TU WIEN Informatics

A grammar-driven and decoupled abstract syntax tree for Xtext

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Niklas Mischkulnig
Matrikelnummer 11809607

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.-Ing. Dr.techn Andreas Krall

Wien, 16. März 2022

Niklas Mischkulnig	Andreas Krall

TU WIEN Informatics

A grammar-driven and decoupled abstract syntax tree for Xtext

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Niklas Mischkulnig
Registration Number 11809607

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ.-Prof. Dipl.-Ing. Dr.techn Andreas Krall

Vienna, 16th March, 2022

Niklas Mischkulnig | Andreas Krall

Erklärung zur Verfassung der Arbeit

Niklas Mischkulnig

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. März 2022

Niklas Mischkulnig

Kurzfassung

Xtext ist ein Framework zur Implementierung von Editoren und Compilern für domänenspezifische Sprachen. Es funktioniert ähnlich wie herkömmliche grammatikbasierte Parsergeneratoren, erzeugt aber außerdem aus den Parserregeln der Grammatik automatisch ein Klassenmodell. Dieses Modell ist allerdings nicht sehr flexibel, falls sich der gewünschte abstrakte Syntaxbaum von der konkreten Syntax unterscheidet, insbesondere nachdem die Grammatik zur Auflösung von Konflikten oder Linksrekursionen umgeschrieben werden musste. Normalerweise würde der abstrakte Syntaxbaum und der Code zum Übersetzen getrennt von der Grammatik spezifiziert werden und so zu dupliziertem und schwer wartbarem Code führen. Stattdessen stellen wir einen Ansatz vor, um die Struktur des abstrakte Syntaxbaums direkt in der Grammatik der konkreten Syntax zu deklarieren. Damit die Grammatik dennoch lesbar bleibt, beschreiben Attribute lediglich den Unterschied zum Modell, das von Xtext abgeleitet wurde. Dies ermöglicht sowohl das automatische Erzeugen von Übersetzungscode als auch das Wiederverwenden der Xtext Klassenmodell-Informationen für den Großteil der Klassen. Trotzdem ist es gleichzeitig möglich, eigene Klassen mit zusätzlichen Attributen zu beschreiben und beliebige Berechnungen im Übersetzungscode durchzuführen, inklusive dem Umwandeln eines Knotens in eine Liste von Knoten.

Abstract

Xtext is a framework for implementing editors and compilers for domain-specific languages. It works similar to traditional grammar-based parser generators but also automatically generates a class model from the parser rules in the grammar. However, this model is not very customizable if the intended abstract syntax tree differs from the concrete syntax. This becomes an even bigger concern if the grammar has to be refactored to resolve parsing ambiguities or left recursion. Traditionally the AST and mapping code would be specified separately from the grammar leading to duplicated code that is hard to maintain. Instead, we present a way to declare the structure of the AST directly inside the concrete syntax grammar. To keep the grammar readable, the attributes are specified in relation to the model inferred by Xtext. This allows for the automatic generation of mapping code and also leveraging the Xtext class model information for a majority of the AST nodes while still supporting custom AST classes with additional attributes and mapping code performing arbitrary computations including mapping a node to a list of nodes.

Contents

CHAPTER 1

Introduction

Xtext [20] is an open-source framework for implementing domain-specific languages with the main advantage over traditional parser generators that it can also be used to generate compilers, type checkers, and editors with syntax highlighting and autocompletion. Languages are specified with an EBNF-like grammar that additionally specifies a mapping between consumed tokens and attributes of an abstract syntax tree (with nodes having attributes that are either Java objects or child nodes).

The parse tree as described by the grammar is mostly identical to the resulting abstract syntax tree even though Xtext has some mechanisms to change the structure of the abstract syntax tree (for example to create less deep hierarchies after left recursion was resolved with additional, non-semantic parser rules). But these can still be limiting or hard to maintain because the structure of this syntax tree is similar to the grammar's structure and cannot be changed arbitrarily without also rewriting parts of the grammar (to change the parse tree and therefore the abstract syntax tree).

Decoupling the resulting tree model from the Xtext tree model induced by the parser rules would provide several benefits:

- The existing Xtext mechanism mentioned above can be either too limited or too complex and hard to read (when declaring the intended language syntax as a LL(*) grammar).
- It can also be useful to exclude some unneeded attributes (either to improve memory usage and runtime performance if the removed attributes represent a significant portion of the tree, or to reduce the API surface of the objects passed to user-defined code).
- On the other hand, additional attributes can be computed based on other attributes and information in the subtree (for example by aggregating child nodes into a

different data structure like a hash map). It's also possible to use different list implementations for node lists that are better suited for the specific application (regarding performance or functionality, like more built-in traversal methods).

The traditional solution for building this second syntax tree model (which we from now on call abstract syntax tree/AST, in contrast to Xtext's existing model that is closer to the grammar being the concrete syntax tree/CST) is to manually write classes and mapping code based on the grammar and the desired output structure. The problem with that approach is maintainability: the definitions of the grammar, the mapping code, and the AST classes are split across multiple files, and modifications of the grammar or model require changing multiple files, which is labor-intensive and error-prone.

We developed an approach that retains Xtext's core philosophy of deriving all necessary code from an expressive grammar by extending the grammar with declarations that describe the AST's structure as declaratively as possible, while still allowing custom code snippets for arbitrary computation and aggregation of attributes (as outlined above).

Chapter 2 gives an introduction into Xtext and chapter 3 lists related work. In chapter 4, we describe the Xtext grammar extensions including their benefits regarding flexibility but also their limitations. Chapter 5 describes how this extension was implemented in the Xtext system.

CHAPTER 2

Basics of Xtext

Internally, Xtext uses ANTLR [1] as a top down LL(*) parser by translating the Xtext grammar into an ANTLR grammar. Parser rules have a head and a body, the head contains the name and optionally more information about the CST structure. The body contains the pattern that should be matched by the rule, where literal tokens are specified with quotes (and are otherwise treated as a call of another rule).

2.1 Nodes and Attributes

The main difference between Xtext grammars and (pure) ANTLR grammars is that tokens can be stored into attributes in the CST using assignments (these named properties are called *features* in Xtext, regardless of whether they are child nodes or attributes). This grammar

```
VariableDeclaration:
  'var' name=('a'..'z') ';';
```

generates the following CST interface class (together with a corresponding implementation class). `EObject` is the supertype of all CST classes and provides various getters to access cross references and features by name.

```
public interface VariableDeclaration extends EObject {
  String getName();
  void setName(String value);
}
```

During parsing, an object for the CST node is created when the first assignment in the rule is encountered. By default, this is an instance of the CST class corresponding to the parser rule, containing attributes whose names and types are inferred from the rule body (and potentially other rules, recursively).

These attribute assignments should be treated as wrapping the specified subpattern, so for example to make such an attribute optional in the parser, it should be made optional as a whole: `(value=ID)?`. In addition to the `x=...` syntax, it is also possible to store a list of values or child nodes in an attribute. Every time the right hand side of the assignment `x+=...` is consumed, the resulting object is added to the attribute's value (a list):

```
Program:
  (declaration+=VariableDeclaration)*;
```

Xtext's type inference creates classes with all possible attributes that could be assigned (in all alternatives, optional clauses etc. of the rule - even if some of these attributes are actually mutually exclusive at runtime). Unassigned rule calls can be used to introduce a class hierarchy, particularly when storing the resulting node as an attribute. Without introducing the `Statement` parser rule in this example, the attribute `stmt` would have the declared type `EObject` instead of some supertype representing statements.

```
Program:
  stmt=Statement;
Statement:
    CallStatement | AssignmentStatement;
```

Common attributes (with the same name and type) that occur in all alternatives are hoisted into the parent type.

A similar effect can also be achieved with `returns`. The main difference is that `returns` doesn't create a hierarchy but instead creates a type with a union of the attributes of the usage (having two parser rules with attributes with the same name and different types is forbidden). In this case, there are only two classes: `Declaration` with the attributes `name` and `init` (even though `init` is only used in one rule), and `Program`. So `returns` usually leads to less type-safe class models.

```
Program:
  body=(VariableDeclaration|FunctionDeclaration);

VariableDeclaration returns Declaration:
  'var' name=ID "=" init=INT ';' ;

FunctionDeclaration returns Declaration:
  'function' name=ID '()' '{' '}' ;
```

As a variant of regular assignments with rule calls, Xtext supports cross references which are treated as identifiers that reference another node of the given type. This is a very common situation in programming languages that is handled automatically by Xtext without needing any additional custom code. `[Entity]` in the following example is treated as an `ID` during parsing but is afterwards set to the entity node which has a name corresponding to the parsed identifier. (This step is called *linking* and also described in section 2.3).

```
Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        ...
    '}';
```

2.2 Structural Rewriting

This alone would result in a language model that is tightly coupled with the grammar's structure, so Xtext already has multiple ways of decoupling rules and model structure:

Rules can create different node types in alternative clauses (`MyRule` becomes the supertype of `TypeA` and `TypeB`) using simple actions (so this is similar to an unassigned rule call with explicit alternative rules as described above):

```
MyRule:
    "A" {TypeA} name=ID |
    "B" {TypeB} name=ID;
```

Additional features can be added to a node that was produced by a rule call (so first, the rule `Token` is applied and then `cardinality` is optionally set on the node created by the rule call):

```
AbstractToken:
    Token (cardinality=('?'|'+'|'*'))?;
```

Assigned actions can be used to wrap the current node in another node, for example to ensure that rules for binary operations create simpler hierarchies and don't unnecessarily wrap a single literal with an otherwise empty operation node:

```
Expression:
    TerminalExpression (
        {Operation.left=current}
        op='+' right=TerminalExpression
    )*;
```

First, `TerminalExpression` is called and the current node is a terminal expression node. When the optional part in the parenthesis is matched, a new (empty) `Operation` node becomes the current node, with the attribute `left` set to the terminal expression node from before. After that, `op` and `right` are set.

Note that the places where node types are referenced often also serve as the implicit definition of the type. The `Operation` type in the last example is automatically inferred (that it is a subtype of `Expression` and has the three attributes `Expression left`, `String op` and `TerminalExpression right`), as is `TypeA` and `TypeB` above.

2.3 Compilation and Runtime

When authoring a language using Xtext, the build process consists not only of compiling Java code (i.e. for code generators, error message providers or validators) but also generating code programmatically: the Xtext grammar is first translated into an ANTLR grammar and then processed by ANTLR to produce Java code for parsing the described language. The accompanying Java classes for the CST are also generated based on the inferred types together with a class called *grammar access* containing getters for accessing these CST classes and also grammar information from the parser code. (There are more generated classes but these are the most relevant and important.)

At runtime (e.g. when using an Eclipse instance with a plugin enabling use of that language or when using the parser standalone from plain Java code), the following steps are performed in order:

1. **Lexing/Parsing** The input code is processed by the generated parser and the callbacks inserted by Xtext into the ANTLR grammar construct the CST nodes as described in the preceding sections.

2. **Linking** Now that the whole program is parsed, cross references are resolved (this has to be deferred to allow forward references) and the attributes in the CST nodes are set.

3. **Validation** Validators can generate errors and warnings after analyzing the parse result. (This step is not strictly relevant for this work.)

CHAPTER 3

Related Work

In the broader context of how to describe concrete syntax, the resulting abstract syntax tree and the corresponding mapping, there are at least two opposing approaches with the first one being grammar-based (just like Xtext):

The Program System Generator [17] uses a DSL to specify the AST structure using sum and product types (without field names) together with a grammar. It also acknowledges how left factoring can cause a divergence between concrete syntax and the AST which can be solved using special grammar rules similar to Xtext's tree rewrite actions. [15] also has a similar syntax though it only describes the mapping and not the AST itself.

[19] describes a „heuristic“ to generate an AST model from an EBNF-style grammar with the downside of generating non-mnemonic names for the attributes of the nodes. The algorithm proposed in [10] takes the same approach while allowing to mark parts of a parser rule that should dropped in the AST (such as keywords or punctuation) but also suffers from the same problem regarding attribute names. [12] describes one of the first of such algorithms and focuses on producing a tree model with as few unnecessary or duplicate nodes as possible.

TCSSL [4] is a specification language which is semantically very similar to a Xtext grammar (and also using ANTLR for a prototype implementation) but doesn't have tree rewrite actions to accommodate left factoring in the concrete syntax.

The MontiCore grammar language [9] is even more similar to Xtext grammars in that it also allows describing the subtyping relation between the inferred AST structure. An additional feature particularly relevant to our work is that it additional Java methods for the AST classes can be specified directly in the grammar.

One problem with Xtext's and MontiCore's class models is that they cover a superset of the instances that can be produced by the grammar itself. This can be improved by generating AST classes with additional invariants for cardinality and mutual exclusivity [2].

[13] covers the very same concepts also used by the Xtext grammar (omitting and renaming features, boolean features, subtyping relationship between rules) with a merely theoretical grammar syntax.

Both [7] and [11] describe a grammar for a bidirectional mapping between a concrete syntax and an existing metamodel for the AST (and therefore doesn't cover how such a metamodel should be derived or described). Both papers use metamodels described using the KM3 DSL [6] which has a syntax similar to Java classes with fields but also allows specifying graph related properties such as cardinality and how two fields in parent and child nodes refer to the same relationship. It also differentiates between attributes, referenced nodes and contained nodes explicit. Another such DSL that describes the AST's structure is Zephyr [18] which is less verbose than KM3 but cannot model as many constraints.

A fundamentally different approach is viewing the AST and CST as a reactive system where updates should be applied bidirectionally [16]. Their main contribution is a system of defining triggers that listen to changes, while the code performing the update still has to be written by hand (though this does allow for an arbitrarily complex relationship between the two models).

On the other end of the spectrum are tools that use manually written AST classes with annotations describing the mapping to the concrete syntax [14]. This provides a very natural way to add custom functionality to the AST nodes (specific to the desired use case such as a `eval` method for computing expressions or even abstract interpretation). [3] additionally explores how this approach can be used to compose multiple such language definitions and their rules.

The question whether the abstract or concrete syntax is more important for the language itself and how they should be developed in parallel is discussed in [8], while [5] explores how grammar-based language construction compares to meta-modelling regarding to the available toolchains and how some selected languages are specified.

CHAPTER 4

Extending the Xtext Grammar

There are two parts that need to be generated from the extended grammar: the AST classes themselves with their attributes and child nodes, and code to generate this new AST. These grammar additions should be as concise as possible to not overload the grammar file and keep it readable (because a single file containing the grammar, complete AST class declarations and the full mapping code would be just as unmaintainable as having these parts in separate files).

Xtext doesn't require explicitly declaring the node attributes for its CST because it infers the node structure from the parser rules based on the assignments and rule calls used in the rule body. So to prevent redundancy in the grammar, only the difference between the Xtext CST structure and the intended AST structure has to be specified by the user. For most cases, no manual Java mapping code has to be provided, but this is still always possible for more complex situations (e.g. when custom attributes are added that aren't copied from the CST).

As a consequence, the CST node objects are converted into AST nodes after Xtext has parsed the input into the CST tree instead of directly creating AST nodes in the parser (though this approach also causes some limitations, as described in section 4.6). For this reason, the additional declaration that appear to be part of a parser rule actually belong to the node class that this parser rule creates at runtime, which can lead to problems when a node class doesn't correspond to a unique parser rule (see subsection 4.6.1).

4.1 Basic Usage

In the base case when a CST node structure should be carried over into the AST unmodified (with the same attributes and child nodes), only `becomes` has to be added after the rule head:

```
Program becomes:
  (imports+=ImportDeclaration)*;

ImportDeclaration becomes:
  "import" source=STRING ";";
```

This generates the Java classes `ASTProgram` and `ASTImportDeclaration`, and two methods to convert each CST node type into the corresponding AST node. Child nodes are recursively translated into the corresponding AST type (like the contents of `imports` in `Program`), both in the class declarations (so `Program#imports` is now a `List<ASTImportDeclaration>` instead of `List<ImportDeclaration>`) and the mapping code (which first maps all child nodes and then the node itself in a depth-first traversal).

The generated AST classes are similar to C structs, with public fields and no method (the mapping code and how it is invoked is described in chapter 5):

```
public class ASTProgram {
  public List<ASTImportDeclaration> imports;
}
public class ASTImportDeclaration {
  public String source;
}
```

The classes are stored in the `ast` subpackage (next to the `validator` and `serializer` packages in the Xtext project template), similar to the Xtext CST classes.

At runtime, there is a new `doGenerateAST` method on generators, where the final AST is passed and can be used at runtime (similar to how the existing `doGenerate` method is used to access the CST):

```
class MyDslGenerator extends AbstractGenerator {
  void doGenerateAST(Object input, IFileSystemAccess2 fsa,
                                IGeneratorContext context) {
    ASTProgram ast = (ASTProgram) input;
    // ...
  }
}
```

Attributes and child nodes can be added or removed by specifying them after `becomes`. When listing just the name, the CST feature is copied over as before (`becomes` without parenthesis is equivalent to `becomes (a, b, c)` with `a, b, c` being a list of all features):

```
ImportDeclaration becomes (symbol):
  "import" symbol=ID "from" source=STRING ";";
```

All features that aren't listed are discarded during the mapping, so to keep for example all but one feature, all other features need to be explicitly listed.

Xtext also supports enum rules, which are translated into Java classes as well:

```
enum ChangeKind becomes:
  ADD='add' | MOVE='move' |
  REMOVE='remove';
```

becomes a Java enum `ASTChangeKind` with the values `ADD`, `MOVE` and `REMOVE`. Xtext's behaviour of initializing uninitialized enum features to the first value is also respected. Enum rules don't allow further customization regarding the AST conversion, in contrast to regular parser rules as described in the next section.

4.2 Custom Mapping

To add an additional, custom attribute to an AST node, a type is specified in addition to the name (just as in a Java method declaration). In this case, the intended mapped value cannot (and should not) be inferred, so to assign a value, a Java code snippet (enclosed in `$$`) should be provided after listing the AST features:

```
ImportDeclaration becomes (symbol, boolean isRemote) $$
  this.isRemote = node.source.startsWith("http");
$$:
  "import" symbol=ID "from" source=STRING ";";
```

The Java code is copied into a constructor in the mapping code (so only statements are allowed and not class declarations, etc.) after generated assignments for copied attributes. The available variables for reading from CST node and assigning values to the AST node are:

`this` The new AST object (containing the public fields declared in the `becomes` declaration).

`node` the Xtext CST node, which it intended to be used to read attributes from the node, like strings and numbers (getters have to be used to read feature values, in contrast to the AST node).

`children` An object containing the converted child AST nodes as public fields. This is a separate object because `node` contains the child CST nodes, not the previously converted AST nodes.

This approach enables arbitrary computations to be performed either based on attributes (via `node.getX()`) or child nodes (using the converted AST nodes in `children` or the original CST `node`), such as normalizing optional attributes based on other attributes. In this example, the `local` attribute defaults to the value of `name`:

```
Import becomes $$
  if(this.local == null) this.local = this.name;
$$:
  'import' name=ID ('as' local=ID)? 'from' source=STRING;
```

The Java code is concatenated into a method, meaning imports cannot be inserted this way. To simplify the usage of AST classes (classes in the `ast` subpackage), they can be referred to with double angle brackets « and » (inspired by Xtend[1] template expressions) in the Java code snippet or the feature declaration. Other classes (e.g. from the Java Class Library) should be used via the fully qualified name (also illustrated by the following two examples). This was done to prevent naming conflicts, as opposed to inserting `import myDsl.ast.*;`.

One application is aggregating child nodes into a different data structure (such as a hash map) by iterating over the child nodes and adding the values into a hash map stored as a feature. In this case, the actual child nodes aren't needed anymore and therefore not included in the AST (because only `data` is listed in the declaration):

```
Map becomes (java.util.HashMap<String, String> data) $$
  this.data = new java.util.HashMap<String, String>();
  for(«ASTMapEntry» entry: children.entries) {
    this.data.put(entry.key, entry.value);
  }
$$:
  '{' entries+=MapEntry '}';

MapEntry becomes:
  key=STRING ":" value=STRING;
```

Another common case is using a different list implementation for child nodes (to use a more efficient list implementation or have specific helper methods available). This could be achieved by declaring a feature and then assigning the desired list type in a code snippet:

```
Program becomes («NodeList»<«ASTImportDeclaration»> imports) $$
  imports = new «NodeList»<«ASTImportDeclaration»>(imports);
$$:
  (imports+=ImportDeclaration)*;
```

This is however rather verbose and repetitive, which is why there is a a shorthand for this constructor call (this can also be used for any type and not just lists, provided that a suitable constructor exists) - adding an ampersand before the name:

[1]Xtend is a language developed as part of the Xtext project (and implemented in Xtext) with a more functional syntax and type-inference.

```
Program becomes («NodeList»<«ASTDeclaration»> &imports):
  (imports+=ImportDeclaration)*;
```

4.3 Custom Classes

While automatically generated AST classes are very concise and easy to maintain, it can still be necessary to manually create specific AST classes to add methods or extend super types. By specifying a class name in the rule, only mapping code and no class is generated:

```
ImportDeclaration becomes ASTSomeManualClass(symbol, source):
  "import" symbol=ID "from" source=STRING ";";
```

With a custom class, the features listed after the class name (`symbol` and `source`) are also copied (so this is a shorthand for a code snippet containing `this.symbol = ...` assignments). The syntax using both type and name (`String symbol`) doesn't make sense in this case because the type is only needed to generate the class definition.

The referenced classes need to be provided in the `ast` (the same package as the automatically generated classes) and need to have an accessible default constructor so that the mapping code can instantiate an object and then assign the features (and run any provided Java snippet.)

4.4 Turning a Node Into a List

There are situations where a list of elements is modeled as a single node containing a list of child nodes (e.g. to have the intended parser precedence or a readable grammar) instead of a list without an intermediate node.

By adding square brackets `[]` after `becomes`, the code snippet of the node is run with `this` being an `ArrayList` of the AST node and all calls of the corresponding node are also replaced with a list of nodes.

This example creates an `ASTMap` class with the field `List<ASTMapEntry> entries;` and an `ASTMapEntry` class with the field `String x` and `ASTValue y`

```
Map becomes:
  '{' entries=MapEntry '}';

MapEntry becomes[] (String x, int y) $$
  for(String key: node.getKeys()) {
    this.add(new «ASTMapEntry»(key, node.getValue()));
  }
$$:
  "[" keys+=STRING ("," keys+=STRING)* "]" ":" value=INT;
```

Without any additional logic, the example above would break with `entries+=MapEntry` because a `List<ASTNode>` would be added to a `List<ASTNode>`. For this reason, if an AST node is mapped to a list but the node is already contained in a list, the new list is merged into the parent list.

This mechanism can be used to flatten lists of nodes containing lists of features into a list of simpler nodes. For example to flatten lists of variable declarations containing multiple names each (assuming that `var a, b;` and `var a; var b;` are semantically equivalent). In this example, `FunctionDeclaration` nodes are mapped individually but `VariableDeclaration` nodes are mapped to a list that is then merged into the `body` node list of the parent `Program` node.

```
Program becomes:
  {Program} '{' (body+=Declaration)* '}';

Declaration becomes:
  VariableDeclaration | FunctionDeclaration;

VariableDeclaration becomes[] (String name) $$
  for(String n: node.getName()) {
    this.add(new «ASTVariableDeclaration»(n));
  }
$$:
  'var' name+=ID ("," name+=ID)* ';';

FunctionDeclaration becomes:
  'function' name=ID '()' '{'
    /* ... */
  '}';
```

Similarly to the non-list conversion, the node class and the list implementation can also be changed by specifying the list class inside the square brackets and specifying a node class name instead of a list of features:

```
MapEntry becomes[«NodeList»<«ASTMapEntry»>] ASTMapEntry $$
  ...
$$
```

In this case, the node name is only used to change nodes containing a `MapEntry` to instead contain a `NodeList<ASTMapEntry>` - the node objects themselves need to be created manually in the code snippet.

4.5 Full Grammar Additions

This is an excerpt of Xtext grammar amended with the described functionality. The first two rules for parser and enum rules are amended with the `becomes` clause, all other rules didn't previously exist.

```
// the existing Xtext rule for enum rules
EnumRule:
  /* ... */
  (becomes?='becomes')?
  /* ... */
;

// the existing Xtext rule for parser rules
ParserRule:
  /* ... */
  ('becomes' becomes=BecomesDecl)?
  /* ... */
;

BecomesDecl returns BecomesDecl:
  (list?="["
    (listType=JavaTypeReference)?
  "]")?
  descriptor=(BecomesDeclGeneratedClass|BecomesDeclManualClass)
  (code=JAVA_STRING)?
;

JavaTypeReference returns ecore::EString:
  (('«' ID '»') | ID)
  ('<' (('«' ID '»') | ID) '>')?
;

terminal JAVA_STRING returns ecore::EString: '$$' -> '$$';

BecomesDeclGeneratedClass:
  {BecomesDeclGeneratedClass} (
  '('
    attributes+=(BecomesDeclCopyAttribute | BecomesDeclCustomAttribute)
    (','
      attributes+=(BecomesDeclCopyAttribute | BecomesDeclCustomAttribute)
    )*
  ')'
  )?
;

BecomesDeclManualClass:
  type=ID
```

```
    (
    '('
      attributes+=(BecomesDeclCopyAttribute)
      (',' attributes+=BecomesDeclCopyAttribute)*
    ')'
    )?
;

BecomesDeclCopyAttribute: name=JavaTypeReference;

BecomesDeclCustomAttribute: type=JavaTypeReference (copy?='&')? name=ID;
```

4.6 Limitations

The described system also has some limitations: some of them are fundamental problems or tradeoffs that were made, others are arbitrary features that are not implemented but could be added in the future.

4.6.1 Correlation between parser rules and CST nodes

Some Xtext-specific grammar constructs produce a CST class hierarchy with no exact mapping between CST classes and the original parser rules, so declaring how rules map to the AST via this non-unique CST-node-to-rule lookup doesn't work in these situations.

When using `returns` as in the following example, the rule creates a CST node of the type `X` and no `A` or `B` CST class is generated (same goes for the generated AST class hierarchy). This means that any mapping code specified for these two rules is never used at runtime because the node cannot be matched to its originating rule; the non-uniqueness is why using the type specified by `returns` to perform this matching isn't possible.

```
A returns X becomes (String value) $$ ... $$:
  'a' value=ID;
B returns X becomes (String value) $$ ... $$:
  'b' value=ID;
```

A very similar situation can be constructed using simple actions, where the rule becomes an (empty) interface extended by the type specified in the simple action (same goes for the AST class hierarchy). For the same reasons as before, custom mapping code cannot be used here.

```
ActionNew1 becomes $$ ... $$: {ActionNewImpl} 'b' value=ID;
ActionNew2 becomes $$ ... $$: {ActionNewImpl} 'b' value=ID;
```

Furthermore, simple actions and alternatives make it possible to have multiple branches in the rule body while only being able to specify the mapping code for all of them in the rule head.

```
ActionNew becomes $$
  this.value = "prefix: " + node.getValue()
$$: {ActionNewX} 'b' value=ID | 'c' value=ID;
```

In both of these examples, the usage of simple actions with a different type than the rule name is statically detected and causes a build error. Instead, the alternatives should be split into multiple rules:

```
ActionNew becomes: ActionNewX | ActionNew2;
ActionNew2 becomes $$ this.value = "prefix: " + node.getValue() $$:
  'c' value=ID;
ActionNewX becomes $$ this.value = "prefix: " + node.getValue() $$:
  'b' value=ID;
```

4.6.2 Turning a List into a Node

There are four possible cases when converting between CST and AST nodes. The two simple cases are the ones where the type doesn't change: a CST nodes becomes an AST node, or a list of CST nodes becomes a list of AST nodes.

When the type does change however, the type of the attribute that contains the converted node has to be taken into account as well. So when converting a CST node into an AST node list where the containing attribute is already a list, the result should be merged into the list (as described in section 4.4).

	To Node	To List
From Node	✓	✓
From List		✓

Table 4.1: Matrix of supported CST-AST conversions between nodes or lists of nodes.

The last case is converting (reducing) a list of CST nodes into a single AST node, but the CST node could also appear in the grammar as a (non list) attribute in itself. So in that sense, the reduction actually happens in the container of the nodes rather than the contained nodes. This perspective also leads to an implementation of such a mechanism without any special syntax: by accessing the child nodes in a code snippet on the parent and storing it in a custom attribute.

4.6.3 Correlation between CST and AST nodes

The AST is not suited for performing formatting/linting with autofixes or validation because the source information is lost, so an error cannot be correlated to the original position in the source file. But furthermore there is no 1:1 correlation between CST and AST meaning the translation isn't reversible, so hand written logic to generate source code from the AST would be required (and since the source positions aren't known, it would also not retain the original formatting or comments).

4.6.4 Interplay with other Xtext features

Xtext performs linking as a separate step after parsing. Because the AST conversion happens directly after parsing, cross references aren't populated (yet) and are instead always `null`.

When using AST conversion, extending arbitrary grammars isn't very useful because you cannot modify the rules in the extended grammar without duplicating them (to add a `becomes` declaration).

Xtext allows using Java keywords as attribute names because the CST classes always use getters and setters, so for example naming collisions with Java keywords (e.g. `final` → `getFinal()`) aren't a problem. Automatically generated AST classes use public fields however, so the user is responsible to choose valid names to avoid a compile time error.

4.6.5 Expressiveness of class descriptors and code snippets

The current syntax for specifying custom fields of automatically generated AST classes could be augmented to allow declaring superclasses and interfaces, this would be especially useful for marker interfaces like `Serializable`.

Both the class field declarations and the code snippet don't allow importing arbitrary Java types (only the usual classes from `java.lang` that are always in scope, and AST classes via «`...`» are easy to use), so these always have to be referenced with their fully qualified name.

CHAPTER 5

Implementation

The implementation has two independent parts, one being executed at build time and the second part being executed at runtime (as described in section 2.3). As much work as possible is performed during build time for better runtime performance (because classes and conversion methods were already generated from the grammar, they are just invoked).

5.1 Generating classes and methods from the grammar

At build time, the grammar is traversed to generate AST classes (if the grammar uses them) and methods for the CST-AST-conversion of a single node. The CST class structure that was already inferred by Xtext at this stage is also used for this.

`org.eclipse.xtext.xtext.generator.StandardLanguage` lists all modules that are invoked by default to generate code at build time (such as `XtextAntlrGeneratorFragment2` which generates the ANTLR parser Java classes). A new class `ASTClassesFragment2` was added to `StandardLanguage` which iterates over all classes in the inferred CST classes and tries to map the class back to the corresponding grammar rule.

In the general case, if there is a rule with a `becomes` declaration, then an AST class or conversion code should be created. However, if there is no corresponding rule for a specific CST node class, then the code still has to be generated (even if it might remain unused at runtime). Because this can break backwards compatibility for existing grammars (for these, the generated code shouldn't change), these unmatched rules are ignored unless the first (entry) rule has a `becomes` declaration.

Unassigning rules (so rules of the form `Op: A|B|C;` without assignments) become empty interfaces in the AST hierarchy, and superclass/-interface relations are also carried over from the CST hierarchy. All other CST node classes directly correspond to an AST node class.

Attributes containing child nodes are rewritten to contain their AST equivalent instead (and potentially wrapped in a `List<...>`) if they should be included in the generated class, while other attributes (such as primitives or objects like a string) always have the same type.

Attributes of superclasses of the CST node are only added to a class if that superclass was made an interface in the AST. This flattens the attributes as much as possible to prevent conflicts with the declared attributes in the `becomes` declaration (otherwise the class could have more attributes than declared) while still working correctly (without duplicated attributes in superclasses) with simple actions (the problem was described in subsection 4.6.1).

Enum rules become regular Java enum classes, without any fields or any of the Xtext-specific extensions in the CST classes (this rewriting does not require any of the logic described above). The value order is retained to also ensure that the default value of enum attributes is the same as in the CST (uninitialized enum attributes in Xtext default to the first value of the enum).

The methods that convert CST nodes to AST nodes are generated in a similar way by adding them to the existing generated grammar access class (which is being used to access the grammar from the ANTLR parser). This class is generated by `GrammarAccessFragment2` which now also calls out to `ASTConversionFragment2`.

The inferred CST classes are once again traversed and mapped back to the rules (where possible), and for each of them a conversion method and a helper class is generated. This class is the type of the `children` variable as described in section 4.2, which stores the converted children of the CST node. The generated conversion method then takes a node of the corresponding CST class together with this children object and returns the resulting AST node.

The method body contains an assignment for each attribute (if declared as such in the grammar) of the form `this.op = children.op;` for child nodes and `this.op = node.getOp();` for other attributes. Custom mapping code (if specified in the grammar) is appended afterwards.

To be able to use `this` in the custom mapping code as though it were a constructor, the conversion method instantiates an anonymous subclass of the AST class and inserts the code in a method of that class.

5.2 Converting the syntax tree

In the `AbstractInternalAntlrParser` class, the CST tree is converted into the AST tree after every successful parsing. Notably, this is done immediately after parsing, so also before Xtext's linking and validation stages.

The new `convertAST` method performs the recursive (so depth-first left-to-right) conversion: it first retrieves handles to the generated conversion method and children class via reflection. Each child node of the current CST node mapped through a recursive `convertAST` call and set as a field on an instance of the children class. If a child node attribute is a list, each element of that list is converted separately. In that case, if the conversion turns a child node into a list, the list is merged into the parent list (performing the merging as described in 4.4).

Finally, the conversion method is invoked with the current CST node and the populated children object, and the return value is returned from `convertAST`.

CHAPTER 6

Conclusion

The approach described in this work tries to achieve a balance to enable a more customizable abstract syntax tree while still using the grammar as the single source of truth without overloading it with too much (unnecessary) information. The core idea is that the additional syntax in the grammar gets progressively more verbose and expressive the more the abstract syntax differs from Xtext's inferred concrete syntax tree. Compared to the existing work for specifying ASTs and their mappings, this makes language definitions whose syntaxes differ in only a few key places easier to maintain and unterstand.

A fundamental shortcoming of supporting custom mapping code and being able to remove unneeded features during the CST-to-AST-conversion is that a bidirectional mapping isn't possible anymore (because the custom mapping code isn't reversible and/or there isn't enough information left). This means that for refactoring or pretty-printing, printing the AST as code would have to be implemented manually (which for many use cases defies the reasoning of using a framework like Xtext in the first place). A similar problem of the current implementation itself (which could be future improvement) is that source location information isn't copied from the CST to the AST (unless done manually for each parser rule).

These two problems currently restrict the usefulness to code generators (which output some different language, be it machine code or a source-to-source translation) and analyzers such as abstract interpreters which don't usually require source information to communicate warnings and errors to the user (unlike a type-checker or linter).

Another idea for future improvement (though it is yet unclear how often this situation occurs in practice) is an easier way of specifying super-interfaces and super-classes for the automatically generated AST classes (for example to inherit some basic functionality or to implement marker interfaces such as `Serializable`).

This thesis is available at `https://github.com/mischnic/bachelors-thesis`, the implementation is available at `https://github.com/mischnic/xtext-core/tree/nmischkulnig/cst-ast`.

Bibliography

[1] *ANTLR parser generator.* URL: `https://www.antlr.org/`. (accessed: 20.01.2022).

[2] Arvid Butting et al. „Translating Grammars to Accurate Metamodels". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering.* SLE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 174–186. ISBN: 9781450360296. DOI: `10.1145/3276604.3276605`.

[3] Sergej Chodarev et al. „Abstract syntax driven approach for language composition". In: *Open Computer Science* 4.3 (2014), pp. 107–117. DOI: `doi:10.2478/s13537-014-0211-8`.

[4] Frédéric Fondement et al. *Metamodel-aware textual concrete syntax specification.* Tech. rep. 2006.

[5] Terje Gjøsæter, Andreas Prinz, and Markus Scheidgen. „Meta-model or Grammar? Methods and Tools for the Formal Definition of Languages". In: *Nordic Workshop on Model Driven Engineering (NW-MoDE 2008).* University of Iceland Press, 2008, pp. 67–82.

[6] Frédéric Jouault and Jean Bézivin. „KM3: A DSL for Metamodel Specification". In: *Formal Methods for Open Object-Based Distributed Systems.* Ed. by Roberto Gorrieri and Heike Wehrheim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 171–185. ISBN: 978-3-540-34895-5. DOI: `10.1007/11768869_14`.

[7] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. „TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering.* GPCE '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 249–254. ISBN: 1595932372. DOI: `10.1145/1173706.1173744`.

[8] Anneke Kleppe. „A language description is more than a metamodel". In: *Fourth international workshop on software language engineering.* Vol. 1. 2007, pp. 1–4.

[9] Holger Krahn, Bernhard Rumpe, and Steven Völkel. „Integrated Definition of Abstract and Concrete Syntax for Textual Languages". In: *Model Driven Engineering Languages and Systems.* Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 286–300. ISBN: 978-3-540-75209-7.

[10] Andreas Kunert. „Semi-automatic Generation of Metamodels and Models From Grammars and Programs“. In: *Electronic Notes in Theoretical Computer Science* 211 (2008). Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), pp. 111–119. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.034.

[11] Pierre-Alain Muller et al. „Model-driven analysis and synthesis of textual concrete syntax“. In: *Software & Systems Modeling* 7.4 (2008), pp. 423–441. DOI: 10.1007/s10270-008-0088-x.

[12] Robert E Noonan. „An algorithm for generating abstract syntax trees“. In: *Computer Languages* 10.3-4 (1985), pp. 225–236.

[13] Jeffrey L. Overbey and Ralph E. Johnson. „Generating Rewritable Abstract Syntax Trees“. In: *Software Language Engineering*. Ed. by Dragan Gašević, Ralf Lämmel, and Eric Van Wyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 114–133. ISBN: 978-3-642-00434-6. DOI: 10.1007/978-3-642-00434-6_8.

[14] Jaroslav Porubän, Michal Forgáč, and Miroslav Sabo. „Annotation based parser generator“. In: *2009 International Multiconference on Computer Science and Information Technology*. 2009, pp. 707–714. DOI: 10.1109/IMCSIT.2009.5352763.

[15] Luis Quesada, Fernando Berzal, and Juan-Carlos Cubero. „A Domain-Specific Language for Abstract Syntax Model to Concrete Syntax Model Mappings“. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2014. Lisbon, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2014, pp. 158–165. ISBN: 9789897580079. DOI: 10.5220/0004671701580165.

[16] István Ráth, András Ökrös, and Dániel Varró. „Synchronization of abstract and concrete syntax in domain-specific modeling languages“. In: *Software & Systems Modeling* 9.4 (2010), pp. 453–471. DOI: 10.1007/s10270-009-0122-7.

[17] G. Snelting. „Experiences with the PSG — Programming System Generator“. In: *Formal Methods and Software Development*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 148–162. ISBN: 978-3-540-39307-8.

[18] Daniel C. Wang et al. „The Zephyr Abstract Syntax Description Language“. In: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. DSL'97. Santa Barbara, California: USENIX Association, 1997, pp. 213–228.

[19] David S. Wile. „Abstract Syntax from Concrete Syntax“. In: *Proceedings of the 19th International Conference on Software Engineering*. ICSE '97. Boston, Massachusetts, USA: Association for Computing Machinery, 1997, pp. 472–480. ISBN: 0897919149. DOI: 10.1145/253228.253388.

[20] *Xtext framework*. URL: https://www.eclipse.org/Xtext/. (accessed: 20.01.2022).