TECHNISCHE UNIVERSITÄT DRESDEN

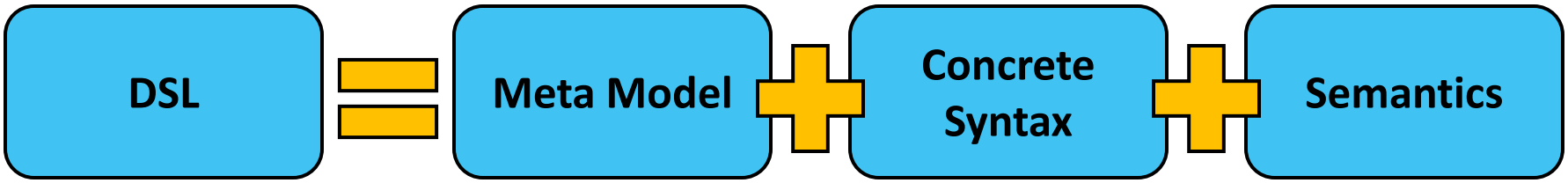
Fakultät Informatik Institut Software- und Multimediatechnik, Lehrstuhl Softwaretechnologie

Model-based Language Engineering with EMFText

Slides based on GTTSE'11 tutorial by Sven Karol, Florian Heidenreich and Christian Wende

ACSE 04.12.2012

What's in a DSL?



Domain concepts, relations, attributes

Symbols, shapes, text

Behavior, meaning

Motivation – Why DSLs?

+ Use the concepts and idioms of a domain
+ Domain experts can understand, validate and modify DSL programs
+ Concise and self-documenting
+ Higher level of abstraction
+ Can enhance productivity, reliability, maintainability and portability
+ Embody domain knowledge, enabling the conservation and reuse of this knowledge

But:

- Costs of design, implementation and maintenance
- Costs of education for users
- Limited availability of DSLs

From: http://homepages.cwi.nl/~arie/papers/dslbib/

Motivation – Why textual syntax?

Why use textual syntax for models?

- Readability
- Diff/Merge/VCS
- Evolution
- Tool autonomy
- Quick model instantiation

Why create models from text?

- Tool reuse (e.g., to perform transformations (ATL) or analysis (OCL))
- Know-how reuse
- Explicit representation of text document structure
- Tracing software artifacts
- Graphs instead of strings

EMFText – Philosophy and Goals

Design principles:

- Convention over Configuration
- Provide defaults wherever possible
- Allow customization for all parts of a syntax

Syntax definition should be

- Simple and easy for small DSLs
- Yet powerful for complex languages

EMFText – Features

Generation Features

Generation of independent code, Generation of Default Syntax

Customizable Code Generation

Specification Features

Modular Specification, Default Reference Resolving

Comprehensive Syntax Analysis

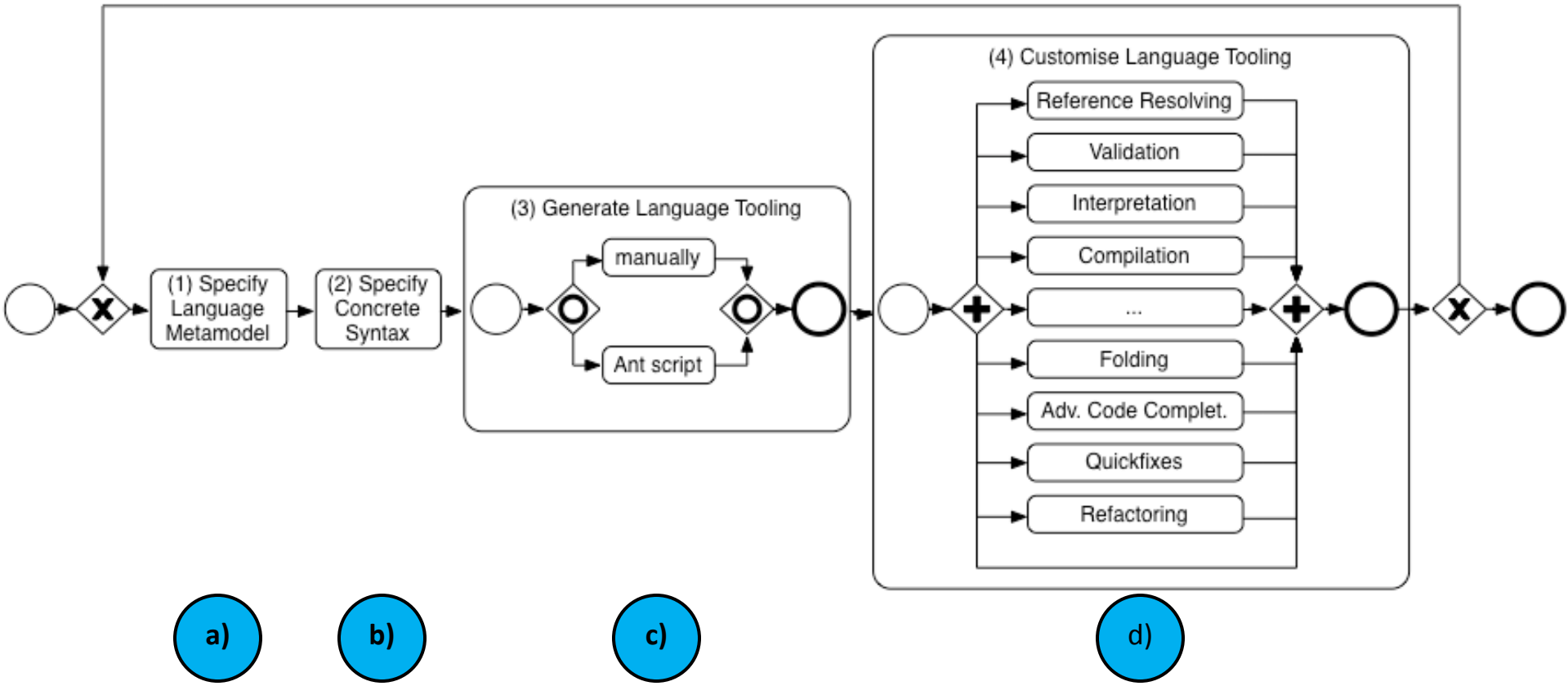
Editor Features

Code Completion, Customizable Syntax and Occurence Highlighting, Code Folding, Hyperlinks, Text Hovers, Outline View, ...

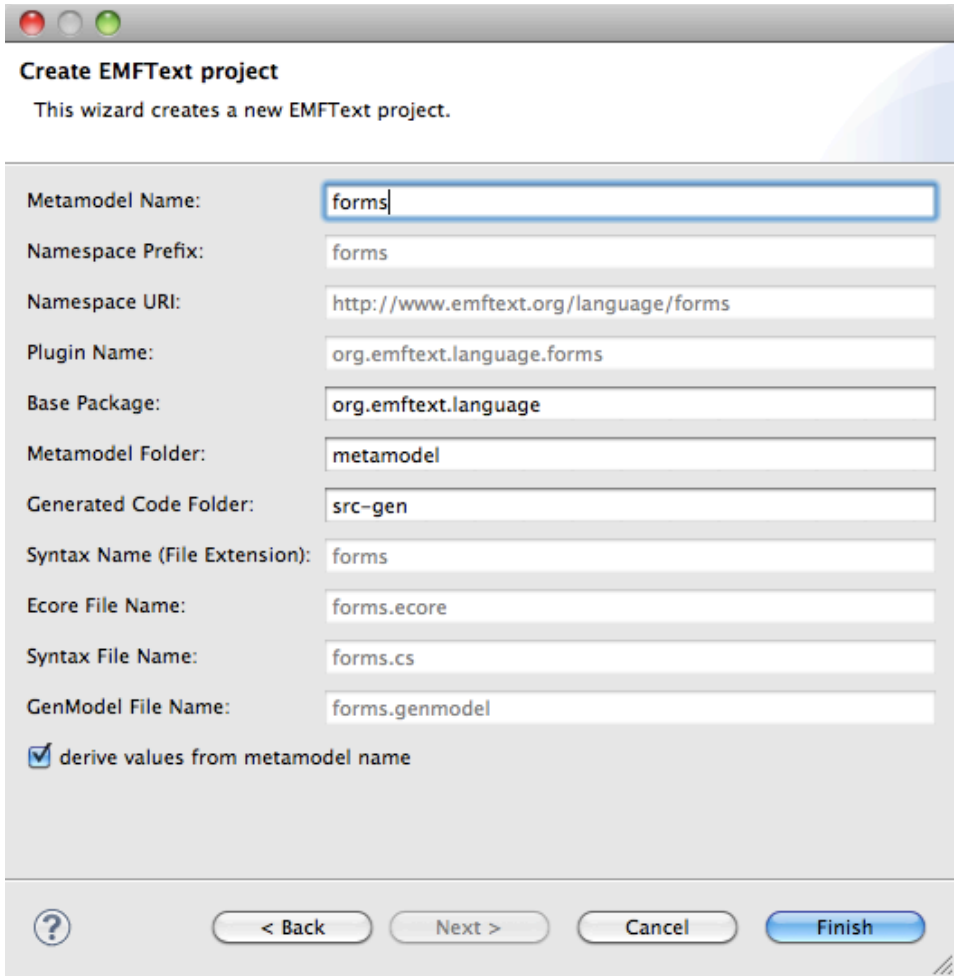
Other Highlights

ANT Support, Post Processors, Builder and Interpreter Stubs, Quick Fixes

EMFText – Language Development Process



Designing a Simple DSL: forms



EMFText project wizard

In Eclipse

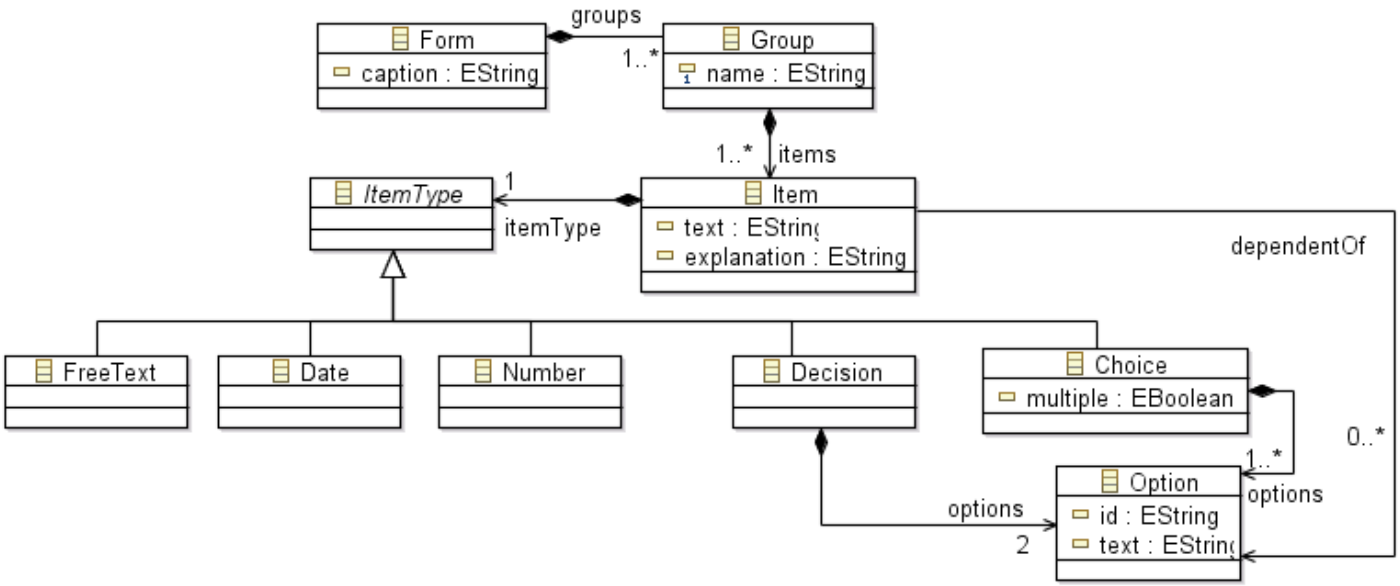
File >
New >
Other... >
EMFText Project

Specifying the Language's Metamodel

a)

Creating a new metamodel:

- Define concepts, relations and properties in an Ecore model



Existing meta models can be imported (e.g., UML, Ecore, …)

Specifying the Language's Metamodel

a)

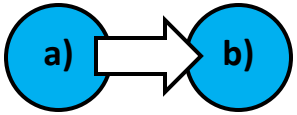
Meta model elements:

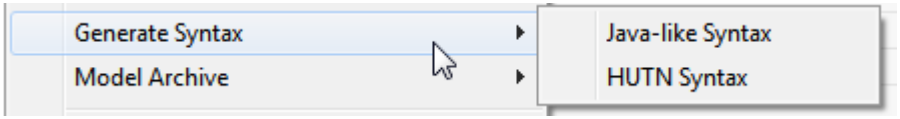
- Classes
- Data Types
- Enumerations

- Attributes
- References (Containment, Non-containment)
- Cardinalities

- Inheritance

Deriving an Initial Syntax (HUTN)





forms.ecorediag | forms.cs

```
SYNTAXDEF forms
FOR <http://www.emftext.org/language/forms>
START Form

TOKENS {
    DEFINE COMMENT $'//'(~('\n'|'\r'|'\uffff'))*$;
    DEFINE INTEGER $('-')?('1'..'9')('0'..'9')*|'0'$;
    DEFINE FLOAT $('-')?(('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ $;
}

TOKENSTYLES {
    // ... lots of token styles
}

RULES {
    Form ::= "Form" "{" ( "caption" ":" caption['"','"'] | "groups" ":" groups )* "}" ;
    Item ::= "Item" "{" ( "itemType" ":" itemType | "dependentOf" ":" dependentOf[]| "text" ":" text['"','"'] | "explanation"
    FreeText ::= "FreeText" "{" "}" ;
    Choice ::=multiple[]? "Choice" "{" ( "options" ":" options )* "}" ;
    Option ::= "Option" "{" ( "id" ":" id['"','"'] | "text" ":" text['"','"'] )* "}" ;
    Date ::= "Date" "{" "}" ;
    Number ::= "Number" "{" "}" ;
    Group ::= "Group" "{" ( "items" ":" items | "name" ":" name['"','"'] | "form" ":" form[] )* "}" ;
    Decision ::= "Decision" "{" ( "options" ":" options )* "}" ;
}
```

Initial HUTN Syntax – Example Document

HUTNForm.forms

```
Form {
    caption : "A simple form written in HUTN syntax"
    groups : Group {
        name : "A first group"
        items : Item {
            text : "Do you like this tutorial so far?"
            explanation : "It's ok to please the presenters instead of telling the truth"
            itemType : Choice {
                options : Option {
                    text : "Yes, its amazing!"
                }
                options : Option {
                    text : "Well, it's a little boring so far."
                }
            }
        }
    }
}
```

Specifying the Language's Concrete Syntax

b)

Structure of a concrete syntax specification .cs file:

- Header
  - File extension
  - Meta model namespace URI, *location*
  - Start element(s)
  - *Imports (meta models, other syntax definitions)*
- *Options*
- *Token Definitions*
- Syntax Rules

Specifying the Language's Concrete Syntax

b)

- One Syntax Rule per metaclass
  - Syntax: **`MetaClassName ::= `*`Syntax Definition`*** `;`

- Definition elements:

• Static strings (keywords)	"public"	
• Choices	a|b	
• Multiplicities	+,*,?	
• Compounds	(ab)	
• Terminals	a[]	(Non-containment References, Attributes)
• Non-terminals	a	(Containment Reference)

Customized Syntax

b)

forms.cs

```
RULES {
    Form ::= "FORM" caption['"','"'] !1 groups*;
    Group ::= !0 "GROUP" name['"','"'] !0 items*;

    Item      ::= "ITEM" text['"','"'] ( explanation['"','"'] )?
                  ("ONLY" "IF" dependentOf[])? ":"
                  itemType !0;

    Choice    ::= "CHOICE" (multiple[MULTIPLE])? "(" options ("," options)* ")";
    Option    ::= ( id[] ":")? text['"','"'];
    Date      ::= "DATE";
    FreeText  ::= "FREETEXT";
    Number    ::= "NUMBER";
    Decision  ::=  "DECISION" "(" options "," options ")";
}
```

Generic Syntax vs. Custom Syntax

b)

HUTNForm.forms

```
Form {
    caption : "A simple form written in HUTN syntax"
    groups : Group {
        name : "A first group"
        items : Item {
            text : "Do you like this tutorial so far?"
            explanation : "It's ok to please the presenters instead of telling the truth"
            itemType : Choice {
                options : Option {
                    text : "Yes, its amazing!"
                }
                options : Option {
                    text
                }
            }
        }
    }
}
```

HUTN

HUTNForm.forms

```
FORM "A simple form written in HUTN syntax"
GROUP "A first group"

ITEM "Do you like this tutorial so far?"
     "It's ok to please the presenters instead of telling the truth" :
    CHOICE (
        "Yes, its amazing!",
        "Well, it's a little boring so far."
    )
```

Custom

Customizing Language Tooling – Attribute Mapping

d)

Putting strings into EString attributes is easy

How about EInt, EBoolean, EFloat, ..., custom data types?

- Solution A: Default mapping
  The generated classes use the conversion methods provided by Java (java.lang.Integer, Float etc.)

- Solution B: Customize the mapping using a token resolver

```
public void resolve(String lexem, EStructuralFeature feature,
    ITokenResolveResult result) {
  if ("yes".equals(lexem)) result.setResolvedToken(Boolean.TRUE);
  else result.setResolvedToken(Boolean.FALSE);
}
public String deResolve(Object value, EStructuralFeature feature,
    EObject container) {
    if (value == Boolean.TRUE) return "yes"; else return "no";
}
```

Customizing Language Tooling – Compilation

d)

Generating HTML or PDF documents from forms instances
Generated Builder stub is extended

Customizing Language Tooling – Interpretation

d)

Generating a wizard-style SWT-based application based on a language instance
Generated Interpreter stub is extended

DSLs built with EMFText can be used with

- GMF
- Acceleo
- ATL/QVT/…
- EMF Validation Framework
- Reuseware
- Refactory
- FeatureMapper
- …

basically any tool that is based on EMF (the ones above are the ones we've tried so far)

Challenges for MDSD

- Developers are required to use different tool machinery for DSLs and GPLs.
- Explicit references between DSL and GPL code are not supported. Their relations are, thus, hard to track and may become inconsistent
- DSLs can not reuse (parts of) the expressiveness of GPLs
- Naive embeddings of DSL code (e.g., in Strings) do not provide means for syntactic and semantic checking
- Interpreted DSL code is hard to debug
- Generated GPL code is hard to read, debug and maintain

Integrating DSLs and GPLs

Approach

(1) Use EMFText to *lift* GPLs to the technical space of DSLs
(2) Language integration by metamodel and grammar inheritance

Integrating DSLs and GPLs

Approach

(1) Use EMFText to *lift* GPLs to the technical space of DSLs
(2) Language integration by metamodel and grammar inheritance

JaMoPP: Lifting Java to the technical space of DSLs

- Ingredients:

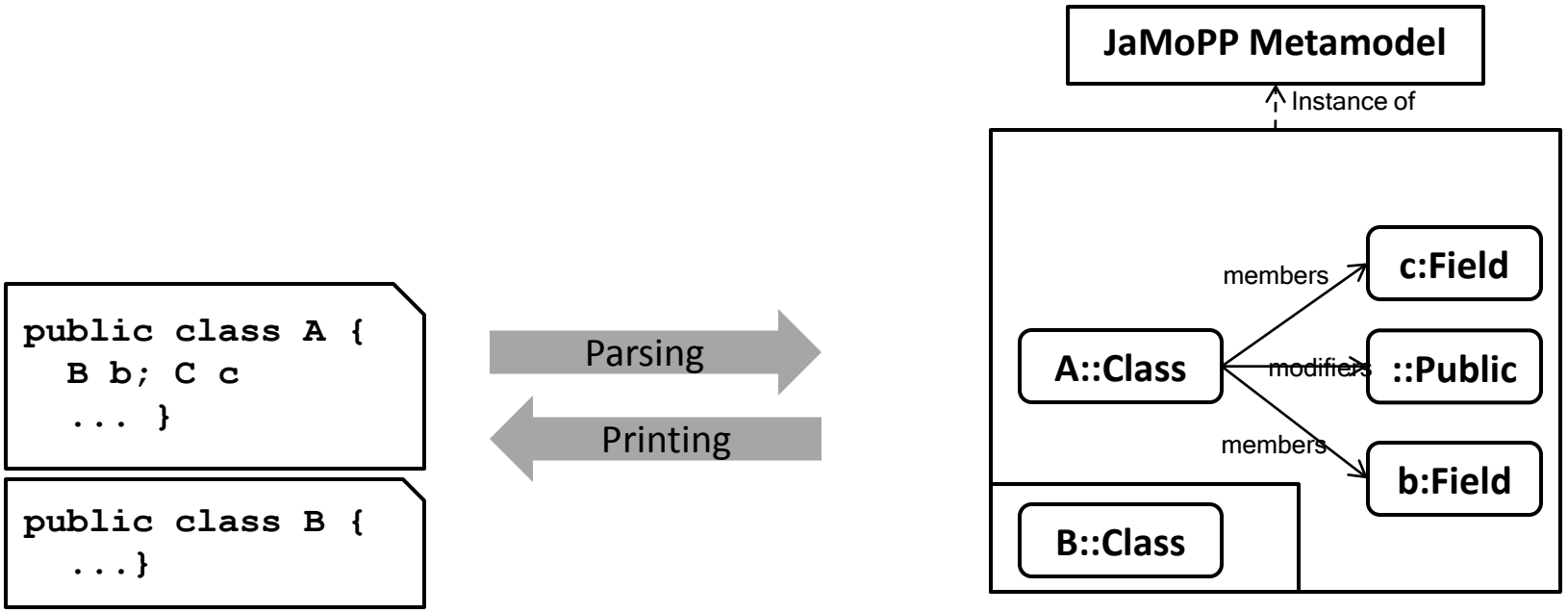
JaMoPP: Lifting Java to the technical space of DSLs

- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)

JaMoPP Metamodel

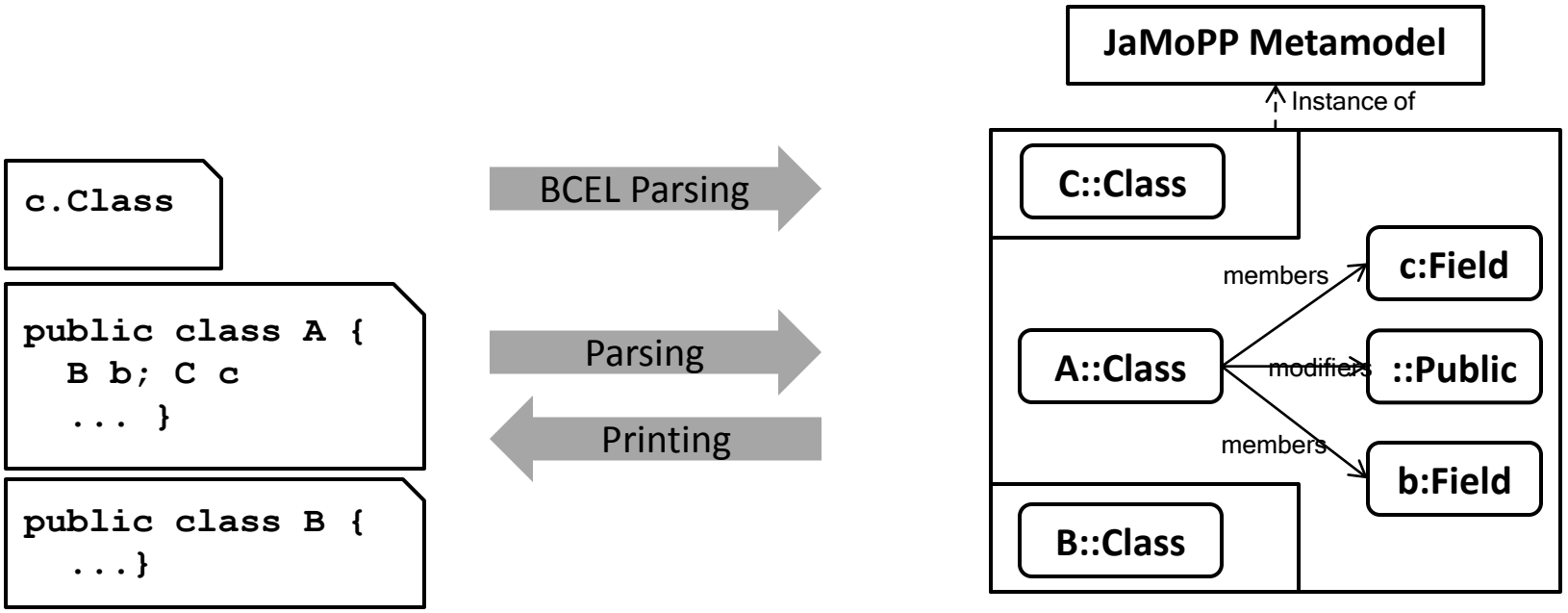
JaMoPP: Lifting Java to the technical space of DSLs

- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class



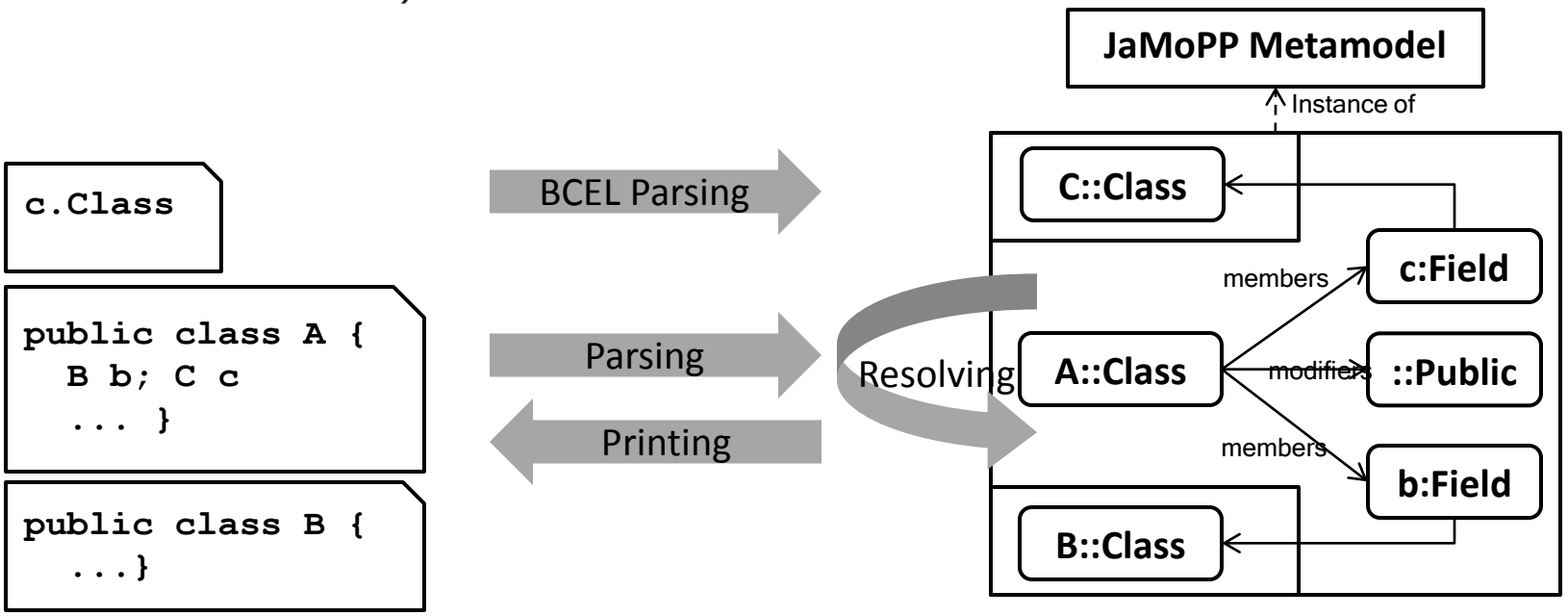
JaMoPP: Lifting Java to the technical space of DSLs

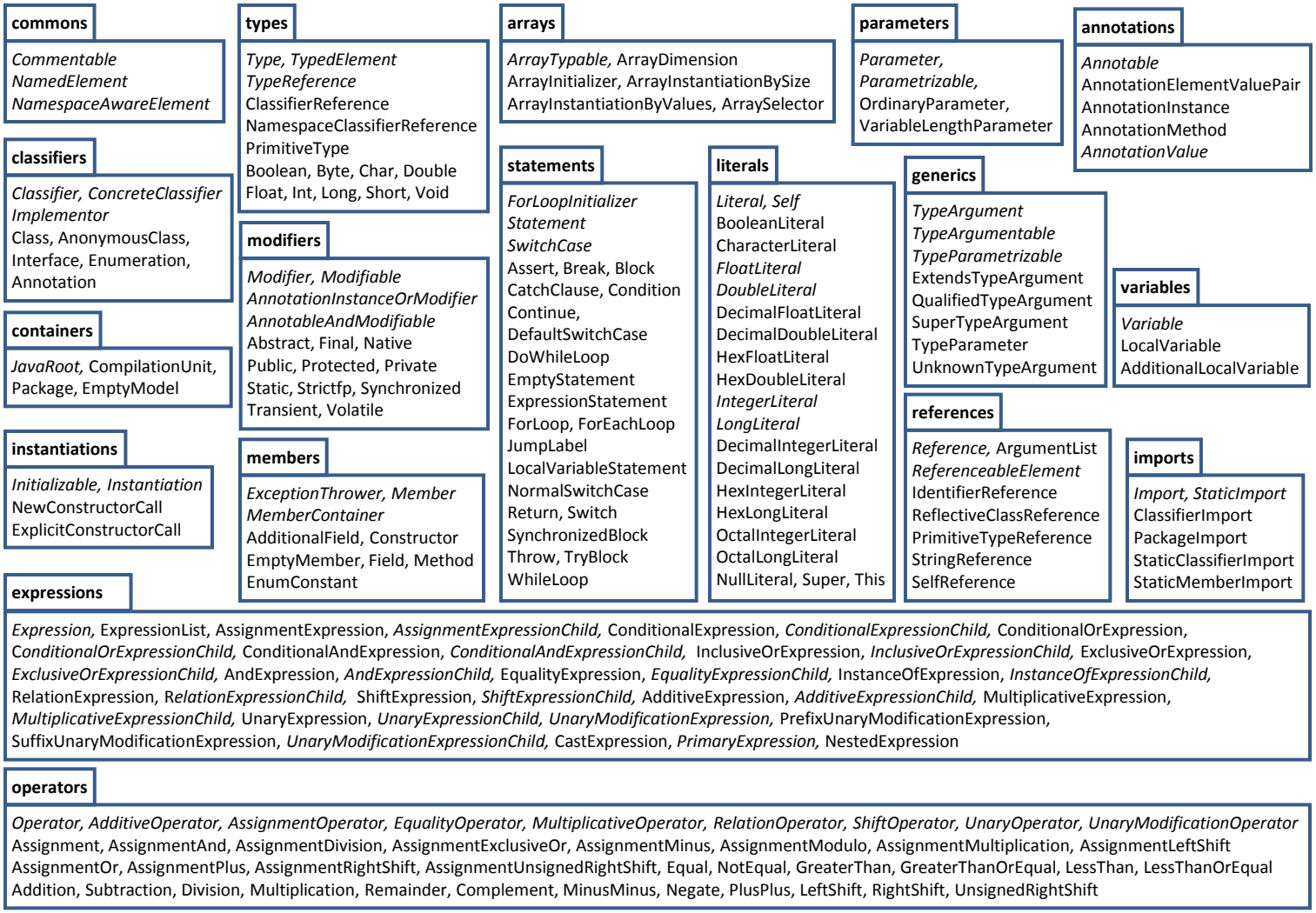
- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class
  - BCEL Bytecode-Parser – to handle third-party libraries



JaMoPP: Lifting Java to the technical space of DSLs

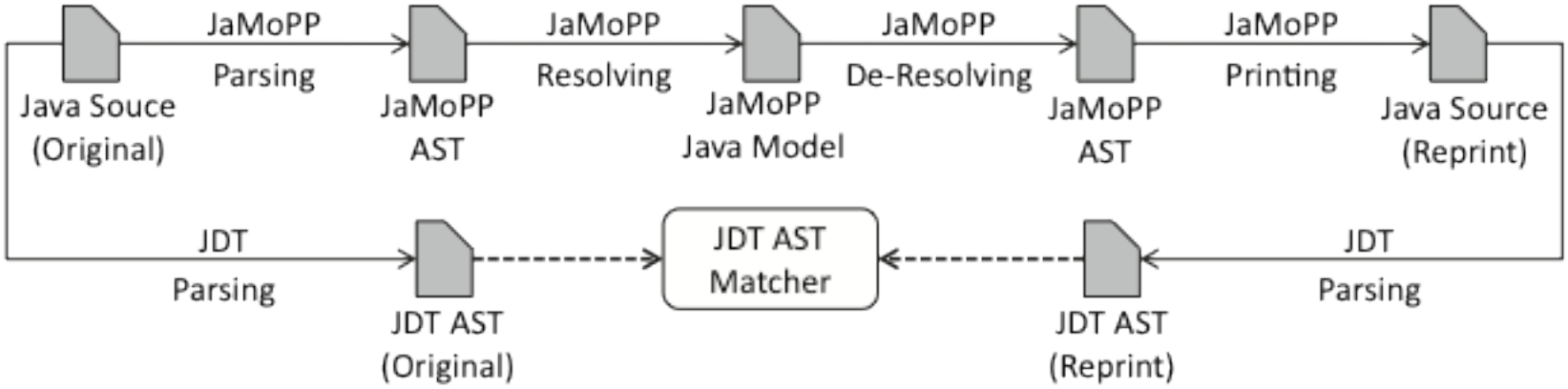
- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class
  - BCEL Bytecode-Parser – to handle third-party libraries
  - Reference Resolvers that implement java-specific scoping (static semantics)




commons
Commentable
NamedElement
NamespaceAwareElement
classifiers
Classifier, ConcreteClassifier
Implementor
Class, AnonymousClass,
Interface, Enumeration,
Annotation
containers
JavaRoot, CompilationUnit,
Package, EmptyModel
instantiations
Initializable, Instantiation
NewConstructorCall
ExplicitConstructorCall
types
Type, TypedElement
TypeReference
ClassifierReference
NamespaceClassifierReference
PrimitiveType
Boolean, Byte, Char, Double
Float, Int, Long, Short, Void
modifiers
Modifier, Modifiable
AnnotationInstanceOrModifier
AnnotableAndModifiable
Abstract, Final, Native
Public, Protected, Private
Static, Strictfp, Synchronized
Transient, Volatile
members
ExceptionThrower, Member
MemberContainer
AdditionalField, Constructor
EmptyMember, Field, Method
EnumConstant
arrays
ArrayTypable, ArrayDimension
ArrayInitializer, ArrayInstantiationBySize
ArrayInstantiationByValues, ArraySelector
statements
ForLoopInitializer
Statement
SwitchCase
Assert, Break, Block
CatchClause, Condition
Continue,
DefaultSwitchCase
DoWhileLoop
EmptyStatement
ExpressionStatement
ForLoop, ForEachLoop
JumpLabel
LocalVariableStatement
NormalSwitchCase
Return, Switch
SynchronizedBlock
Throw, TryBlock
WhileLoop
literals
Literal, Self
BooleanLiteral
CharacterLiteral
FloatLiteral
DoubleLiteral
DecimalFloatLiteral
DecimalDoubleLiteral
HexFloatLiteral
HexDoubleLiteral
IntegerLiteral
LongLiteral
DecimalIntegerLiteral
DecimalLongLiteral
HexIntegerLiteral
HexLongLiteral
OctalIntegerLiteral
OctalLongLiteral
NullLiteral, Super, This
parameters
Parameter,
Parametrizable,
OrdinaryParameter,
VariableLengthParameter
generics
TypeArgument
TypeArgumentable
TypeParametrizable
ExtendsTypeArgument
QualifiedTypeArgument
SuperTypeArgument
TypeParameter
UnknownTypeArgument
references
Reference, ArgumentList
ReferenceableElement
IdentifierReference
ReflectiveClassReference
PrimitiveTypeReference
StringReference
SelfReference
annotations
Annotable
AnnotationElementValuePair
AnnotationInstance
AnnotationMethod
AnnotationValue
variables
Variable
LocalVariable
AdditionalLocalVariable
imports
Import, StaticImport
ClassifierImport
PackageImport
StaticClassifierImport
StaticMemberImport
expressions
Expression, ExpressionList, AssignmentExpression, AssignmentExpressionChild, ConditionalExpression, ConditionalExpressionChild, ConditionalOrExpression,
ConditionalOrExpressionChild, ConditionalAndExpression, ConditionalAndExpressionChild, InclusiveOrExpression, InclusiveOrExpressionChild, ExclusiveOrExpression,
ExclusiveOrExpressionChild, AndExpression, AndExpressionChild, EqualityExpression, EqualityExpressionChild, InstanceOfExpression, InstanceOfExpressionChild,
RelationExpression, RelationExpressionChild, ShiftExpression, ShiftExpressionChild, AdditiveExpression, AdditiveExpressionChild, MultiplicativeExpression,
MultiplicativeExpressionChild, UnaryExpression, UnaryExpressionChild, UnaryModificationExpression, PrefixUnaryModificationExpression,
SuffixUnaryModificationExpression, UnaryModificationExpressionChild, CastExpression, PrimaryExpression, NestedExpression
operators
Operator, AdditiveOperator, AssignmentOperator, EqualityOperator, MultiplicativeOperator, RelationOperator, ShiftOperator, UnaryOperator, UnaryModificationOperator
Assignment, AssignmentAnd, AssignmentDivision, AssignmentExclusiveOr, AssignmentMinus, AssignmentModulo, AssignmentMultiplication, AssignmentLeftShift
AssignmentOr, AssignmentPlus, AssignmentRightShift, AssignmentUnsignedRightShift, Equal, NotEqual, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual
Addition, Subtraction, Division, Multiplication, Remainder, Complement, MinusMinus, Negate, PlusPlus, LeftShift, RightShift, UnsignedRightShift

JaMoPP: Rigorous Testing

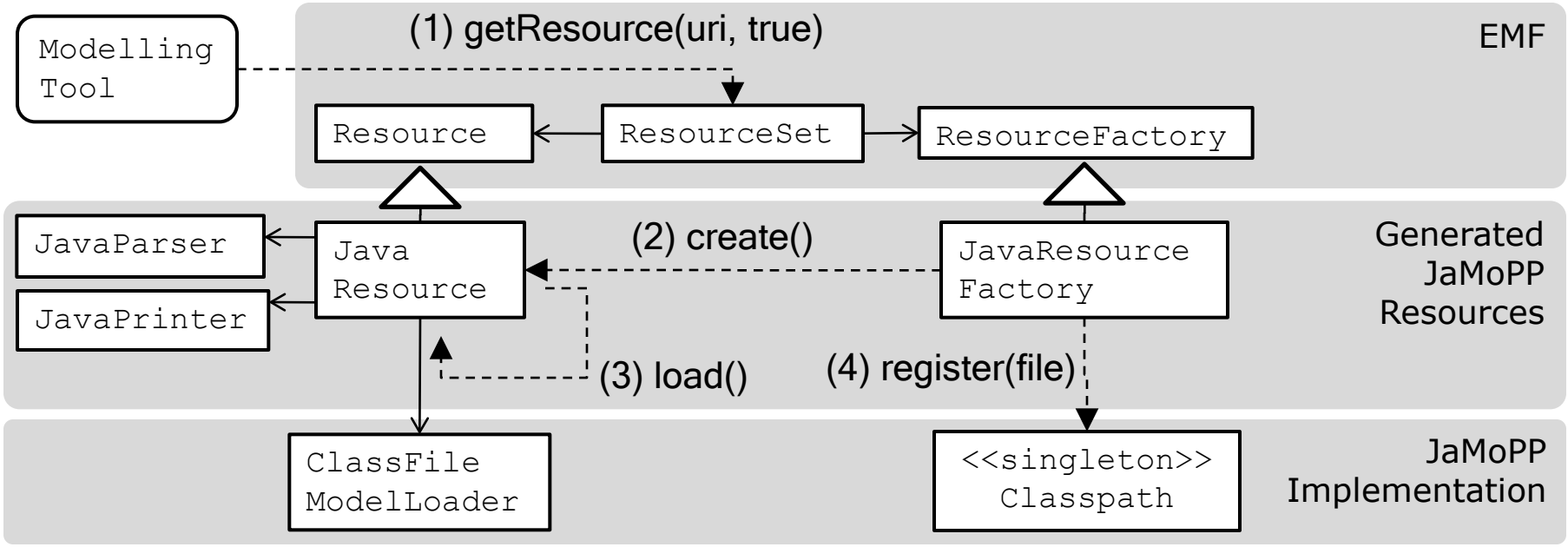
- Parsing public class A is easy, but parsing Java 5 is not (Unicode, Generics, Annotations and lots of weird things allowed by the JLS)
- We wanted JaMoPP to be complete



- Test suite:
  - 88.595 Java files (14.7 million non-empty lines including comments)
  - Open Source projects:
    AndroMDA 3.3, Apache Commons Math 1.2, Apache Struts 2.1.6, Apache Tomcat 6.0.18, Eclipse 3.4.1, Google Web Toolkit 1.5.3, JBoss 5.0.0 GA, Mantissa 7.2, Netbeans 6.5, Spring 3.0.0M1, Sun JDK 1.6.0 Update 7, XercesJ 2.9.1

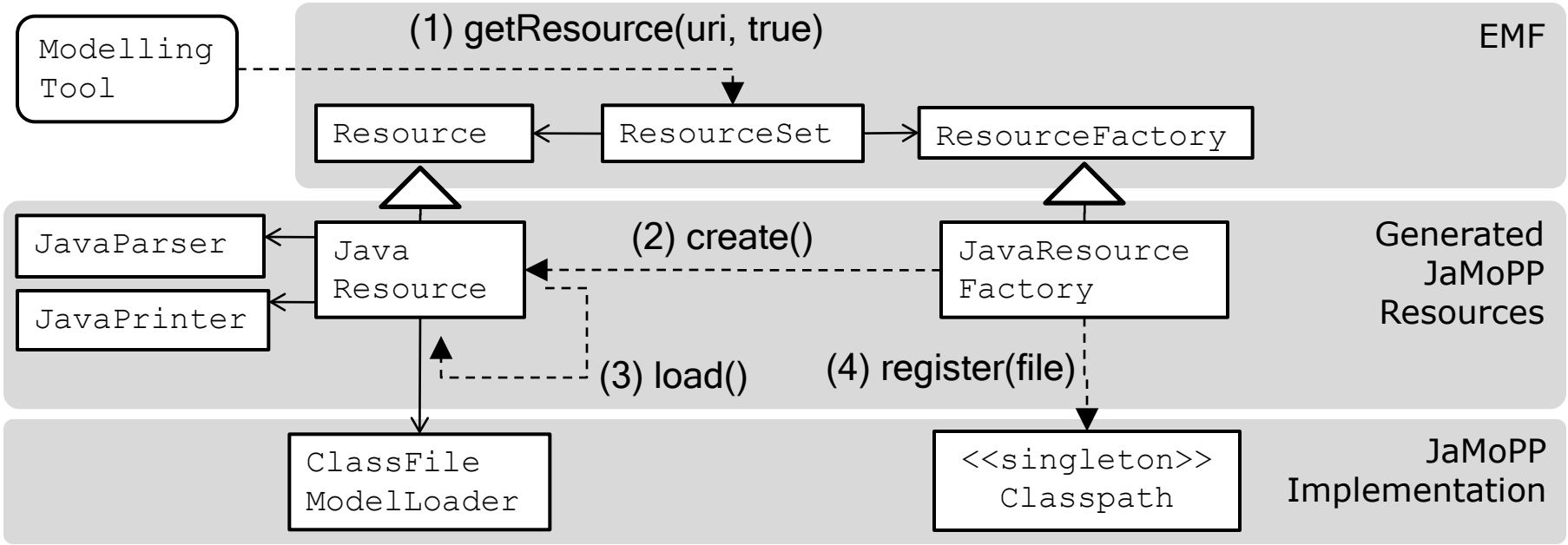
JaMoPP: Tool Integration

- JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools



JaMoPP: Tool Integration

- JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools

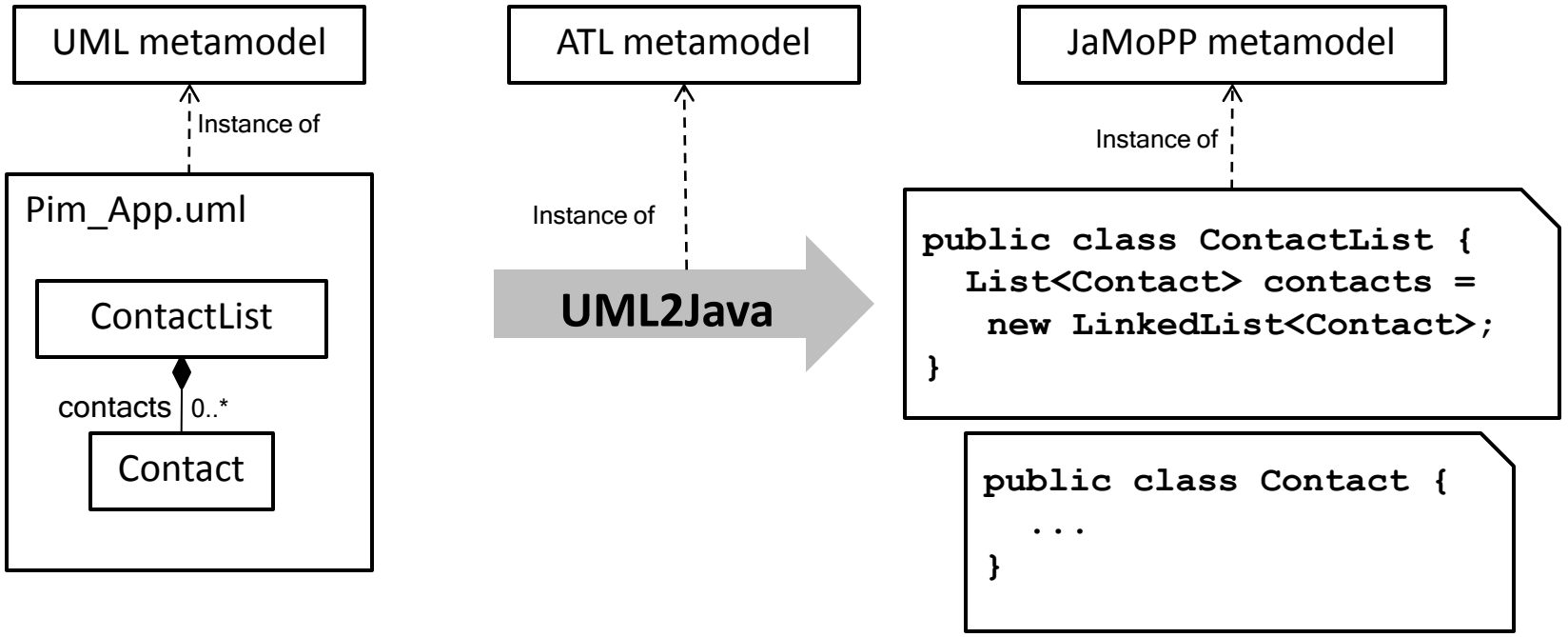


- Parsing Java files to models and Printing Java Files is simple

```
ResourceSet rs = new ResourceSetImpl();
Resource javaResource = rs.getResource(URI.createFileURI("A.java"),true);
//parsing
javaResource.save(); // printing
```

JaMoPP Application: Code generation (with ATL)

- Design UML model, apply M2M transformation, print JaMoPP model
- Syntactic and semantic correctness



JaMoPP Application: Code generation (with ATL)

- Design UML model, apply M2M transformation, print JaMoPP model

```
rule Property {
   from umlProperty : uml!Property
   to javaField : java!Field (
     name <- umlProperty.name,
     type <- typeReference
   ),

   typeReference : java!TypeReference (
     target <- if (umlProperty.upper = 1) then umlProperty.type
       else
         java!Package.allInstances()->any(p | p.name = 'java.lang').compilationUnits->collect(
           cu | cu.classifiers)->flatten()->any(c | c.name = 'LinkedList')
       endif,
     typeArguments <- if (umlProperty.upper = 1) then
         Sequence{} -- empty type argument list
       else
         Sequence{typeArgument}
       endif
   ),

   typeArgument : java!QualifiedTypeArgument(
     target <- umlProperty.type
   )
 }
```

JaMoPP Application: Code analysis (with OCL)

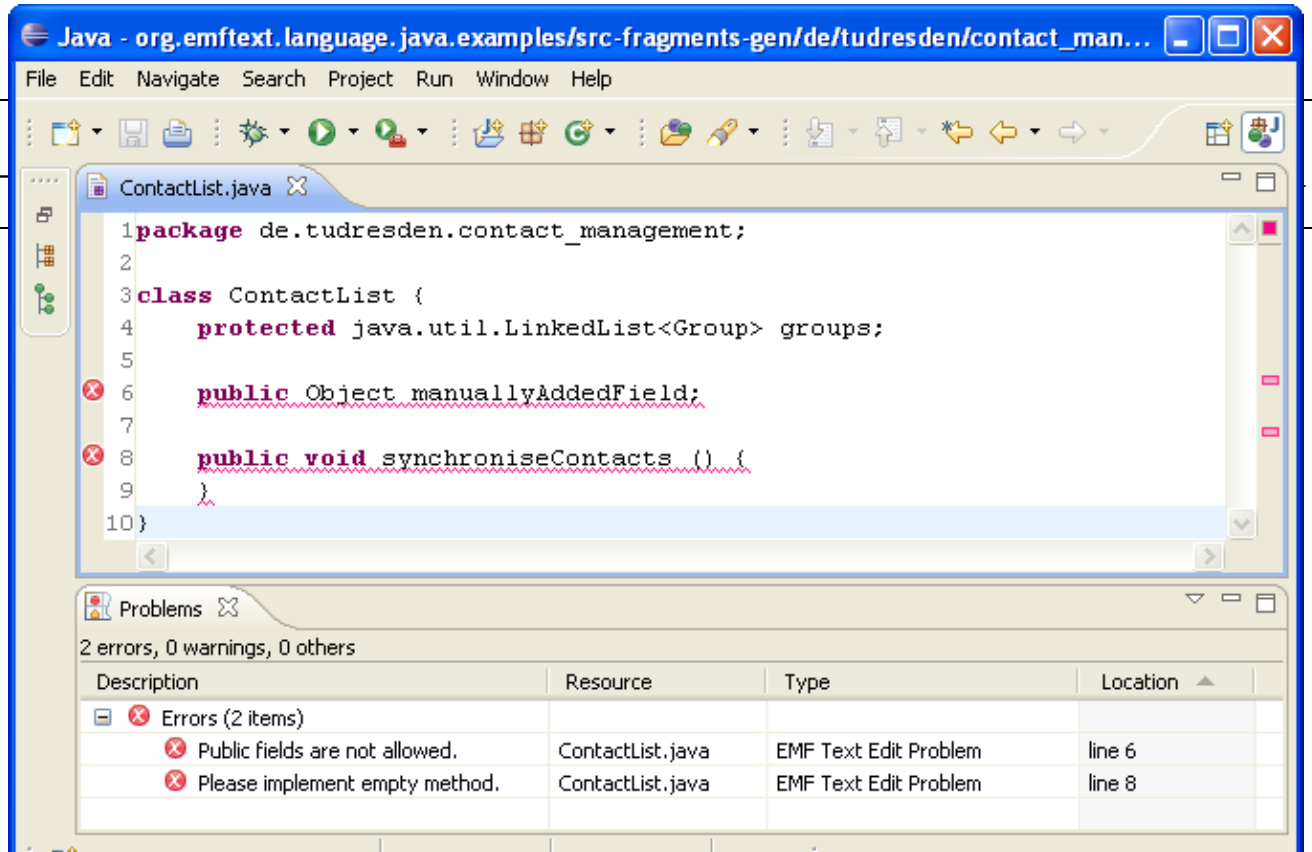
- Parse Java source files to model instances
- Run OCL queries to find undesired patterns

```
context members::Field inv:
   self->modifiers->select(m|m.oclIsKindOf(modifiers::Public))->size() = 0
```

JaMoPP Application: Code analysis (with OCL)

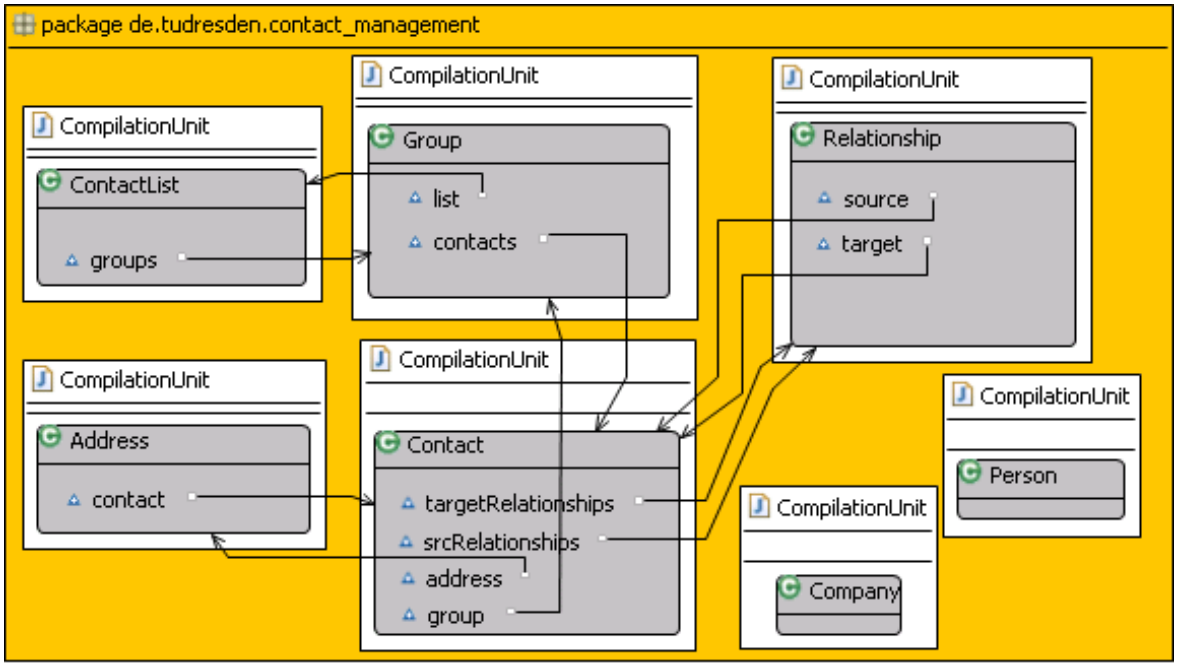
- Parse Java source files to model instances
- Run OCL queries to find undesired patterns

```
context
    self-                                              ze() = 0
```

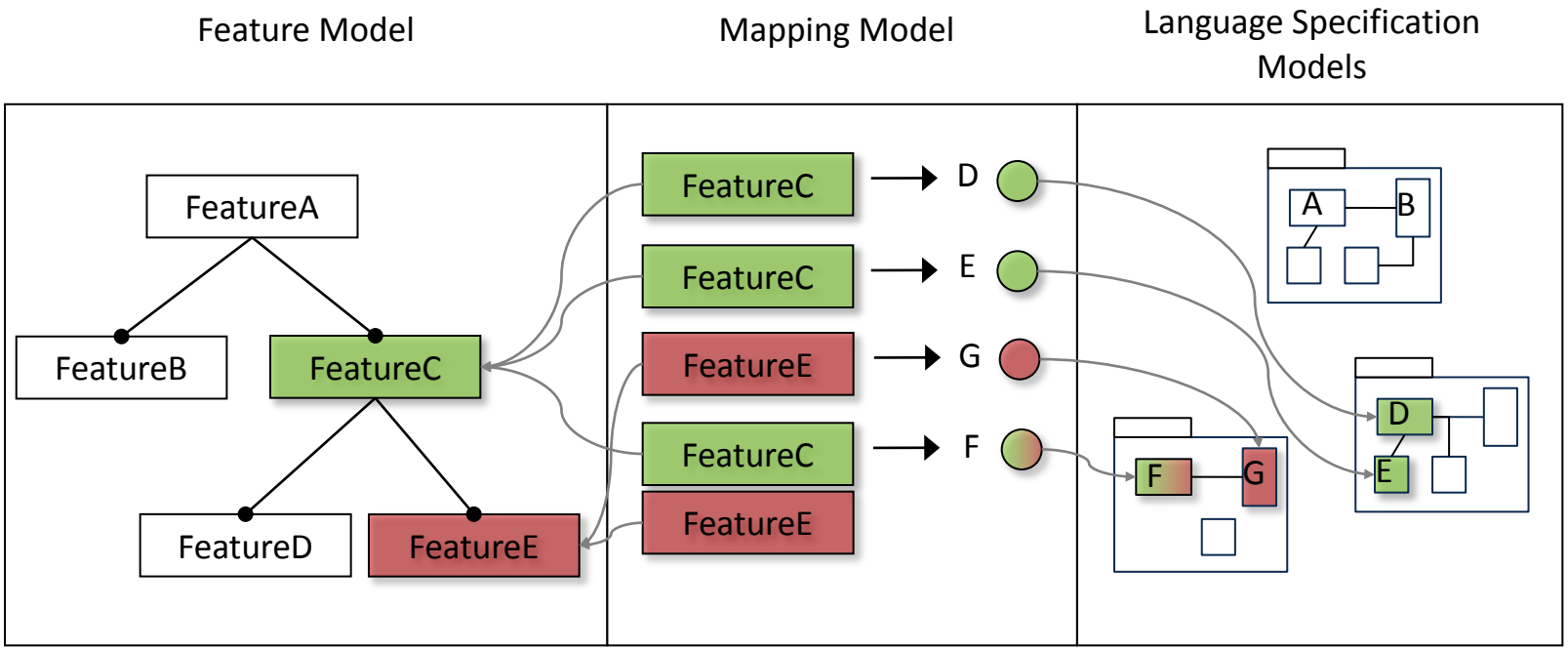


JaMoPP Application: Code visualisation (with GMF)

- Create .gmfgraph, gmftool, and gmfmap model
- Generate Graphical Editor for Java



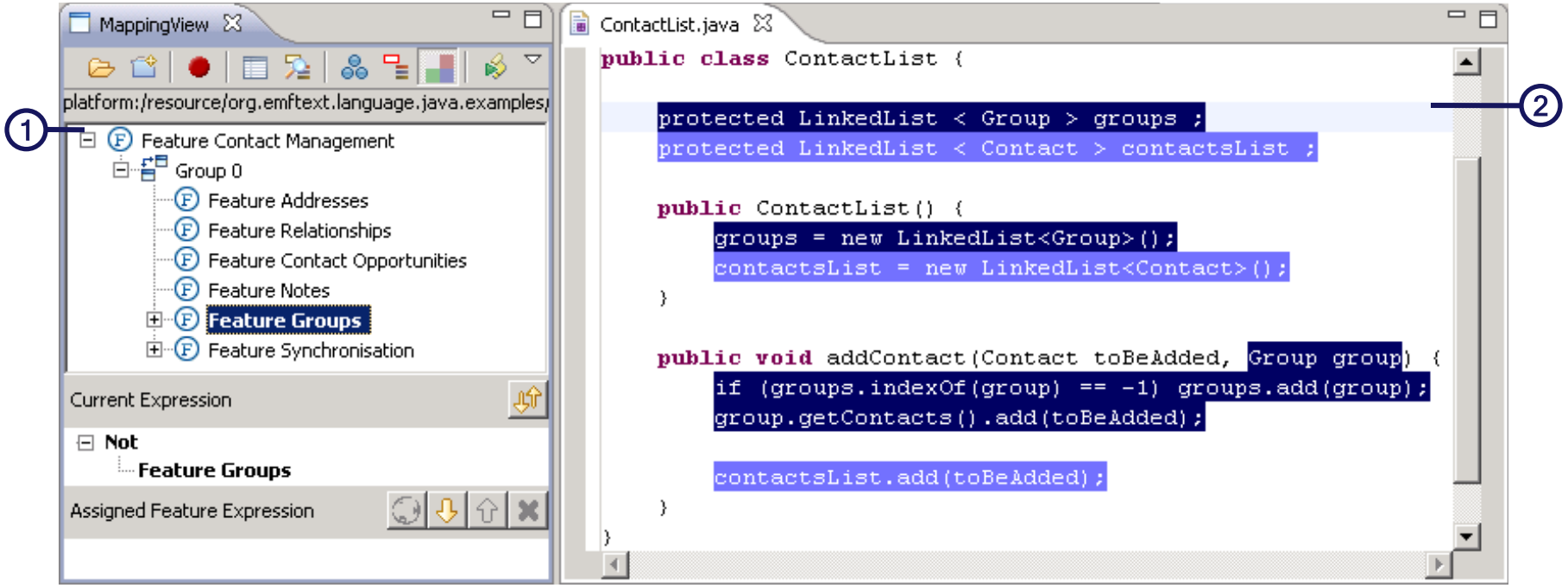
JaMoPP Application: Software Product Line Engineering (with FeatureMapper)



TECHNISCHE UNIVERSITÄT DRESDEN

emftext

JaMoPP Application: Software Product Line Engineering (with FeatureMapper)



1 – Feature Model

2 – EMFText Editor for Java (code for feature highlighted)

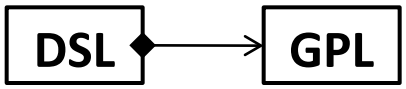
JaMoPP: What else can it be used for?

- Typesafe Template Languages
  - Same syntax as string-based templates
- Round-trip Support for template-based code generators
- Refactoring, Optimization using model transformations
- Traceability-related activities
  - Certification (Map code to the model elements)
  - Impact analysis (How much of the code will change if I do this?)
- Model-based compilation to byte code
- …

Some Language Integration Examples

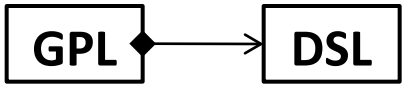
eJava

- Provides metamodels with EOperations implementations without touching the generated java files



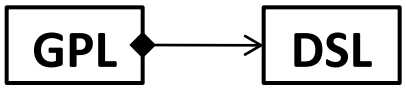
JavaTemplate

- Syntax safe templates with JaMoPP



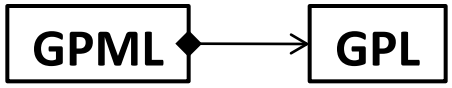
PropertiesJava

- Experimental extension for Java to define C# like properties



JavaBehaviour4UML

- An integration of JaMoPP and the Unified Modelling Language
- Methods can be directly added to Classes in class diagrams

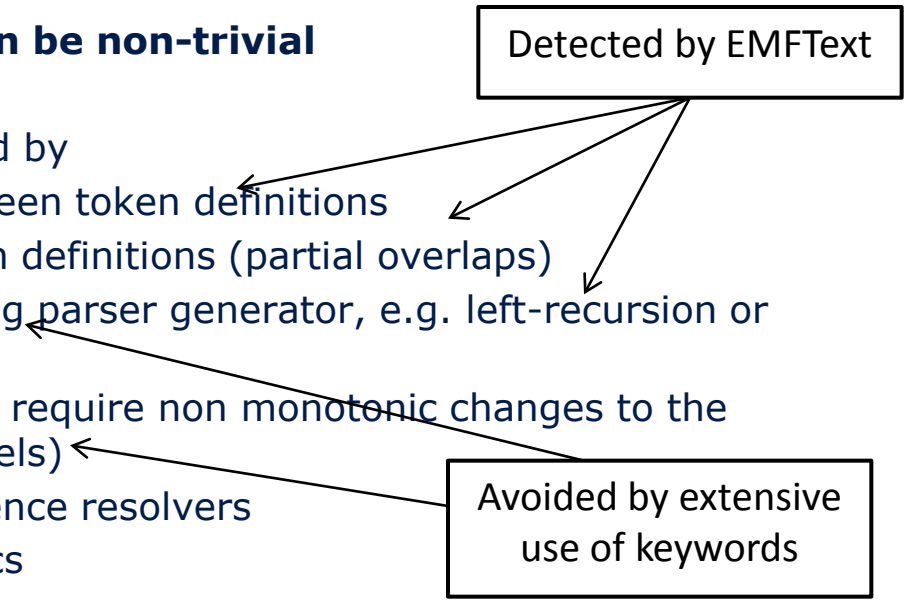


Disclamer: Pitfalls of Language Integration (with EMFText)

Syntax and model extensions can be non-trivial

In EMFText problems may be caused by

- Unexpected inclusions between token definitions
- Intersections between token definitions (partial overlaps)
- Problems with the underlying parser generator, e.g. left-recursion or extensive backtracking
- Ambiguous grammars (may require non monotonic changes to the grammar and the metamodels)
- Interference between reference resolvers
- Different language semantics



Alternative parsing technologies: Scannerless Parsing, Context-Aware Scanning, SDF/SGLR, MPS, Packrat Parsing, Parsing Expression Grammars ….

Thank you!