Mind the Gap: Lessons Learned from Translating Grammars between MontiCore and Xtext

Manuela Dalibor
Software Engineering
RWTH Aachen

http://www.se-rwth.de/
Motivation

- Model-driven systems engineering relies on software languages that support different stakeholders
- Checking consistency, tracing, and change propagation of models developed by different stakeholders
- Integration of heterogeneous software languages
- Translation in an automated toolchain and present lessons learned along the way
- Reuse existing languages in different context and domains
Outline

1. Preliminaries
2. Evaluation criteria for translations
3. Cases that we identified while translating between MontiCore and Xtext
4. Results
Software Language Engineering

- Software Language Engineering (SLE) is the discipline to design useful software languages and their tool infrastructure in an efficient, systematic way.

- A language defines a set of sentences, models (the elements of the language)

- Formal definition should be flexible to allow adapting the language

Language consists of:

- Conrete Syntax: Representation of Models
- Abstract Syntax: Structure of a language
- Semantic Domain: Meaning
- Semantic Mapping: Connecting Language Elements and the semantic domain

```plaintext
grammar Automaton extends Literals, Expressions {
    Automaton = Name (State |Transition)*;
    symbol State = ("initial"|"final")* Name;
    Transition = from:Name@State input:Name to:Name@State;
}
```
A language workbench (LWB) is a development tool to define new software languages (DSLs) and provide assistance for their analysis, manipulation and transformation.

**Xtext**
- Facilitates the development of domain-specific languages
- Code generator in Xtend can be hooked in for any language
- Customizable IDE

**MontiCore**
- Modular definition of languages and language fragments
- Assistance for model composition and transformation
- Generation using FreeMarker templates

MontiCore (LWB) generates DSL Tool (generator) generates Product
Classifying Translations

**Language Equivalence**

- Two grammars are equivalent if they represent the same language
  - There exists a **bijective (one-to-one) function** which maps a set of structural descriptions of the first grammar to a set of structural descriptions to the second

- The problem of whether two context-free grammars represent the same language is undecidable

- When translating domain-specific languages, we also have to consider the **translation of well-formedness rules**

**Bijectivity**

- **Bidirectional** translation
  - From Xtext to MontiCore and vice versa

- Translating a language from MontiCore to Xtext and back yields the **initial language**

- For any grammar in the source technique and for any grammar in the target technique the translation is surjective and injective

- This requires that every grammar in the source technique is mapped to exactly one grammar in target technique and vice versa
Convergence

- Bijectivity is hard to achieve if
  - Translation between meta-languages requires transformations
  - Two concepts in the source grammar map to the same concept in the target grammar

- If we can find a maximal number of translations, we call the translation to be convergent.
  - Convergence after 0 steps gives us a bijective translation
  - Convergence after 1 step is a translation between languages that are not fully compatible
  - Convergence in more than 1 step should be further investigated

- A non-converging translation may be incorrect
  - Concepts are translated cyclically
  - Indicates that two equal concepts should be reduced to one
Translation Concept

- CoCos as helper

- Simplification Trafo
1. Base Rules

• Every metagrammar has **basic concepts for defining productions** and terminals:
  
  – The standard for this is the extended Backus–Naur form (EBNF)
  – EBNF is a metalanguage for context-free grammars
  – It is possible to reduce any context-free grammar to EBNF

• If possible, **preserve the original structure** of the language

• Translate base rules (according to EBN) directly

```
Automaton = (State | Transition)* ;
```

```
Automaton: (states+=State | transitions+=Transition)* ;
```
2. Simplification Rules: Interfaces

- One example of a simplification rule in MontiCore is the definition of interfaces

- If an interface is declared and used at different points in the grammar, at every point the interface is used, all implementing productions are valid options for the parser

- Xtext does not support interface productions
  - Transform grammars that contain interfaces before translating them to Xtext

```plaintext
01 StartRule      = InterfaceProd*; //implementing nonterminals must have a name
02 interface InterfaceProd    = Name;
03 FirstImpl implements InterfaceProd = "first" Name;
04 SecondImpl implements InterfaceProd = "second" Name;
```

```plaintext
01 StartRule : interfaceProds+=InterfaceProd*;
02 FirstImpl : "first" name=Name;
03 SecondImpl : "second" name=Name;
04 InterfaceProd : firstImpl=FirstImpl|secondImpl=SecondImpl;
```
2. Simplification Rules: Unordered Groups

- All elements of an unordered group need to appear exactly once but in **arbitrary order**
  - For an unordered group of size n, we need n! many alternatives in EBNF

- MontiCore does not provide an equivalent language concept

- Translator creates a list in MontiCore to enable the occurrence in arbitrary order
  - adds an **AST rule** that ensures that each element of the list appears exactly once

```plaintext
enum Visibility  = public | private | protected;

Modifier: static?='static'? & final?='final'? & visibility=Visibility;

Modifier = (a:ModifierA|b:ModifierB|c:ModifierC)+;
astrule Modifier = as:ModifierA min=0 max=1
  bs:ModifierB min=0 max=1
  cs:ModifierC min=1 max=1;

ModifierA = "static";
ModifierB = "final";
ModifierC = Visibility;
enum Visibility = public | private | protected;
```
3. Recursion: Expressions

- Expressions always bring two problems:
  1. Concerning parsing, differentiate left (or right) recursion
  2. Xtext bases on ANTLR3, and hence, does not support left recursion
  3. MontiCore, on the other hand, uses ANTLR4 which already supports left recursion
- Detect left recursion and apply left factoring before translation
- If a construct recurses on the left hand side, put it into a delegation chain according to the operator precedence.
- The non-terminal that recurses delegates to the rule with the next higher precedence
4. Keyword Escaping

- MontiCore supports adding an ampersand (&) to the Name nonterminal to support keywords as names.

- Xtext supports prefixing a name with a caret (^) that is removed during parsing to escape keywords:
  - This concept is not translatable into MontiCore.
  - Models are still parsable, but the escape character will be part of the name.

- Ampersand must be handled to ensure parsable models:
  - Production `NameWithKeywords` that refers either to a Name or to all possible keywords.

- When we retranslate a grammar from Xtext back to MontiCore, we production called NameWithKeywords to change it back to Name&.
5. Inheritance

- **Grammar Inheritance:**
  - Multi,
  - Single,
  - No inheritance

- **Transformation required if the target technology is stricter**
  - Reduce the inheritance, e.g., by merging all super grammars

- **Maintain the inheritance structure wherever possible**
  - Subgrammars may redefine or override productions
  - Merge super grammar stepwise

- **No inheritance:** Insert all rules of the super grammar into the translated grammar to keep expressiveness
6. AST Transformations

- Rewrite rules directly change the created AST or the classes of which the AST consists.
  - In Xtext language engineers can change the AST node that is produced by a production.

  **01** Addition returns Expression:
  **02** Multiplication ('+' Multiplication)*;

- These rules are workbench-specific → not possible to provide a general concept for their translation.

- Rules that support adding arbitrary attributes or methods to an AS class cannot be translated in general.
  - Cannot guarantee that the names and types are present in the result.
  - Adding of an attribute may incorrectly override an existing attribute of the target, or may incorrectly not override an attribute that is not existing in the target grammar.

- Result in a semantically non-equivalent translation, and should be forbidden to ensure the stability of the translation.
7. Symbols and Scopes

- Symbols, symbol tables, and scopes are an essential factor in the structuring of languages:
  - Referencing of model elements at a different point in the model

- MontiCore supports references to symbols that have names that are of type Name

- Xtext supports references to nonterminals with an arbitrary identifier
  - Rename the ID production and all its occurrences to Name
  - Reduce the second reference to an element of type ValidID

```plaintext
01 State: "state" name=ID ";" ;
02 Transition: from=[State] "->" to=[State|ValidID] ";" ;
03 ValidID: ID ("." ID)* ;
```

**reference to a state via full qualified name**

```plaintext
01 symbol State = "state" Name";" ;
02 Transition = from:Name@State "->" to:ValidID ";" ;
03 ValidID = Name ("." Name)* ;
```

**reference to a state via its Name**
Conclusion

Language equivalence cannot be achieved with grammar translations only

AS-conservatism is not achieved as Xtext and MontiCore produce different AS

CS-conservatism is achieved, so the same model can be parsed

The translation between MontiCore and Xtext is not bijective

The sequential translation from Xtext to MontiCore converges after at most two steps

<table>
<thead>
<tr>
<th>Element</th>
<th>MontiCore</th>
<th>Xtext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scopes</td>
<td>Grammar</td>
<td>Xtend</td>
</tr>
<tr>
<td>IDE</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Grammar Inheritance</td>
<td>Multiple</td>
<td>Single</td>
</tr>
<tr>
<td>Production Inheritance</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Change of return Type</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Code Actions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Tree Rewriting</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ASTRule</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explicit Start Rule</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Unordered Group</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Left Recursion</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Interface/ Abstract NTs</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Names with Keywords</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fragment Rules</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Thank You!