Automated Analysis and Code Generation for Domain-Specific Models

George Edwards
University of Southern California

Yuriy Brun
University of Massachusetts

Nenad Medvidovic
University of Southern California

Overview

- Background: DSLs and MDE
- Research Challenge: Building Tools for DSLs
- Our Solution Approach
- The LIGHT Platform
- Evaluation Results

Domain-Specific Languages (DSLs)

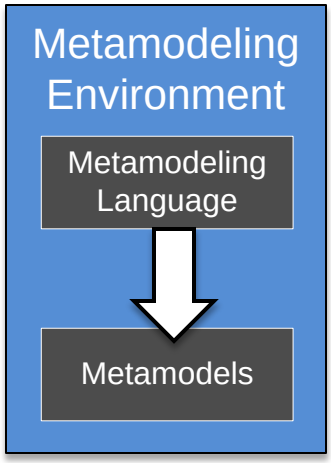
- Modeling languages that are customized for a particular problem
- Concisely express system designs
  - No missing or extraneous features
  - Capture common, reusable patterns
  - Enforce architectural constraints
  - Use symbols native to the application domain
- Easily modified, evolved, and composed

Model Driven Engineering (MDE)

- **Model-driven engineering** (MDE) combines DSLs with model interpreters
  - **Metamodels** define elements, relationships, views, and constraints
  - **Model interpreters** leverage domain-specific models for analysis, code generation, and transformation

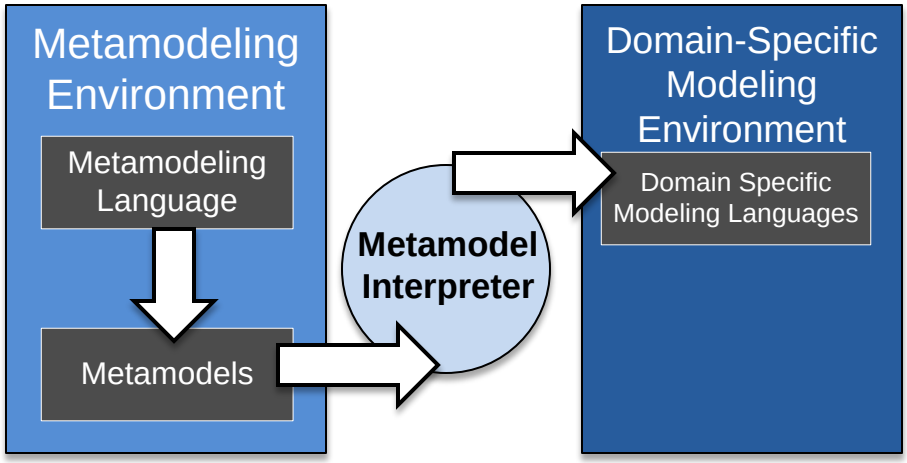
Model Driven Engineering (MDE)

- **Model-driven engineering** (MDE) combines DSLs with model interpreters
  - **Metamodels** define elements, relationships, views, and constraints
  - **Model interpreters** leverage domain-specific models for analysis, code generation, and transformation



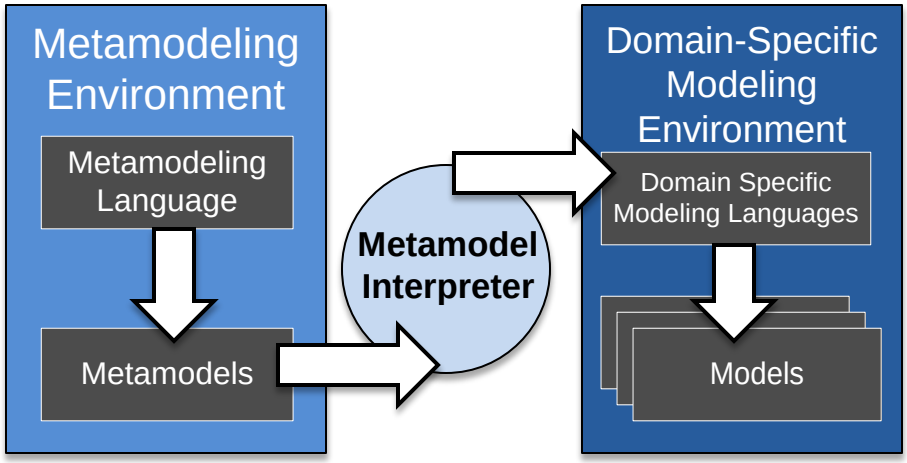
Model Driven Engineering (MDE)

- **Model-driven engineering** (MDE) combines DSLs with model interpreters
  - **Metamodels** define elements, relationships, views, and constraints
  - **Model interpreters** leverage domain-specific models for analysis, code generation, and transformation



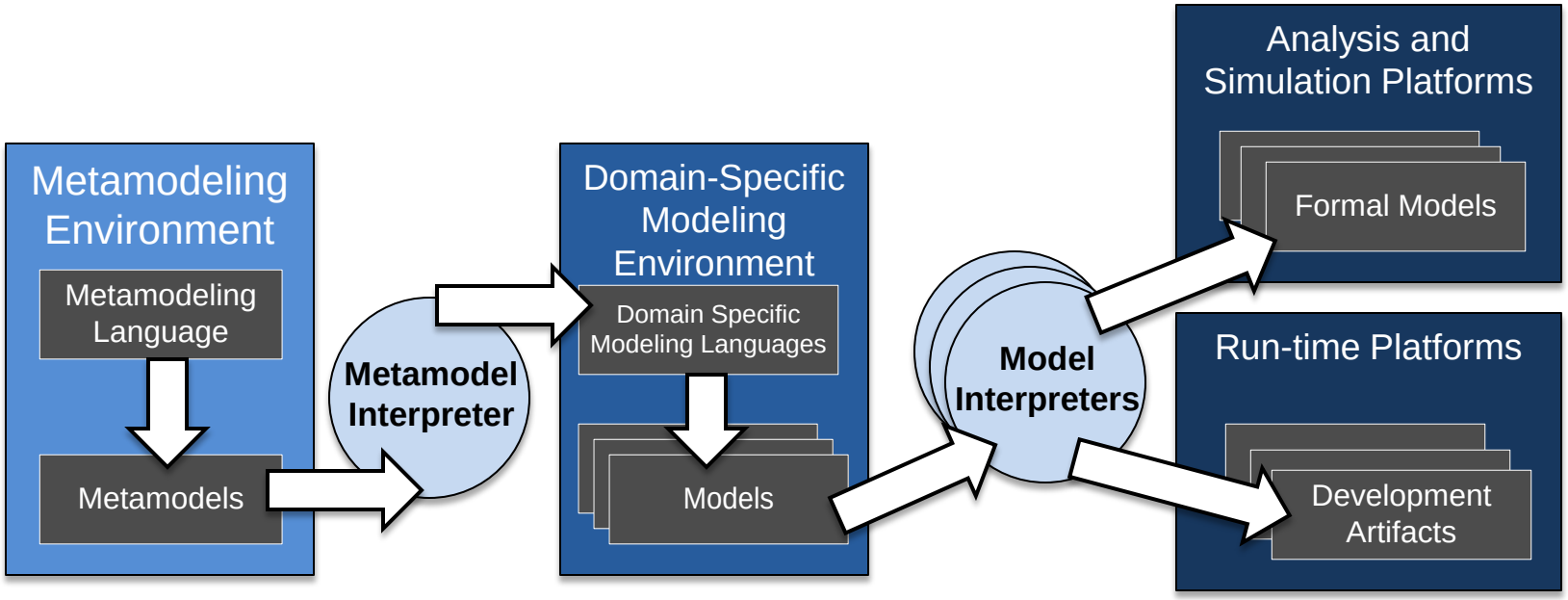
Model Driven Engineering (MDE)

- **Model-driven engineering** (MDE) combines DSLs with model interpreters
  - **Metamodels** define elements, relationships, views, and constraints
  - **Model interpreters** leverage domain-specific models for analysis, code generation, and transformation



Model Driven Engineering (MDE)

- **Model-driven engineering** (MDE) combines DSLs with model interpreters
  - **Metamodels** define elements, relationships, views, and constraints
  - **Model interpreters** leverage domain-specific models for analysis, code generation, and transformation

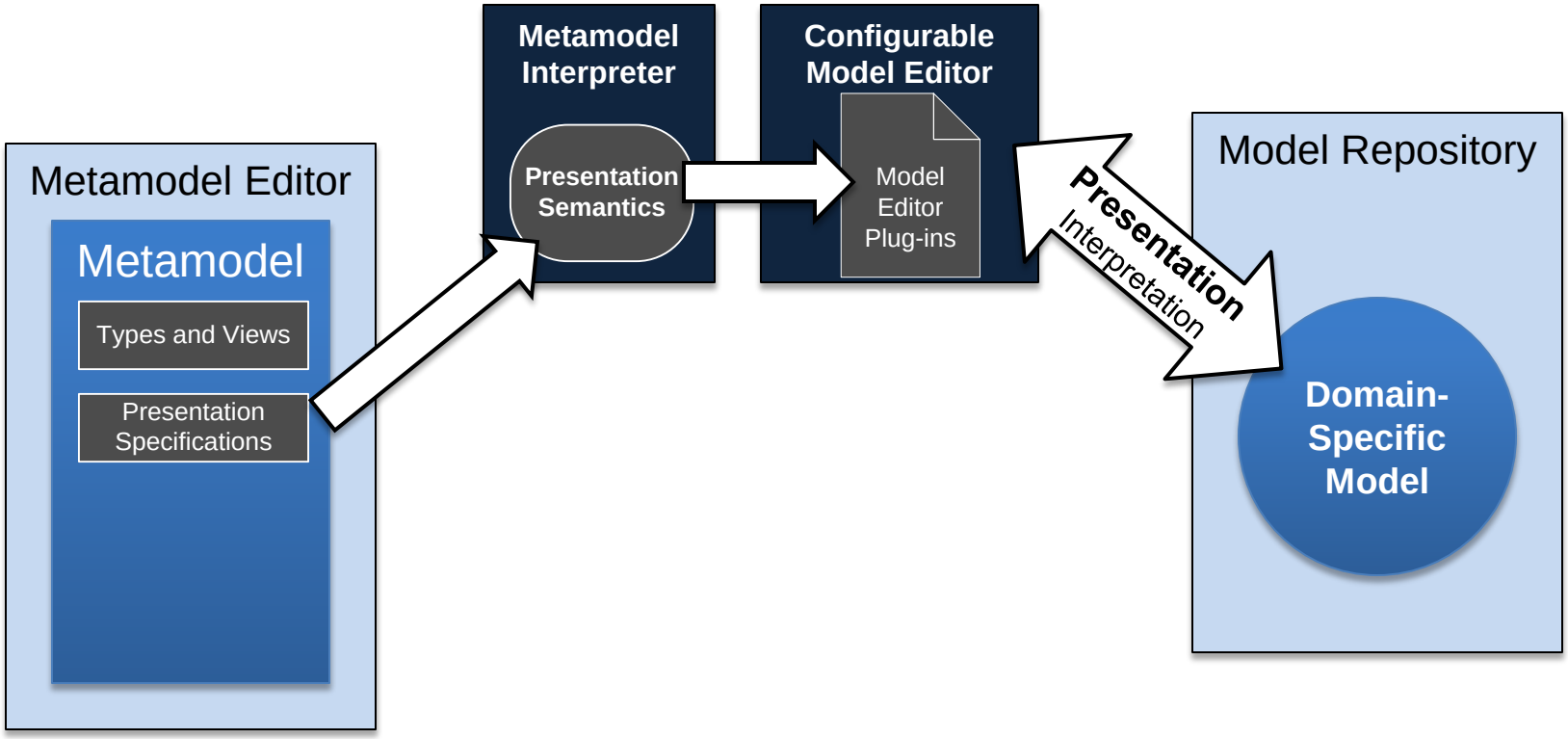


Challenge: Building Interpreters

- Today, we have to write these tools by hand
- For a DSL of modest size, tools average 18K SLOC and approximately 4 person-months
- Developing and maintaining DSLs and interpreters is hard
  - Reusing model interpreters for different DSLs is hard
  - Little guidance on how to construct DSLs and interpreters
  - Semantics applied to models are opaque
  - Requires particular types of expertise

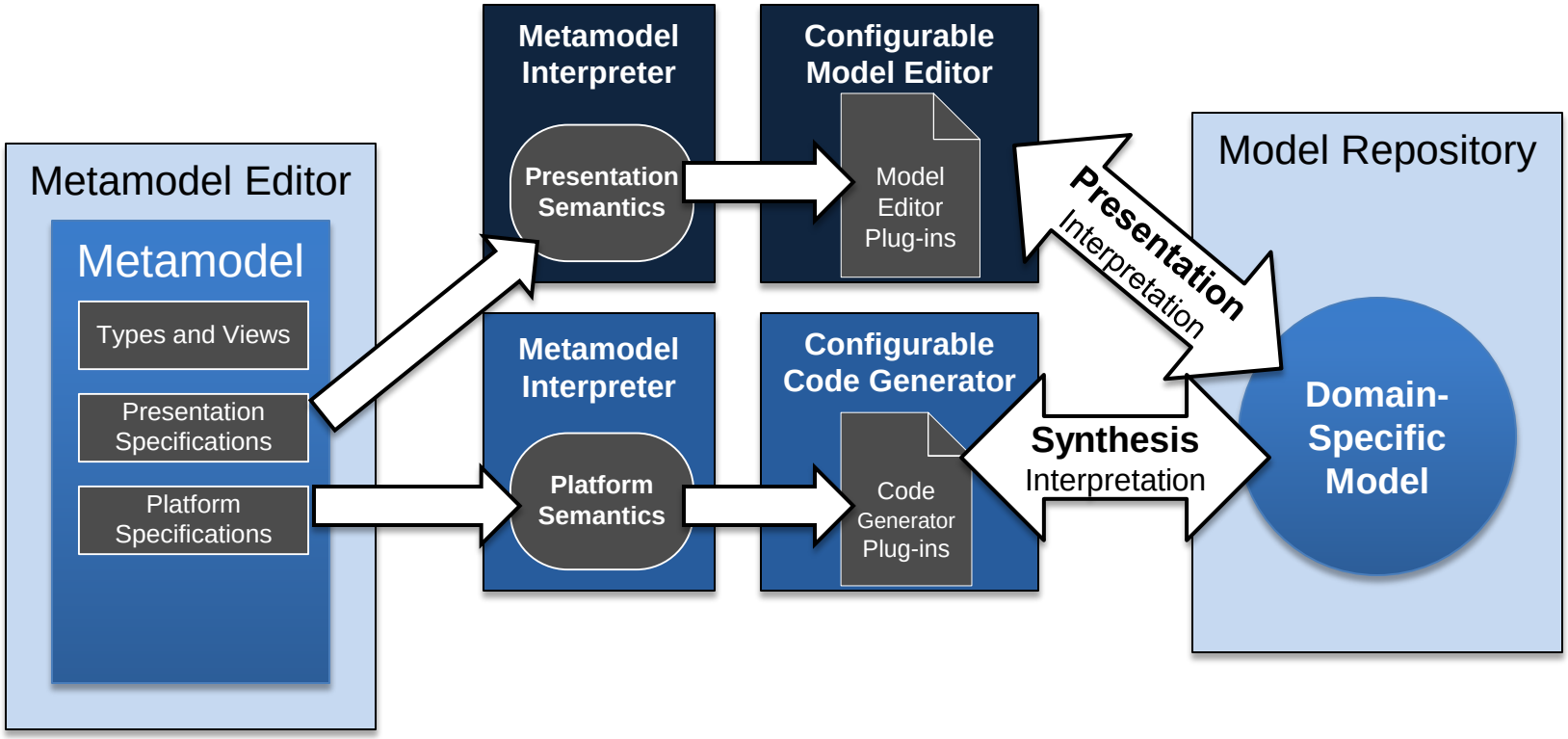
Simplifying Insight

Automatically synthesize domain-specific model interpreters the same way that domain-specific model editors are synthesized



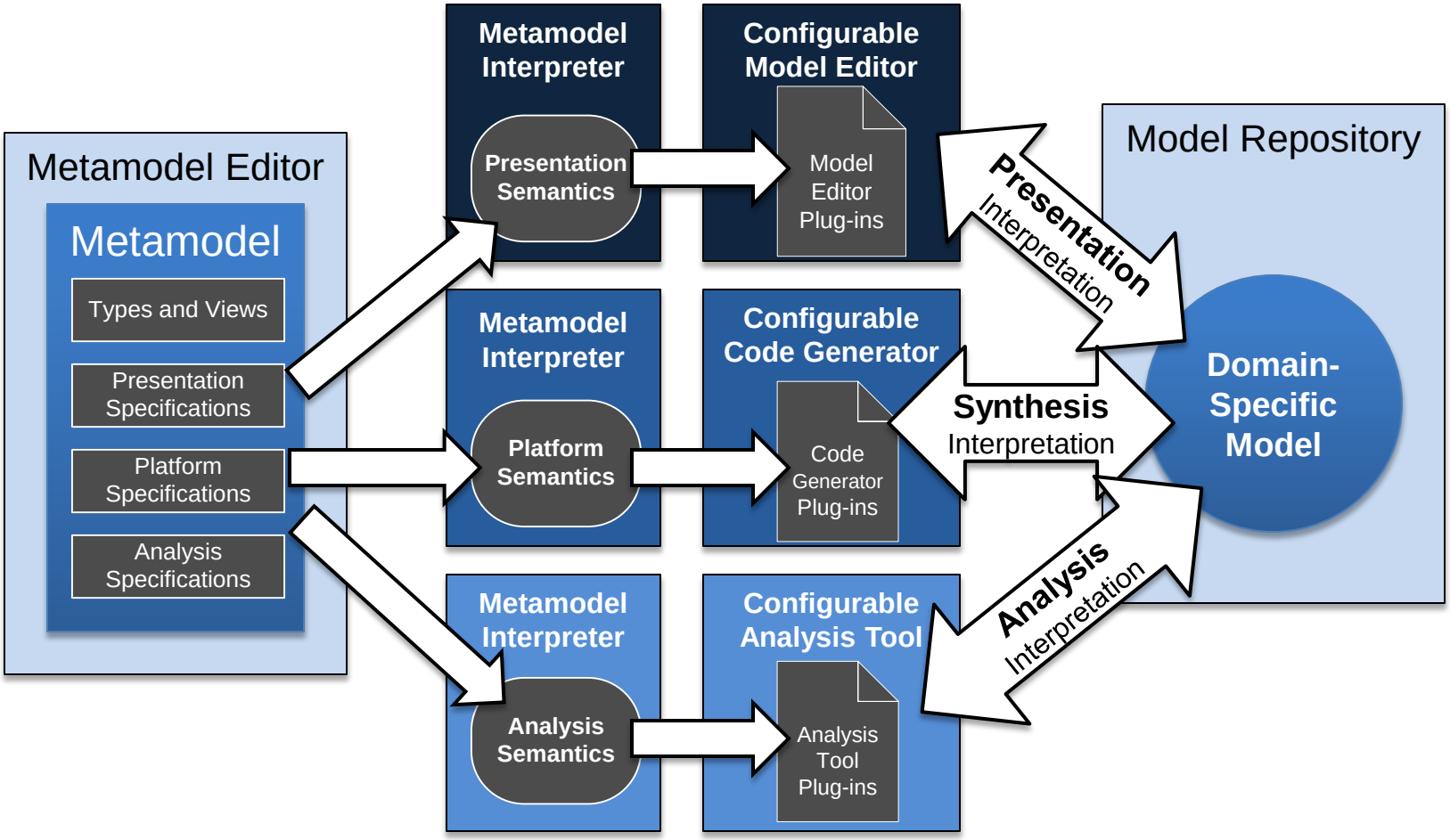
Simplifying Insight

Automatically synthesize domain-specific model interpreters the same way that domain-specific model editors are synthesized



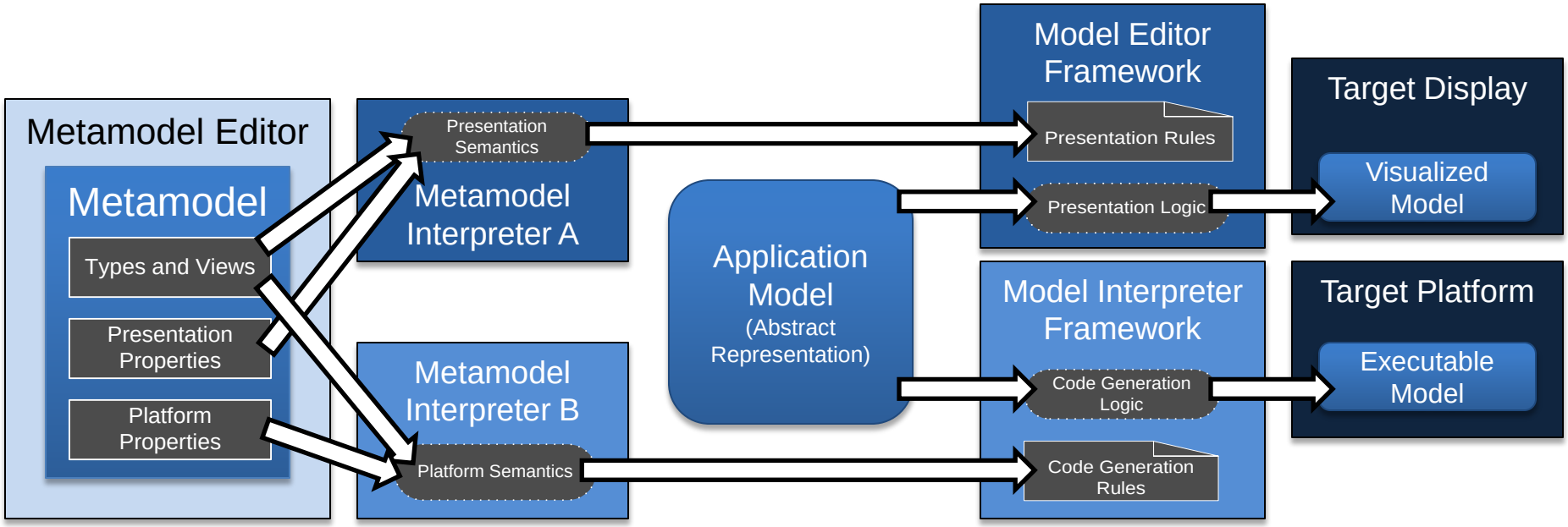
Simplifying Insight

Automatically synthesize domain-specific model interpreters the same way that domain-specific model editors are synthesized



Solution Approach

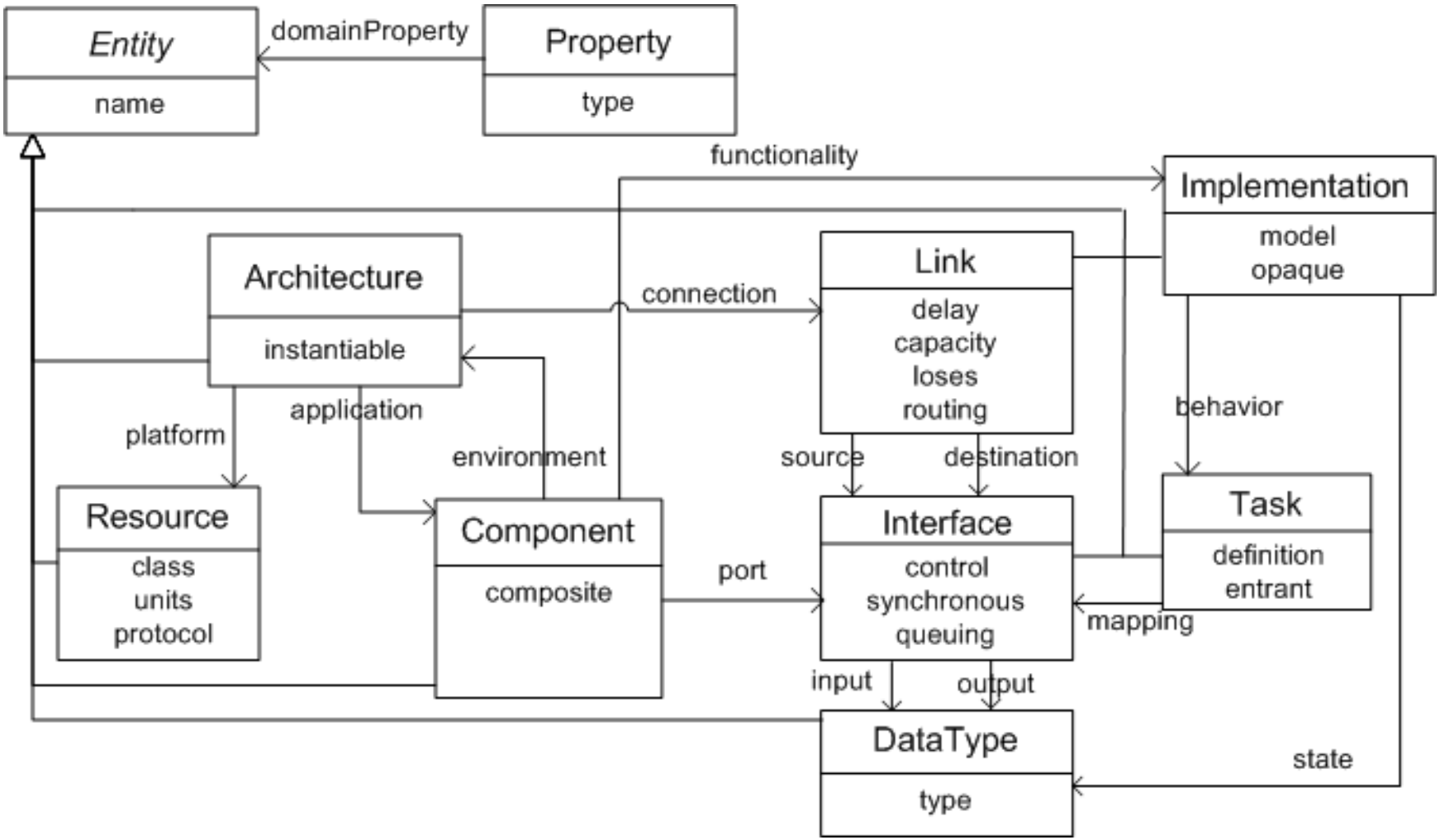
- Embed semantics in metamodels as properties of metatype instances
- Use a **metamodel interpreter** to derive transformation rules from property values
  - Transformation rules are captured in a framework extension
- Use a **model interpreter framework** to implement transformation logic
  - Transformation logic is applied according to transformation rules



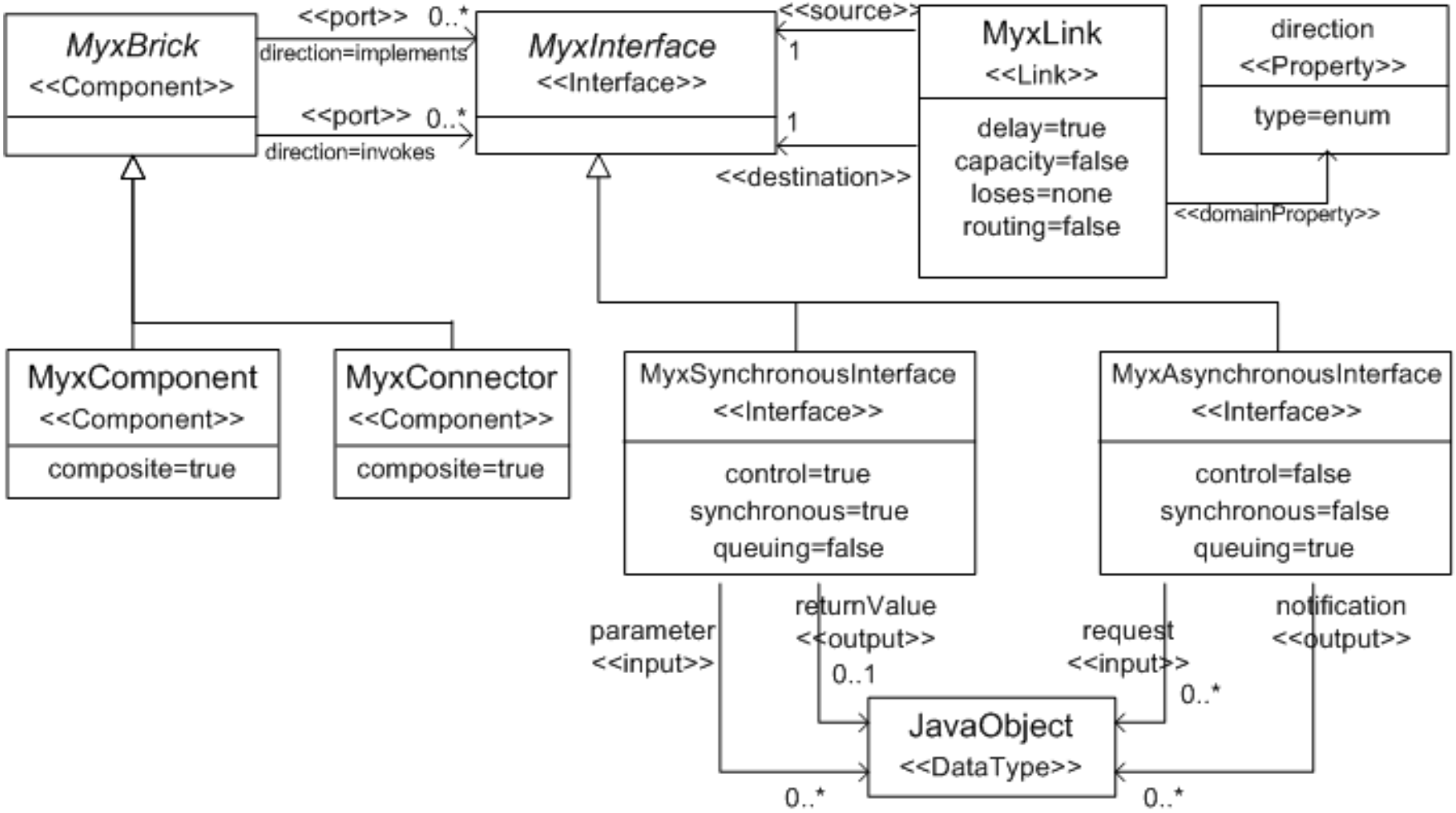
The LIGHT Platform

- A MDE platform for software architectures
- Includes:
  - Metamodeling language and metamodel editor
  - Two metamodel interpreters with paired model interpreter frameworks
  - Example metamodels and framework extensions
- Provides the extensibility to accommodate new language features and architectural analyses

Metamodeling Language



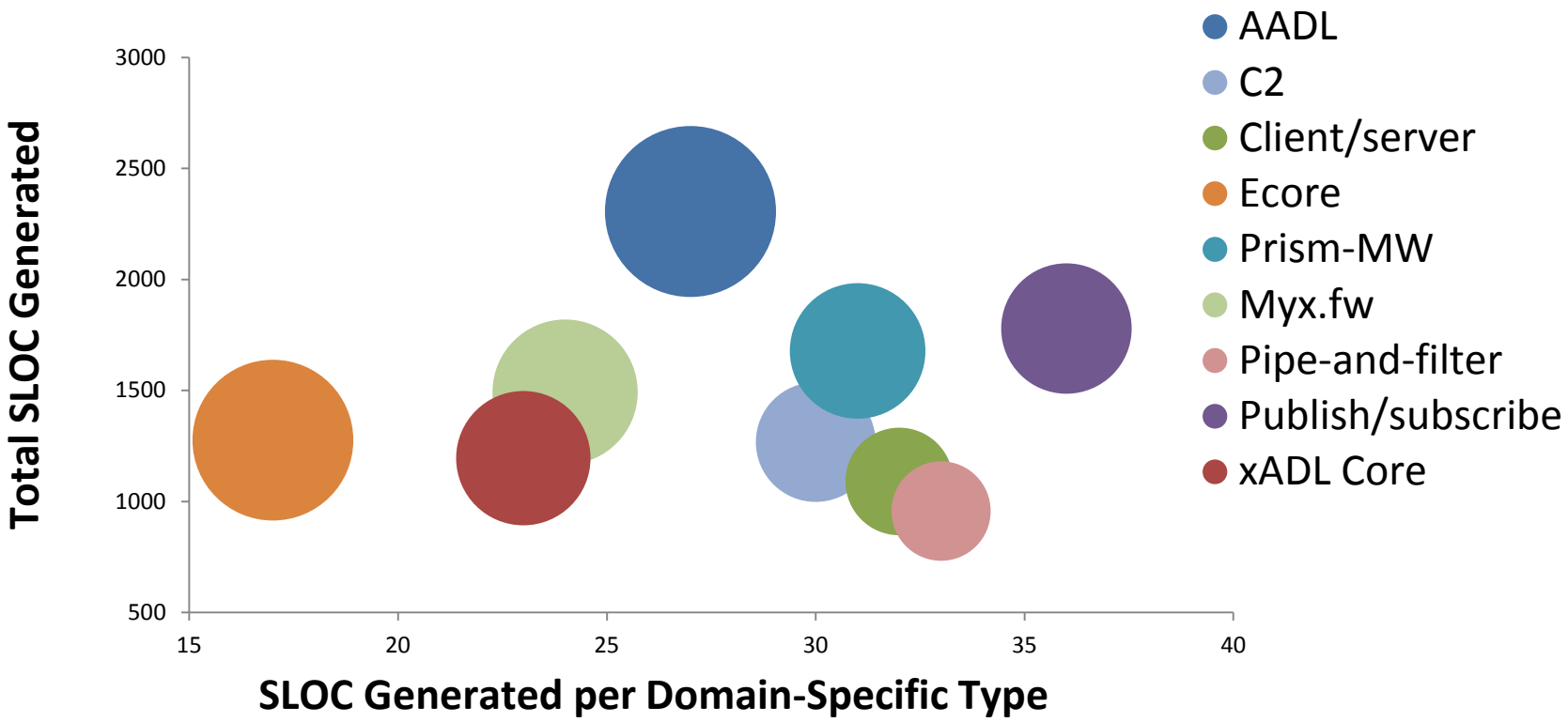
Example Metamodel Snippet



LIGHT Benefits

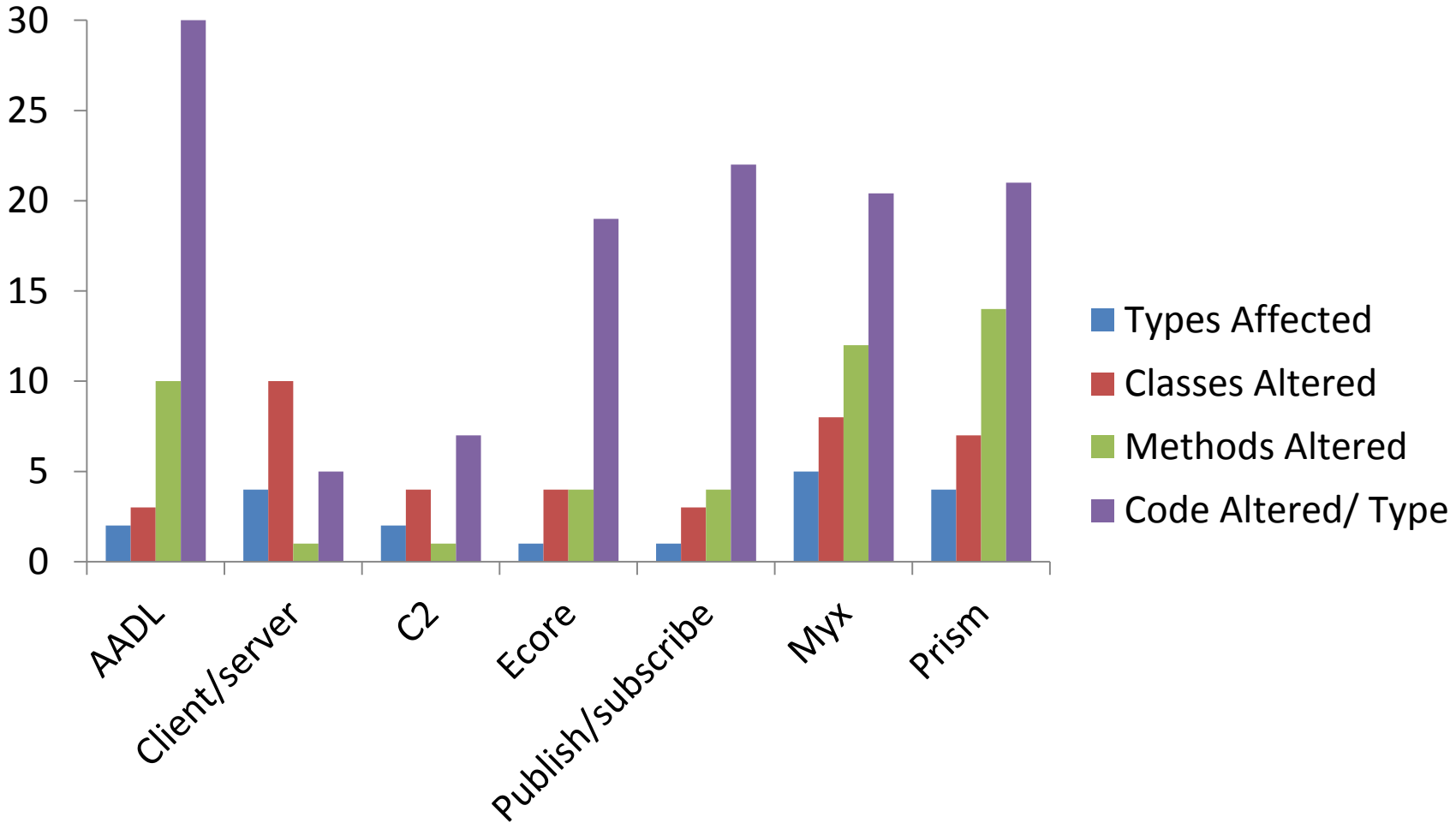
- Reduced **implementation** effort
  - Effort saved through code generation and reuse
  - Quantified by:
    - Lines of generated interpreter code
    - Total lines of reused interpreter code
    - Lines of generated code per domain-specific type
    - Lines of reused code per domain-specific type
- Reduced **maintenance** effort
  - Due to relative ease of performing DSL modifications within a metamodel rather than within model interpreter source code
  - Quantified by number of metamodel objects altered vs. number of classes, methods, and SLOC altered

Implementation Effort Metrics



COCOMO Estimates (avg): Nominal settings → 23.4 person-months
Favorable settings → 4.2 person-months

Maintenance Effort Metrics



Conclusions

- Building and maintaining DSL tools is hard
- Automatic synthesis of modeling tools reduces the cost of using DSLs
- Tradeoffs in our approach:
  - Reduced flexibility
  - Additional metamodeling effort
  - Analysis and code generation tools must be chosen *a priori*

Questions?