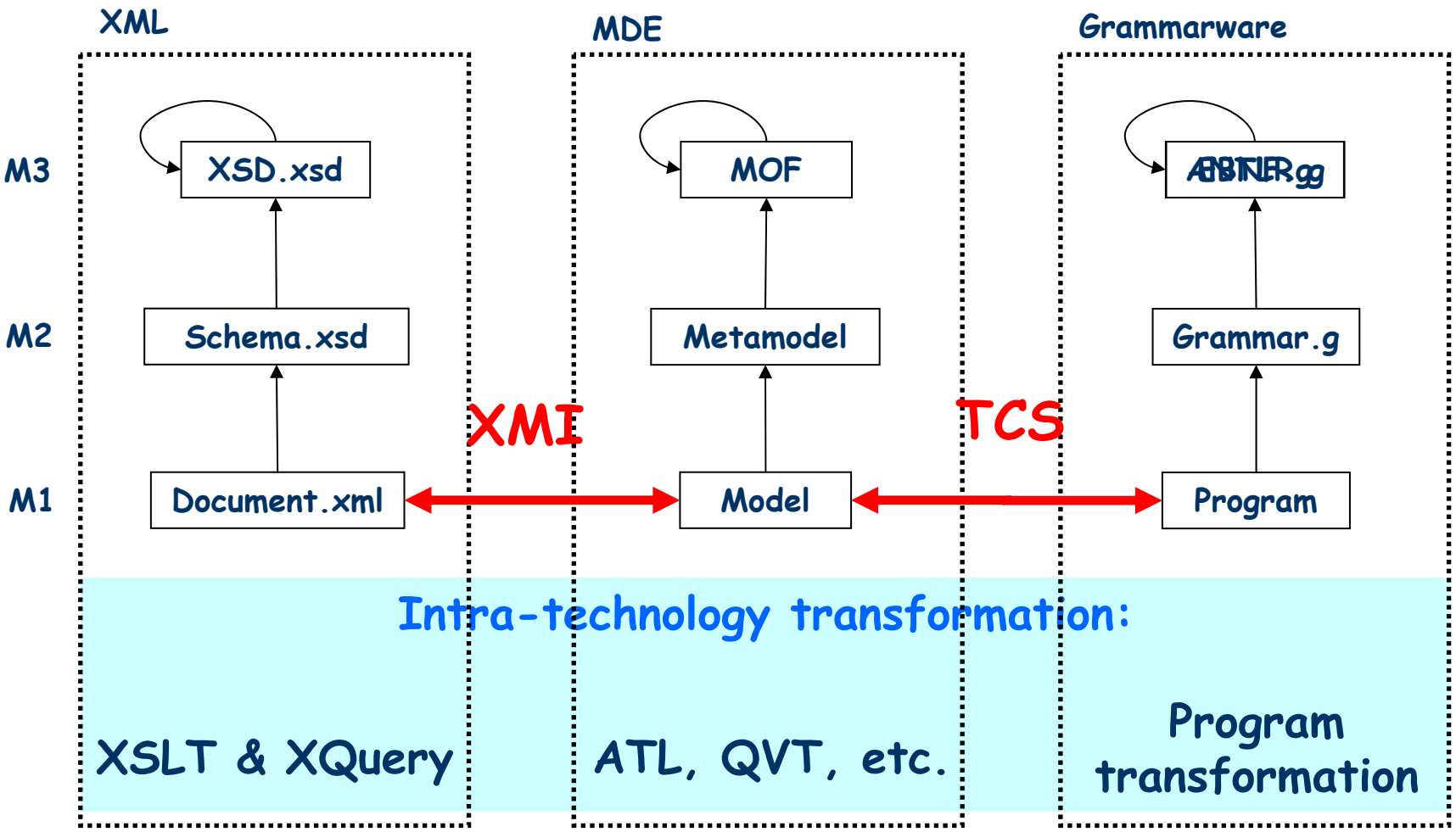
Textual Syntax Specification with TCS

Frédéric Jouault
AtlanMod (INRIA & EMN)
frederic.jouault@{inria,emn}.fr

Reading about TCS

- Jouault, F., Bézivin, J., and Kurtev, I.: *TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering*. In: GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering, Portland, Oregon, USA, pages 249—254. 2006.

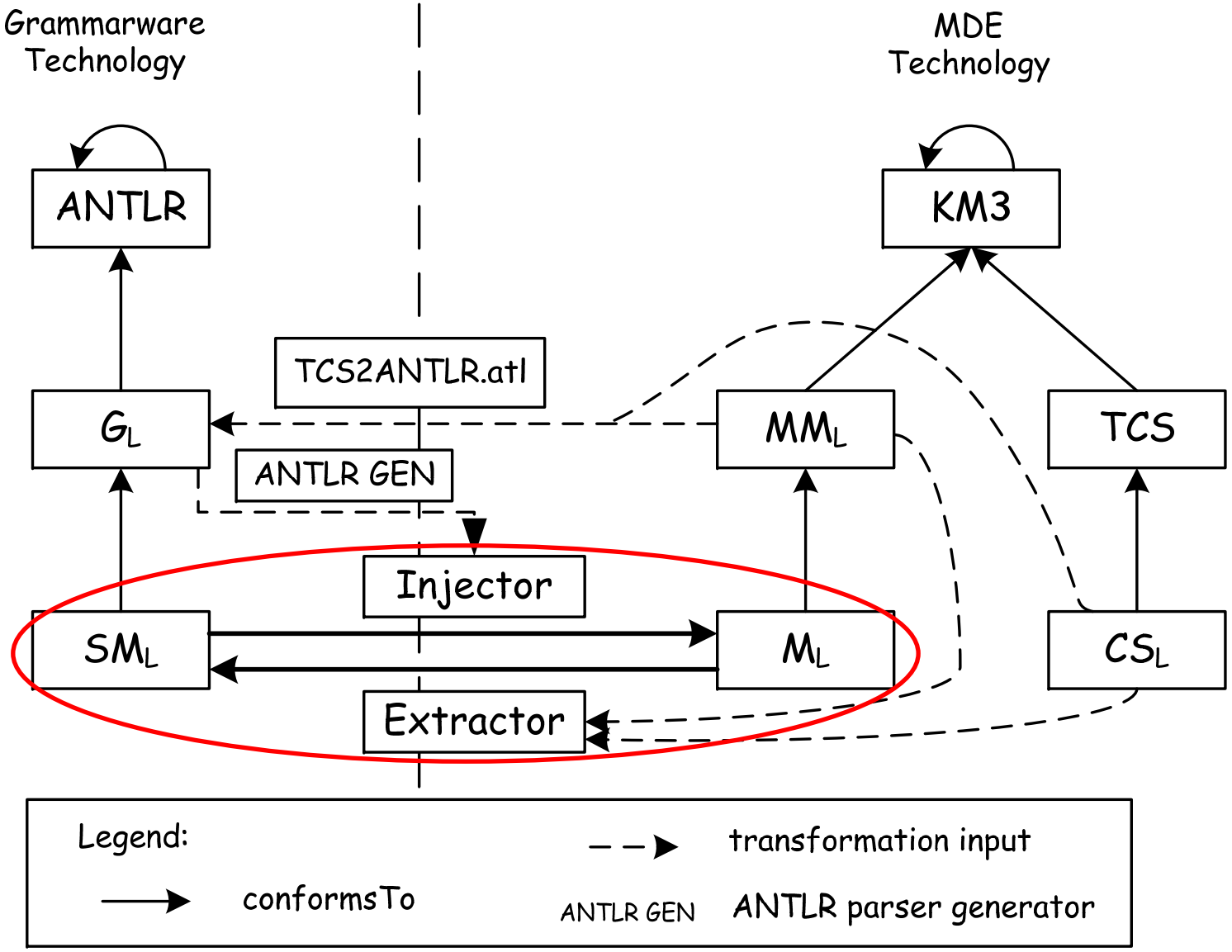
Context


XML
MDE
Grammarware
M3
M2
M1
XSD.xsd
MOF
Schema.xsd
Metamodel
Grammar.g
Document.xml
Model
Program
XMI
TCS
Intra-technology transformation:
XSLT & XQuery
ATL, QVT, etc.
Program transformation
conformsTo
Inter-technology transformation

TCS Overview

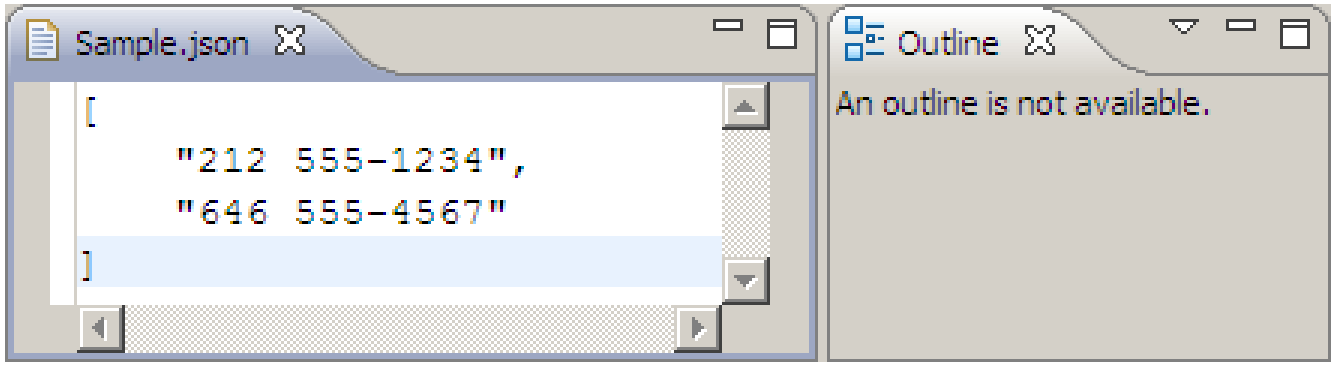
- Declaratively specify textual concrete syntaxes for metamodels:
  - Customizable/user-friendly,
  - Without repeating what is in the metamodels,
- Automatically parse programs into models:
  - Keep tracing information (e.g. line number),
- Automatically serialize models into programs:
  - Pretty printing (automatic indentation),
- Provide a featured editor.

TCS Overview: Megamodel

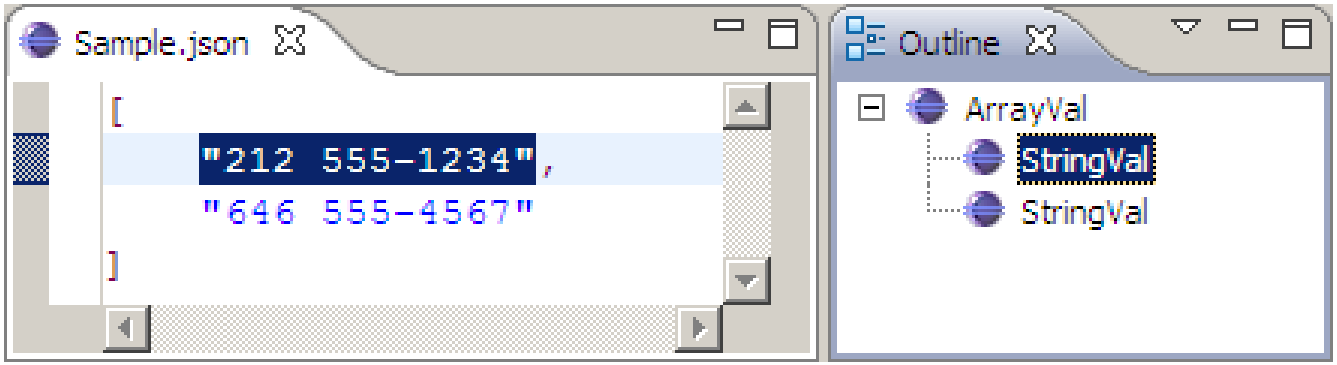


Example: JSON (JavaScript Object Notation)

- Without TCS:



- With TCS:



Example: excerpt from the definition of JSON (in KM3 and TCS)

Metamodel excerpt:

```
abstract class Value extends LocatedElement {}
class StringVal extends Value {
    attribute value : String;
}
class ArrayVal extends Value {
    reference elements[*] ordered container : Value;
}
```

Excerpt of corresponding TCS model:

```
template Value main abstract;
template StringVal
    :   value{as = stringSymbol}
    ;

template ArrayVal
    :   "[" elements{separator = ","} "]"
    ;
```

Excerpt of generated grammar:

```
value : stringVal | arrayVal;

stringVal : STRING;

arrayVal : LSQUARE (value (COMA value)*)?
RSQUARE;
```

Example: excerpt from the definition of KM3 (in KM3 and TCS)

Metamodel excerpt:

```
abstract class ModelElement {
        attribute name : String;
        reference "package" : Package oppositeOf contents;
}

class Package extends ModelElement {
        reference contents[*] ordered container : ModelElement oppositeOf "package";
}
```

Excerpt of corresponding TCS model:

```
template ModelElement abstract;

template Package main context
        :       "package" name "{"
                        contents
                "}"
        ;
```

Excerpt of generated grammar:

```
modelElement : package_ ;

package_ : "package" identifier LCURLY
        (modelElement ( modelElement )* |)
        RCURLY ;
```

Discussion

- There is little redundancy between the metamodel and the TCS:
  - The links between metamodel and TCS elements are done by name (e.g. Class & Template, properties, etc.),
  - The (primitive) type of the **name** attribute is known from the metamodel whereas its position in the text is known from the TCS,
  - The (complex) type and multiplicity of the **contents** reference are known from the metamodel whereas its position in the text is known from the TCS.
- Structural elements are defined in the metamodel.
- Syntax elements are defined in TCS:
  - Keywords as alpha-numeric strings between double quotes,
  - Symbols as non-alpha-numeric strings between double quotes.

More constructs: conditionals, symbol table handling

- Metamodel excerpt:

```
abstract class ModelElement {
        attribute name : String;
}

class Class extends Classifier {
        attribute isAbstract : Boolean;
        reference supertypes[*] : Class;
        reference structuralFeatures[*] ordered container : StructuralFeature
                                                                    oppositeOf owner;
}
```

- Corresponding TCS excerpt:

```
template Classifier abstract addToContext;

template Class context
        :       (isAbstract ? "abstract") "class" name
                (isDefined(supertypes) ?
                   "extends" supertypes{refersTo = name, separator = ",", autoCreate = never}
                )
                "{"
                        structuralFeatures
                "}"
        ;
```

More constructs: remarks

- TCS handles the symbol table:
  - Elements are marked as being contexts using the "context" keyword,
  - Elements are added in the current context using the "addToContext" keyword,
  - Elements are referred to by the value of one of their properties using the "refersTo" keyword.
- Conditional elements:
  - Using the value of Boolean properties (e.g. **isAbstract** here),
  - Testing the value of a property (not shown yet),
  - Testing whether a property is set for multiplicities 0-n, 1 ≤ n (e.g. **supertypes** here).
- Miscellaneous:
  - Separators can be specified for multi-valued properties (e.g. **supertypes** here).

Advanced constructs: operators, function templates

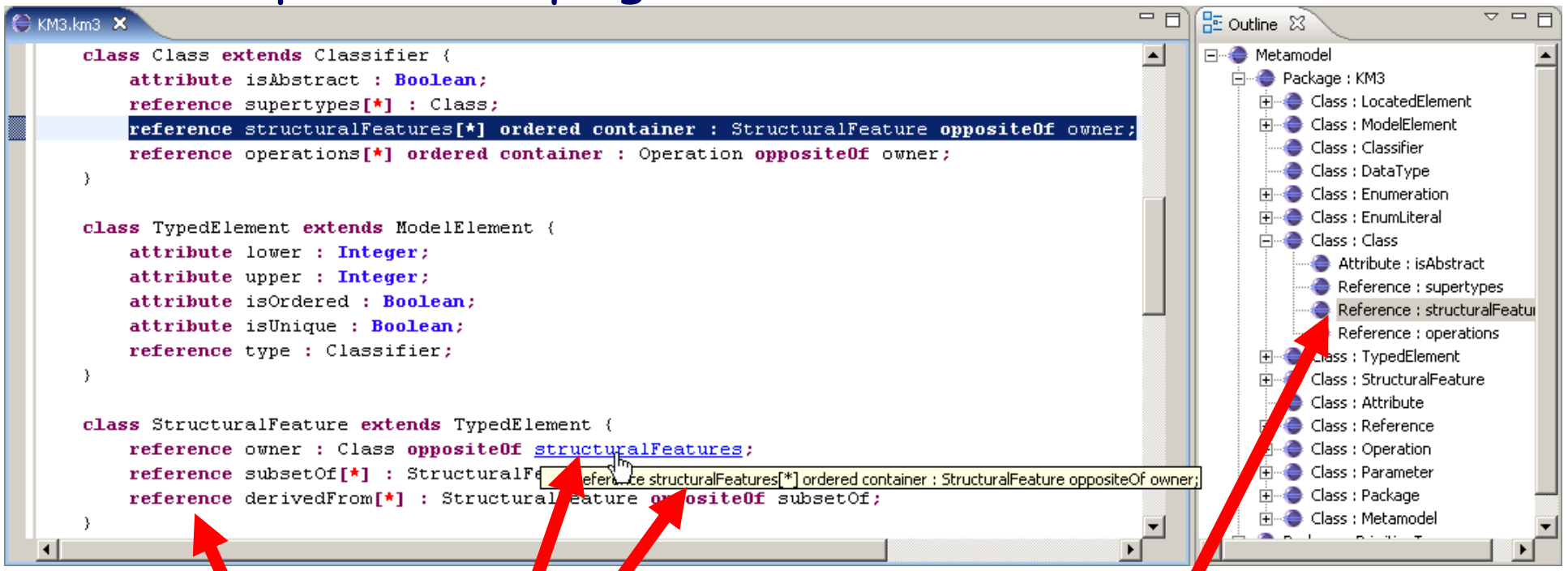
- TCS can also deal with operators:
  - Operators and their priorities are first defined,
  - operatorTemplates specify their usage,
  - The notation can be chosen:
    - Infix: 1 + x * 4
    - RPN: (+ 1 (* x 4))
- More complex symbol table handling can be performed:
  - Context importation (e.g. to deal with class inheritance),
  - Search for a target element in another context than the current one using the "lookIn" keyword (see the Reference template later in this presentation).
- Functions can be defined to factorize code (see later in this presentation).

Other features

- Pretty printing actually needs more than the syntax:
  - Indentation blocks can be defined,
  - Specific separators can be used (new line, blank, tab, etc.).
- Program-to-model traceability is provided:
  - The location attribute of each generated element is set with:
    - Line and column numbers of beginning,
    - Line and column numbers of ending,
  - Comments may be kept (and serialized back).
- Textual Generic Editor…

Textual Generic Editor for Eclipse

- Eclipse editor plugin:

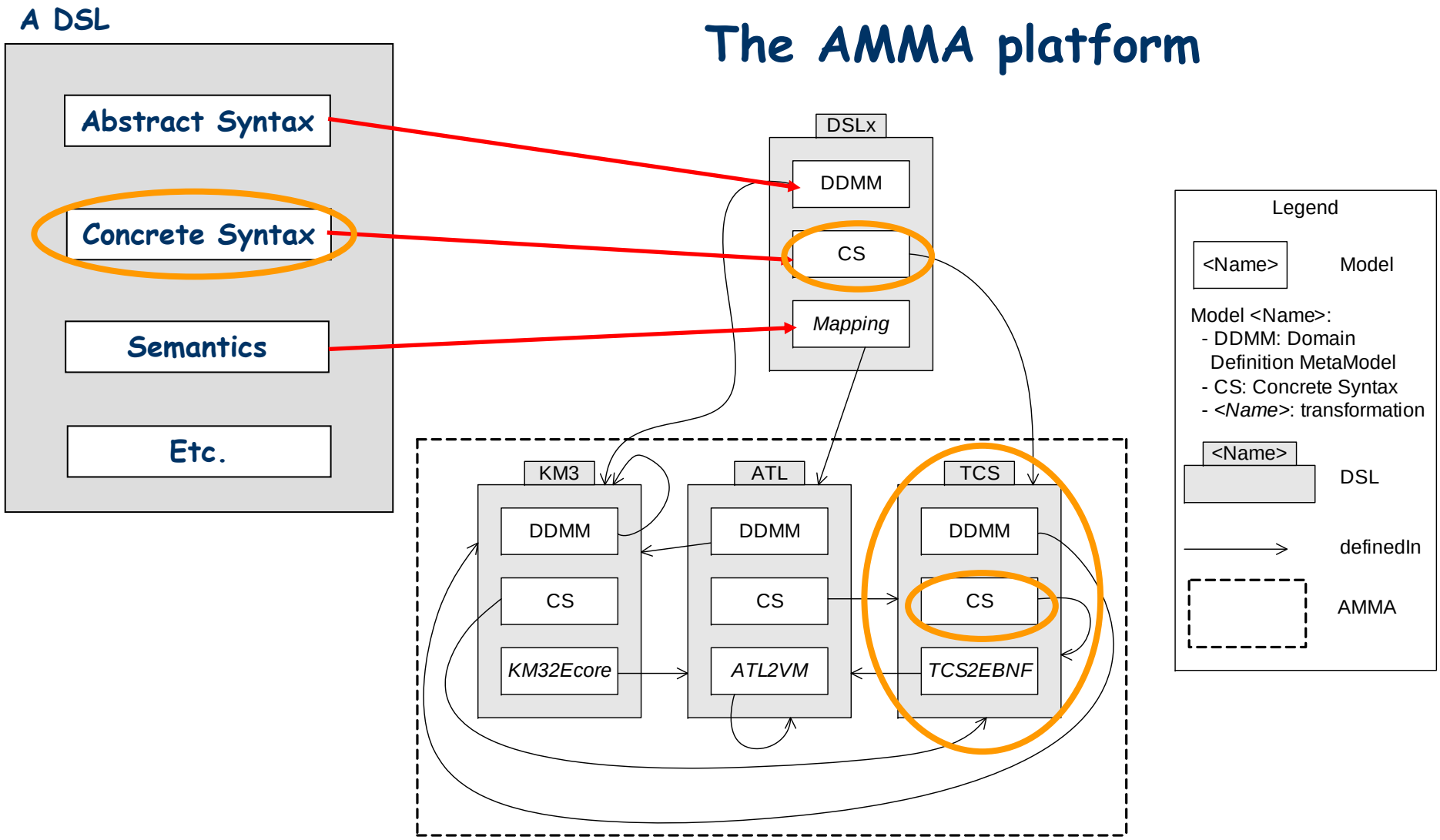


- Hyperlinks
- Text hovers
- Syntax highlighting
- Outline with bidirectional synchronization

Textual Generic Editor for Eclipse

- Language definition is an EMF model specifying:
  - Comment blocks,
  - Keywords list,
  - Highlighting format (font and color),
  - ➔ Can be generated by *TCS2Editor.atl*.
- Outline definition is an EMF model specifying:
  - Nodes to display,
  - Label format,
  - ➔ Can be generated by *KM32Outline.atl*.
- Uses TCS-generated parser:
  - To populate the outline,
  - To provide text hovers and hyperlinks.

Representing DSLs as sets of coordinated models: the role of TCS



Summary

- Modeling using a DSL can be done textually.
- Specifying a bidirectional mapping between a metamodel and a textual syntax is possible and rather straightforward using TCS.
- TCS has been used for several DSLs: KM3, TCS, ATL, ACG, AM3, Editor, SQLDDL, etc.
- Eclipse-based textual editor (TGE) for "free".

➔ This is how the KM3 editor works.

Demonstration: JSON

- Information from:

  http://en.wikipedia.org/wiki/JSON
- JSON's basic types are
  - Number (integer, real, or floating point)
  - String (double-quoted Unicode with backslash escapement)
  - Boolean (true and false)
  - Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
  - Object (collection of key/value pairs, comma-separated and enclosed in curly brackets)
  - null

Demonstration: JSON

- Information from:
  http://en.wikipedia.org/wiki/JSON
- Sample:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Lexical Considerations

- TCS makes the following assumption about the syntax:
  - Non-significant blanks (space, tabulation, newline)
  - Case-sensitive (keywords and identifiers)
- The default lexer provides:
  - Integers: [0-9]+
  - Floats: [0-9]+ '.' [0-9]+
  - Strings: between single quotes
  - Identifiers:
    - Simple form: [ALPHA][ALPHA|DIGIT]*
    - Quoted form: any string between double quotes

    ➔ The quoted form is especially useful to overcome collision between an identifier and a keyword

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Multivalued properties – without separator

- KM3 excerpt:

```
class List {
    attribute values[*] : Integer;
}
```

- TCS excerpt:

```
template List
    :    "{" values "}"
    ;
```

- Example:

{1 2 3 4}

Multivalued properties – with separator

- KM3 excerpt:

```
class List {
    attribute values[*] : Integer;
}
```

- TCS excerpt:

```
template List
    :    "{" values{separator = ","} "}"
    ;
```

- Example:

{1, 2, 3, 4}

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Conditionals

- Purpose: choice between two alternatives depending on property values
- Syntax:

  (*<condition>* ? *<then>* [: *<else>*])
- The *<condition>* can test:
  - A Boolean property
  - The presence of a value for a multivalued (i.e., [n-m] with m > 1) or an optional (i.e., [0-1]) property
  - The value of an Integer property
  - A conjunction (with the **and** keyword) of these
- The *<then>* part is mandatory (but can be empty)
- The *<else>* part is optional

Using Conditionals - Boolean Properties Examples

- Different keywords depending on the value
  - KM3 excerpt

```
attribute isLeft : Boolean;
```

  - TCS excerpt

```
(isLeft ? "left" : "right")
```

  ➔ The value of the property corresponds to an alternative between two different sequences of syntactic elements
- Keyword if *true*, nothing if *false*
  - KM3 excerpt

```
attribute isAbstract : Boolean;
```

  - TCS excerpt

```
(isAbstract ? "abstract")
```

  ➔ The value of the property corresponds to the presence or absence of syntactic elements (e.g., a keyword)

Using Conditionals - Optional Property Examples

- KM3 excerpt

```
reference supertype[0-1] : Class;
```

- TCS excerpt

```
(isDefined(supertype) ? "extends" supertype)
```

➔ This enables having keywords or symbols only if there is a value to represent

! The property must be used in the *<then>* part

Using Conditionals - Integer Property Example

- KM3 excerpt

```
attribute upper : Integer;
```

- TCS excerpt

```
(upper = -1 ? "*" : upper)
```

➔ This enables specific representation of special values

! The property must be used in the *<else>* part but not in the *<then>* part

Advanced Usage of Conditionals: KM3 Multiplicities (simplified)

- Definition of *lower* and *upper* in KM3:

```
attribute lower : Integer;
attribute upper : Integer;
```

- Representation of *lower* and *upper* in TCS:

```
(lower = 1 and upper = 1 ?
    -- nothing
:(lower = 0 and upper = -1 ?
    "[" "*" "]"
:(upper = -1 ?
    "[" lower "-" "*" "]"
:
    "[" lower "-" upper "]"
)))
```

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Functions

- Purpose: reuse parts of concrete syntax specification
- Syntax:
  - Definition:

    function *<name>*(*<type>*)
    : *<body>*
    ;
  - Call:

    $*<name>*
- *<name>* is used to call the function
- *<type>* specifies the Class for which calling the function is valid (and therefore the properties that can be used in <body>)
- *<body>* defines a textual representation

Functions - Example

- KM3 *StructuralFeature* (simplified)

```
abstract class StructuralFeature {
    attribute name : String;
    attribute lower : Integer;
    attribute upper : Integer;
} -- Both Attribute and Reference extend StructuralFeature.
class Attribute extends StructuralFeature {}
class Reference extends StructuralFeature {}
```

- Corresponding TCS excerpt:

```
template Attribute
    :       "attribute" name $multiplicity
    ;
template Reference
    :       "reference" name $multiplicity
    ;
function multiplicity(StructuralFeature) -- We must specify the
    :       "[" lower "-" upper "]"  -- type for which it works.
    ; -- The representation of multiplicity is factorized.
```

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Representing Expressions – introducing *Calc*

- Let us consider *Calc*: a simple expression language with addition, subtraction, multiplication, division, negation, and exponentiation
- The abstract syntax of *Calc* can be represented as the following grammar:

```
expression ::=
        expression '+' expression
      | expression '-' expression
      | expression '*' expression
      | expression '/' expression
      | '-' expression
      | expression '^' expression
      | '(' expression ')'
      | INTEGER
```

- This grammar cannot be used to parse *Calc* sentences because it is ambiguous: different derivations can be constructed for the same sentence
- Ambiguities in expressions are generally solved by introducing operator priority (or precedence) and associativity

Operator priority and associativity in *Calc*

Category	Operators	Associativity	Examples
Exponentiation	^	right	x ^ y ^ z = x ^ (y ^ z)
Unary negation	-	right	- - x = - (- x)
Multiplicative	* /	left	x * y * z = (x * y) * z x / y / z = (x / y) / z
Additive	+ -	left	x + y + z = (x + y) + z x - y - z = (x - y) - z

increasing priority

x + y * z = x + (y * z)
x * y ^ z = x * (y ^ z)
- x ^ y = - (x ^ y)

Note: This table corresponds to what is commonly used in math.

Designing a grammar that can parse *Calc* sentences

- The solution depends on:
  - The class of grammars (e.g., LL, LR)
  - The actual technology (i.e., parser generator), which may add some constraints or offer additional possibilities
- Usually, it is necessary to rewrite the grammar:
  - One production rule per priority
  - Specific constructs depending on the associativity
- Sometimes, it is possible to declaratively specify priority and associativity (e.g., with Yacc)
- Extra work is generally necessary to construct Abstract Syntax Trees (ASTs)
  - Because the additional production rules often introduce unwanted nodes

Calc with LALR(1) parsers (e.g., Yacc) – expanded version

```
expression ::= additive
additive ::= multiplicative
    | additive '+' multiplicative
    | additive '-' multiplicative
    /* left recursion is used for left associativity */
multiplicative ::= unaryNegation
    | multiplicative '*' unaryNegation
    | multiplicative '/' unaryNegation
unaryNegation ::= exponentiation
    | '-' unaryNegation
exponentiation ::= primaryExpression
    | primaryExpression '^' exponentiation
    /* right recursion is used for right associativity */
primaryExpression ::= INTEGER | '(' expression ')'
```

Calc with LALR(1) parsers (e.g., Yacc) – automatic version

```
/* operators are listed in increasing priority order */
%left '+' '-'
%left '*' '/'
%right NEG
%right '^'
expression ::=
          expression '+' expression
        | expression '-' expression
        | expression '*' expression
        | expression '/' expression
        | '-' expression %prec NEG
        | expression '^' expression
        | '(' expression ')'
        | INTEGER
```

Calc with LL(k) parsers (e.g., ANTLR)

```
expression ::= additive
additive ::= multiplicative (('+' | '-') multiplicative) *
   /* repetition is used for left associativity */
multiplicative ::= unaryNegation (('*' | '/') unaryNegation)*
unaryNegation ::= exponentiation
   | '-' unaryNegation
exponentiation ::= primaryExp ('^' exponentiation)?
   /* right recursion is used for right associativity */
primaryExp ::= INTEGER | '(' expression ')'
```

Using KM3 and TCS to represent *Calc*

- The abstract syntax is specified as a KM3 metamodel
  - This defines the structure of the abstract syntax,
  - With few constraints with respect to the concrete syntax
- A projector enables the translation to and from the Grammarware technical space
  - Translates programs into models
  - Translates models into programs
- TCS is this projector, with the following features:
  - Customizable syntax: you can often use the syntax you would have specified with a grammar
  - Bidirectional: a single TCS model is enough for both directions (i.e., program-to-model, and model-to-program)

Representing Expressions – a metamodel for *Calc*

```
abstract class Expression {}

class IntegerExp extends Expression {
    attribute value : Integer;
}

abstract class OperatorCallExp extends Expression {
    attribute opName : String;
}

class BinaryOperatorCallExp extends OperatorCallExp {
    reference left container : Expression;
    reference right container : Expression;
}

class UnaryOperatorCallExp extends OperatorCallExp {
    reference operand container : Expression;
}
```

Priority and associativity with TCS

```
operators {
    priority 0, right {        -- highest priority
        opExp  = caret, 2;
    }    -- ^ operator name
    priority 1, right {        -- right associative
        opMinus1 = minus, 1;
    }                  -- ^ operator symbol
    priority 2 { -- left associative (by default)
        opStar = star, 2;
        opSlash = slash, 2;
    }                      -- ^ operator arity
    priority 3 {
        opPlus = plus, 2;
        opMinus2 = minus, 2;
    }
}
```

Priority and associativity with TCS (alternative encoding)

```
operators {
    priority 0, right {          -- highest priority
        opExp  = "^", 2;
    }
    priority 1, right {
        opMinus1 = "-", 1;
    }                -- ^ operator symbol
    priority 2 {
        opStar = "*", 2;
        opSlash = "/", 2;
    }
    priority 3 {
        opPlus = "+", 2;
        opMinus2 = "-", 2;
    }
}
```

Concrete syntax of *Calc* elements with TCS

```
template Expression abstract operatored;
        -- The root of the expressions inheritance hierarchy
        -- must be declared « operatored ».

template IntegerExp
   :    value
   ;

operatorTemplate BinaryOperatorCallExp(operators =
   opStar opSlash
   opPlus opMinus2
   opExp,        -- source identifies the first operand
   source = 'left', storeOpTo = opName,
   storeRightTo = 'right');
-- storeRightTo identifies the second operand
operatorTemplate UnaryOperatorCallExp(operators =
   opMinus1,
   source = operand, storeOpTo = opName);
   -- storeOpTo identifies operator (a String property)
```

Additional considerations about operators

- Some expressions require special handling
  - The generated grammar makes use of recursion to encode priority and associativity ➔ TCS needs to know about other uses of recursion
- Use operators if you have left recursion (i.e., an expression on the left)
  - Example: 1.toString()

```
operatorTemplate OperationCallExp
    (operators = opPoint, source = 'source')
    :      operationName "(" ")"
    ;
```

- Use **nonPrimary** if you have right recursion (i.e., an expression on the right)
  - Example: let x = 1 in x * 2

```
template LetExp nonPrimary
    :      "let" varName "=" value "in" in
    ;
```

Calc with LL(k) parsers (e.g., ANTLR) – handling nonPrimary

```
expression ::= additive
   | nonPrimary /* if we have nonPrimary expressions */
additive ::= multiplicative (('+' | '-') multiplicative) *
   /* repetition is used for left associativity */
multiplicative ::= unaryNegation (('*' | '/') unaryNegation)*
unaryNegation ::= exponentiation
   | '-' unaryNegation
exponentiation ::= primaryExp ('^' exponentiation)?
   /* right recursion is used for right associativity */
primaryExp ::= INTEGER | '(' expression ')'
nonPrimary ::= 'let' IDENT '=' expression 'in' expression
```

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Pretty Printing

- Purpose: customize the output of the model-to-program translation
- Only blanks are customizable ➔ this generally does not impact the syntax (i.e., the grammar)
- There are three pretty printing constructs in TCS:
  - Blocks: used to specify indentation
  - Special symbols can have specific options for spaces
  - Custom separators to force specific handling of blanks at specific places

Pretty Printing: without Blocks

- KM3 excerpt:

```
class List {
    attribute values[*] : Integer;
}
```

- TCS excerpt:

```
template List
    :     "{" values "}"
    ;
```

- Sample output:

{1 2 3 4}

Pretty Printing: with Blocks

- KM3 excerpt:

```
class List {
    attribute values[*] : Integer;
}
```

- TCS excerpt:

```
template List
    :     "{" [ values ] "}"
    ; -- [] defines an indentation block
```

- Sample output:

```
{
    1
    2
    3
    4
}
```

Pretty Printing: with Blocks, and Options

- KM3 excerpt:

```
class List {
    attribute values[*] : Integer;
}
```

- TCS excerpt:

```
template List
    :     "{" [ values ] {indentIncr = 0} "}"
    ;-- indentIncr specifies the indentation increment
```

- Sample output:

```
{
1
2
3
4
}
```

Pretty Printing: Special Symbols and Spaces

- By default, no blanks are inserted

```
symbols {
    coma        = ",";
    lparen      = "(";
    rparen      = ")";
    semi        = ";";
    colon       = ":";
    -- operator symbols
    minus       = "-";
    star        = "*";
    slash       = "/";
    plus        = "+";
    caret       = "^";
}
```

Example: 1+(2*3)(5),6

Pretty Printing: Special Symbols and Spaces

- Each symbol can specify how to serialize spaces before it is printed:
  - `leftSpace` to force a space before

  OR
  - `leftNone` to prevent having a space before (even if the previous symbol has `rightSpace`, see below)
- Each symbol can specify how to serialize spaces after it is printed:
  - `rightSpace` to force a space after
  - `rightNone` to prevent having a space after (even if the next symbol has `leftSpace`)

Pretty Printing: Special Symbols and Spaces

```
symbols {
    coma        = "," : leftNone, rightSpace;
    lparen      = "(";
    rparen      = ")" : leftNone, rightSpace;
    semi        = ";" : leftNone, rightSpace;
    colon       = ":" : leftSpace, rightSpace;
    -- operator symbols
    minus       = "-" : leftSpace, rightSpace;
    star        = "*" : leftSpace, rightSpace;
    slash       = "/" : leftSpace, rightSpace;
    plus        = "+" : leftSpace, rightSpace;
    caret       = "^" : leftSpace, rightSpace;
}
```

Example: 1 + (2 * 3) (5), 6

PrettyPrinting: Custom Separators

- <space>: to force a space
- <no_space>: to prevent a space from being used
- <newline>: to force a new line
- <tab>: to force a tabulation

Additional remarks

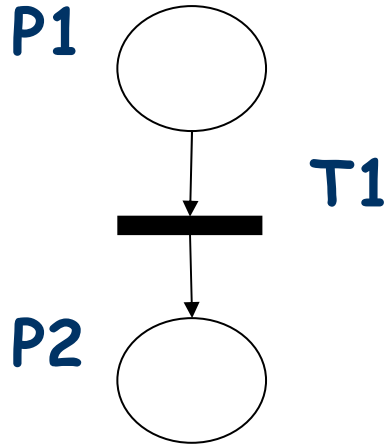
- Do not hesitate to have a look at the generated grammar to see how TCS works
  - how conditionals are encoded,
  - how operators and priorities are encoded,
  - etc.
- You can try different pretty printing options and compare the results.

Demonstration: Creating a Petri Net DSL

- Sample:

```
petrinet {
  place P1;
  place P2;
  transition T1;
  arcPT P1 -> T1;
  arcTP T1 -> P2;
}
```

a) Textual Syntax



b) Visual Syntax

Common TCS Constructs - Outline

- Lexical Consideration
- Separators
- Conditionals
- Functions
- Expressions
  - Using grammars
  - Using KM3 and TCS
- Pretty Printing
  - Blocks
  - Special symbols spacing
- Common Patterns

Separating contained elements

- KM3 excerpt:

```
class A {
    reference bs[*] container : B;
    reference cs[*] container : C;
}
class B {}
class C {}
```

- TCS excerpt:

```
template A: "a" "{" bs cs "}";
template B: "b";
template C: "c";
```

➔ Pros: Getting contained *B*s or *C*s is as easy as navigating one reference.

➔ Cons: All *B*s must appear before all *C*s in the concrete syntax.

Separating contained elements: Example

- KM3 excerpt:

```
class PetriNet {
    reference places[*] container : Place;
    reference transitions[*] container : Transition;
}
class Place { attribute name : String; }
class Transition { attribute name : String; }
```

Mixing contained elements

- KM3 excerpt:

```
class A {
    reference borcs[*] container : BOrC;}
abstract class BorC {}
class B extends BorC {}
class C extends BorC {}
```

- TCS excerpt:

```
template A: "a" "{" borcs "}";
template BorC abstract;
template B: "b";
template C: "c";
```

➔ Pros: *B*s and *C*s can appear in any order.

➔ Cons:

- *BorC* may not capture any other common property of *B* and *C* except that they can syntactically appear intermixed in the concrete syntax.
- Navigation in models is more complex (need to check the type of elements in *borcs*).

Mixing contained elements: Example 1

- KM3 excerpt:

```
class PetriNet {
    reference nodes[*] container : Node;
}
abstract class Node { attribute name : String; }
class Place extends Node {}
class Transition extends Node {}
```

➔ Here, class *Node* is also used to define the *name* attribute.

Mixing contained elements: Example 2

- KM3 excerpt:

```
class PetriNet {
   reference elements[*] container : Element;
}
abstract class Element {}
class Place extends Element { attribute name : String; }
class Transition extends Element { attribute name : String; }
class Arc extends Element {}
```

➔ Because *Arcs* do not have names in our example, class *Element* cannot be used to define the *name* attribute. Its only purpose is to enable mixing *Places*, *Transitions*, and *Arcs* in the concrete syntax.

Representing different kinds of elements (1/3)

- KM3 excerpt:

```
class Arc {
    attribute isP2T : Boolean;
    reference place : Place;
    reference transition : Transition;
}
```

- TCS excerpt:

```
template Arc: (isP2T ?
        "arcPT" place{refersTo = name} "->" transition{refersTo = name} ";"
:
        "arcTP" transition{refersTo = name} "->" place{refersTo = name} ";"
)
;
```

Representing different kinds of elements (2/3)

- KM3 excerpt:

```
abstract class Arc {
    reference place : Place;
    reference transition : Transition;
}
class ArcPT extends Arc {}
class ArcTP extends Arc {}
```

- TCS excerpt:

```
template Arc abstract;
template ArcPT:
        "arcPT" place{refersTo = name} "->" transition{refersTo = name} ";";
template ArcTP:
        "arcTP" transition{refersTo = name} "->" place{refersTo = name} ";";
```

Representing different kinds of elements (3/3)

- KM3 excerpt:

```
abstract class Arc {}
class ArcPT extends Arc {
   reference source : Place; reference target : Transition;
}
class ArcTP extends Arc {
    reference source : Transition; reference target : Place;
}
```

- TCS excerpt:

```
template Arc abstract;
template ArcPT:
      "arcPT" source{refersTo = name} "->" target{refersTo = name} ";";
template ArcTP:
      "arcTP" source{refersTo = name} "->" target{refersTo = name} ";";
```

End of the presentation

- Thanks
  - Questions?
  - Comments?