Translating Grammars to Accurate Metamodels

Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
www.se-rwth.de

Abstract

There is a software language engineering gap between metamodel-based languages and grammar-based languages. Grammars can support integrated definition of concrete syntax and abstract syntax, which facilitates processing models, but usually prevents reusing the variety of language tools operating on Ecore metamodels (such as editors, interpreters, debuggers, *etc.*). Existing work on translating grammars to Ecore metamodels features very cursory translations only, which requires re-engineering intricacies natural to grammars for the metamodels again. We conceived a translation from an EBNF-like syntax to Ecore metamodels that considers the grammars' intricacies. This translation is realized as a fully automated toolchain from grammars into Ecore & OCL using the language workbench MontiCore. Using this translation enables grammar-based languages to leverage the benefits of Ecore-compatible language tools while supporting natural definition of concrete and abstract syntax.

CCS Concepts • **Software and its engineering → Software notations and tools**; **Domain specific languages**;

Keywords Grammarware, Metamodels, Language Translation

ACM Reference Format:
Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2018. Translating Grammars to Accurate Metamodels. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276605

1 Introduction

Model-driven development (MDD) [15, 44] lifts models to primary development artifacts. It has been successfully applied to various challenging domains, including automotive [4], avionics [14], and robotics [45]. The automated analysis and transformation of models requires their adherence to software languages, which can be general modeling languages (*e.g.,* UML [27], SysML [16]), applicable to a variety of challenges and domains, or domain-specific languages (DSLs).

Independent of their nature, the success of software languages depends on the available tooling (editors [42], compilers [1], interpreters [5], *etc.*). For metamodel-based languages implemented with Ecore [35], a large body of tooling is available in the Eclipse environment. However, metamodels describe abstract syntax (structure) of a language only and their concrete syntax usually is defined through editors.

The ongoing success of textual languages for software developers indicates that text is their preferential model representation. This is in line with the observation that textual model presentation facilitates reusing established tooling for common activities (editing, differencing, *etc.*). Textual languages can be defined with grammars [22] easily [13, 33, 39, 40]. However, these usually lack integration with Ecore to benefit from the the wealth of existing tooling. Consequently, such tooling must be co-developed and evolved alongside the grammar, which requires significant additional efforts. On the other hand, the parsers generated from grammars often support forms of model validation (*e.g.,* checking complex cardinalities) that cannot be expressed by metamodels only and require additional efforts from metamodel-based languages. To facilitate integration of language concerns realized within different technological spaces and reusing Ecore-based tooling with grammar-based languages, bridging these different spaces is necessary [21].

To reduce the gap between grammar-based languages and Ecore-based languages, and to facilitate the development of language processing tooling for textual languages, we conceived and developed a translation from grammars to Ecore metamodels. In contrast to naive translations, the generated metamodels are augmented with OCL [17] constraints and Java methods to capture grammar intricacies, such as complex grammar expressions (*e.g.,* multiply nested iterations and disjunctions), that represent several systems of linear equations to ensure the metamodel adheres to the implied cardinalities. Using the MontiCore [33] language workbench, the translation not only supports EBNF, but leverages features of MontiCore's metalanguage to establish associations between different parts of the abstract syntax. Overall, the contributions of this paper towards bridging the gap between grammarware and modelware are:

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276605

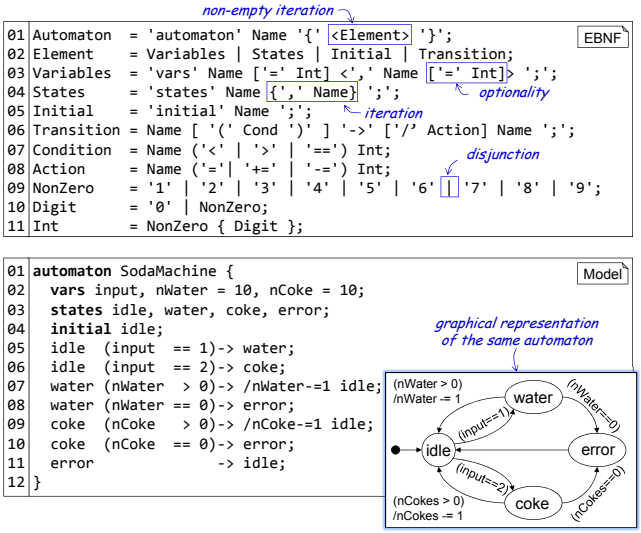

Figure 1. EBNF grammar of an automaton language (top), corresponding textual model (bottom left), and graphical representation (bottom right).

- A concept for translating EBNF-like grammars into accurate metamodels that leverages OCL constraints and additional Java solvers to capture the intricate and implicit grammar cardinalities.
- A realization of this translation that translates MontiCore grammars into Ecore metamodels.

In the following, Section 2 motivates the benefits of bridging both worlds by example before Section 3 presents preliminaries. Section 4 introduces our translation concept and Section 5 presents its realization. Afterwards, Section 6 illustrates its benefits through a case study and Section 7 discusses observations. Ultimately, Section 8 highlights related work and Section 9 concludes.

2 Motivating Example

Consider developing a textual domain-specific language to describe manufacturing processes and their integration with cyber-physical production systems (CPPS). These languages are becoming common [47] with Industry 4.0 [6] and related advances, such as the Japanese Industrial Value Chain Initiative [37] or the Advanced Manufacturing Initiative in the United States [38]. Part of this language is describing the discrete state-based behavior of CPPS through finite automata by the CPPS developers (*e.g.*, [24, 29]).

The simplified grammar in EBNF for the syntax of such automata over variables and integer numbers is illustrated in Figure 1. It defines that an automaton features at least one `Element` (l. 1), which can be a list of variables, a list of states, an initial state, or a transition (l. 2). Variable lists feature optional initial value assignments per variable (l. 3), list of states consist of an iteration of names (l. 4), and initial

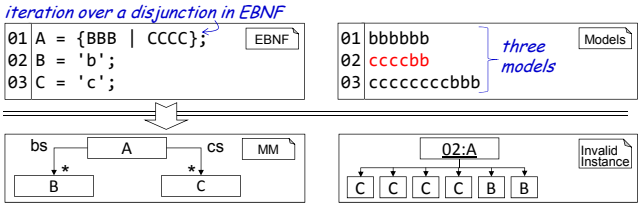

Figure 2. EBNF grammar (top left), corresponding textual models (top right), naively derived metamodel (bottom left), and corresponding instance (bottom right).

state declarations yield a single name (l. 5). Transitions feature two names directed at state, a single Boolean condition over an integer and an action assigning, incrementing, or decrementing a value (ll. 6-8).

Below this grammar, Figure 1 depicts a conforming model that describes a soda dispensing machine. The machine initially provides ten bottles of water and ten bottles of coke (l. 2). Upon pushing one of its two buttons, it either dispenses a bottle of water or a bottle of coke - if available (ll. 5-10). Otherwise, it reaches an error state from which it can only return to its initial state idle (l. 11).

Developing a language for such automata, however, requires more than syntax [9], such as well-formedness rules, realization of semantics, analyses, editors, documentation, and more. Hence, prior to producing a potentially shippable prototype to the customer, producing corresponding tooling requires significant effort and the Ecore infrastructure can help with that. Therefore, instead of developing, for instance, an editor, from scratch, reusing frameworks such as Sirius [42] can facilitate language tool engineering.

However, translating grammars manually to metamodels is costly, error prone, and only half the way: instances of textual models must be translated into Ecore metamodel instances also. Existing approaches to this [2, 19, 43], however, do not consider the constraints imposed by *implicit cardinalities* of grammar rules in the metamodel. Considering, for instance, the grammar depicted in Figure 2 (top left), which accepts multiples of three `bs` or four `cs`. A naive translation, as performed, *e.g.*, by Xtext [43], derives a concept `A` with lists `bs` and `cs` relating to `B` and `C` instances accordingly and some validity checking in the generated editors. While this enables capturing all models of the grammar, it also supports nonconforming models, such as the second model `ccccbb` (top right) as depicted by the compatible instance (bottom right), when instantiating the models through other means (*e.g.*, programmatically or via graphical editors that are not directly related to the Xtext approach itself). Consequently, when developing tooling for such a language, these intricacies must be considered manually (*e.g.*, within its editors, generators, *etc.*). Automatically deriving context conditions restricting the numbers of `bs` and `cs` from the grammar and integrating these directly into the metamodel liberates from

their manual definition. This facilities engineering of tooling for textual languages.

To this end, we propose generating linear equation systems (LESs) for each concepts' context conditions that have a solution if a model provides a correct number of model element instances. For the metamodel of Figure 2 (bottom left), the LES over variables x, y representing the iterations of block alternatives could be:

$$|bs| = 3 * x \quad (1)$$

$$|cs| = 4 * y \quad (2)$$

Then, each data structure with fixed numbers of `bs` and `cs` is a valid instance of the metamodel, if there is a solution to this LES. For instance, this would rule out the second model, `ccccbb`, as there is no integer solution for $|bs| = 2$ and $|cs| = 4$.

The next sections present a translation from grammars into accurate metamodels that adhere to the grammars' implicit cardinalities by adding generated LESs to corresponding solvers invoked by OCL invariants.

3 Preliminaries

We realize our translation from grammars to metamodels using the MontiCore language workbench and Ecore metamodels with attached OCL. This section describes all three.

3.1 MontiCore

MontiCore [18, 33] is a language workbench for engineering compositional DSLs[1]. Its languages are defined as extended context-free grammars (CFGs) that enable integrated development of concrete syntax and abstract syntax. From this grammar, MontiCore generates abstract syntax classes, a parser translating textual models into abstract syntax trees (ASTs), *i.e.,* instances of these classes, symbol tables, a model checking infrastructure, and an infrastructure for template-based code generation. The model checking infrastructure is used, *e.g.,* to execute handcrafted context conditions to check well-formedness properties not expressible with CFGs (*e.g.,* duplicate occurrence of the same name). Symbol tables store *symbols*, *i.e.,* essential information on model parts, meant as interface for other languages. Symbol tables of a class diagram model, for instance, could comprise information about the names of the contained classes, their method signatures or relations, but hide information on their methods' implementations. Once a model is translated into an AST, its symbol table is created, and its well-formedness is checked. Code generators then transform ASTs of well-formed models into target language artifacts (*e.g.,* Java, XML, or anything else that can be represented as text). For compositional language engineering, MontiCore also supports (1) aggregation: a loose coupling between symbols of different languages; (2) embedding: a combination of abstract

[1]MontiCore on GitHub: https://github.com/MontiCore/monticore

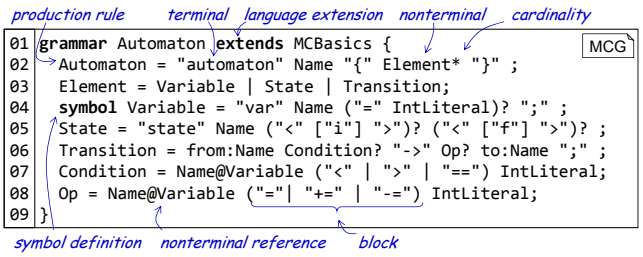

Figure 3. MontiCore grammar of an automata language.

syntax classes through underspecification in the host language; and (3) extension: in which one language inherits all productions of its parent languages).

Figure 3 illustrates the quintessential elements of a MontiCore grammar: Each grammar begins with a name and a list of possibly extended grammars (l. 1). Here, `Automaton` extends `MCBasics` to inherit the productions `Name` (*cf.* l. 1) and `IntLiteral` (*cf.* l. 4), which are translated into the data types String and Integer. A grammar's body contains production rules that define the abstract syntax types. Here, for instance, the type `Automaton` is defined to yield an attribute `Name` and a list of `Element` instances. This rule also defines the concrete syntax of `Automaton` instances, which begin with the keyword `automaton`, followed by a name, and a body of elements delimited by curly brackets. To define production rules, MontiCore supports the usual EBNF operations, *i.e.,* iteration (l. 2), disjunction (l. 3), and optionality (l. 4). Additionally, it supports the definition of symbols (l. 4), which can be referenced from ASTs, if declared via nonterminal references (l. 7). Here, `Name@Variable` denotes that the name used in the left-hand side of a `Condition` must reference the name of an existing `Variable` instance of the same model. The generated abstract syntax classes are augmented with methods resolving instances of the referenced nonterminals through the symbol tables. Leveraging the symbol table enables MontiCore to express not only the containment trees typical for grammar-based metalanguages, but also associations between the tree's different branches, *i.e.,* graph structures of the abstract syntax. As with EBNF, MontiCore supports the definition of languages with intricate implicit cardinalities by nesting disjunctions and iterations in hierarchical blocks.

3.2 EMF Ecore and OCL

The Eclipse Modeling Framework (EMF) [34] is a Java framework for model-driven engineering that is part of the Eclipse infrastructure. To describe domain models, it features the graphical Ecore metalanguage (similar to class diagrams) and generates a tree-based editor to instantiate these. In combination with various other Eclipse-based frameworks and tools (*e.g.,* Sirius [42] for more precise graphical concrete syntax,

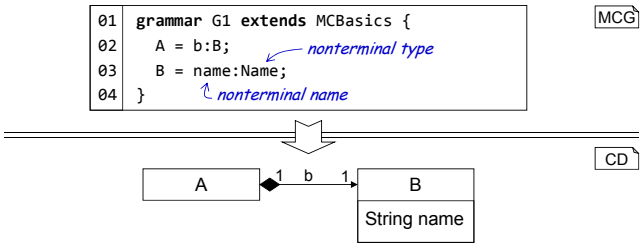

Figure 4. Simple nonterminal attributes are translated into compositions between the class representing the grammar rule and its nonterminal attribute.

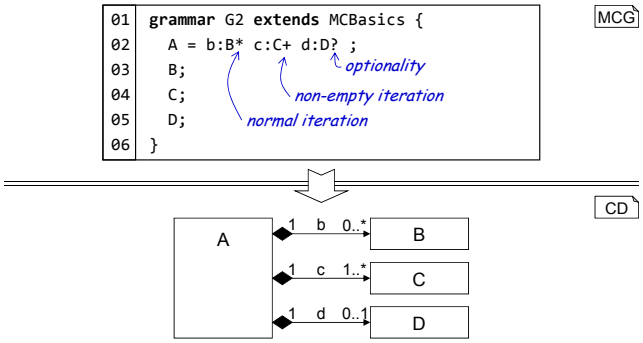

Figure 5. Grammar cardinalities are translated into compositions with the corresponding cardinalities.

OCL [17] for well-formedness checking, or Xtend [3] for code generation), Ecore is suitable for language engineering.

The Object Constraint Language (OCL) [10, 30] is a declarative specification language over object structures developed to constrain UML [27] models. It enables specifying constraints that cannot be (easily) represented by the underlying model itself. For instance, OCL enables restricting class diagrams to yield no classes featuring methods of the same names. The Eclipse OCL plugin enables specifying constraints for Ecore models to describe their well-formedness in terms of invariants, preconditions, and postconditions for metamodel concepts.

4 Translating Grammar Rules to Metamodel Concepts

This section introduces our translation from grammar rules into metamodel concepts by presenting how the different kinds of grammar rules are derived into metamodel concepts with related OCL constraints. The translation rules cover all kinds of grammar rules of MontiCore's extended CFGs, which covers complete EBNF [33]. Therefore, this translation concept enables translating arbitrary EBNF grammars to metamodels. We chose MontiCore for illustration nonetheless, as its support for nonterminal references enables metamodels that are graphs instead of trees only.

Exclusive disjunctions, however, cannot be represented into metamodels natively. For this, we present OCL invariants to ensure both representations of the language accept the same models. Moreover, the implicit cardinalities prescribed by nested blocks of exclusive disjunctions entail LESs (*cf.* Figure 2) that cannot be represented by comprehensible OCL easily. We delegate this to a solver, which is invoked from OCL instead. This section presents derivation of metamodel classes, construction of production graphs, and their translation into LESs that are invoked from the generated OCL invariants.

4.1 Derivation Rules

MontiCore's extended CFGs comprise nonterminal definitions, iterations, optionality, explicit cardinalities, disjunctions, interface production rules, abstract production rules, and nonterminal references. This section describes their translation into accurate metamodels.

Nonterminal Definitions: Nonterminal definitions become concepts of the same name in the metamodel. For each nonterminal used in the right-hand side of a nonterminal definition, the corresponding concept yields an association of the nonterminal's name and type. References to basic types (*e.g.,* numbers, strings, Booleans) instead become basic data types in the metamodel. Figure 4 illustrates this. The grammar `G1` (top) comprises the nonterminal definitions `A` (l. 2) and `B` (l. 3). The production rule of `A` contains the nonterminal `b` of type `B`. The production rule `B` contains the nonterminal `name` of type `Name`. The corresponding metamodel (bottom) consequently consists of the two classes `A` and `B`. Class `A` contains a composition of name `b` and cardinality one to class `B`. Class `B` yields an attribute `name` of type `String`, because `Name` is a special non-terminal that translates to the type `String`.

Cardinalities: As EBNF, MontiCore supports different kinds of iterations that implicitly express cardinalities. These are the normal, possibly empty, iteration (∗), the non-empty iteration (+), the optional, one or none, iteration (?), and the default cardinality (*i.e.,* 1). All of these become explicit association cardinalities in the metamodels as illustrated in Figure 5: The production rule for `A` (l. 2) references the nonterminals `B`, `C`, and `D` (ll. 3-5) – which yield right-hand sides that are omitted from now on – with different cardinalities each. For the normal iteration, we implement an association with an arbitrary number (0..∗) of entries. Non-empty iterations become an association of at least one entry (1..∗). Optional iterations become associations with zero or one entries (0..1). Moreover, MontiCore supports explicit specification of cardinalities. To this end, nonterminal definitions

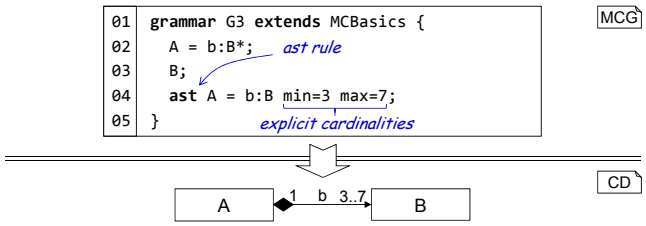

Figure 6. Grammar rules support custom boundaries for the iteration of nonterminals. These are translated into corresponding cardinality values in the metamodel.

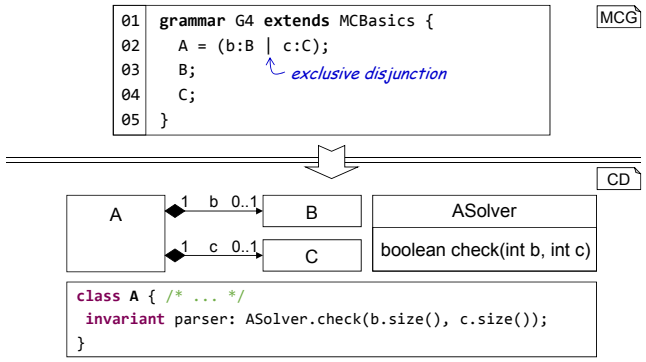

Figure 7. Exclusive disjunctions are translated as independent compositions with cardinality 0..1. A generated `Solver` verifies that exactly one alternative is present in a model.

can be accompanied by so-called AST rules that support attaching additional properties to the resulting abstract syntax classes, such as methods, derived fields, or boundaries for iterations. Figure 6 illustrates this with grammar `G3`, which contains a nonterminal definition `A` featuring an iteration of `B` instances (l. 2). Additionally, the grammar features an AST rule describing the boundaries for `A`'s `B` instances (l. 4). Additional AST rule features are presented in [33].

MontiCore also supports defining custom cardinalities by specifying an explicit number of nonterminal instances. The production `A = BB`, for example, describes two occurrences of nonterminal `B`. In this case, the generated abstract syntax class `A` contains a list `bs` of arbitrary cardinality. Here, the parser ensures the correct number of `B` instances (*i.e.,* two). The translation principle that considers such intricacies is explained in the following.

Disjunctions: As EBNF, MontiCore supports exclusive disjunctions (*i.e.,* requiring that exactly one of the alternatives holds). Together with iterations, these prevent a naive translation from grammars to metamodels as discussed in Section 2. Figure 7 (top) illustrates a simple disjunction in the nonterminal definition of `A` (l. 2), which can feature exactly one instance of `B` or exactly one instance of `C`, but not both. In a naive translation, this disjunction between alternatives gets lost in the resulting metamodel as this cannot be expressed

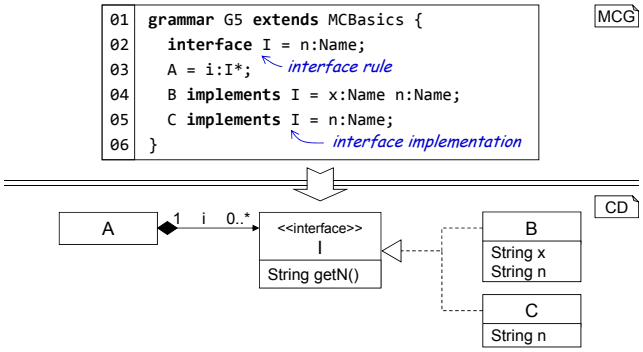

Figure 8. Nonterminal interfaces are translated into metamodel interfaces with method signatures for defined attributes.

with metamodels without introducing artificial aggregation concepts. To enable holding the instance of `B` or of `C`, we introduce associations from `A` to one instance of each. To ensure that neither both are absent, nor both are present, we introduce an invariant to class `A` checking this.

Generally, this can be ensured by requiring the invariant `(self.b->size() == 0) xor (self.c->size() == 0)`. However, for more complicated, *e.g.,* nested disjunctions, this becomes less comprehensible with each alternative (*cf.* Figure 2) as OCL is not tailored to specify equation systems. Instead, we opted for hiding these 'parsing constraints' from the developer within a Java class generated for each metamodel class that takes care of evaluating the resulting LES as depicted in Figure 7 (bottom). Here, the generated Java class `ASolver` wraps multiple LESs per derived metamodel concept. Moreover, it yields a single public method taking the numbers of instances of associations of the related concept and returning true, iff these are a solution of one of the LESs. The generated invariant `parser` for metamodel class `A` holds exactly if the solver found a solution to the LESs related to `A`. Thus, we do not pollute the OCL with LESs, but keep these out of the developers sight and can solve these more efficiently than through OCL.

Interface Productions and Abstract Productions: For underspecification and flexible language reuse, MontiCore's extended CFGs support interface productions and abstract productions that operate similar to interfaces and abstract classes in object-oriented programming. Interface productions can specify abstract syntax properties of implementing productions. Abstract productions can provide abstract syntax properties to inheriting productions. Both are transformed to metamodels as expected.

Figure 8 shows how MontiCore grammars can realize interfaces (top) and how we translate these to metamodels (bottom). The grammar `G5` features the interface production `I` (l. 3), which prescribes that interfaces must yield a nonterminal `n` of type `Name`, and two implementations of this

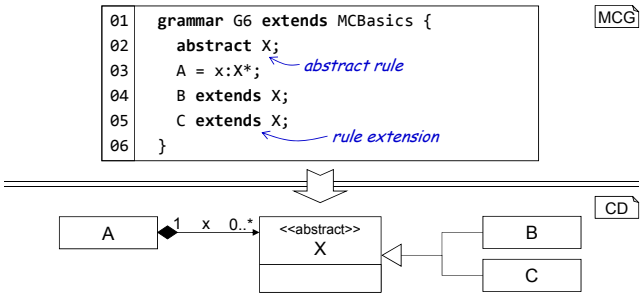

Figure 9. Abstract nonterminals are translated into abstract classes that can be extended by subclasses.

interface (ll. 4-5). From the productions of grammar G5, we derive the four metamodel classes depicted below the grammar. As expected, the class A contains a list of I instances, which can be either Bs or Cs. For abstract grammar nonterminals this translation is similar, as depicted in Figure 9. Here, X is an abstract nonterminal that is translated into an abstract metamodel class and the derived classes B and C inherit from it accordingly.

Nonterminal References: Grammars generally express containment trees, *i.e.,* nonterminal definitions contain instances of other nonterminal definitions, whereas metamodels generally define multi-graphs of classes. MontiCore alleviates this by featuring nonterminal references that enable creating associations from nonterminals to symbols (*cf.* Section 3.1) of other nonterminals. Grammar productions that define referenceable nonterminals must start with the keyword `symbol` and must contain a name on their right-hand side. Other grammar productions then may reference this name, *i.e.,* the symbol with this name. For each nonterminal reference, MontiCore generates the abstract syntax classes such that these use the symbol table to resolve the symbol corresponding to the targeted nonterminal. From this symbol, its AST can be retrieved.

Figure 10 illustrates this with the grammar G7, which contains the nonterminal A, for which MontiCore also produces a symbol (l. 2). Further, it contains the nonterminal B, which features the nonterminal reference `ref` that points to an A instance. We translate this into a metamodel class A as usual and a metamodel class B that is associated to one instance of A, but instead of featuring a normal attribute `ref` of type `String`, this attribute is derived and points to the associated A instance. Consequently, we also introduce two constraints. The first marks attribute `name` as `id`, which requires that all names used by instances of class A are unique. The second defines the derived attribute `ref`. The `derivation` rule ensures that `ref` actually holds the name of the referenced A instance. We prescribe the latter to prevent prescribing specific implementations of the metamodel classes.

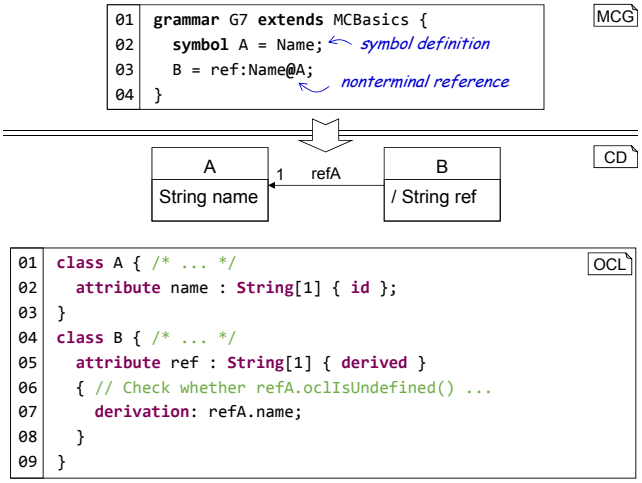

Figure 10. Nonterminal references are translated into explicit associations. An OCL rule defines the derivation for the original `String` attribute.

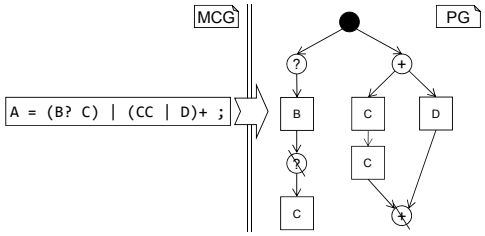

Figure 11. Production rule (left) and its corresponding graph (right). Nonterminals are represented as squares and blocks as circles.

4.2 Production Graphs

Considering more complex productions with nested implicit cardinalities, the fixed cardinalities of pure metamodels are insufficient to represent exclusive disjunctions. In Figure 11, we illustrate this with the nonterminal definition A that either yields a single instance of C that can optionally be preceded by an instance of B (left side of the disjunction) or an even number of C instances (right side) with optional single occurrences of D instances before or after each second C. While this could be mapped to a metamodel class A with multiple associations to Cs of different cardinalities, maintaining this in the language processing software is unfeasible.

To consider these implicit cardinalities properly, we distinguish the grammar rules' explicit cardinalities – which become cardinalities in the metamodel – from the grammar rules' implicit cardinalities. The latter are delegated to a solver in the related concepts' invariants. To prevent confusing the different cardinalities, we denote the explicit metamodel cardinalities as *global* cardinalities (in the sense that these define the global lower and upper bounds of an

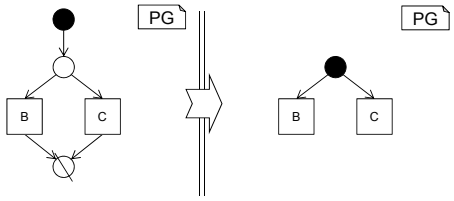

Figure 12. Eliminating blocks of default cardinality.

association) and the implicit metamodel cardinalities as *local* cardinalities.

Figure 11 illustrates the difference between the global cardinality of A's C instances, which overall can occur an arbitrary number of times, and their local cardinality, which describes under which circumstances nonterminal instances may occur. This also includes the coherence with the distribution of other variables. For example, the disjunction's left part features an optional variable B followed by an instance of C. This implies that if B occurs, we require exactly one instance of C. If no B occurs, we may have an arbitrary even number of Cs, or a single C. Thus, an instance of A is valid exactly if it either features (1) a single instance of B and a single instance of C; (2) a single instance of C; or (3) an even number of instances of C and an arbitrary number of instances of D.

In the following, we describe computing the *local* cardinalities and attaching these to generated metamodel's classes. To this end, we propose constructing so-called *production graphs* (PGs) for each production rule in the grammar. Each graph represents the possible 'paths' a parser can follow through the corresponding production rule and these paths are translated into a set of LESs describing the production rules' local cardinalities. The PGs are hierarchically layered as usual with trees and each graph yields two types of nodes: nonterminal nodes and block nodes. The former are labeled with nonterminal names and denote that the specified nonterminal must occur at this specific position on each path through this node. The latter are labeled with block identifiers and denote opening or closing of a block. Succession in paths through the production graph denotes concatenation of nonterminals and sibling relations (*i.e.,* nodes on the same hierarchy level of the PG) denote alternative paths. Moreover, each production graph yields a single root node.

Figure 11 illustrates translating the production rule A into its corresponding production graph. According to the disjunction of A, the production graph yields two alternatives. The first (left) begins with an optional block containing a nonterminal node requiring an instance of B if this path is taken. The second (right) begins with an iteration block containing either two subsequent instances of C or a single instance of D. From these paths, we can derive the LESs that yield solutions if an instance of A is correct regarding As implicit cardinalities.

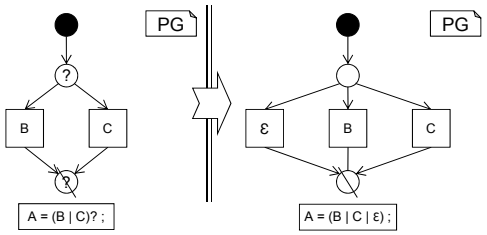

Figure 13. Replacing blocks of optional (*i.e.,* ?) cardinality by blocks of default cardinality featuring an ε node.

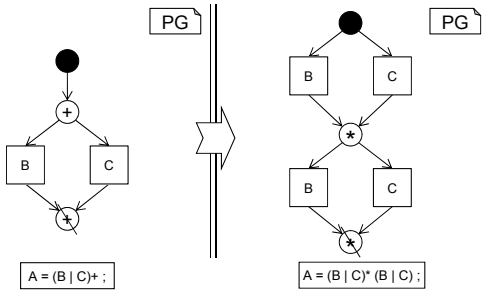

Figure 14. Replacing blocks of non-empty iteration (*i.e.,* +) by blocks of default cardinality followed by blocks of arbitrary iteration (*i.e.,* ∗).

In the following, we normalize PGs to facilitate their translation in LESs. To this end, we convert the branches in such a way that the graph only contains blocks of iterations (*e.g.,* ∗ cardinalities). The next paragraphs therefore describe elimination of block nodes labeled with the default (*i.e.,* 1), ?, and + cardinalities. Figure 12 illustrates elimination of blocks with default cardinality. Here, the block node is removed and its content is directly attached to its parent (the predecessor node one level higher in the production graph hierarchy). If that block contains a disjunction, its alternative paths become attached to the parent. Blocks denoting optional cardinality are eliminated by introducing the alternative of not processing any nonterminal. We denote this by nodes labeled with ε. Paths featuring this node do not have to process a nonterminal at this position. As depicted in Figure 13, this introduces a new branch to express omitting a nonterminal. The non-empty iteration requires for each contained block that at least one of its contained elements occurs. Regarding the number of related instances, this amounts to enforcing a single occurrence of the contained elements followed by an arbitrary number of iterations. Consequently, we express blocks of non-empty iteration as a single mandatory block followed by a block indicating arbitrary iterations. In Figure 14, the block containing at least one B or one C becomes a block containing exactly one B or C followed by a block containing arbitrary numbers of both. After iteratively applying all eliminations, the normalized production graph

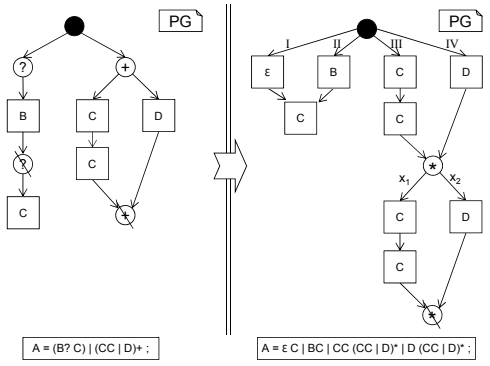

Figure 15. Normalizing a production graph and variable assignment to branches of normal iteration (*star*) blocks.

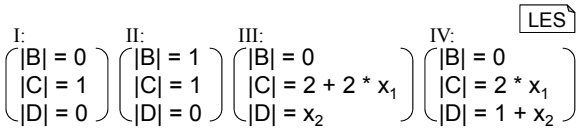

Figure 16. Resulting set of LESs for existing variables of the production in Figure 11.

consists of a single root node, nonterminal nodes, and arbitrary iteration blocks only. For the production graph depicted in Figure 11, this results in the normalized production graph depicted in Figure 15 (right), which is equivalent to its former representation, but facilitates further analysis.

Moreover, we labeled the different paths leaving the root node, which entail different LESs. Also, we labeled the distinct branches of the * cardinality blocks. These labels correspond to pairwise disjoint variables. With the normalized production graph at hand, we can extract equations for each variable concerning the existing branches. For evaluation, we interpret different branches of * blocks as multiplication of the content of the block with the labeled variable (x_i). The remaining parent-child relations between nonterminal nodes are evaluated as an addition. Different branches from * blocks that contribute to the same variable also become additions. Consequently the normalized production graph of Figure 15 entails four LESs as depicted in Figure 16.

For each branch of the root node, we derive a dedicated LES that describes cardinality constraints for each variable of the complete production rule. Following, *e.g.,* branch `III`, we observe that a valid instance of `A` with respect to branch `III`, (1) cannot yield instances of `B`; (2) must yield two or more instances of `C`; (3) must yield an even number of `C` instances; and (4) can yield an arbitrary number of instances of `D`. This results in the three equations that are depicted in Figure 16 (`III`).

After transforming the production graph, we obtain a set of LESs that represents the local cardinalities of the processed production rule. To integrate the LESs into the metamodel,

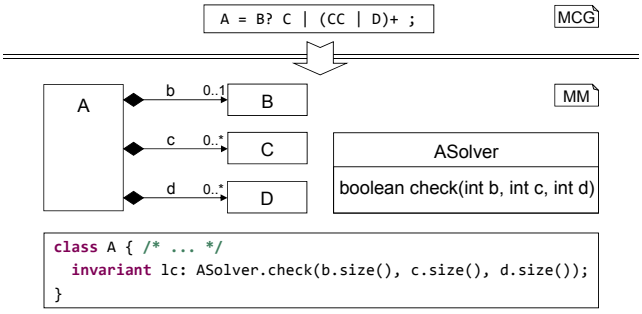

Figure 17. Complete transformation of a production rule into metamodel classes, OCL, and a Java solver ensuring its *local* cardinalities.

we generate a solver for the production rule's specific set of LESs that takes the number of instances as input and applies Gaussian elimination [36]. Therefore, we impose additional constraints: (1) Only accept whole numbers as solution since the algorithm may find arbitrary decimals otherwise. (2) Nested iterations require an additional condition ensuring that the inner loop demands at least one iteration of the outer loop. (3) Variables marked as `unset` are always considered to have zero instances despite their actual value. If one of the LESs has at least one solution, the model is valid; otherwise, it is rejected.

We connect the solver to the metamodel by generating OCL invariants for the metamodel classes that hold if the solver finds a solution. Figure 17 illustrates this for the production rule depicted in Figure 11. First, we translate the nonterminal definitions to metamodel classes and add corresponding associations hold their references instances of other metamodel classes (nonterminals). For structural reference, we impose the corresponding *global* cardinalities onto each association. To check the *local* cardinalities, we generate the additional Java class `ASolver`. Its `check()` method takes the numbers of instances and invokes the LES solver. An OCL invariant invokes the method and holds if this check was successful. This concept – naive translation of grammar rules into metamodel concepts, construction of production graphs, derivation of LESs, and generation of a solver attached to the metamodel classes through OCL – enables translating complex grammars into accurate metamodels for many grammar-based language definition techniques.

5 From MontiCore to Ecore

Translating grammars to accurate metamodels enables to reuse the available metamodel-based tooling. However, without parsing textual models into metamodel instances, this loses the benefits of textual modeling. Thus, this section presents the toolchain translating MontiCore grammars to Ecore metamodels and translating textual MontiCore models into Ecore instances.

5.1 MontiCore Grammars to Ecore Metamodels

MontiCore translates CFGs into Java abstract syntax classes and generates a parser translating textual models into instances of these classes (the AST). To achieve this, it first translates the CFGs into UML/P [32] abstract syntax class diagrams (AS CDs), which normally are translated into Java artifacts. We extended this toolchain with components (1) enriching the AS CD with EMF-specific methods and attributes, such as the `eGet()` and `eSet()` methods, and ensuring its classes implement the EMF interfaces related to serialization; (2) calculating production graphs from MontiCore CFGs; (3) transforming the AS CD into Java implementations and the PGs into corresponding solvers; and (4) translating this Java AS implementation into an accurate Ecore metamodel.

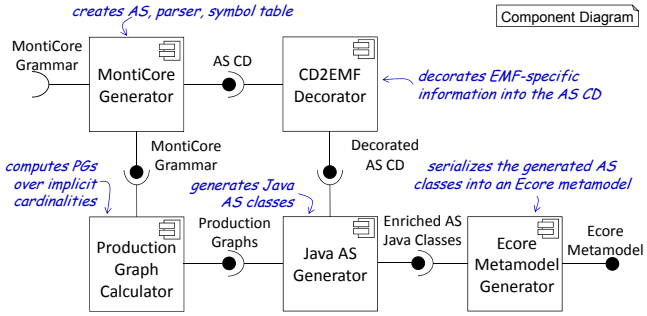

Figure 18. Component diagram of the MontiCore toolchain with Ecore metamodel generation.

Figure 18 illustrates these components and their interaction. First, the MontiCore generator processes the grammar and produces an AS CD, an ANTLR [28] parser capable of translating textual models into abstract syntax trees, and the corresponding symbol table infrastructure. The CD2EMF decorator adds EMF-specific information to the AS CD, which is necessary to serialize the contained classes into naive metamodels (*cf.* Section 4.1). The production graph generator takes the processed grammar, extracts the production rules, translates these into graph representations, and normalizes the latter (*cf.* Section 4.2). The Java AS generator takes the decorated AS CD and the productions graphs and produces LES solvers from the production graphs as well as Java AS classes enriched with information about related solvers. Finally, the Ecore metamodel generator takes the enriched AS Java classes and solvers and serializes these into an accurate metamodel with attached OCL rules invoking the solvers.

5.2 Textual Models to Ecore Instances

Realizing a useful bridge between MontiCore and EMF also requires translating textual models into metamodel instances. To this effect, we leverage the EMF-enriched AS classes: After MontiCore parses and translates the textual models into ASTs, their AS classes' EMF attributes and methods already are EMF-conform models, which only need to be serialized prior to further use. In detail, we leverage EMF's `Resources` as an interface for managing its `EObjects`. As all AST nodes are `EObjects` by construction, MontiCore can serialize the ASTs into a resource by adding the models' root nodes to the corresponding resources.

```
MCG
01 grammar MontiArc extends MCBasics {
02   Component = (itf:"interface")? "component" Name Signature TypeArgs?
03               "{" ArcElement* "}";
04   Signature  = Parameters? ("extends" Type)?;
05   Parameters = Parameter ("," Parameter)*;
06   Parameter  = Type Name ("=" Expression)?;
07   ArcElement = (
08     // Structural Elements
09     Ports | SubComponent | Connector |
10     // Component Mode Elements
11     ModeController | InitialMode | RecMode |
12     // Embedded Behavior Elements
13     Variable | JavaBehavior | Automaton
14     // Reuse Elements
15     RuntimeEnv);
16
17   // Component Body Elements
18   Ports          = "port" Port ("," Port)+ ";";
19   Port           = Name (["in"] | ["out"]) Type Names?;
20   Names          = Name ("," Name)*;
21   SubComponent   = "component" Type Arguments? instances:Names ";";
22   Arguments      = Expression ("," Expression)*;
23   Connector      = "connect" source:Name@Port "->" targets:Names ";";
24
25   // Component Modes
26   ModeController = "transitions" "{" (transitions:Transition)* "}";
27   InitialMode    = "initial" Name ";";
28   RecMode        = "mode" modes:Names "{"(Connector | UseStatement)* "}";
29   UseStatement   = "use" components:Names ";";
30
31   // Embedded Behavior Elements
32   Variable       = Type Names? ";" ;
33   Automaton      = "automaton" Name? "{"
34                      (States | InitialStates | Transition )*
35                    "}";
36   States         = "state" Names ";" ;
37   InitialStates  = "initial" Names ("/" Block)? ";";
38   Transition     = source:Name ("->" target:Name)? ("[" Expression "]")?
39                    stimulus:Block? ("/" reaction:Block)? ";";
40   Block          = "{" (Name "=")? Expression ("," Expression)* "}";
41 }
```

Figure 19. Simplified grammar of the MontiArc ADL.

6 Case Study

Consider developing a textual modeling language for software architectures tailored to software developers. When analyzed by system engineers, these, however, prefer a graphical representation. Translating the language's grammar to an Ecore metamodel enables reusing the rich available tooling for this. For this case study, we illustrate translating a simplified grammar of the textual MontiArc [7, 8, 31] component & connector architecture description language (ADL) into an Ecore metamodel whose instances are visualized using the Sirius [42] editor framework.

Generally, a MontiArc architecture consists of hierarchically composed component types that exchange messages through their typed and directed ports. Components either are composed and contain configurations of subcomponents or are atomic and yield a behavior model, which describes their input-output behavior. The behavior of composed components emerges from the behavior of their subcomponents. The simplified MontiArc grammar depicted in Figure 19 illustrates this: The `MontiArc` grammar extends MontiCore's

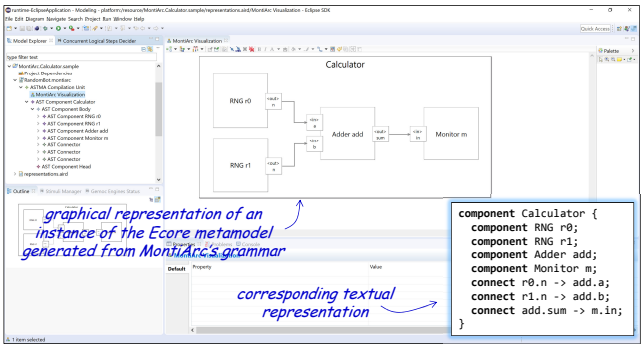

Figure 20. Sirius visualization of a component and connector architecture, modeled with MontiArc.

built-in `MCBasics` grammar to inherit its literals, types, and expressions (l. 1). It defines components with names, signatures, type arguments, and a body of `ArcElement` instances (ll. 2-3). The latter is a disjunction over everything allowed in a component body, such as ports, subcomponents, connectors, reconfiguration models, behavior automata, *etc.* (ll. 7-15). Afterwards, the grammar defines these nonterminals (ll. 18-41).

From this grammar, we can automatically derive the metamodel illustrated in Figure 21. This metamodel is the product of the derivation rules presented in Section 4.1 and contains OCL invariants for each class derived from a production featuring disjunctions. For all these classes, such as `ASTArcElement`, we also generate a solver holding the related LESs and attach it through the class' OCL. In case of `ASTArcElement`, its invariant invokes the `check()` method of the related `ASTArcElementSolver`. The latter takes the sizes of all nine outgoing associations of `ASTArcElement` and passes these to the solver's `check()` method (*cf.* Section 4.2) to ensure that each `ASTArcElement` holds at most one instance and the others are empty.

Translating MontiArc's grammar to an Ecore metamodel facilitates developing graphical editors for it. For this case study, we visualize MontiArc components with Sirius [42]. Components become rectangles containing their name and subcomponents. Ports become smaller squares at each components' border. Incoming ports are on the left, outgoing ports at the right. Connectors are arrows starting from one port and ending in another port. Figure 20 presents a Sirius-based Eclipse editor displaying an example architecture. This architecture is textually modeled with MontiArc (presented as overlay in the bottom-right), translated into an Ecore instance (*cf.* Section 5.2) and visualized via Sirius.

Even if the generated metamodel is instantiated through a graphical editor, its constraints on implicit cardinalities remain ensured through the generated solvers without requiring additional constraint checking in the employed editor (or other metamodel-based tooling). Preventing modeling errors this way facilitates using the language overall.

7 Discussion

Our approach of bridging the gap between grammars and metamodels is based on generating accurate metamodels with embedded LESs that represent the implicit cardinalities raised by (nested) blocks of disjunctions and iterations. For MontiCore and Ecore, it is realized by decorating the AST derived from a MontiCore grammar with EMF-specific information that ultimately is transformed into (a) abstract syntax classes conforming to Ecore interfaces, and (b) an Ecore metamodel interacting with a generated solver for each concept's derived implicit cardinalities. The latter integration is necessary to prevent instantiating invalid (with respect to the grammar) metamodel instances. Although it is possible to integrate the LESs directly into the metamodel's OCL invariants, this becomes confusing even for compact productions. This is an effect of the different paths through the production graphs and cannot be mitigated aside from minor syntactic improvements. Instead we opted for encapsulating these, ultimately, parser-related checks in the generated solvers. This prevents polluting the metamodel's OCL with the LES.

Regarding the solver, we provide a straightforward implementation based on Gaussian elimination. However, due to the loose coupling and compact interface between OCL invariants and the solver, employing more sophisticated LES solvers requires minimal effort.

The availability of efficient LES solving (in cubic time) also motivated us to use LESs over deterministic accepting automata (due to determinization in exponential time) for recognizing (in)valid instantiation of implicit cardinalities in derived metamodel concepts. However, similar to LESs, encoding large automata in OCL reduces its comprehensibility and should be moved to external Java artifacts.

Our contribution investigates forward translation from grammars to metamodels only. We currently investigate to leverage grammar inheritance to derive grammars with a default concrete syntax from metamodels, such that specific syntaxes can be integrated through production inheritance. The feasibility of this approach has to be shown. To this effect, modifications to the metamodels generated by our approach currently cannot be mapped back to modifications in the grammar. Furthermore, modifications of the metamodel can entail that the generated constraints do not apply anymore.

To express well-formedness of models, many language workbenches [12] employ specification mechanisms more expressive than the ones used to describe the languages' abstract syntaxes. For Ecore, these are OCL constraints, for MontiCore these are Java-based context conditions. As these are vital to express well-formedness of models, future work investigates reusing well-formedness rules from one technological space to another. As Ecore metamodels support Java

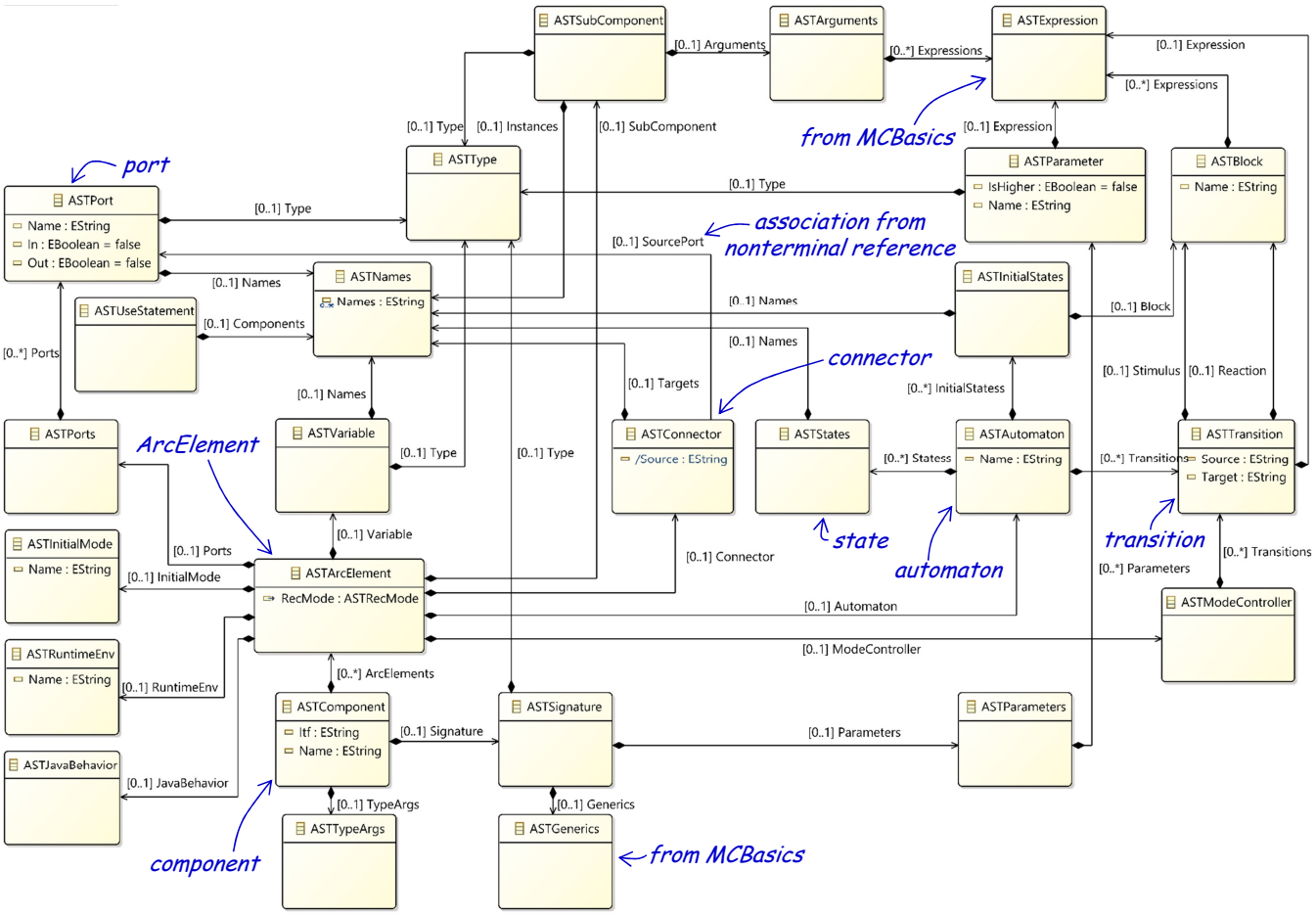

Figure 21. Excerpt of the metamodel generated from MontiArc's grammar omitting some classes generated from `MCBasics`.

well-formedness rules as well, adapting MontiCore's context conditions to the corresponding Ecore interfaces might suffice for the translation of MontiCore grammars into Ecore metamodels. Direct translation between the different formalisms can be more challenging.

8 Related Work

The gap between grammar-based and metamodel-based languages is subject to ongoing research. There are various approaches to combine textual DSLs with metamodels that support a coarse translation only that forfeits considering implicit cardinalities properly. In the following, we discuss related approaches, which perform transformations between grammars and metamodels and compare these to our concept as well as to its realization in MontiCore.

Xtext [43] is a language development framework that defines DSLs via grammars based on EBNF as well. Based on these, Xtext generates an abstract syntax EMF metamodel. Hence, Xtext performs a direct translation from grammars to metamodels. Moreover, Xtext also produces parsers capable of transforming textual models into EMF-based models [11]. Xtext, however, does not derive constraints capturing the implicit cardinalities imposed by its grammars for the metamodels. Consequently, the generated metamodels support different (more) instances than specified by the grammars. Thus, these constraints must be handcrafted instead.

EMFText is an infrastructure for developing textual modeling languages, which is based on EMF metamodels [19] as well. With EMFText, the concrete and abstract syntax of a language are defined separately. First, the Ecore metamodel is created to specify the abstract syntax of a language. After that, a syntax specification language can be used to specify the concrete syntax with respect to the underlying metamodel. With Ecore metamodels not supporting the intricacies of the syntax specification languages, developers either are restricted to less complex concrete syntaxes or need to address the challenges of implicit cardinalities manually as well.

Rascal [23] is a metaprogramming language for developing modeling languages. Rascal languages also are based on grammars, from which Rascal derives a parse tree data structure. This data structure can be manually or automatically

mapped to a handcrafted abstract syntax tree data structure. The obtained ASTs serve as input for generators, which transform the models into GPL artifacts. There is ongoing work to bridge Rascal and ECore [41], *i.e.,* enabling processing of Ecore models in Rascal and deriving metamodels from Rascal grammars. To the best of our knowledge, there currently is no publication detailing the concepts of Rascal's attempt.

Similar to our concept, the approach presented in [46] suggests parsing grammars and translating their distinct language elements into analog MOF [25] metamodel representation. Here, structural grammar elements, such as sequences, iterations, or optionalities, are also represented in the abstract syntax of the metamodel as annotations. Further optimizations decrease the number of structural classes, if possible, to discard unnecessary overhead. However, in general, there remain structural parts in the metamodel. This yields a complete representation of grammar rules in the abstract syntax of the metamodel, but pollutes the metamodels with annotations. Moreover, the optimizations applied by this approach on the metamodel introduce a conceptual gap between grammar concepts and metamodel realizations.

Another approach to transform grammars into EMF metamodels leverages Xtext as intermediate language representation [2]. Afterwards, Xtext transforms the grammar into an Ecore metamodel. To this end, an EBNF meta-grammar is implemented in Xtext that supports parsing designed EBNF grammars as Xtext models. These models are translated into EMF and then transformed into a corresponding Xtext grammar. Finally, the new grammar serves as input for Xtext and is processed to derive the EMF metamodel. Consequently, this approach is subject to the same challenges as plain translation with Xtext.

XMLText [26] is a framework for constructing textual DSLs for XML-based languages relying on XSD. XMLText combines different approaches of EMF and Xtext. An XSD importer automatically retrieves an Ecore metamodel for a given XML language. Afterwards, the derived metamodel is normalized to overcome conceptual gaps between the XML schema and the corresponding Ecore representation. Finally, XMLText generates an Xtext grammar. Again, this omits enforcing proper constraints for implicit cardinalities.

The Grammar-to-Model Language (Gra2MoL) [20] is a transformation language for bridging grammarware and modelware. It processes textual models conforming to grammars and translates these into model representations conforming to target metamodels. Gra2MoL uses ANTLR [28] to construct a parser that processes textual models and derives their abstract syntax trees. Additionally, the language developer defines model transformations via Gra2MoL's rule-based transformation language. These rules define the bridge between the parsed textual model and the metamodel-conform representation. Finally, the transformations are applied to the syntax tree to retrieve the target model. With Gra2MoL, only instances of grammars and metamodels are bridged, but not their definitions. This prevents reusing tooling between the different technological spaces.

9 Conclusion

We presented a concept for translating EBNF-like grammars into accurate metamodels that employs OCL constraints to specify the implicit cardinalities imposed by grammar rules featuring disjunctions and iterations. At its core, implicit cardinalities of grammar rules are translated into linear equation systems that are solved at modeling time to check validity of models. We also presented a realization of our concept that translates MontiCore grammars into Ecore metamodels with OCL incorporating a solver generated for each metamodel class. Using this realization, we presented a case study translating the grammar of the MontiArc ADL into an accurate Ecore metamodel. Leveraging our transformation concept liberates language developers from manually enforcing that metamodels representing grammars are accurate. This facilitates bridging grammarware and modelware and contributes to successful software language engineering with truly heterogeneous modeling languages.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. 1977. *Principles of Compiler Design.* Addison-Wesley Longman Publishing Co., Inc.

[2] Alexander Bergmayr and Manuel Wimmer. 2013. Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques.. In *MDEBE@ MoDELS*. 22–31.

[3] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd.

[4] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiaro, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design* (2013), 456.

[5] 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.

[6] Bundesministerium für Bildung und Forschung. 2017. Zukunftsprojekt Industrie 4.0. https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html. Accessed: 2017-04-20.

[7] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*. Springer International Publishing, 53–70.

[8] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*. Athens, Greece, 213–218.

[9] Tony Clark, Mark den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*. Springer, 7–20.

[10] Tony Clark and Jos Warmer. 2002. *Object Modeling With the OCL: The Rationale Behind the Object Constraint Language*. Vol. 2263. Springer, Dagstuhl Castle, Germany.

[11] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, Vol. 32. EclipseCon, Santa Clara, CA 95054, 118.

[12] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Fsuture. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[13] Moritz Eysholdt and Heiko Behrens. 2010. Xtext - Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (SPLASH '10)*. ACM, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

[14] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

[15] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.

[16] Sanford Friedenthal, Alan Moore, and Rick Steiner. 2014. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

[17] Object Management Group. 2010. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01).

[18] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. 2015. Composition of Heterogeneous Modeling Languages. In *International Conference on Model-Driven Engineering and Software Development*. Springer, Cham, 45–66.

[19] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. 2013. Model-based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*. Springer, Berlin, Heidelberg, 322–345.

[20] Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, and Jesús Garcıa 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.

[21] Jean-Marc Jézéquel, Manuel Leduc, Olivier Barais, Tanja Mayerhofer, Erwan Bousse, Walter Cazzola, Philippe Collet, Sébastien Mosser, Benoit Combemale, Thomas Degueule, Robert Heinrich, Misha Strittmatter, Jörg Kienzle, Gunter Mussbacher, Matthias Schöttle, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures* 54 (2018), 139 – 155.

[22] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Trans. Softw. Eng. Methodol.* 14, 3 (July 2005), 331–380. https://doi.org/10.1145/1072997.1073000

[23] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 168–177.

[24] David Lechevalier, Seung-Jun Shin, Sudarsan Rachuri, Sebti Foufou, Y Tina Lee, and Abdelaziz Bouras. 2017. Simulating a virtual machining model in an agent-based model for advanced analytics. *Journal of Intelligent Manufacturing* (2017), 1–19.

[25] MOF 2004. OMG. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/ptc/03-10-04.pdf. Accessed: 2018-06-26.

[26] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. 2015. XMLText: From XML Schema to Xtext. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 71–76.

[27] Object Management Group. 2010. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05). http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ [Online; accessed 2015-12-17].

[28] Terence Parr. 2007. *The Definitive ANTLR Reference: Building Domain-Specific Languages* (first ed.). Pragmatic Bookshelf.

[29] Gang Ren, Qingsong Hua, Pan Deng, Chao Yang, and Jianwei Zhang. 2017. A Multi-Perspective Method for Analysis of Cooperative Behaviors Among Industrial Devices of Smart Factory. *IEEE Access* 5 (2017), 10882–10891.

[30] Mark Richters and Martin Gogolla. 1998. On Formalizing the UML Object Constraint Language OCL. In *International Conference on Conceptual Modeling*. Springer, Bremen, Germany, 449–464.

[31] 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).

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

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

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

[35] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA.

[36] Josef Stoer, Friedrich L Bauer, and Roland Bulirsch. 1989. *Numerische Mathematik*. Vol. 8. Springer, Berlin, Heidelberg.

[37] The Industrial Value Chain Initiative. 2018. https://iv-i.org/wp/en/about-us/whatsivi/. Accessed: 2018-06-04.

[38] The U.S. Advanced Manufacturing Initiative. 2018. https://www.nist.gov/sites/default/files/documents/2017/04/28/Molnar_091211.pdf. Accessed: 2018-06-06.

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

[40] Tijs van der Storm. 2011. *The Rascal language workbench*. CWI. Software Engineering.

[41] Tijs Van Der Storm. 2018. Bridging Rascal and EMF. Slides of LangDev Meetup: http://langdevcon.org/slides/RascalEcore.pdf. Accessed: 2018-06-05.

[42] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A Rapid Development of DSM Graphical Editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, Tihany, Hungary, 233–238.

[43] 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.

[44] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

[45] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics* (2017).

[46] Manuel Wimmer and Gerhard Kramler. 2005. Bridging Grammarware and Modelware. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 159–168.

[47] Andreas Wortmann, Benoit Combemale, and Olivier Barais. 2017. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*. IEEE, 281–291.