[DJK+19] M. Dalibor, N. Jansen, J. Kästle, B. Rumpe, D. Schmalzing, L. Wachtmeister, A. Wortmann:
Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext.
In: International Workshop on Domain-Specific Modeling (DSM'19), pp. 40-49, ACM, Athens, Oct. 2019.
www.se-rwth.de/publications/

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

Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
www.se-rwth.de

Abstract

Model-driven systems engineering relies on software languages that support different stakeholders. These languages often operate in different technological spaces. Checking consistency, tracing, and change propagation of models developed by different stakeholders, thus demands methods to bridge the gaps between these spaces. Research on the integration of heterogeneous software languages often considers heterogeneity within specific technological spaces only. We outline a systematic method to translate grammars between the technological spaces of the MontiCore and Xtext language workbench (LWB) and report observations on general grammar translation challenges. We have realized this translation in an automated toolchain and present lessons learned along the way. This can significantly facilitate bridging different technological spaces and, thus, improve model-driven systems engineering.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Syntax*.

Keywords Domain-specific languages, Grammarware, Grammar Translation, Xtext, MontiCore

ACM Reference Format:
Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, Andreas Wortmann. 2019. Mind the Gap: Lessons Learned from Translating Grammars between MontiCore and Xtext. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358501.3361236

This research has partly received funding from the German Research Foundation (DFG) under grant no. EXC 2023 / 390621612. The responsibility for the content of this publication is with the authors.

DSM '19, October 20, 2019, Athens, Greece

ACM ISBN 978-1-4503-6984-8/19/10…$15.00
https://doi.org/10.1145/3358501.3361236

1 Introduction

Large-scale model-driven systems engineering (MDSE) [18] often involves the collaboration of experts from various domains across space and time. These experts use various modeling paradigms, software languages, and tools that operate in different technological spaces. Translating models from one technological space into another is crucial for the seamless and automated model processing required to make MDSE projects successful.

Software language engineering (SLE) is the discipline of conceiving, engineering, maintaining, and evolving software languages [9]. Its subjects are software languages in different shapes, including graphical, textual, and projectional languages. Textual languages, *i.e.,* languages whose models are represented textually, have been successful in a variety of domains, including automotive [5], cloud systems [26], robotics [19], and software engineering [21]. Such languages often are defined in terms of (context-free) grammars [8, 25, 28] that define their structure (abstract syntax) and presentation (concrete syntax) in an integrated fashion.

Automating translation of textual models between different technological spaces leveraging SLE techniques can be a greatly facilitate automating model analyses and syntheses in MBSE projects. We, therefore, investigate the challenges of translating the concrete syntax and the abstract syntax of grammar-based languages between different technological spaces. To this end, we discuss challenges in their translation, achievable language equivalencies, and discuss the translation of various grammar constituents between the technological spaces of MontiCore and Xtext. The contributions of this paper, hence, are:

- A discussion of grammar translation challenges.
- A systematic investigation of grammar translations based on the technological spaces of MontiCore [8] and Xtext [28].
- Lessons learned about such transformations

The insights reported in this paper can support language engineers in systematically developing translations between other technological spaces by shedding light on typical challenges for these.

In the remainder, Section 2 illustrates the challenges by example. Section 3 introduces preliminaries before Section 4 discusses forms of model conservativity that imply different challenges for grammar translation. Afterward, Section 5

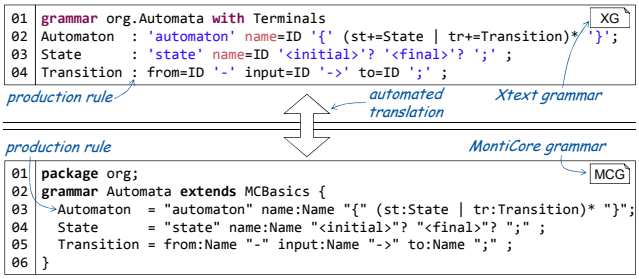

Figure 1. Xtext grammar (top) for a simple automata DSL with states and transitions. Automatic translation into a corresponding MontiCore grammar (bottom).

discusses various cases of grammar element translation and outlines a systematic methodology for their translation. Section 6 illustrates its application via a case study and Section 7 presents lessons learned. Section 8 discusses observations, Section 9 highlights related work, and Section 10 concludes.

2 Example

Consider a DSL for describing state-based behavior automata in the development of cyber-physical production systems (CPPS) as identified in [31]. There are various LWBs (*e.g.,* MontiCore [8], Neverlang [25], Spoofax [15], or Xtext [28]) that support engineering such DSLs. Figure 1 (top) depicts a simplified grammar for automata that was developed using the Xtext LWB as Xtext automatically generates editors for models of the DSL. Furthermore, Xtext operates in the technological space of Ecore [22], therefore model comparison tools [24], execution engines [2], and much more are available. In this DSL, an automaton has a name and consists of multiple states and transitions (l. 2). States again have a name and can be marked as initial or final (l. 3). Finally, transitions go from one state to another (identified via the state's name) and have an input that defines on which pattern the transition is triggered (l. 4). But if the CPPS should change to support real-time operations, the DSL needs to change as well to support timing constraints. To systematically migrate existing models of the DSL, these need to be augmented with timing information carefully. To prevent doing this manually for hundreds of models, the CPPS expert modelers should provide suitable model-to-model (M2M) transformations. Instead of forcing these to learn ATL [14] or a similar generic model transformation language, they should be enabled to use the syntax of the automata DSL within the transformations. Domain-specific transformation languages (DSTL) enable this by using their base DSL's syntax in the pattern and replacement parts of transformations [6]. Within the technological space of Xtext, DSTLs are not available, whereas in the technological space of MontiCore, they are. To transform these models using a DSTL, the DSTL first needs to be derived from the base DSL, for which its representation as a MontiCore grammar is necessary. However, the manual transformation between both LWBs is usually time-consuming and error-prone. DSL developers would have to ensure soundness and completeness of the translation, *i.e.,* that the MontiCore language depicts exactly the set of models that the Xtext language recognizes. Hence, they would have to be experts in both technological spaces. To mitigate this, we investigate on bridging technological spaces for textual DSLs while considering the intricacies of the corresponding language workbenches. For instance, Figure 1 (bottom) shows a MontiCore grammar that recognizes models that are syntactically identical to the original Xtext grammar. Obtaining this representation systematically and automated enables leveraging the combined advantages of both technological spaces, *i.e.,* benefiting from the editors generated for the Xtext version of this DSL and from the DSTLs at the same time. This increases efficiency and productivity and thus, fosters software language engineering [10].

3 Preliminaries

MontiCore [8] is a workbench for engineering and composing textual DSLs and facilitates language development by employing context-free grammars (CFGs) for the integrated definition of abstract and concrete syntax. From a CFG, MontiCore generates model processing infrastructure, comprising corresponding abstract syntax classes and a parser, as well as symbol tables, visitors, model checking and code generation infrastructure. Models that adhere to a grammar definition are parsed into corresponding abstract syntax trees. Afterward, their symbol tables are created, which store essential information of model elements as symbols for easy access. Their well-formedness is checked via handcrafted context conditions using MontiCore's model-checking infrastructure. To facilitate modular DSL development, MontiCore supports language composition of language constituents via language extension, embedding, and aggregation [8].

Xtext [28] also is a workbench for engineering textual software languages that is part of the Eclipse Modeling Framework (EMF) [23]. In Xtext, a language is defined by a grammar that simultaneously defines abstract and concrete syntax. From such a grammar, Xtext infers a metamodel and generates a parser translating models into abstract syntax trees. Furthermore, Xtext provides additional language infrastructure, *e.g.,* hooks for well-formedness rules, and editors including syntax highlighting, auto-completion, and refactoring. As abstract syntax trees in Xtext are instances of Ecore models, Xtext facilitates integrating of EMF-based tooling, such as Sirius [27] for graphical modeling or EMF Compare [3].

4 Classifying Language Translations

A grammar translation is a function from one grammar metalanguage to another. The translation takes a grammar of one metalanguage (*e.g.,* MontiCore) and translates it to another.

Usually, this entails a translation between different technological spaces.

Equivalence Two grammars are equivalent if they represent the same language [11]. The problem of whether two context-free grammars represent the same language is undecidable. Furthermore, when translating domain-specific languages, we also have to consider the translation of well-formedness rules and generators to ensure that semantics are maintained. As equivalence between MontiCore and Xtext languages can generally not be achieved if our translation only works for grammars, we focus on conservatism a less strong characteristic of a translation.

Conservative Translation We derive the term conservative translation from the established term of conservative extension. A conservative extension [8] is an extension of a grammar that introduces new model instances into the extended grammar and enables reusing of all original models within the extended grammar. Similarly, we define a conservative translation. An abstract syntax (AS) -conservative translation of a grammar preserves all nonterminals, cardinalities, as well as semantically relevant terminals. It, however, is allowed that additional semantically relevant entities extend the AS of the translated grammar. For example, a translation could introduce new nonterminals. The goal that is achieved by AS-conservation is that all functionalities for the original AS still exist in the translated AS. A conservative translation of the concrete syntax preserves and only extends the concrete syntax of a language. A concrete syntax (CS) -conservative translation of a grammar results in a grammar that represents all models that the original grammar can represents but might also represent more models. Thus all models that conform to the original grammar can be reused. This entails that for any class of the input grammar's AS, there is a class in the output grammar's AS such that the output grammar's AS contains at least the same information of the input grammar's AS. If concrete and abstract syntax are preserved during a translation, but the AST representation of the same model differs in the original and the translated language, we cannot reuse tooling that was developed for the original language. CS-AST-compliance enforces that the same model results in the same abstract syntax tree.

Bijectivity Our translation works in both directions, that means we can translate from Xtext to MontiCore and vice versa. When translating a grammar from MontiCore to Xtext and back to MontiCore we want to obtain exactly the grammar that we started with. Therefore, we define bijectivity. The translation is bijective if and only if for any grammar in the source technique and for any grammar in the target technique the translation is surjective and injective [30]. This requires that every grammar in the source technique is mapped to exactly one grammar in target technique and vice versa. Hence, neither in the source nor in the target technique, there exist grammars that are not mapped to a grammar in the other technological space.

Convergence Bijectivity is hard to achieve as soon as the translation between meta-languages requires transformations since then there exist two concepts in the source grammar that map to the same concept in the target grammar. A transformation is, for example, the reduction of a language concept that the target language does not support into an equivalent statement that is translatable. If it is not possible to achieve bijectivity, there is a gap between the source and the resulting grammar. To measure the quality of the translation, it is relevant to investigate whether this gap is increasing when applying the translation multiple times. When concatenating translations from one technological space to another we analyze whether the translation achieves bijectivity at some point. 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 because it means that concepts are translated ambiguously. A non-converging translation is either an incorrect translation or concepts are translated cyclically. While this cyclic translation is not problematic by itself, it shows that there are two equal concepts that could be reduced to one.

5 Translating Grammars Across Technological Spaces

We developed a tool to realize a bidirectional translation between MontiCore and Xtext grammars. During developing the grammar translation, we identified different concepts that are part of grammar definitions and must be considered when translating grammars between different LWBs in general. In the following, we present these cases and discuss how we solved them in our translation. We distinguish eight different cases and the translation results based on whether we have to add transformations before the translation or if we generate additional well-formedness rules.

1. Base Rules Every metagrammar has basic concepts for defining productions and terminals. The standard for this is the extended Backus–Naur form (EBNF) [12]. EBNF is a metalanguage for context-free grammars. Since EBNF supports the definition of arbitrary CFGs, it is possible to reduce any context-free grammar to EBNF. If possible, we try to maintain as much of the original grammar as possible when translating between Xtext and MontiCore and thus preserve the original structure of the language. Base rules, according to EBNF, can be translated directly as depicted in Figure 2. Furthermore, MontiCore and Xtext both support the definition of lexer rules. Lexer rules are rules that a lexer breaks up an input stream of characters into vocabulary symbols

```
01 Automaton = (State | Transition)* ;                                  MCG
   alternatives in MontiCore and Xtext        iterations
01 Automaton: (states+=State | transitions+=Transition)* ;              XG
```

Figure 2. Translating EBNF-conform productions between MontiCore and Xtext.

for the parser. In MontiCore they are called `tokens` and in Xtext they are called `terminals`. Thus, we translate lexer rules also directly as Figure 3 shows.

```
01 grammar UntilGrammar {                                              MCG
02   StartElement = (element:UNTIL_ELEMENT);
03   token UNTIL_ELEMENT: 'x' .*? 'y';
04   token WS = (' '|'\t'|'\r'|'\n')+ : -> skip;
05 }

01 grammar de.se_rwth.babel.UntilGrammar hidden(WS)                    XG
02   StartElement: (element=UNTIL_ELEMENT);
03   terminal UNTIL_ELEMENT: 'x' -> 'y';
04   terminal WS: (' '|'\t'|'\r'|'\n')+;
```

Figure 3. Bijective translation of tokens in MontiCore and Xtext terminals.

2. Simplification Rules Because a grammar that consists only of EBNF rules can get complicated and unreadable, additional concepts are introduced in the respective language workbenches that increase the structure and readability of the grammar. One example of a simplification rule in MontiCore is the definition of interfaces. If an interface is declared

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

01 StartRule     : interfaceProds+=InterfaceProd*;                     XG
02 FirstImpl     : "first" name=Name;
03 SecondImpl    : "second" name=Name;
04 InterfaceProd : firstImpl=FirstImpl|secondImpl=SecondImpl;
```

Figure 4. Translating interfaces from MontiCore to Xtext, by introducing an alternative with all productions that implement the interface in MontiCore.

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. Therefore, our translator transforms grammars that contain interfaces before translating them to Xtext. An example is depicted in Figure 4. The grammar fragment consists of an interface `InterfaceProd` that is implemented by the two productions `FirstImpl` and `SecondImpl`. The `Name` on the right hand side of interface `InterfaceProd` ensures that all implementing nonterminals contain a name. After applying the upon stated transformation, the interface is converted into a regular production with two alternatives. A simplification in Xtext grammars are 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. Hence, we need a transformation. When translating a grammar containing unordered groups from Xtext to MontiCore, our translator creates a list in MontiCore to enable the occurrence in arbitrary order and adds an AST rule that ensures that each element of the list appears exactly once Figure 5.

```
01 Modifier:                                                           XG
02 static?='static'? & final?='final'? & visibility=Visibility;
03 enum Visibility: public | private | protected;

01 Modifier         = (a:ModifierA|b:ModifierB|c:ModifierC)+;          MCG
02 astrule Modifier = as:ModifierA min=0 max=1
03                    bs:ModifierB min=0 max=1
04                    cs:ModifierC min=1 max=1;
05 ModifierA        = "static";
06 ModifierB        = "final";
07 ModifierC        = Visibility;
08 enum Visibility  = public | private | protected;
   astrule ensures that each element occurs at most once in the model
```

Figure 5. Translating unordered groups from Xtext to MontiCore, by introducing a list and an ast-rule that ensure that each element of the list occurs exactly once.

3. Expressions A standard task for every LWB is the definition of expressions. An expression is a type of rule or rule set that calls itself recursively. The most standard case are algebraic expressions. Expressions always bring two problems to the language engineer. Concerning parsing, she has to consider left (or right) recursion and left or right associativity. Xtext bases on ANTLR3, and hence, does not support left recursion. MontiCore, on the other hand, uses ANTLR4 which already supports left recursion. Therefore, when our translation encounters left recursion in a MontiCore grammar, it first transforms the left recursive rules. Figure 6 shows how we transform a left recursive grammar in MontiCore before we can translate it into Xtext according to [11]. First we identify all nonterminals that are unambiguous, for example because they have a terminal prefix that identifies them. Our translator groups these nonterminals into a new nonterminal called `UnambiguousExpr`. Next, the rule with the lowest priority is moved to the top. In the example, the `AddExpr` is moved to the top and its righthand side is expressed as a sum of `MultExpr`. The right-hand side of `MultExpr` is also changed to `UnambiguousExpr` which is the expression with the highest priority.

4. Decision Rules for the Parser It is sometimes impossible to write unambiguous grammars. For example, a production with two alternatives that are both applicable at a point in the parsing process is ambiguous because the parser can take both parsing paths. Therefore, it is necessary to specify decision rules, usually in the form of predicates. Translating predicates is very hard because the underlying

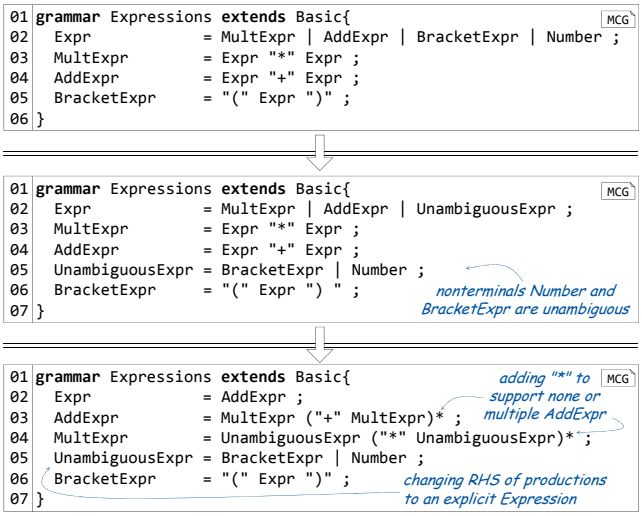

Figure 6. Removing left recursion in a MontiCore grammar before translating to Xtext.

parser technology of LWBs defines the structure for predicates, and these structures can vary between metalanguages. Some parsers can even resolve ambiguities directly, so the language engineer is not aware of developing an ambiguous language, which impedes the translation even more.

5. Keywords We also have to consider the handling of keyword escaping. While lexer and grammar-based languages forbid keywords as names or arbitrary tokens based on the context, some concepts enable the usage of keywords as names. MontiCore supports adding an ampersand (&) to the `Name` nonterminal to support keywords to as names. Xtext has no such feature. Instead, it supports prefixing a name with a caret that is removed during parsing to escape keywords. Because the caret is handled on the model level and depends on Xtext as source platform, this concept is not translatable into MontiCore. The models are still parsable, but the escape character will be part of the name. The loss of the ampersand from MontiCore to Xtext is unacceptable because it makes models unparsable. The suggested way of the Xtext developers for using keywords as names is the creation of a production `NameWithKeywords` that refers either to a Name or to all possible keywords as depicted in Figure 7. The `NameWithKeywords` nonterminal in the Xtext grammar can either be a `Name` or the keyword `state`. When we retranslate a grammar from Xtext back to MontiCore, we try to find a production that is called `NameWithKeywords` to change it back to `Name&`.

6. Grammar Inheritance When dealing with grammar inheritance, we consider three possible cases: multi, single, and no inheritance. If the target metalanguage permits a more general kind of inheritance, the translation is simple. If it is stricter, we reduce the inheritance, *e.g.,* by merging all super grammars into a single grammar to transform multi

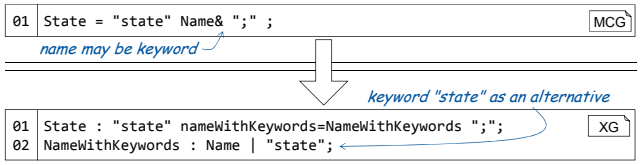

Figure 7. Resolving MontiCore's ampersand in Xtext to enable keywords as names.

to single inheritance. This process is recursive because we prefer maintaining the inheritance structure wherever possible. During the merging process, we also have to take into account that subgrammars may redefine or override productions of their super grammars. Therefore, we merge super grammar stepwise and remove all productions that are overridden. If no inheritance is permitted, we apply the same approach, but insert all rules of the super grammar into the translated grammar to keep the same expressiveness. Of course, when translating back into MontiCore the translator cannot restore the original super grammar as the translator cannot identify which productions originate from which grammar.

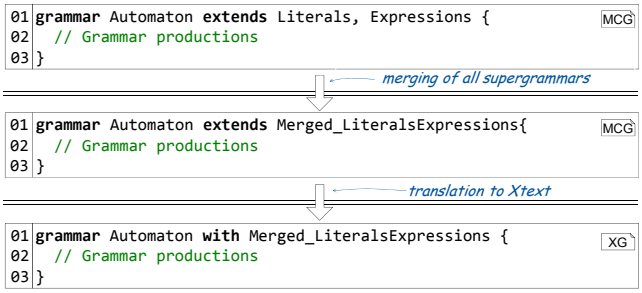

Figure 8. Transforming multi-inheritance in MontiCore to single inheritance in Xtext, by merging supergrammars.

7. Rewrite Rules Rewrite rules directly change the created AST or the classes of which the AST consists. For example, in Xtext language engineers can change the AST node that is produced by a production as depicted in Figure 9. Many of these rules are workbench-specific. Therefore, it is not possible to provide a general concept for their translation. Some concepts are not translatable, for example, if the source grammar changes the return type of a production, and the target metalanguage does not support this type of change. In consequence, this grammar cannot be translated AST-equivalently.

Rules that support adding arbitrary attributes or methods to an AS class cannot be translated in general. Although it is possible to copy these rules into a handwritten extension using, for example, the generation gap or the TOP pattern, we cannot guarantee that the names and types are present in the result, as we do not know the generated structure yet. Similarly, the adding of an attribute may incorrectly override an

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

Figure 9. Specifying the return type of a production in Xtext.

existing (renamed) attribute of the target, or may incorrectly not override an attribute that is not existing in the target grammar. In summary, this behavior would result in a semantically non-equivalent translation, and should, therefore, be forbidden to ensure the stability of the translation.

8. Symbols and Scopes Symbols, symbol tables, and scopes are an essential factor in the structuring of languages and are often specified in grammars. The most important use case for symbols is the referencing of model elements at a different point in the model. This process, called cross-referencing, is idiomatic to most grammar-based languages. MontiCore supports references to symbols that have names that are of type `Name` whereas Xtext supports references to nonterminals with an arbitrary identifier. The Xtext grammar in Figure 10 specifies transitions that reference two states. The `from` state is referenced via the ID specified in the nonterminal `State`. The target state is specified via the nonterminal `ValidID` that specifies a full-qualified name. While translating `Transition` to MontiCore, we rename the ID production and all its occurrences to `Name`. Next, we reduce the second reference to an element of type `ValidID`. This preserves model-equivalence if context conditions are generated that check for the existence of the referenced `State`.

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

reference to a state via its Name — *reference to a state via full qualified name*

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

Figure 10. Translating non-terminal references between MontiCore and Xtext.

6 Case Study

Suppose a software engineering team is developing a modeling language for the textual description of software architectures using MontiCore's language composition mechanism to reuse existing language components. When employed by software architects, these wish for editor support to craft architecture descriptions effectively. Translating a grammar from MontiCore to Xtext enables to reuse Xtext's rich editor features. Another reason why a software team might want to change the technological space is that they want to integrate languages that are developed in MontiCore with languages that are developed in Xtext. We illustrate the translation of a grammar using the example of MontiArc, an architecture description language (ADL) for modeling component and connector architectures.

```
01 grammar MontiArc extends MCBasics {                             MCG
02   MACompilationUnit = Package ImportStatements* Component;
03   // Component Head
04   symbol Component implements ArcElement
05             = "component" Name Signature "{" ArcElement* "}";
06   Signature  = Parameters? ("extends" Type)?;
07   Parameters = Parameter ("," Parameter)*;
08   Parameter  = Type Name ("=" Expression)?;
09   interface ArcElement;
10
11   // Component Body Elements
12   SubComponent implements ArcElement
13        = "component" Type Arguments? instances:Names ";";
14   Connector implements ArcElement
15        = "connect" source:Name "->" targets:Names ";";
16   Ports implements ArcElement
17        = "port" Port ("," Port)+ ";";
18   Port = (["in"] | ["out"]) Type Names?;
19   Names = Name ("," Name)*;
20
21   // Embedded Behavior Elements
22   Variable implements ArcElement = Type Names? ";" ;
23   Automaton implements ArcElement = "automaton" Name? "{"
24                    (States | InitialState | Transition )* "}";
25   symbol State = "state" Name ";" ;
26   InitialState = "initial" Name ("/" Block)? ";";
27   Transition   = source:Name@State ("->" target:Name@State)?
28                  ("[" Expression "]")? ("/" reaction:Block)? ";";
29   Block        = "{" (Name "=")? Expression ("," Expression)* "}";
30 }
```

Figure 11. Simplified excerpt of the MontiArc grammar.

Architectures in MontiArc consist of hierarchically composed components that communicate over typed, directed ports. Components are either composed or atomic. A composed component's behavior is induced through the behavior of its subcomponents, whereas atomic components yield behavior descriptions, such as automata, to define their input and output behavior. The MontiCore grammar of the MontiArc ADL defines the abstract and concrete syntax of all MontiArc architectural models. A simplified excerpt of this grammar is shown in Figure 11. First, the MontiArc grammar extends `MCBasics` (l. 1) to inherit and avoid re-engineering of commonly used productions, such as names, types, expressions, packages, and import statements. Next, this grammar defines MontiArc models files such that they have a package declaration, some import statements, and a component definition (l. 2). Components have a name, a signature, and a component's body consists of instances of `ArcElements`, which are structural component elements (ll. 11-19), such as ports and connectors, and behavioral elements (ll. 21-29), *e.g.,* behavior automata.

From the MontiCore grammar of the MontiArc ADL, we automatically derive the Xtext grammar shown in Figure 12. To this end, the translator automatically applies the transformation rules presented in Section 5. First, the input grammar is transformed internally to resolve MontiCore specifics before individual productions are translated into their respective counterparts in Xtext. Productions without language workbench specifics, such as `MACompilation`, are directly translated into the respective counterpart in Xtext using transformation rule 1. To resolve interfaces, such as

```
01 grammar MontiArc with MCBasics                                    XG
02   MACompilationUnit : package=Package importStatements+=ImportStatements*
03                       component=Component;
04   // Component Head
05   Component  : "component" name=ID signature=Signature
06                "{"arcElements+=ArcElement* "}";
07   Signature  : parameters?=Parameters? ("extends" type?=Type)?;
08   Parameters : parameters+=Parameter ("," parameters+=Parameter)*;
09   Parameter  : type=Type name=ID ( "=" expression?=Expression )?;
10   ArcElement : Component | SubComponent | Connector |
11                Ports | Variable | Automaton ;
12
13   // Component Body Elements
14   SubComponent : "component" type=Type arguments?=Arguments?
15                  instances=Names ";";
16   Connector : "connect" source=ID "->" targets=Names ";";
17   Ports     : "port" ports+=Port ( "," ports+=Port)+ ";";
18   Port      : ( in ?="in" | out ?="out") type=Type names?=Names?;
19   Names     : names+=ID ("," names+=ID)*;
20
21   // Embedded Behavior Elements
22   Variable  : type=Type names?=Names? ";";
23   Automaton : "automaton" name?=ID? "{" (states+=State |
24               initialStates+=InitialState | transitions+=Transition)*"}";
25   State        : "state" name=ID ";";
26   InitialState : "initial" name=ID ("/" block?=Block)? ";";
27   Transition   : source=[State] ("->" target?=[State])?
28                  ("["expression?=Expression"]")?("/"reaction?=Block)?";";
29   Block        : "{" ( name?=ID "=")? expressions+=Expression
30                  ("," expressions+=Expression )* "}";
```

Figure 12. Xtext grammar of the MontiArc ADL.

`ArcElement`, which are specific to MontiCore, transformation rule 2 is applied, resulting in an alternative off all its implementing nonterminals (l. 10). Furthermore, using transformation rule 8, the symbols `Component` (l. 4) and `State` (l. 25) of Figure 11 are transformed into simple productions in Xtext, as Xtext does not specify symbols explicitly. The state references in the `Transition` are represented as equivalent references in Xtext (l. 27). The other specified transformations rules are not needed for this example but some are used in the transformation of its super grammars.

By this, the translation enables importing MontiArc models into Xtext and reuse the tooling landscape there for further development. Thus, a software architect who uses our approach is enabled to use language features of MontiCore such as the definition of the interface nonterminal `ArcElement` as well as the tool support for Xtext by automatically deriving the Xtext grammar.

7 Lessons Learned

In the previous sections, we implemented a bidirectional translation between MontiCore and Xtext. This translation consists of four steps: Parsing, checking, transforming and translating. First, the translator parses a MontiCore or Xtext grammar. Next, the translator ensures that the grammar complies to the specified well-formedness rules. Validation checks not only the metalanguage's context conditions but also some translation-dependent restrictions. For example, an Xtext grammar must have a package; this is optional for MontiCore. Therefore, we cannot translate a MontiCore grammar to Xtext if it does not have a package. If the grammar is well-formed and also translatable, the translator checks whether the grammar contains concepts that cannot be expressed in the target meta-language and thus requires simplification before translation. During the simplification, we transform concepts that do not have an equivalent in the target language into concepts that the target language supports, *e.g.,* explicit start rules.

The simplification consists of chained transformations that are preserving equivalence and can be executed in arbitrary order. After the necessary simplifications have been performed, the tool translates the grammar. In the generation step, the AST is given to the generation engine, which produces the target grammar with the help of templates and the helper structures. The translation is performed rule-by-rule. That means that the generator iterates through all rules one by one and translates them individually. Only if transformations affect multiple rules, our translator also transforms multiple rules at once. We consider all grammar concepts of both metalanguages. The translator translates all concepts that are generally translatable and gives warnings or errors for all other concepts.

Table 1 lists the language constructs where Xtext and MontiCore differ. MontiCore supports the definition of symbols and scopes at the grammar level. Furthermore, MontiCore provides more functionalities regarding inheritance between productions and grammars. Xtext provides an IDE for DSLs which is beneficial for developers and prevents errors. Also unordered groups are useful when the order in which elements occur in a model is not relevant. Although we found a workaround for this in MontiCore this impedes readability. Both language workbenches support handwritten extensions but the constructs provided in the grammar differ. For example, in Xtext grammar language engineers can explicitly change the return type of a production while MontiCore supports code injection in grammars.

In Section 4, we distinguished between equivalence, conservatism, bijectivity, and convergence. To each of which we identified a lesson based on our experience

Lesson 1: Equivalence cannot be achieved with grammar translations only. *Equivalence* means that two languages contain exactly the same models in terms of abstract and concrete syntax, well-formedness and semantics. Since our translation operates on grammar only and neglects well-formedness rules and semantics, we were unable to achieve equivalence. Furthermore, due to transformations that our translation performs before translating grammars from one technological space to another the AS of languages changes. Hence, equivalence is not achieved. For that reason, future approaches, which aim to achieve language equivalence must address well-formedness rules and semantics in addition to grammar translations.

Lesson 2: EOFs are problematic as grammars with EOFs are not AS-conservative. In Section 4, we distinguished different kinds of conservatism. *AS-conservative* translation

Table 1. Differences of language constructs and infrastructure in Xtext and MontiCore

Element	MontiCore	Xtext
Scopes	Grammar	Xtend
IDE	no	yes
Grammar Inheritance	Multiple	Single
Production Inheritance	yes	no
Change of Return Types	no	yes
Code Actions	yes	no
Tree Rewriting	no	yes
ASTRule Additions	yes	no
Until	no	yes
Explicit StartRule	yes	no
Unordered Groups	no	yes
Left Recursion	yes	no
Constant Groups	yes	no
Interfaces / Abstract NT	yes	no
Names with Keywords	yes	no
Fragment Rules	no	yes

of a grammar preserves all nonterminals, cardinalities, as well as semantically relevant terminals. MontiCore creates for every nonterminal an object in the AST and Xtext does the same. We accept the renaming of nodes and enforce only an equal structure of the tree for an AS-conservative translation from Xtext to MontiCore. The EOF token is not translatable as it is handled differently in both LWBs. When the translation encounters EOF tokens it warns the language engineer that it cannot translate this concept. Therefore, our translation is AS-conservative for grammars that do not contain EOF tokens. From this, we can learn that there are implicit tokens and token definitions such as the EOF token, which cannot be translated between grammars such as MontiCore and Xtext.

Lesson 3: Semantic Predicates and EOF are problematic as grammars with semantic predicates or EOF rules are not CS-conservative. A translation is *CS-conservative* if the resulting parser of the grammar can parse at least all models of the input grammar. If we forbid the usage of semantic predicates and the EOF rule in the input MontiCore grammar, the translation is CS-conservative. Every model that is parsable using the original grammar stays parsable in the generated grammar. For equivalence between the original and the translated grammar, we need to check if every unparsable text stays unparsable for the translated grammar. This property does not hold. For example, in MontiCore `astrules` enable cardinality restrictions. These are not translated to Xtext as Xtext does not support a similar concept. Thus, the translated Xtext grammar produces models, that the original grammar developed in MontiCore did not produce.

Lesson 4: Resolving inheritance in MontiCore prevents CS-AST compliance. *CS-AST-compliance* demands that the same model results in the same AST structure. Our current translation cannot guarantee CS-AST compliance, as we have to resolve inheritance in MontiCore because Xtext does not provide this concept as it, for example, does not support interface productions. This transformation of grammars ultimately affects the structure of the abstract syntax of the grammar and thus the AST of models. Thus, the translation between a grammar that supports language inheritance and interface productions and a second grammar which has none of these concepts is not CS-AST compliant.

Lesson 5: The translation between MontiCore and Xtext is not bijective. *Bijectivity* is achieved if each grammar in the source technique is mapped to exactly one grammar in the target technological space and vice versa. MontiCore and Xtext both have concepts that are not expressible in the other workbench. Contradicting bijectivity, two MontiCore grammars may be translated into the same Xtext grammar, as the translation requires transformations. For example, interface productions are removed during translation. The reverse translation cannot distinguish whether there was production inheritance. The same holds for parameters in an Xtext grammar. Therefore, we cannot achieve bijectivity. To overcome this problem, it could be possible to identify best practices for the creation of grammars which could allow an improved bidirectional (but still not bijective) translation. We will discuss our ideas to this respect again in Section 8.

Lesson 6: The sequential translation from Xtext to MontiCore converges after at least two steps. *Convergence* requires that sequential translations from Xtext to MontiCore and back do not change the input grammar. Our tool chain translates LWB specific concepts into a common concept, so it achieves convergence after one step. The only exceptions are translations that require additional context conditions or AST rules, as these concepts are not translated yet and therefore lost after the second translation. Grammars that include these concepts converge after two steps. Hence, the translation of LWB specific concepts into a common concept ensures that the sequential translation does not change the input grammar and thus converges after two steps.

8 Discussion

Even though the approach presented in this paper implements a bidirectional translation, it is not bijective but converges after at most two steps. If the source language has a concept that is simplified before the translation (*cf.* Section 5), we have two source grammars (the non-simplified and the simplified are valid grammars of the source metalanguage) that are mapped to the same target grammar. Thus, we can, for example, not decide whether a production inheritance was used after we refactored it. For that reason, the benefits of language features such as MontiCore's language composition are lost. Moreover, if the target metalanguage supports

concepts that cannot be represented in the source language, a grammar that uses these concepts cannot be created by the translation. As Xtext does not use a symbol table, we cannot decide whether a production was a symbol before we translated it to Xtext. Although bijectivity is impossible due to these reasons, there are ways to improve our approach concerning bidirectional translations. One possible extension would be a metalanguage dependent transformation, which reintroduces concepts such as inheritance by applying refactoring transformations, which aim to implement the best practices of the grammar definition. By this, a refactored language that conforms with its best practices would reduce the grammar ambiguity. Nonetheless, we do not necessarily receive the initial source grammar using these rules. Moreover, the translation between MontiCore and Xtext is not fully language equivalent as some language concepts are incompatible. For example, MontiCore inherits the concept of semantic predicates from ANTLR 4, whereas Xtext forbids using semantic predicates. Hence, languages that use semantic predicates are not directly expressible in Xtext.

In addition to the language inequivalence, also the abstract syntax of the languages are not entirely equivalent, as the implemented translations are not bijective. Therefore, it is not possible to directly reuse all well-formedness rules, transformations, or generators that are based on the AS. However, there are two possibilities to synchronize the well-formedness rules. First, it could be beneficial to map every AS class of the source AS to a class of the target AS by extending the translation with AST-conservative rules and newly implement the well-formedness rules for the target AS. Second, we could alternatively rewrite one of the grammars until the resulting abstract syntax is CS-AST compliant. We expect that changing the source grammar to be a more beneficial solution, as changing the source AS to conform to the target AS corresponds to reducing the source grammar to a new version that is consistent with the target metagrammar.

In addition to the AST translation, also the translation of well-formedness rules is not covered in our approach yet. Since the implementation of these rules in MontiCore is done in Java, it is not straightforward to translate into a different language workbench, as different generated infrastructure might be required to check these rules in the target implementation. As grammars alone are not capable of fully defining a language, different concepts for the specification and translation between language workbenches could be investigated or developed in future works to improve the translation of languages that were created using different language workbenches. It could be beneficial to translate grammars to metamodels and back, as this concept could facilitate the integration of other language workbenches or integration areas for the language itself. As there already exist approaches for this translation in other related works, we will discuss this again in Section 9 when we take a closer look at these works. Finally, lessons learned during the translation between MontiCore and Xtext should be generalized to other LWBs.

9 Related Work

In contrast to most other related works, which focus on transcompilation or metamodel translation, we focused on the translation of grammars between language workbenches.

Additionally, the approach presented in [29] derives an M3-level translation to bridge the technology gap between grammarware and modelware. Grammarware includes all artifacts that are directly related to grammar. The M3-level for grammarware is the metagrammar. Modelware includes all model-based artifacts. The M3-level for modelware is the MOF. Consequently, an M3 level translation translates grammars to metamodels. The work from [29] chooses EBNF as metagrammar as it is the most commonly used one. Similarly, the concept presented in [16] develops a related concept that additionally translates programs (the models of a grammar) to models of the derived metamodel.

Another concept based on MontiCore translations is described in [4] and introduces a concept and an implementation for the translation of MontiCore grammars to metamodels with Ecore and MontiCore models into Ecore instances. Since MontiCore is already capable of generating AST classes that conform to EMF, it is only necessary to derive an Ecore metamodel. This generator serializes the meta AST, *i.e.,* the classes that build the AST. They use OCL constraints to map cardinalities from the grammar in the metamodel. Following this, we can leverage tools like Sirius for graphical modeling and the visualization of languages. Additionally, in [1] an approach to create Ecore metamodels based on Xtext grammars is presented. Based on these approaches future works could investigate whether the bidirectional translations between MontiCore and Xtext could be improved by using the Ecore metamodel as an intermediate translation step.

The idea of deriving metamodels from grammars leveraging Xtext has been done previously [1, 13]. Both works take EBNF grammars as input and translate them to Xtext. From there, Xtext takes over for the metamodel translation. For EBNF grammars, this is intuitive because Xtext uses a syntax that is close to EBNF's. The Grammar-to-Model Language (Gra2Mol) [13] uses an early version of Xtext. Thus, the derived metamodels are not matured, yet.

10 Conclusion

Model-driven systems engineering demands integration of truly heterogeneous modeling languages to automate consistency checking, tracing, and change propagation of models developed by different stakeholders. This demands to bridge the gaps between the different technological spaces of the

participating modeling languages. To this end, we have presented challenges for translation of grammars across different technological spaces, a systematic method to translate grammars between the technological spaces of MontiCore and Xtext, and lessons learned while developing this method. Overall, varying degrees of language compatibility are feasible and must be considered when developing a language in a particular technological space to enable its translation in another. When considering these challenges, automated translation of grammars, and, hence, seamless integration of solutions engineered by stakeholders from different domains becomes possible. This can greatly facilitate pervasive model-driven systems engineering in the future.

References

[1] Alexander Bergmayr and Manuel Wimmer. 2013. Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. *CEUR Workshop Proceedings* 1104 (01 2013), 22–31.

[2] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, New York, NY, USA, 84–89.

[3] Cédric Brun and Alfonso Pierantonio. 2008. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional* 9, 2 (2008), 29–34.

[4] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2018. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*. ACM, 174–186.

[5] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Software: Practice and Experience* (February 2019).

[6] Robert Eikermann, Katrin Hölldobler, Alexander Roth, and Bernhard Rumpe. 2018. Reuse and Customization for Code Generators: Synergy by Transformations and Templates. In *International Conference on Model-Driven Engineering and Software Development*. Springer, 34–55.

[7] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Lecture Notes in Computer Science, Vol. 8225. Springer International Publishing.

[8] Katrin Hölldobler and Bernhard Rumpe. 2017. *MontiCore 5 Language Workbench Edition 2017*. Shaker Verlag.

[9] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures* 54 (2018), 386–405.

[10] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures* 54 (2018), 386–405.

[11] John E. Hopcroft. 2008. *Introduction to automata theory, languages, and computation*. Pearson Education India, Delhi.

[12] International Organization for Standardization. 1996. ISO / IEC 14977: 1996 (E). *Information technology — Syntactic metalanguage — Extended BNF* (1996).

[13] Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, and Jesús García Molina. 2008. Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. In *Workshop on Model-Driven Software Evolution*. 1–8.

[14] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of computer programming* 72, 1-2 (2008), 31–39.

[15] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. *ACM sigplan notices* 45, 10 (2010), 444–463.

[16] Andreas Kunert. 2008. Semi-automatic Generation of Metamodels and Models From Grammars and Programs. *Electronic Notes in Theoretical Computer Science* 211 (2008), 111–119. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).

[17] Bernhard Merkle. 2010. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 139–148.

[18] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. 2015. Systems of systems engineering: basic concepts, model-based techniques, and research directions. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 18.

[19] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. 2014. A survey on domain-specific languages in robotics. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 195–206.

[20] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2015. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics* (2015), 33–57.

[21] Bernhard Rumpe. 2016. *Modeling with UML: Language, Concepts, Methods*. Springer International.

[22] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education, Indianapolis, Indiana 46240.

[23] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[24] Antoine Toulmé. 2006. Presentation of EMF Compare Utility. In *Eclipse Modeling Symposium*. Intalio Inc, Redwood City, CA 94065, USA, 1–8.

[25] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[26] Eelco Visser. 2007. WebDSL: A case study in domain-specific language engineering. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 291–373.

[27] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. 233–238.

[28] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. 2013. *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[29] Manuel Wimmer and Gerhard Kramler. 2006. Bridging Grammarware and Modelware. In *Satellite Events at the MoDELS 2005 Conference*, Jean-Michel Bruel (Ed.). Springer, Berlin, Heidelberg, 159–168.

[30] Robert S. Wolf. 1998. Proof, Logic, and Conjecture the Mathematician's Toolbox. (1998), 421.

[31] Andreas Wortmann, Benoit Combemale, and Olivier Barais. 2017. A Systematic Mapping Study on Modeling for Industry 4.0. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 281–291.