSMLs together.
- Presently, there is little support for formal composition and evolution of DSMLs.

Goal

- Use semantic formalization for automatic generation of model interpreters, simulators, debuggers and verifiers, which would have significant impact on the current practice of model-driven engineering in terms of automating many tasks that are currently done ad hoc in a manual hand-crafted manner.

Semantics-Based Tools in Domain-Specific Modeling



Conclusion

- Grammarware may be the framework for many software engineering systems
- Modelware is one of the areas in software engineering that may benefit from grammar-based approaches
- Semantics are necessary to build complete systems

Acknowledgments

- This material is based upon work supported by the U. S. National Science Foundation under Grants CCF-0811630 and OISE-0968596.
- Collaborators:
  - Peter Clarke – Florida International University, USA
  - Robert France – Colorado State University, USA
  - Danielle Gaither – UNT
  - Jeff Gray – University of Alabama, USA
  - Dejan Hrnčič, Marjan Mernik – University of Maribor, Slovenia
  - Gabor Karsai – Vanderbilt University, USA
  - Faizan Javed, Qichao Liu, Upendra Sapkota, Alan Sprague, Xiaoqing Wu – University of Alabama at Birmingham, USA

Further Information

Contact:

Barrett.Bryant@unt.edu

http://www.cse.unt.edu/~bryant