AST Matching based on Concrete Syntax Patterns: Exploration of the Specification Challenges

Arjan J. Mooij[1,2,3], Piërre van de Laar[1]

[1]TNO-ESI, Eindhoven, The Netherlands
[2]Delft University of Technology, Delft, The Netherlands
[3]Zürich University of Applied Sciences, Winterthur, Switzerland

Abstract

Software analysis often relies on pattern matching in terms of Abstract Syntax Trees (ASTs), but AST patterns are known to be tedious to specify. Concrete syntax patterns with placeholders have been proposed as a user-friendly alternative. Several designs for this proposal have been implemented, but these typically focus on specific parsing technologies. In this paper we explore the overarching challenges of specifying AST patterns using concrete syntax with placeholders.

Using our experience with industrial applications, we take the perspective of an analyst who creates concrete syntax patterns to find matches in a code base. We identify two specification challenges: (1) understanding the underlying AST structure, and (2) ambiguities caused by placeholders. For designs based on black-box parsers we also inventorize the challenge of (3) encoding and recognizing the placeholders in concrete syntax patterns. We illustrate these challenges with examples in the Ada and C/C++ programming languages. Our results can serve as warnings to users of concrete syntax patterns, as additional requirements for parser front-ends, as attention points for language specifications, and as starting points for further research on pattern matching.

Keywords

Abstract Syntax Tree, pattern matching, concrete syntax patterns

1. Introduction

Pattern matching on Abstract Syntax Trees (ASTs) is used widely for analyzing software [1], and for defining operational semantics [2]. In practice, when specifying code patterns in terms of abstract syntax, it is tedious to specify all the nodes and attributes: it quickly becomes unreadable, and you easily forget to specify that certain nodes should be absent and certain attributes should have default values. Similar observations are made in [1]. Specifications in terms of abstract syntax have a steep learning curve that easily distracts from the analysis. This is illustrated by the example in Fig. 1.

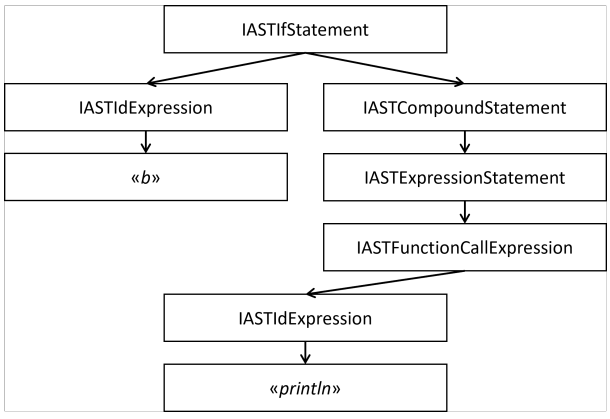

Figure 1: Abstract Syntax Tree for the following C++ pattern in concrete syntax: `if (b) { println(); }`

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
arjan.mooij@zhaw.ch (A. J. Mooij); pierre.vandelaar@tno.nl (P. van de Laar)
0009-0005-9566-7696 (A. J. Mooij); 0009-0008-6113-7672 (P. van de Laar)

As explained in [3], "*the abstract syntax must be provided behind-the-scenes to enable model management and manipulation*", whereas "*the concrete syntax must be designed so that end-users have a familiar and accessible syntax*". Parsers first produce a concrete syntax tree, and afterwards reduce it to an abstract syntax tree by eliminating irrelevant information such as whitespaces and comments. The mapping from concrete to abstract syntax may be complex.

The idea of "native patterns" [4] is that code patterns should resemble normal code in the object language. The object language is the language of the code being analyzed, whereas the meta language describes other aspects of the analysis. Ideally a user of concrete syntax patterns would not need to understand the underlying AST.

We study AST matching based on concrete syntax patterns. The ASTs are matched structurally, but the AST patterns are described using concrete syntax. Some pattern fragments, each corresponding to adjacent AST nodes/attributes, can be replaced by meta-language placeholders. As a convention, similar to [1], we use `~x` to denote a placeholder with name `x` for a single (1..1) AST node/attribute, and `~*xs` to denote a placeholder with name `xs` for a list of multiple (cardinality 0..*) AST nodes/attributes. Each placeholder can occur multiple times in a single pattern, e.g., `~x + ~x`, denoting that the AST nodes that match with these placeholder occurrences should be identical. In practice, parsers may store file positions and comments as part of the AST, but we exclude such annotations from AST matching. For example, an Ada-pattern that describes an if-then-else statement with equal then- and else-clauses could be specified as:

```
if ~c then ~*stats; else ~*stats; end if;
```

Concrete syntax patterns have been implemented in, e.g., Stratego, TXL, and Rascal. Their designs for using concrete syntax are described in, e.g., [1], [5], and [6], respectively, but these descriptions typically focus on specific parsing technologies. Generally the AST patterns are obtained by parsing using either:

- Grammar of the object language: embed the object language inside the meta language, and create a new parser;
- Parser for the object language: encode the placeholders using the object language, and use a black-box parser.

Regarding the first approach, [4] indicates that "*a grammar is always necessary in order to parse the code*" and hence the "*restriction that a grammar is necessary for generating a pattern language does not impose extra work*". However, "*developing such grammars is a costly and error-prone endeavor, especially for real-life programming languages*" [6].

In this paper we discuss our experience with concrete syntax for AST patterns from industrial reengineering projects. [7, 8, 9, 10]. "*Reengineering is an area where technology exchange with the IT industry is crucial. ... The real problems that are known in the IT industry should become available to academics*" [4]. In this paper we illustrate our experiences using snippets of code in the programming languages C++ and Ada. The C++ examples are based on the Eclipse C/C++ Development Tools (CDT) parser [11], and the Ada examples are based on the Libadalang parser [12]. We use such parsers as "*The C++ grammar is known to pose a significant parsing challenge*" [13], "*The C++ grammar is context-dependent and ambiguous. This makes the creation of a C++ parser a complex hence difficult task*" [14], and "*keeping up with the pace of development of new C++ language features is challenging*" [15].

In this paper we focus on specifying placeholders inside a single concrete syntax pattern, and ignore aspects like:

- Dealing with macros and pre-processors,
- Meta-language fragments beyond only placeholders (e.g., more complex expressions),
- Linked find/replace patterns for code transformations.

Although the goal of "*pattern matching with concrete syntax is WYSIWYG*" (what you see is what you get) [2], in this paper we identify three specification challenges:

- Section 2: Understanding the underlying AST structure;
- Section 3: Ambiguities caused by placeholders;
- Section 4: Encoding and recognizing the placeholders.

These are overarching challenges for AST matching based on concrete syntax patterns. We link these challenges to specific designs in Section 5. Conclusions and future work can be found in Section 6.

2. Understanding the Underlying AST Structure

Concrete syntax patterns provide a convenient alternative for writing AST patterns. In this section, we illustrate that users of concrete syntax patterns are still, sooner or later, confronted with the formal language structure and/or AST, which may be unfamiliar to them. This can happen in two phases:

- Specification of the concrete syntax patterns;
- Post-processing of AST matches from the code base.

As far as we know, this challenge does not get attention in the literature on implementing concrete syntax patterns. In the subsections, we identify groups of related issues. We frequently discuss differences between parsers, even though our goal is not to use the same pattern with different parsers. Our goal is to indicate dependencies on implementation decisions from specific parsers, which get exposed to users of concrete syntax patterns.

2.1. Mental Model may differ from the Formal Language

Even experienced language users may be surprised by the AST structures of familiar languages. For example, look at the C++ declaration pattern

```
~t x
```

which consists of a specifier `~t` and a declarator `x`. Suppose we try to match this pattern with the following C++ code fragments:

- **`int`** `x` is a match;
- **`const int`** `x` and **`static int`** `x` are also matches, as type qualifiers like **`const`** and storage classes like **`static`** are part of the specifier;
- **`int*`** `x` and **`int`**`&` `x` are no matches, as pointer operators like `*` and `&` are part of the declarator.

The last two categories may surprise many C++ developers that did not study the formal language specification of C++.

Sometimes there are differences between related constructs in different languages. For example, AST nodes for if-statements in C++ just contain a condition, then-clause and (optional) else-clause. However, in Ada there is an additional (possibly-empty) list of elsif-clauses, each with a condition and statement, which semantically correspond to else-clauses with nested if-statements. This has an impact on concrete syntax patterns for Ada as the elif-clauses explicitly need to be considered.

Similarly, the AST of a case-statement in Ada consists of a condition and a list of when-clauses (containing statements), whereas the AST of the related switch-statement in C++ consists of a condition and a list of statements (including possibly-nested case- and break-statements, but not grouped by case-statement). This has an impact on the post-processing of AST matches, even though C++ code often does not use the full flexibility of the language.

There are also differences in multi-variable declarations. A multi-variable declaration in Ada, such as

```
int x, y := 42;
```

consists of a list of identifiers and a single optional expression as initial value for all variables. A multi-variable declaration in C++, such as

```
int x, y = 42;
```

consists of a list of declarators, each consisting of an identifier and an optional expression as initial value. Hence a pattern such as

```
int ~*xs := 3;
```

looks useful in Ada as placeholder `~*xs` can be matched with multiple AST nodes in Ada. However, the similar pattern

```
int ~*xs = 3;
```

is suspicious in C++, as placeholder `~*xs` can only be matched with a single AST node in C++.

The challenge discussed in this subsection is caused by matching using the programming languge's AST. This challenge may (to some extent) be solvable using different matching algorithms like [16].

2.2. AST Representation may be Parser Specific

The representation of the AST by a specific parser also has an impact. For example, in CDT the AST node type for unary operators is also used to represent prefix operators, postfix operators, parentheses and special operators like `noexcept(...)`, thus abstracting from their different concrete syntax. This may give unexpected matching results, as C++ expression pattern `~op x` matches `++x`, `x++`, `(x)` and `noexcept(x)`. Moreover C++ function call expression pattern `~func(~expr)` does not match `noexcept(x)`.

Another example is the representation of if-then and if-then-else statements. We could imagine the use of different AST node types for these statements, but have only seen a single node type for both statements in the studied parsers. However, in CDT the else-clause is represented as a single statement node, which is null if there is no else-clause. In Libadalang the else-clause is represented by a list of statement nodes, which is empty if there is no else-clause. So in Ada, the pattern

```
if ~c then ~*ts; else ~*es; end if;
```

indicates an optional else-clause, as `~*es` denotes any number of AST nodes, which may be counter-intuitive from the concrete syntax. This could be solved by introducing a notation like `~+es` to denote a placeholder for a non-empty list (1..∗) of AST nodes/attributes.

Similarly, a qualified name `a.b.c.d` could be represented as a list [a, b, c, d] or as a tree ((a, b), c), d). This impacts the behavior of placeholders for lists of nodes, e.g., `a.~*ps.d`. Only in case of a list representation, `~*ps` can consist of list of nodes.

Finally, C++ type qualifiers, like `const` and `volatile`, are semantically order-independent. CDT stores them not as a list, but as a boolean attribute per possible qualifier. Hence `const volatile int x` and `volatile const int x` have identical ASTs, and hence match with each other, which may look counter-intuitive from the concrete syntax. Also the concrete syntax pattern `~*qs int x` cannot be used to specify a declaration with any combination of type qualifiers, as `~*qs` expects an (order-dependent) list. Side note: it may also be problematic because all qualifiers are reserved words, but that is a separate issue that we will address in Section 4.1.

The challenge discussed in this subsection is caused by design decisions from specific languages and parsers. This challenge may (to some extent) be solvable by taking AST matching into account during language and parser design; see also the suggestion from [17] for further research on language design principles.

2.3. Parser-Specific AST may not fulfill your AST Expectations

In Ada, function calls without arguments must omit the bracket pair `()` in the concrete syntax. In the AST from Libadalang, this does not lead to a FunctionCall node, but to an Identifier node similar to a variable reference. According to [18], the semantic analyzer should use resolution routines to resolve

ambiguous nodes, for example to distinguish procedure calls from entry calls. According to [3], this disambiguation should already be resolved in the mapping from concrete syntax to abstract syntax; see also Section 3.3 on ambiguities caused by missing placeholder context. These expectations are not fulfilled as no single concrete syntax pattern matches all function calls with any number of arguments.

In addition, decimal literals in Ada can contain optional underscores, like `1_000`, to increase readability, but these underscores are just syntactic sugar. The AST from Libadalang contains the concrete syntax of the decimal literal, so including any underscores, and hence the ASTs for `1000` and `1_000` differ. Also, the AST from CDT often stores the concrete syntax, causing that for example the long value `2147483648` is different from `2147483648L` and that none of the following representations of the same integer value are identical: `0x10`, `0X10`, `16`, `020`, `0b10000`, and `0B10000`. Yet, the ASTs from CDT for the strings `"con""cat"` and `"concat"` are identical, despite their different concrete syntax.

Furthermore, explicit specifications often do not match their implicit, equivalent specification. In CDT, the type `signed int` does not match `int` and the function declaration with explicitly no parameters, i.e., `void f(void);`, does not match the implicit one, i.e., `void f();`.

Finally, you may not expect to find parenthesis nodes in an AST, but they are present in Libadalang and CDT. They are represented in different ways: Libadalang represents them using a dedicated ParenExpr node type, whereas CDT represents them as a unary operator. Thus parenthesis become relevant during pattern matching, as `(x)` and `x` are different.

The challenge discussed in this subsection is caused by design decisions from specific languages and parsers. This challenge may (to some extent) be solvable by taking AST matching into account during language parser design; see also the suggestion from [17] for further research on language design principles.

3. Ambiguities Caused by Placeholders

Specifying AST patterns using concrete syntax with placeholders is less direct than using abstract syntax, especially when aiming for compactness. In this section, we illustrate that this can easily lead to ambiguities. In each subsection we identify a source of ambiguity. Additional types of C++ ambiguities are documented in [13].

3.1. Indistinguishable Object- and Meta Language-Identifiers

Some authors argue for "*as few squiggles as possible*" [1] and "*minimal syntactic overhead*" [19], and argue that the $\sim$ symbol for placeholders should be optional or omitted. However, this can lead to specifications that are ambiguous in whether an identifier should be interpreted in the meta language as placeholder name, or in the object language as variable name.

In general, users of concrete syntax patterns may not realise that they are writing a combination of two languages: the object and meta language. Our own practical experience is that it is useful to have clearly recognizable placeholder identifiers that are easily distinguishable from variable names.

The challenge discussed in this subsection is caused by trying to be too compact. This challenge can be solved by explicitly distinguishing these two types of identifiers.

3.2. Implicit AST Node Type of Placeholders

In some approaches it is possible, or even required, to specify the AST node type of placeholders, either explicitly or by convention on the placeholder name [1, 6]. Sometimes it is possible, or even required, to omit the AST node type from placeholders. When an AST node type needs to be provided, users of concrete syntax patterns are exposed to the underlying AST, which is what we aim to avoid or minimize. However, omitting the AST node type can also introduce ambiguities [19]. In this section, we focus on placeholders without explicitly specified AST node type.

Consider the C++ pattern

$$\sim \texttt{x}\,;$$

for a single statement. Would the type of the placeholder be (any) Statement; an ExpressionStatement; (any) Expression, nested inside an ExpressionStatement; an IdExpression; or a Name, nested inside an IdExpression?

Also consider the C++ enum-declaration pattern

```
enum { ~member }
```

where placeholder `~member` can have two possible AST node types, viz., Enumerator or Name:

- `enum { name }` is a match for both AST node types.
- `enum { name = 0 }` is only a match for AST node type Enumerator.

We make the following observations:

- The more shallow AST node types, e.g., Enumerator, enable the matching of placeholders with composite AST nodes. Shallow refers to the distance to the root node of the AST, so not to the node type hierarchy, if any.
- A single AST node type for each placeholder in a concrete syntax pattern avoids case analysis in the post-processing, e.g., to get the name in all possible matches.

The combination of these observations suggests to use for each placeholder the most shallow AST node type within the concrete syntax pattern, which is indeed useful in our experience. This means that for specifying an enum-member without a default value, either we would need placeholders with cardinality zero (0..0) in the meta language, or we need to filter out the unwanted matches as post-processing. The latter option keeps the meta language simpler, yet distributes knowledge over the pattern and the filter.

A similar situation occurs with the C++ function declaration pattern

```
void ~func(~x) { ~*body; }
```

where placeholder `~x` can have three possible AST types, viz., ParameterDeclaration, NamedTypeSpecifier, or Name:

- `void f(int) { }` is a match for all three placeholder AST types.
- `void f(int i) { }` is only a match for a placeholder of type ParameterDeclaration.

Again, the most shallow node type, so ParameterDeclaration, seems to be appropriate. To specify a parameter declaration without a name, we would again need placeholders with cardinality zero.

But what about the type of placeholder identifiers that occur more than once? Imagine the C++ pattern for swapping,

```
int ~t = ~x; ~x = ~y; ~y = ~t;
```

in which each placeholder occurs twice. Recall that the pattern matching is applied on the AST, so the binding of names is not considered. The first occurrence of `~t` has to be a Name, whereas all other occurrences of placeholders can be IdExpression and Name. Again, for each placeholder, the most shallow node type within the full scope of the pattern (so considering all occurrences of the placeholder) seems to be appropriate, so Name for `~t` and IdExpression for `~x` and `~y`.

The examples so far can be addressed using the unique most shallow node type, but this is not always possible. Ada's SubtypeIndication node type contains an optional constraint of a node type for which there are 5 possible subtypes. The 2 so-called composite subtypes, Index and Discriminant, just consist of a non-empty list, but with unrelated node types, DiscreteRange and DiscriminantAssociation, that can contain just a Name. So there is no unique most shallow node type of placeholder `~*xs` in pattern `subtype S is T(~*xs);`.

The challenge discussed in this subsection is caused by trying to be too compact. This challenge could be solved by being explicit about the AST node types, but then we would loose some of the benefits of concrete syntax patterns. An alternative direction would be the use of generalized parsers that produce all possible ASTs; see also Section 5.1.

3.3. Parsing Dependencies on Name and Type Resolution

The parsing of languages like C++ requires name and type resolution to resolve ambiguities. For example, without context about symbols g and ∼x, statement pattern

```
g(∼x);
```

is ambiguous in C++. It can be parsed both as a DeclarationStatement that declares a variable ∼x of type g, and as an ExpressionStatement that invokes a method g with argument ∼x. In both cases Name is a valid AST node type for placeholder ∼x.

More C++ examples like this are known [14], including statement

```
∼x * ∼y;
```

that can be parsed as a DeclarationStatement, declaring a variable ∼y of type ∼x *, and as an ExpressionStatement, applying binary operator * to variables ∼x and ∼y. In these cases, we noticed that CDT prefers the parsing as DeclarationStatement, unless we provide a context in which ∼x and/or ∼y are declared as variables.

Such ambiguity aspects have an impact on the specification of concrete syntax patterns in languages like C++. If we want to disambiguate these patterns, there are two obvious candidates:

1. Solve the ambiguity by annotating some concrete syntax fragments, which are not necessarily placeholders and which possibly occur within a larger pattern, with the intended AST node type;
2. Solve the ambiguity using a parsing context with declarations of variables, functions, and types.

Note that only the latter candidate requires no additional AST awareness from the user, apart from knowing when a parsing context needs to be specified. An alternative direction would be the use of generalized parsers that produce all possible ASTs; see also Section 5.1.

4. Encoding and Recognizing the Placeholders

In Section 1 we have described two ways to obtain the ASTs for the patterns. In this section, we focus on approaches that use a black-box parser for the object language. Such approaches typically consist of three conceptual steps for handling concrete syntax patterns with placeholders:

1. Encode the placeholders within the concrete syntax;
2. Parse the concrete syntax into abstract syntax;
3. Recognize the placeholders within the abstract syntax.

For the encoding, we see two overall approaches:

- The user directly encodes the placeholders;
- Specifications by users are translated into an encoding.

In this section, we identify challenges for directly encoding the placeholders; note that this is also relevant when designing a translation to an encoding. In each subsection we identify an aspect of such a placeholder encoding.

4.1. Grammatical Constructs

The notation ∼p for placeholders in the previous sections suggests an encoding using an identifier in the object language, possibly with certain naming conventions. However, usually not all code fragments, that fully represent one AST node or a list of multiple AST nodes, in the object language can be replaced by a single identifier. In what follows we discuss two possibilities, which may also be combined, to address this issue:

- Encode the placeholders using larger language fragments;

- Parse the object language with sufficient error recovery.

As a simple example of an encoding using larger language fragments, consider the pattern for any statement in C++. The pattern `~x` is not a valid statement. Appending a semi-colon to the pattern, i.e., `~x;`, results in a larger, yet valid statement pattern. This approach is taken by many designs based on black-box parsers.

A more complex example is the parameter declaration for functions in Ada. The Ada language does not support a single name to be used at the place of a parameter declaration, as in `function fun(~*xs)`. Instead, a larger pattern like

```
function fun(~*xs : ~Dummy)
```

could be used, where the special identifier `~Dummy` would indicate that we do not intend to match a list of names with this specific type, but a list of (parameter) declarations. In the context of JSON, the concrete holes in figure 2 from [6] illustrate the same phenomenon. We have encountered similar situations, for example, when specifying any list of elsif's in Ada. The drawback of adding such special identifiers is that they reduce the understandability of the concrete syntax pattern for users.

As an example of parsing with sufficient error recovery, consider the C++ declaration pattern

```
class X { ~*f; };
```

which seems to be a natural way to specify a class with any number of declarations. Even with the extra semi-colon at the end, this is strictly speaking not valid C++ code, but CDT's error recovery [14] parses `~*f` as a UsingDeclaration with Name `~*f`. The most shallow node type as discussed in Section 3.2 could then be used to interpret the placeholder as any number of declarations.

Typical examples of concrete syntax fragments that often cannot be resolved in these ways are reserved words in the object language, as identifiers and reserved words are usually disjoint. Reserved words could be useful for matching C++ type classifiers such as **`const`** and **`volatile`**. However, recall that C++ type classifiers also give another problem in CDT as discussed in Section 2.2.

4.2. Name Resolution

It is convenient if placeholder names can be used in the concrete syntax pattern without explicit declaration. Although in general the parsing may use declarations to resolve ambiguities, see also Section 3.3, in our experience the declarations of placeholder names can usually be omitted without causing problems. The capabilities of the used parsers to deal with incomplete input are key in this respect.

4.3. Naming Scheme

For each placeholder, we usually want to specify:

- Identifier: name to refer to the specific placeholder;
- Cardinality: how many AST nodes to match, e.g., exactly one node, or any number of nodes;
- AST node types (optional): type of the AST node(s).

The placeholders must be encoded in the concrete syntax, and afterwards be recognized in the abstract syntax. As noted by [6], a challenge in concrete syntax patterns is to avoid capture between placeholder identifiers from the meta language, and normal identifiers from the object language. Some alternative approaches that we see:

- use a pattern-specific list of placeholder identifiers;
- use a predefined class of placeholder identifiers, which should not be used as normal identifiers in any pattern.

The first approach might be preferable since each identifier from the language can be used for a variable and for a placeholder, only not both in the same pattern. However, object and meta-language identifiers are indistinguishable within a pattern (without considering the list of placeholder identifiers), a drawback already mentioned in Section 3.1, and a change might affect both the pattern and the list of placeholder identifiers.

For brevity reasons, the second approach may be preferable for manually-specified placeholder encodings. A drawback is that matching actual code is impossible when that code uses variable names that are in the predefined class of placeholder identifiers. In some cases you can use identifiers that are illegal according to the object language, but that are supported by the parser. In the Ada language, for example, identifiers cannot start with a dollar-sign, but Libadalang supports parsing of such identifiers. In these cases, the drawback is absent and the object and meta-language identifiers can be easily distinguished.

5. Related Work

In this section, we discuss various approaches for concrete syntax patterns from the literature, and relate them to the specification challenges that we have presented. The subsections correspond to the two ways to obtain the ASTs for the patterns, as described in Section 1.

5.1. Grammar of the Object Language

There are two types of parsing with different outputs:

- Deterministic, like LR (Left-to-right Rightmost derivation) and LL (Left-to-right Leftmost derivation): produces one single AST;
- Generalized, like GLR (generalized LR) and GLL (generalized LL): produces all possible ASTs.

With respect to the ambiguities from Section 3, generalized parsers can be used to detect ambiguities in concrete syntax patterns, and to produce all possible interpretations. However, they may complicate the further processing of matching code fragments, by requiring filtering of duplicate matches, and case distinctions on specific AST node types.

"*Most modern source transformations tools, such as ASF+SDF, Stratego, and DMS, use generalized LR (GLR) parsers*" [5]. In particular ASF+SDF is used as basis for the native patterns from [4] and Spoofax. In contrast, Rascal uses a "*scannerless variant of the GLL parsing algorithm*" [20], whereas "*TXL ... yields a deterministic unambiguous parse in all cases*" [5].

Stratego in Spoofax uses an approach [1, 19] based on language embedding using grammars for the meta and object language. For meta variables, a squiggle operator (∼) is introduced that does not occur in the object language. To limit the number of squiggles in object patterns, the grammars for meta variables and object variables may (partially) overlap. Ambiguities are resolved by giving preference to meta variables over object variables.

DMS ([21]) uses escapes for meta-variables, such as `\p`. Rascal uses placeholders that require a mandatory indication of the AST node type, e.g., `<T p>`. TXL [5] uses `p[T]` to denote a meta variable `p` with AST node type `T`; cardinality can be expressed using `[opt T]` and `[repeat T]`. Native patterns from [4] do not visually distinguish between meta and object language variables; they postfix the placeholder variable for lists with a `*` or `+` to denote the cardinality.

The approach from [16] also uses concrete syntax patterns, where `:[p]` is used to denote holes. The patterns are parsed using a highly-reusable base grammer that focuses on delimiter pairs. In between the delimiters, everything is parsed as a sequence of language-specific tokens. This approach avoids some of our challenges on understanding the underlying AST structure, but cannot handle patterns that rely on the AST structure beyond the delimiter structure.

Coccinelle [22] uses the Semantic Patch Language (SmPL) to specify patterns using concrete syntax. However, the underlying pattern matching is based on paths in the control flow graph instead of structures of the AST. The placeholders are declared with an explicit AST node type.

5.2. Parser for the Object Language

ClaiR (C Language Analysis in Rascal) [6] is based on Eclipse CDT and Rascal. The placeholders are specified using Rascal's approach, in which it is mandatory for syntax safety to indicate the AST node type of placeholders. Placeholders are then automatically encoded in concrete syntax by the lower-step, and recognized in the abstract syntax by the lift-step. Internally the encoding uses generated placeholder identifiers. Afterwards "*Rascal's ordinary pattern evaluation logic takes over*" [6].

Mooij et al. [7] report about an AST pattern matcher for Delphi based on concrete syntax. The same approach was applied in [8] for C/C++ using Eclipse CDT, and in [9] for Ada using Libadalang. During matching, for each placeholder the most shallow matching AST node type is used. Users encode the placeholders directly in the concrete syntax using special identifiers, where p can be any valid name:

	Single AST node	Multiple AST nodes
Delphi	`_p`	`__p`
C/C++	`$p`	`$$p`
Ada	`$S_p`	`$M_p`

This table illustrates the intricacies of the placeholder naming scheme for different languages. The differences originate from the identifier syntax that is supported by the parsers. The AST pattern matcher identifies and interprets the placeholders on-the-fly based on this naming convention. The following diagnosis information is available in case of failing matches:

- formatted stack-trace with the related pattern and instance nodes, represented both in concrete and abstract syntax;
- collected placeholders with their values.

6. Conclusions and Future Work

We have identified two challenges for specifying concrete syntax patterns for AST matching: (1) understanding the underlying AST structure, and (2) ambiguities caused by placeholders. For designs based on black-box parsers we have also inventorized the challenge of (3) encoding and recognizing the placeholders. Our results can serve

- as warnings to users of concrete syntax patterns,
- as additional requirements for parser front-ends,
- as attention points for language specifications, and
- as starting points for further research on pattern matching.

Note that the third point relates to a research area that was already suggested by [17], viz., "language design principles for easily manipulable and transformable programs".

Future work could explore the debugging functionality in case of unexpectedly failing (or successful) pattern matches, and the detection of suspicious patterns as discussed in Section 2.1. Other directions could include ways to specify linked find/replace patterns for code transformations, and other matching algorithms for concrete syntax patterns.

Acknowledgments The authors thank Jeroen Ketema and Rosilde Corvino for sharing their experience with the approach from [8] in industrial contexts, Jeff Smits for Spoofax support during our experiments with the approach from [1], and Rodin Aarssen for clarifying discussions on the approach from [6]. The authors thank Rosilde Corvino and the anonymous reviewers for their feedback on an earlier version of this article.

Part of the research is carried out as part of the HybridAnalysis program under the responsibility of TNO-ESI in cooperation with Philips Image Guide Therapy Systems. The research activities are co-funded by Holland High Tech | TKI HSTM via the PPP Innovation Scheme (PPP-I) for public-private partnerships.

References

[1] E. Visser, Meta-programming with concrete object syntax, in: GPCE 2002, volume 2487 of *LNCS*, Springer, 2002, pp. 299–315. doi:10.1007/3-540-45821-2_19.

[2] T. van der Storm, Semantics engineering with concrete syntax, in: EVCS 2023, volume 109 of *OASIcs*, 2023, pp. 29:1–29:11. doi:10.4230/OASIcs.EVCS.2023.29.

[3] A. S. Herrera, E. D. Willink, R. F. Paige, A domain specific transformation language to bridge concrete and abstract syntax, in: ICMT 2016, volume 9765 of *LNCS*, Springer, 2016, pp. 3–18. doi:10.1007/978-3-319-42064-6_1.

[4] M. P. A. Sellink, C. Verhoef, Native patterns, in: 5th Working Conference on Reverse Engineering, WCRE '98, IEEE Computer Society, 1998, pp. 89–103. doi:10.1109/WCRE.1998.723179.

[5] J. R. Cordy, The TXL source transformation language, Sci. Comput. Program. 61 (2006) 190–210. doi:10.1016/j.scico.2006.04.002.

[6] R. Aarssen, J. J. Vinju, T. van der Storm, Concrete syntax with black box parsers, Programming Journal 3 (2019) 15–es. doi:10.22152/programming-journal.org/2019/3/15.

[7] A. J. Mooij, M. M. Joy, G. Eggen, P. Janson, A. Radulescu, Industrial software rejuvenation using open-source parsers, in: ICMT 2016, volume 9765 of *LNCS*, Springer, 2016, pp. 157–172. doi:10.1007/978-3-319-42064-6_11.

[8] S. Klusener, A. J. Mooij, J. Ketema, H. van Wezep, Reducing code duplication by identifying fresh domain abstractions, in: ICSME 2018, IEEE Computer Society, 2018, pp. 569–578. doi:10.1109/ICSME.2018.00020.

[9] P. van de Laar, A. Mooij, Renaissance-Ada: Tools for analysis and transformation of Ada code, Ada User Journal 43 (2022) 165–170. https://www.ada-europe.org/archive/auj/auj-43-3-withcovers.pdf.

[10] P. van de Laar, R. Corvino, A. J. Mooij, H. van Wezep, R. Rosmalen, Custom static analysis to enhance insight into the usage of in-house libraries, J. Syst. Softw. 212 (2024) 112028. doi:10.1016/J.JSS.2024.112028.

[11] Eclipse Foundation, Eclipse C/C++ Development Tools (CDT), https://github.com/eclipse-cdt, 2024.

[12] AdaCore, Libadalang, https://www.adacore.com/libadalang, 2024.

[13] E. Willink, Meta-Compilation for C++, Ph.D. thesis, University of Surrey, 2001.

[14] D. Piatov, A. Janes, A. Sillitti, G. Succi, Using the Eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser, in: OSS 2012, volume 378 of *AICT*, Springer, 2012, p. 399. doi:10.1007/978-3-642-33442-9_45.

[15] N. Ridge, C++ language support in Eclipse CDT, 2017. https://www.eclipse.org/community/eclipse_newsletter/2017/april/article3.php.

[16] R. van Tonder, C. Le Goues, Lightweight multi-language syntax transformation with parser parser combinators, in: PLDI 2019, ACM, 2019, pp. 363–378. doi:10.1145/3314221.3314589.

[17] R. D. Cameron, M. R. Ito, Grammar-based definition of metaprogramming systems, ACM Trans. Program. Lang. Syst. 6 (1984) 20–54. doi:10.1145/357233.357235.

[18] J. Miranda, E. Schonberg, GNAT: The GNU Ada Compiler, 2004. https://www.adacore.com/uploads/books/pdf/gnat-book.pdf.

[19] M. Bravenboer, R. Vermaas, J. J. Vinju, E. Visser, Generalized type-based disambiguation of meta programs with concrete object syntax, in: GPCE 2005, volume 3676 of *LNCS*, Springer, 2005, pp. 157–172. doi:10.1007/11561347_12.

[20] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, J. J. Vinju, Modular language implementation in Rascal - experience report, Sci. Comput. Program. 114 (2015) 7–19. doi:10.1016/j.scico.2015.11.003.

[21] I. D. Baxter, C. W. Pidgeon, M. Mehlich, DMS®: Program transformations for practical scalable software evolution, in: ICSE 2004, IEEE Computer Society, 2004, pp. 625–634. doi:10.1109/ICSE.2004.1317484.

[22] Y. Padioleau, R. R. Hansen, J. L. Lawall, G. Muller, Semantic patches for documenting and automating collateral evolutions in Linux device drivers, in: PLOS 2006, ACM, 2006, p. 10–es. doi:10.1145/1215995.1216005.