Concrete Syntax for Objects

2

ABSTRACT

Application programmer's interfaces give access to domain knowledge encapsulated in class libraries without providing the appropriate notation for expressing domain composition. Since object-oriented languages are designed for extensibility and reuse, the language constructs are often sufficient for expressing domain abstractions at the semantic level. However, they do not provide the right abstractions at the syntactic level. In this chapter we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level. The method is supported by proven and available technology, i.e. the syntax definition formalism SDF and the program transformation language and toolset Stratego/XT. We illustrate the method with applications in three domains: code generation, XML generation, and user interface construction.

2.1 INTRODUCTION

Class libraries encapsulate knowledge about the domain for which the library is written. The application programmer's interface to a library is the means for programmers to access that knowledge. However, the generic language of method invocation provided by object-oriented languages does often not provide the right notation for expressing domain-specific composition. General purpose languages, particularly object-oriented languages, are designed for extensibility and reuse. That is, language concepts such as objects, interfaces, inheritance, and polymorphism support the construction of class hierarchies with reusable implementations that can easily be extended with variants. Thus, OO languages provide the flexibility to develop and evolve APIs according to growing insight into a domain.

Although these facilities are often sufficient for expressing domain abstractions at the semantic level, they do not provide the right abstractions at the syntactic level. This is obvious when considering the domain of arithmetic or logical operations. Most modern languages provide infix operators using the well-known notation from mathematics. Programmers complain when they have to program in a language where arithmetic operations are made

available in the same syntax as other procedures. Consider writing `e1 + e2` as `add(e1, e2)` or even `x := e1; x.add(e2)`. However, when programming in other domains such as code generation, document processing, or graphical user interface construction, programmers are forced to express their designs using the generic notation of method invocation rather than a more appropriate domain notation. Thus programmers have to write code such as

```
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15,15,15,15));
```

in order to construct a user interface, rather than using a more compositional syntax reflecting the nice hierarchical structure of user interface components in the Swing library. Building in syntactic support for such domains in a general purpose language is not feasible, however, because of the different speeds at which languages and domain abstractions develop. A language should strive for stability, while libraries can be more volatile.

In this chapter we describe MetaBorg[1], a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level [Mernik et al. 2005].

For example, to improve the construction of user-interfaces in Java, we have designed a little *Swing user interface Language* (Swul) that makes the compositional structure of the Swing components visible in application programs. Using our method we have embedded this language in Java, such that it is directly available to application programmers. They can now write *within their Java programs* expressions such as

```
JPanel panel = panel of border layout {
  north = label "Welcome"

  center = scrollpane of
    input : textarea {
      rows    = 20
      columns = 40
    }

  south = panel of border layout {
    east = button for ExitAction
  }
};
```

in order to implement a user interface consisting of a panel with border layout, containing a label, a text area, and another panel with a button. Such a program is *assimilated* into the surrounding Java code by translating it to the sequence of Swing method calls that one would write by hand.

[1] MetaBorg provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to strengthen itself. The ease of implementing embeddings makes resistance futile.

Our work stands in a long line of approaches to add syntactic extensibility to programming languages [Leavenworth 1966, Weise & Crew 1993, Cardelli et al. 1994, Shalit 1996, Batory et al. 1998, Brabrand & Schwartzbach 2002, Begel & Graham 2004]. Although our work has many commonalities with other approaches, it is distinguished by its generality, i.e. the lack of restrictions on either the syntax or the semantics of embedding and assimilation. In addition, implementation of embeddings is high-level and concise; definition and embedding of Swul required only 100 lines of syntax definition and 170 lines of assimilation rules. Our method has the following characteristics:

Syntactic Embedded code fragments are checked syntactically at compile-time. This is in contrast with approaches to compose program fragments using string literals.

No restrictions on syntax definition Our maxim is that it should be possible to design a notation that is fitting for the domain without placing artificial restrictions on the syntax to be used. This means that both lexical and context-free syntax should be definable. Furthermore, all aspects of the embedding, including quotation symbols, if any, should be adaptable. Only the full class of context-free grammars allows such natural syntax definition. This is in contrast to languages with user-definable operators [Hudak 1996], overloading of (a fixed set of) operators, syntax macros [Leavenworth 1966], or grammar formalisms supporting only a subset, such as LL or LALR, of the context-free grammars [Cardelli et al. 1994, Batory et al. 1998]. The only proviso we make is that host and embedded language have a *context-free* syntax.

Not restricted to a single host language The method is not specific to a particular host language [Batory et al. 1998], but can be used to embed any language in any host language.

Interaction with host language Embedded code fragments should be able to refer to artifacts in the host program and vice versa. This is in contrast to approaches based on a separate domain-specific language from which code is generated [Smaragdakis & Batory 2000, Mernik et al. 2005].

Combination of extensions It should be possible to combine multiple domain notations, in contrast to hard-wired language extensions.

No restrictions on assimilation The translation of embedded fragments to the host language should not be limited to a simple homomorphism or other fixed translation order [Brabrand & Schwartzbach 2002], but should allow use of context-sensitive information, global analysis, and multi-stage transformations.

As a consequence of these characteristics we do *not* require that language extensions are implemented *within* programs in the host language [Cardelli et al. 1994], since such approaches lead to restrictions in many of the areas

mentioned above. We also do *not* expect language design and implementation skills from the average application programmer. Instead we opt for a separation of roles between the metaprogrammer defining the language embedding and assimilation, and the application programmer using a domain notation. However, our techniques are sufficiently high-level that a knowledgeable programmer can use them to create new embeddings. That is, the method is based on SDF2 [Visser 1997b, van den Brand et al. 2002], a syntax definition formalism used to define embeddings, and Stratego/XT [Visser 2004, de Jonge et al. 2001] a language and toolset for program transformation used to implement assimilation. These tools are mature and freely available from [SDF Website] and [Stratego Website], respectively. The applications of MetaBorg developed in this chapter are available at `http://www.metaborg.org`.

Organization In Section 2.2 we examine the practice of object-oriented programming in three domains: code generation, document generation, and graphical user interface construction. For each of these domains we show how the readability of programs improves dramatically by employing domain-specific concrete syntax. In particular, we show how to generate Java programs and XML documents in Java. Furthermore, we describe the domain-specific *Swing user interface Language* (Swul), which provides a nice compositional language for user interface composition in Java. In Section 2.3 we explain how concrete syntax embeddings and the corresponding assimilations are realized using the MetaBorg method and illustrate this by the implementation of embeddings for the three applications from Section 2.2. In Section 2.4 we give an overview of the syntax definition formalism SDF2. In Section 2.6 we discuss the relation with competing approaches such as user-definable operators, syntax macros, application generators, and domain-specific languages. We discuss future work in Section 2.7, and conclude in Section 2.8.

2.2 CONCRETE SYNTAX FOR OBJECTS

In this section we examine three application domains that suffer from the misalignment between language notation and the domain: code generation, XML document generation, and graphical user interface construction. For each of these domains we discuss the methods that are used for programming in these domains using an object-oriented language and we show how our concrete syntax method dramatically improves the readability and writability of applications in these domains. For all examples in this chapter we use Java as the host language, but the techniques are equally well applicable to other languages.

2.2.1 *Code Generation*

A *code generator* automates the production of boilerplate code by translating a compact high-level specification of a problem into full blown code. Typical applications include the generation of data types for the representation of *abstract syntax trees*, the generation of *XML data binding* [Bourret 2007] code for convert-

ing XML to a specific data type, and the generation of *object-relational binding* code for connecting an object-oriented program to a database system. Numerous tools are available for these purposes; LLBLGen [Bouma] is an object-relational binding generator; ApiGen [de Jong & Olivier 2004] and JTB [Tao et al.] are abstract syntax tree generators, JAXB [JAXB Website] and Castor [Castor Website] are XML data binding tools, to name but a few.

The implementation of a code generator requires an internal representation of program code and an interface for accessing this representation in order to compose and transform code fragments. Ideally, generators use a structured representation of programs, i.e. a data structure to represent abstract syntax trees. Such a representation makes it easy to compose, analyze, and transform fragments. For example, the XML data binding tool JAXB uses a full abstract syntax tree in its code generator. However, in practice, many generators are string-based, meaning that code is generated by directly printing strings to a file, or by representing fragments as strings and composing those. For example and ironically, the abstract syntax tree generators ApiGen [de Jong & Olivier 2004] and JTB [Tao et al.] are text-based code generators. Castor [Castor Website] is an example of a hybrid approach which uses a combination of an abstract syntax tree for the global structure and text for method bodies.

Neither implementations using a data structure, nor using string literals are satisfactory. The advantage of using string literals or text templates is that one can use *concrete syntax*, i.e. the fragments are readable as program code, and it is trivial to implement in a general purpose language. The approach is illustrated by the Castor example in Figure 2.1, which builds up a string representation of a program fragment. However, the disadvantages far outweigh the advantages. Escaping to the host language in order to insert a fragment of code computed elsewhere is cumbersome. The syntax of the generated code is not checked. No further manipulation of the code is possible. And runtime overhead is incurred for parsing, analysis, compilation or interpretation if code is to be executed.

The advantage of using a data structure is that the generated code is structured and is amenable to further processing. Type correctness of the generator often entails syntactic correctness of the generated program. It is also easy to implement in a general purpose object-oriented language. The approach requires the creation of a class hierarchy which can be substantial for a real language. Code generators such as ApiGen and JTB can be used for that purpose. However, composition of code fragments is done via calls to the code API, leading to very verbose metaprograms. It is usually hard to understand the structure of the generated code when inspecting such metaprograms.

The MetaBorg method combines the best of both worlds. Code fragments are written using the concrete syntax of the programming language, but the implementation is based on an API for code representation. This is illustrated in Figure 2.2 with a fragment of a Java program generating a Java program, corresponding to the example in Figure 2.1. The generator uses ATerms [van den Brand et al. 2000] for the representation of generated code. Instead of using constructors from the ATerm class hierarchy to create a code fragment,

```
String x = "propertyChangeListeners";
jsc.add("if (");
jsc.append(x);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, f, v1, v2);");
jsc.add("");
jsc.add("for (int i = 0; i < ");
jsc.append(x);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(x);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Figure 2.1 Code generation in Castor.

```
ATerm x = id |[ propertyChangeListeners ]|;

ATerm stm = bstm |[ {
    if(x == null) return;
    PropertyChangeEvent event = new PropertyChangeEvent(this, f, v1, v1);
    for(int c=0; c < x.size(); c++) {
      ((PropertyChangeListener) x.elementAt(c)).propertyChange(event);
    }
  }
]|;
```

Figure 2.2 Code generation with concrete syntax.

it is written as a regular piece of Java code. The code fragments are distinguished from the surrounding code by the delimiters |[and]|. The delimiters are not fixed in MetaBorg and can even be left out when appropriate. The `bstm` and `id` tags are used to indicate that the fragments are of syntactic category *statement* and *identifier* respectively. A fragment does not need to be closed, but can incorporate code fragments generated elsewhere. For example, in Figure 2.2 the identifier assigned to variable *x* is used in the fragment assigned to the variable *stm*.

Metaprograms with embedded object-program fragments are translated to pure Java programs in which code construction is expressed directly in terms of calls to the code representation API. This tool also guarantees that the code fragments are syntactically correct. The type system of the host language and a properly defined underlying API will then guarantee that *compositions* are syntactically correct as well.

Note that MetaBorg is not restricted to Java in Java. The same method can be used to embed other languages in Java (e.g. to generate C# code), or to use a different host language (e.g. to generate Java code with a C# program). The realization of these embeddings will be discussed in Section 2.3.

```
out.startDocument();
out.startElement("", "html", "html", noAttrs);
out.startElement("", "body", "body", noAttrs);
out.startElement("", "p", "p", noAttrs);
out.characters(text.toCharArray(), 0, text.length());
out.endElement("", "p", "p");
out.endElement("", "body", "body");
out.endElement("", "html", "html");
out.endDocument();
```

Figure 2.3 Composition of an XML document in Cocoon.

```
out.write doc %>
    <html>
      <body>
        <p><% text :: cdata %></p>
      </body>
    </html>
  <%;
```

Figure 2.4 Composition of an XML document with concrete syntax with underlying SAX ContentHandler invocations.

2.2.2 *XML Document Generation*

XML is used on a large scale for the exchange of data between programs. This requires programs to read and write XML documents and to convert internal data to XML and back. Applications that generate XML documents by filling in templates suffer from the lack of support for XML syntax in general purpose programming languages. The problems are similar to that in code generation. Text-based solutions cannot guarantee that the produced text is well-formed XML. Typical examples are server-side scripting languages such as JSP and ASP .NET, which support the embedding of a programming language in XML or HTML. Further manipulation of the document after generation is impossible. A typical example of post generation transformation is the addition of statistics to a generated web page. Many web page generators just put this information outside the HTML tags, which is of course invalid, but is accepted by web browsers if the page does not claim to be well-formed XML.

Other solutions are based on data structures. In libraries such as W3C DOM, JDOM, and XOM, documents are represented by instances of classes corresponding to the generic structure of XML, e.g. `Document`, `Element`, and `Attribute`. Thus document construction is achieved with the constructors of these classes. Alternatively, construction can be achieved with events emitted to a SAX `ContentHandler`, which can be an XML serializer or a DOM constructor, as illustrated in Figure 2.3 with a code fragment from Cocoon [Cocoon Website]. By using these APIs the code is guaranteed to produce well-formed XML as long as the library does its job properly. However, these solutions result in sequences of method invocations that are hard to read.

The MetaBorg solution is the same as in the case of code generation. There

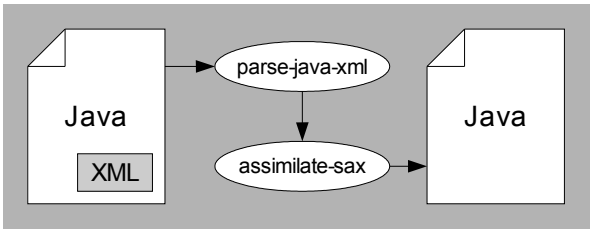

Figure 2.5 A program with embedded concrete syntax (e.g. for XML) is translated to a pure Java program.

is a good notation for this domain, i.e. XML itself. With the MetaBorg method we make this notation available to the application programmer, while keeping the properties of APIs for structured representation of documents. Figure 2.4 shows a statement in a Java program with an embedded XML document. The symbols `%>` and `<%` are used to delimit the embedded XML from the surrounding code. Within the document is a reference to a document fragment defined earlier in the program, using the `<%` and `%>` antiquotation delimiters. The assimilation process (Figure 2.5) transforms a Java program with embedded XML to a pure Java program with calls to a SAX `ContentHandler`.

The big difference between the embedding of XML and the embedding of a programming language is that the lexical structure of XML is completely different from that of, say, Java. This makes the parsing problem more complicated.

2.2.3 *Graphical User Interface Construction*

In the cases of code and XML generation a domain notation is readily available to improve the readability of application programs. However, there are many other domains with APIs representing a 'language' in the sense of a coherent set of concepts and composition facilities, but without a concrete notation. Programming in these domains can also be improved by employing domain-specific notation, which should then first be designed. Consider for example the construction of graphical user interfaces in Java.

Graphical user interfaces can be generated with a visual tool or can be written by hand. A handwritten user interface typically instantiates GUI components such as buttons and textfields and arranges these components in panels by using fixed positioning or a layout manager that allows the user interface to adapt itself to changes in the size of the window. Despite catalogs of design patterns for developing graphical user interfaces, this code is still one of the most unwieldy parts of a program (Figure 2.6). A handwritten user interface takes quite a few lines of code and the resulting code is difficult to understand and maintain.

There are a few proposals for languages specifically geared towards user interfaces, e.g. Mozilla's XUL, W3C's XForms and Microsoft's XAML, but these

```
JTextArea text = new JTextArea(20,40);
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
panel.add(BorderLayout.NORTH, new JLabel("Please enter your message"));
panel.add(BorderLayout.CENTER, new JScrollPane(text));
JPanel south = new JPanel(new BorderLayout(12,12));
JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);
panel.add(BorderLayout.SOUTH, south);
```

Figure 2.6 Sequence of statements for composing a user interface with Swing API methods

```
JPanel panel = panel of border layout {
  hgap = 12
  vgap = 12
  north = label "Please enter your message"
  center = scrollpane of textarea {
    rows    = 20
    columns = 40
  }
  south = panel of border layout {
    east = panel of grid layout {
      hgap = 12
      vgap = 12
      row = {
        button "Ok"
        button "Cancel"
      }
    }
  }
};
```

Figure 2.7 SWUL implementation for composing the same user interface as Figure 2.6

solutions do not integrate well with the host language. They abstract from a GUI toolkit, which is not always an advantage, and restrict the way the GUI can interact with the host language. XUL clones for Java, such as SwiX^{ml} and Luxor, use reflection and a convention for the location of methods (or `Action` fields) to invoke code in the host language. Although this works reasonably well, it limits the way the host language is able to interact with the GUI code. For example, a model-view-controller design, where the GUI components are directly updated from their (Swing) models, is not possible with these toolkits. Furthermore, the XML formatted GUI specifications are interpreted, not checked statically for internal consistency or correct interaction with code in the host language.

Using the MetaBorg method we have developed a solution that provides domain-specific notation for the construction of user interfaces and integrates well with the rest of the program written in the host language. Expressions in the *Swing User Interface Language* (SWUL) correspond to components of a Swing

user interface. The language encourages the compositional construction of complex components, in contrast to the assembly language spaghetti style used with direct calls to Swing. Thus, subcomponents are subexpressions, and do not have to be added afterwards. Attributes of components are named, avoiding the need to continuously look up the order of method parameters. The declaration of the layout style is combined with the instantiation of the subcomponents. Embedded in Java, the Swul language can be used directly to program user interfaces. Swul expressions can refer to elements such as variables and methods from other parts of the program. For example, host language names and expressions can be used to set the value of a layout attribute or to pass an event handler.

The example in Figure 2.7 illustrates the use of Swul, creating in a single Java assignment (be it multiline) the same user interface implemented directly using Swing methods in Figure 2.6. Although the Swul solution uses more lines, the difference in number of non-space characters (507 vs 243) indicates that the Swul solution is significantly more concise.

2.2.4 *Other Applications*

We have discussed three application domains in which concrete domain notation improves programs. The same approach can be applied to many other domains, including embedded query languages such as XPath and SQL. Unfortunately, queries in these languages are usually embedded in string literals. They are often even composed at runtime by concatenating strings. In this case the SQL statements and XPath queries cannot be checked syntactically at compile time. This might result in runtime errors or, even worse, security problems. If a value from the host language is embedded in an SQL query by string concatenation, then there is no guarantee that the string actually is of the syntactic category expected at this point in the SQL statement. This results in a security issue that is rather easy to exploit by entering SQL constructs where plain strings are expected. To prevent this, the embedded string has to be escaped. A missing escape will immediately pose a security threat. This is a general problem of runtime composition of code fragments by string concatenation. XML applications that use string concatenation might be a security risk as well. In such applications proper escaping is required at many different places in the code. This task can be automated by applying our language embedding tools. We discuss the StringBorg method for preventing injection attacks using syntax embedding in Chapter 4.

Another example of a little language that is often embedded in string literals is the language of regular expressions used for pattern matching of character sequences. A regular expression is encoded in a string literal which is interpreted or compiled at runtime to a matcher. The clarity of a regular expression is reduced by requiring special characters to be escaped. C# reduces the number of characters that have to be escaped by adding verbatim strings to the language, where only double quotes have to be escaped. Just as in all string literal based embeddings the regular expression is not checked

syntactically at compile-time.

The difference with the other examples is that the libraries operating on these languages usually only accept textual input. This makes a structured representation of the query more annoying than advantageous. Yet, the MetaBorg method can still be used by assimilating the embedded code fragments to the corresponding string operations. This will make the embedding of meta values safer and easier, and the embedding will guarantee the syntactic correctness of the embedded expressions. To this end the MetaBorg tools support the representation of embedded fragments in a full *parse tree*, which can be yielded to a string.

2.3 REALIZING CONCRETE SYNTAX

Introducing domain-specific notation in a host language requires (1) an *embedding* of the domain-specific language in the host language and (2) *assimilation* of the embedded domain fragments into the surrounding host code. In this section we describe the method of embedding and assimilation from the point of view of the metaprogrammer creating the embedding. We illustrate the method with several examples. In the next sections the technology behind the method will be discussed.

2.3.1 *Embedding and Assimilation*

The architecture of the MetaBorg method is illustrated by the diagram in Figure 2.8. The embedding of domain notation in a host language requires a syntax definition for the host language, a syntax definition for the embedded language, and a syntax definition for the combination of the two languages, embedding the latter in the former. A parser then uses this combined syntax definition to parse programs in the extended language. Next, an assimilator applies a set of rules to assimilate the embedded domain fragments, reducing the program to the pure host language.

The diagram also illustrates the roles of programmer, metaprogrammer, and MetaBorg tooling. An application programmer using the domain extension uses the combination of parser and assimilator for the extension as a single tool, which could even be integrated with the compiler. Thus, programming in the extended language is no different than programming in the host language, except for the additional expressivity that is available. The metaprogrammer implementing a domain extension needs to provide the syntax definitions for host and domain language, the syntax for the embedding and the assimilation rules. Generally, the syntax definition for the host language can be reused from a library of syntax definitions. The syntax definition of the domain language can be reused as well, if the domain language is an existing language that is already used on its own. If the host API is a shared target between several domain notations, then a set of generic assimilation rules can be used. This is typically the case for metaprogramming domains where a standard API for abstract syntax trees is used. Finally, syntax definitions and

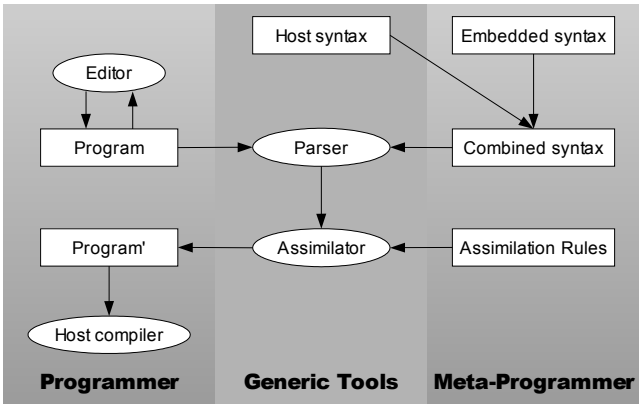

Figure 2.8 Architecture of embedding and assimilation framework.

```
module Java-Tuple
imports Generic-Java
exports
  context-free syntax
    "(" Expr "," Expr ")" -> Expr {cons("NewTuple")}
    "(" Type "," Type ")" -> Type {cons("TupleType")}
```

Figure 2.9 Syntax definition of Java with tuples.

```
module Java-Tuple-Assimilate
imports Generic-Java
rules
  AssimilateTuple :
    expr |[ (e1, e2) ]| -> expr |[ Tuple.construct(e1, e2) ]|

  AssimilateTuple :
    type |[ (t1, t2) ]| -> type |[ Tuple<t1, t2> ]|
```

Figure 2.10 Rewrite rules for assimilation of tuples in Java

assimilation rules are processed by the generic tools provided in the Stratego/XT toolset [Stratego Website, Visser 2004]; MetaBorg can be considered as a particular pattern of usage of these tools.

To make these concepts more concrete we will illustrate the method with several examples. Before diving into the interesting aspects, we look at a very simple example, i.e. the extension of Java 1.5 with concrete syntax for tuples.

Tuples are instances of a class `Tuple<`*A*`, `*B*`>`, which is parameterized with the type of the first and second item of the tuple. We define an extension of Java providing the notation `(x,y)` for the construction and declaration of tuples. Thus, `(Integer, String)` is a type expression and corresponds to the type `Tuple<Integer, String>`. The expression `(1, "foo")` is an instance of `Tuple<Integer, String>`, where the first item in the tuple is the integer 1 and

the second is the String "foo".

To realize this extension all we need is the syntax definition in Figure 2.9 and the assimilation rules in Figure 2.10. The syntax definition in Figure 2.9 defines the language Java-Tuple, which extends Java with syntax for tuple expressions and tuple types. The extension of the Java language is achieved by a simple import of the syntax definition of Java. Thus, it is not necessary to create a copy of that syntax definition and add the rules for tuples to it. The Java-Tuple syntax module is written in SDF, which is the syntax definition formalism employed by MetaBorg. A syntax definition mainly consists of *productions* of the form s_1 ... s_n -> s_0 {cons("c")}, declaring that a phrase of type s_0 can be formed by concatenating phrases of types s_1 ... s_n. The *constructor* c is a name for the production that is used in the construction of abstract syntax trees. The features of SDF that enable concise modular syntax definition will be explained in Section 2.4.

Assimilation of the language extension is achieved by translating the new language constructs to the base language. For example, the Java-Tuple declaration

```
(Integer, String) t = (1, "Hello world!");
```

is translated into the Java declaration

```
Tuple<Integer, String> t = Tuple.construct(1, "Hello world!");
```

This translation is achieved by the *rewrite rules* in Figure 2.10. The assimilation into the host language is implemented in the Stratego program transformation language. Rewrite rules play a central role in Stratego. A labeled rewrite rule L: p_1 -> p_2 rewrites a program fragment matching the pattern p_1 to the program fragment p_2, where the *metavariables* in p_2 are instantiated with the corresponding fragments found when matching p_1. In the first rewrite rule of Figure 2.10, *e1* and *e2* are metavariables denoting Java expressions. In the second, *t1* and *t2* are metavariables denoting Java types. Thus, the first rule replaces a tuple expression (*e1*, *e2*) with a call to the construct method of the Tuple class, passing the expressions *e1* and *e2* [2].

Note that the Stratego rewrite rules use concrete syntax as well. That is, the rules are transformations on Java programs and thus use the syntax of Java to indicate code fragments. However, the rules are *implemented* as transformations on abstract syntax trees. This is achieved by the same method as applied to Java programs [Visser 2002].

2.3.2 *Java with Swul*

Next we examine a serious example, i.e. the embedding of Swul, the *Swing user interface Language*, into Java. Plain Java code using the Swing API is a kind of assembly language in which intermediate results are bound to variables to

[2] We use the static factory method construct rather than the constructor of the Tuple class to work around the lack of inference of type arguments for constructors in Java 5 and 6. See also http://gafter.blogspot.com/2007/07/constructor-type-inference.html

```
module Java-Swul
imports Java-Prefixed Swul-Prefixed
exports
  context-free syntax
    SwulComponent -> JavaExpr {cons("ToExpr")}
    SwulLayout    -> JavaExpr {cons("ToExpr")}

    JavaExpr -> SwulBorder    {cons("FromExpr")}
    JavaExpr -> SwulComponent {cons("FromExpr")}
```

Figure 2.11 Syntax definition of Java with Swul; concrete syntax for user interface construction with Swing

be used elsewhere. This is illustrated in Figure 2.7. We have custom designed Swul in order to provide a compositional syntax for the construction of user interfaces with Swing in Java. Swul is just a more attractive notation for the same code. The assimilation of embedded Swul produces the sequence of statements that one would normally write by hand.

Embedding

Swul is defined as a separate language with its own syntax definition in SDF module `Swul`, introducing productions such as

```
"panel" "of" Layout                   -> Component {cons("Panel")}
"border" "layout" "{" LayoutProp* "}" -> Layout {cons("BorderLayout")}
```

to define syntax for Swing concepts such as panels and layout schemes.

The embedding of the language in Java is achieved by creating a new SDF module `Java-Swul` (Figure 2.11) that imports the syntax of Java and the syntax of Swul. The actual imported modules are prefixed wrappers of the real syntax definition, a feature of SDF further explained below.

Combining the two syntax definitions by importing them in a new module does not actually achieve the embedding of the syntax of Swul into Java. The languages are strictly separated from each other since the productions of Java do not refer to nonterminals of Swul and vice versa. The embedding of Swul in Java is achieved by *adding* productions to the combined syntax definition that allow Swul expressions to be used as Java expressions and vice versa. These productions just connect the languages at the desired location. Note that the embedding of the language is completely user-definable with an ordinary SDF module.

The kind of connection between the languages is indicated by specific constructor names. Productions that allow Swul constructs to be used as Java expressions use the constructor `ToExpr`. Similarly, productions that allow Java expressions to be used in Swul use the constructor `FromExpr`. Thus, in the embedding defined above, the `ToExpr` productions declare that Swul *component* and *layout* expressions can be used as Java expressions. The `FromExpr` productions declare that Java expressions can be used as Swul border or component expressions. Note that no special quotation symbols are needed to inject Swul expressions into Java expressions or vice versa.

```
module Java-Prefixed
imports Java
  [ ImportDec            => JavaImportDec
    CompilationUnit      => JavaCompilationUnit
    TypeDec              => JavaTypeDec
    ...
    Expr                 => JavaExpr ]
```

Figure 2.12 Syntax definition that prefixes all Java nonterminals with the name of the language.

Another point to note about the embedding is the import of the modules `Java-Prefixed` and `Swul-Prefixed` instead of `Java` and `Swul`. The `-Prefixed` SDF modules are (generated) SDF modules that import the actual syntax and rename all nonterminals in this imported definition by prefixing them with the name of the language. An example of such a module is shown in Figure 2.12. Note that these renamings do not require an actual copy of the definitions, but rather an import with a set of renamings applied.

These renamings are necessary to keep the two languages strictly separated, except for the explicitly defined connections. For example, both languages might define an `Id` or `Expr` nonterminal. If these syntax definitions are just imported directly, then there will be just one `Id` or `Expr` nonterminal. Productions using these nonterminals will then refer to the productions for these nonterminals from both languages. This embedding is not explicit and should therefore be prevented. By making the nonterminal names unique for both languages, undesired embedding is avoided[3].

Assimilation

Assimilation is again achieved via rewrite rules implemented in Stratego. The rewrite rules transform Swul expressions to Java expressions. These rules express the knowledge of the Swing API captured in the language by translating each Swul construct to the appropriate sequence of Swing method calls. Figure 2.13 illustrates this for some of the Swul constructs. Note again that although these examples are all written in concrete syntax, the actual representation that is being transformed is a structured abstract syntax tree. The Swul and Java code fragments are all syntactically checked when compiling the generator.

Most of the rewrite rules have a `where` clause. The Stratego construct <*s*> *t* => *p* applies the rewriting *s* to the expression *t* and matches the result of applying *s* to *t* to the pattern *p*. In the examples the patterns are simple variables. In this case, <*s*> *t* => *x* is comparable to an ordinary assignment of *s* applied to *t* to the variable *x*. The rewrite rules use metavariables in the code fragments. That is, `ps*`, `c`, `x`, `e` and `bstm*` are Stratego variables, used directly in the Java code fragment. These variables are bound to abstract syntax trees

[3]In Chapter 5 we improve the method of renaming nonterminals by introducing grammar mixins.

```
Swulc-Component :
  swul |[ scrollpane of c ]| -> expr |[ new JScrollPane(e) ]|
  where <Swulc-Component> c => e

Swulc-Component :
  swul |[ textarea {ps*} ]| ->
  expr |[ {|JTextArea x = new JTextArea(); bstm* |x|} ]|
  where new => x
      ; <map(Swulc-SetProp(|x))> ps* => bstm*

Swulc-AddComponent(|x) :
  swul |[ north = c ]| -> bstm |[ x.add(BorderLayout.NORTH, e); ]|
  where <Swulc-Component> c => e

Swulc-Layout :
  swul |[ grid layout {ps*} ]| -> expr |[ new GridLayout(i,j) ]|
  where <nr-of-rows> ps* => i
      ; <nr-of-columns> ps* => j

Swulc-SetProp(|x) :
  swul |[ border = b ]| -> bstm |[ x.setBorder(e); ]|
  where <Swulc-Border> b => e

Swulc-Component :
  swul |[ x: c ]| -> expr |[ {| t x = e; | x |} ]|
  where <java-type-of> c => t
      ; <Swulc-Component> c => e
```

Figure 2.13 Some of the rewrite rules for assimilation of GUI abstractions using Swing API

and are not matched literally as Java variables. The Stratego primitive `new` (used in the `where` clauses) generates a new unique identifier.

Most rewrite rules in Figure 2.13 are straightforward translations from Swul constructs to corresponding Swing API invocations. For example, the rewrite rule for a `scrollpane` directly translates into a constructor call of the `JScrollPane` class. Some constructs in Swul provide even more abstraction than an alternative syntax. For example, the rewrite rule for the `grid layout` calculates the number of rows and columns. The rewrite rule for the Swul construct `x : c`, the last rule in the figure, shows another abstraction; if possible, it determines the type of the component and declares a variable `x` of this type with the initial value `c`. This inline declaration is very useful since Swul does not cover the full Swing API, but just the most common constructs. If unusual things have to be configured for a component, then this can be done later without ruining the compositional definition of the user interface.

The transformation makes use of a convenience extension of Java with expression blocks, which are removed by a separate transformation. In fact the compositional nature of Swul (and other extensions) is based on this extension. Constructing Java objects is composable in Java itself as long as all required operations can be performed by invoking a single method or constructor. As soon as further method calls are required there is a problem in composing objects without binding them to intermediate variables. The ex-

```
context-free syntax
  Stm -> JavaStm {cons("ToStm")}

  JavaId "." "write"       "%>" Content  "<%" ";" -> Stm {cons("Write")}
  JavaId "." "write" "doc" "%>" Document "<%" ";" -> Stm {cons("Write")}

  "<%" JavaStm  "%>" -> Content              {cons("FromStm")}
  "<%" JavaExpr "%>" -> DoubleQuotedPart     {cons("FromExpr")}
  "<%" JavaExpr "::" "cdata" "%>" -> Content {cons("TextFromExpr")}
```

Figure 2.14 Syntax definition of Java with XML

tension of Java with expression blocks solves this problem introducing two expression: `{| bstm* | e |}` and `{| e | bstm* |}`. An expression block is thus a list of block-level statements followed by an expression or the other way around. The expression is the value of the expression block. In the first case the statements are lifted to the block-level statements *before* the context expression. In the second case they will be lifted to block-level statements *after* the context expression. The syntax of this extension is defined by two SDF production rules and the constructs are translated into ordinary Java by a small Stratego program[4].

The expression block extension makes the definition of assimilation, and code generators in general, much simpler. For example, the declaration of a `textarea{ps*}` is translated to a declaration of a new variable `x` of type `JTextArea`, which is instantiated to a new `JTextArea`. The instantiation of the properties of the textarea is achieved with additional statements, which are executed after creating the textarea. The context in which the `textarea` was placed receives as value the variable `x`. Thus a linear sequence of statements building the components of the user interface is realized.

2.3.3 *Java with XML*

Our next example is the embedding of XML in Java. The application of this embedding was illustrated in Figure 2.4.

Embedding

Some of the production rules of the syntax embedding are shown in Figure 2.14. These production rules are somewhat different from usual embeddings, since the embedded XML syntax is translated to the SAX API. XML construction in SAX is not done using expressions and objects, but by invoking methods of the `ContentHandler` interface. Typically, the methods of a `ContentHandler` are invoked to report parsing events of an XML parser as callbacks to an ap-

[4]Kats [Kats 2007] observed that this transformation is not always possible to implement by trivially lifting the statement to the first surrounding statement level, for example if the expression block is used in a loop condition. The work of Kats proposes a new kind of open compiler based on mixing Java source and bytecode, which makes expression blocks trivial to implement as a language extension by leveraging the greater flexibility of Java bytecode.

plication. However, any code can use the `ContentHandler` interface to report the content of an XML document.

The embedding extends Java with syntax for writing XML content to a `ContentHandler`. The `ContentHandler` instance is specified by a Java identifier. The `ToStm` and `Write` constructors are used to represent the switch from Java to XML. The `Write` constructor has two arguments: a Java identifier of the `ContentHandler` and the XML content to write to it.

In addition to the embedding of XML in Java, we also define production rules for escaping from XML back to Java. First, the escape from XML content to Java is represented by the `FromStm` constructor. This escape consists of a Java statement, which might be surprising, since an escape to the host language usually is an *expression*. However, the SAX `ContentHandler` produces XML content by statements, not by expressions. Second, the embedding defines an escape from XML attribute values (`DoubleQuotedPart`) to Java. Third, the embedding defines a more specific anti-quotation for character data, using the constructor `TextFromExpr`. The result of the Java expression must be a String, which is emitted to the `ContentHandler`.

Assimilation

Most of the Stratego rewrite rules for the assimilation of the embedded XML fragments into Java are shown in Figure 2.15. For a single XML construct the resulting code fragments are now larger than in the previous examples. Hence, the XML syntax of this embedding is a major abstraction from the API interface provided by SAX. In this assimilation, the left-hand side of the rewrite rules are not written in concrete XML syntax, but rather in abstract syntax. The left-hand side patterns are so small that using concrete syntax would only be confusing. This illustrates that programs using concrete syntax can fall-back to the assimilated notation in the host language (in this case abstract syntax for XML) when this is more appropriate.

Figure 2.15 shows the rewrite rules for XML documents, elements, text and character data escapes to Java. The translation of `Document`[5] and `Text`[32] is straightforward. The translation of an `Element`[13] is somewhat more involved, since the list of attributes has to be assimilated into an instance of the `AttributesImpl` class. Character data escapes to Java are translated using expression blocks[26], as discussed before. This translation to an expression block is necessary because we need the value (bound to *y*) at multiple locations; for calculating the length of the string and for translating it into a character array that is passed to the `ContentHandler`. Note that the rules use several metavariables: *x*, *y*, *bstm1**, *bstm2**, and *e**. Also, the translation of an XML element uses a special construct *inside* a string literal: `"~n"`. The `~` construct denotes an escape to the host language, which is Stratego in this case. Anti-quotation and metavariables inside such lexical constructs are a unique feature of SDF and SGLR. This is possible by preserving the structure of lexemes, which will be discussed in more detail in Section 2.4.

```
1  explode-write :
2    ToStm(Write(Id(x), c)) -> bstm |[ { bstm* } ]|
3    where <content-to-stm(|x)> c => bstm*
4
5  content-to-stm(|x) :
6    Document(c) -> bstm* |[
7       x.startDocument();
8       bstm1*
9       x.endDocument();
10   ]|
11   where <content-to-stms(|x)> c => bstm1*
12
13 content-to-stm(|x) :
14   Element(Name(None(), n), atts, kids) -> bstm* |[
15      org.xml.sax.helpers.AttributesImpl y
16        = new org.xml.sax.helpers.AttributesImpl();
17      bstm1*
18      x.startElement("", "~n", "~n", y);
19      bstm2*
20      x.endElement("", "~n", "~n");
21   ]|
22   where <map(content-to-stms(|x))> kids => bstm2*
23       ; new => y
24       ; <map(attr-to-stm(|y))> atts => bstm1*
25
26 content-to-stm(|x) :
27   TextFromExpr(e) -> bstm |[
28     x.characters({| String y = e; | y.toCharArray() |},0,y.length());
29   ]|
30   where new => y
31
32 content-to-stm(|x) :
33   Text(s) -> bstm* |[ x.characters(new char[]{e*}, 0, i); ]|
34   where <explode-string> s => cs
35       ; <length; int-to-string> cs => i
36       ; <map(escape-char)> cs => e*
```

Figure 2.15 Rewrite rules for assimilation of XML to SAX ContentHandler invocations.

2.3.4 *Java with Java*

Lastly, we return to our first example in Section 2.2: Java embedded in Java. This combined language is called JavaJava. JavaJava is a language for Java metaprogramming, i.e. Java programs that generate or manipulate Java programs. The embedding and assimilation of JavaJava are defined with the MetaBorg method. The implementation involves the definition of an embedding of Java in Java, and an assimilation of the embedded Java fragments into the host language.

Embedding

A selected number of productions from the embedding of Java in Java are shown in Figure 2.16. The complete embedding is much larger, since the Java syntax definition contains many nonterminals. For each nonterminal that is to be quoted or anti-quoted in JavaJava, productions have to be defined in the

```
context-free syntax
         "⟦" Expr "⟧" -> JavaExpr        {cons("ToExpr")}
  "expr" "⟦" Expr "⟧" -> JavaExpr        {cons("ToExpr")}

  "type"  "⟦" Type      "⟧" -> JavaExpr  {cons("ToExpr")}
  "bstm*" "⟦" BlockStm* "⟧" -> JavaExpr  {cons("ToExpr")}

  "~"  JavaExpr ->  Expr                 {cons("FromExpr")}
  "~*" JavaExpr -> {Expr ","}*           {cons("FromExpr")}
  "~*" JavaExpr -> {VarInit ","}*        {cons("FromExpr")}
  "~*" JavaExpr -> {FormalParam ","}*    {cons("FromExpr")}
  "~*" JavaExpr -> ClassBodyDec*         {cons("FromExpr")}

variables
  "e"    [0-9]*     -> Expr             {prefer}
  "t"    [0-9]*     -> Type             {prefer}
  "e"    [0-9]* "*" -> {Expr ","}*      {prefer}
  "e"    [0-9]* "*" -> {VarInit ","}*   {prefer}
  "bstm" [0-9]* "*" -> BlockStm*        {prefer}
  [ij]   [0-9]*     -> DeciLiteral      {prefer}
  [xyz]  [0-9]*     -> Id               {prefer}
```

Figure 2.16 Syntax definition of Java with Java

combined syntax definition. The JavaJava syntax definition applies some of the more advanced features in SDF for combining the syntax of the embedded language and the host language.

The first two productions show that the declaration of nonterminals of concrete syntax fragments is optional. To this end, there is a production rule for embedding Java expressions using just the quotation symbols ⟦ and ⟧ and there is a production with an explicit declaration `expr` of the nonterminal. The embedding of `BlockStm*` shows that lists of nonterminals, in this case `BlockStm`, can be embedded as well. The tags `expr`, `type`, and `bstm*` are necessary for disambiguation. In Chapter 3 we will discuss a method for avoiding these tags by employing a disambiguating type-checker. In Chapter 4 we use an alternative, more lightweight method of runtime disambiguation.

The escapes to the meta level use the ~ symbol in this embedding. That is, meta Java expressions can be included by using only a ~ before the meta expression. The `~*` escape can be used to escape to a Java expression that produces a list of a certain nonterminal, even if they are separated by tokens. The `~*` escape is defined by an SDF production rule that produces a list nonterminal (the right-hand side of the production rules). For example, the escape for `{Expr ","}*` is used in method invocations, which take a number of expressions separated by commas. In the argument list of a method in concrete syntax it is possible to escape to the meta level using `foo(~*args)`, but even `foo("bar",~arg,~*args)` and `foo(~*args1,~*args2)` are allowed. All these escape are defined by a single production rule in the embedding of Java in Java. No additional SDF productions are required, since productions that produce list symbols are desugared to a set of more basic productions that are sufficient to handle all these combinations.

JavaJava also provides metavariables, which is an even more compact method for embedding variables from the meta language in embedded code fragments. The SDF syntax definition formalism has been designed for application in metaprogramming and therefore it has the built-in notion of metavariables. Metavariables are defined in a `variables` section. This section consists of SDF productions that define a special meaning for certain identifiers. For example, the first production in the `variables` section of Figure 2.16 defines that the identifier `e` is a metavariable denoting a Java expression. Some other metavariables in JavaJava are `e*` for lists of expressions separated by commas and `x`, `y`, and `z` for identifiers.

Assimilation

The MetaBorg examples of Java with Swul and Java with XML apply assimilations that are specific to the host language *and* the embedded language. The assimilation cannot be reused for different embedded languages since the assimilator rewrites specific constructs of the embedded language to the host language. Such a specific assimilator can incorporate domain-specific knowledge, which adds a level of abstraction to the syntactic embedding. For example, the Swul assimilator calculates the number of rows and columns of a GridLayout.

However, for some applications it is possible to implement a *generic* assimilation of an arbitrary embedded language to an API in the host language. Hence, the assimilation can be reused for the embedding of *any* language in the host language. For instance, generic assimilation is possible for assimilating embedded languages in metaprogramming. Metaprogramming frameworks often follow a certain standard procedure for representing a subject program in a class hierarchy. Examples of such frameworks are JJForester [Kuipers & Visser 2001], ApiGen [de Jong & Olivier 2004], Java Tree Builder [Tao et al.] with JavaCC [JavaCC Website], and SableCC [Gagnon & Hendren 1998]. For a specific metaprogramming framework the mapping from a language definition to a class structure is fixed or at least reproducible. This fixed translation scheme can be used for the implementation of a generic assimilator. Similarly, a generic assimilator is also possible if the target API itself is generic. This is the case if the embedded language is to be represented using a generic class hierarchy for trees, such as ATerms [van den Brand et al. 2000] or XML.

The generic assimilation from an embedded language to the construction of a generic tree representation can be implemented particularly well in a language that supports generic programming. To illustrate the implementation of a generic assimilation we have implemented an assimilation from the embedded Java code in the JavaJava language to the ATerm library. A generic assimilation requires generic programming. Stratego allows generic term construction and deconstruction using the `#` operator. A term `foo(bar(), fred())` can be deconstructed into the constructor of the term, `foo`, and a list of children to which the construct is applied, `bar()` and `fred()`. The pattern

```
AssimilateAppl(rec) :
  fun#([t*]){} -> expr |[ _factory.makeAppl(e, e*) ]|
  where <AssimilateAFun> (fun, <length> t*) => e
      ; <map(rec)> t* => e*

AssimilateAFun :
  (fun, arity) -> expr |[ _factory.makeAFun("~fun", i, false) ]|
  where <int-to-string> arity => i

AssimilateMetaVar :
  meta-var(x) -> expr |[ x ]|

AssimilateInt :
  i -> expr |[ _factory.makeInt(j) ]|
  where <int-to-string> i => j

AssimilateString :
  s -> expr |[ _factory.makeAppl(_factory.makeAFun("~s", 0, true)) ]|
  where <is-string> s
```

Figure 2.17 Generic assimilation of abstract syntax trees in Java with ATerms

`fun#([t*])` binds the constructor name to the variable `fun` and the list of children to the variable `t*`.

The generic assimilation of ATerms to Java is implemented by the Stratego rewrite rules shown in Figure 2.17. The rewrite rules `AssimilateAppl` and `AssimilateAFun` handle ATerm constructor applications and use generic term deconstruction. The right-hand sides of the rewrite rules are Java expressions. For the construction of ATerms in the resulting Java code, methods defined in the `ATermFactory` interface of the ATerm library are invoked. The rules `AssimilateInt` and `AssimilateString` handle the ATerm integer and string constructs. The `AssimilateMetaVar` rule rewrites metavariables to real Java variables.

This generic assimilation can be applied to all embedded languages in the host language Java. Hence, an embedding of a different subject language only needs to define the combined syntax definition.

API specific assimilations for the *same* JavaJava language can be implemented as well. For example, an assimilator could target the class hierarchy for Java abstract syntax trees in Eclipse.

2.4 SYNTAX DEFINITION

The embedding of languages poses some challenges on syntax definition and parser technique. MetaBorg employs the syntax definition formalism SDF [Visser 1997b] and SGLR [Visser 1997a, van den Brand et al. 2002], a Scannerless Generalized LR parser.

In the previous section we described several MetaBorg embeddings. So far, we have not revealed how these combined languages are actually parsed and what the features of the syntax definition formalism SDF are. Full insight in why SDF is the appropriate syntax definition formalism for defining lan-

guage embeddings requires more detailed knowledge of SDF. In this section, the SDF syntax definition formalism is introduced and we show how it is applied to achieve concrete syntax for objects. Several features of the syntax definition formalism allow the concise definition of the syntax of embeddings. Two features of this parsing technology are essential for defining the syntax of language embeddings: the *GLR* algorithm [Tomita 1985, Rekers 1992] and *scannerless* parsing [Salomon & Cormack 1989]. These features are combined in SGLR [Visser 1997a, van den Brand et al. 2002].

2.4.1 *SDF Overview*

The SDF syntax definition formalism supports concise and natural expression of the syntax of context-free languages. SDF integrates lexical and context-free syntax in a single formalism. The complete syntax of a language is thus defined in a single definition. SDF supports the entire class of context-free grammars. SDF does therefore not restrict the grammars to a subclass of the context-free grammars, such as LL or LALR. SDF syntax definitions can be split into modules, and SDF modules can be reused in different syntax definitions. Disambiguation of grammars is not done by grammar hacking, but by applying special purpose disambiguation facilities in SDF, such as priorities, reject productions, and follow restrictions. To illustrate the key features we explain a syntax definition of expressions. For more documentation and examples we refer to [SDF Website, Visser 1997b].

Context-Free and Lexical Syntax

Syntax is defined in syntax sections, which are either *context-free* or *lexical*. The difference between these two kinds of syntax sections is that in lexical syntax no layout is allowed between symbols. In context-free syntax sections layout is allowed between the symbols of a production. The term *context-free syntax* should not be confused with the expressiveness of the production rules. The expressiveness of the lexical and context-free syntax sections is not different: lexical syntax is not restricted to a regular grammar. In fact, lexical and context-free syntax are even translated into a single core syntax definition formalism [Visser 1997b].

Lexical syntax sections are used to define constructs such as identifiers, layout, comments, and literals. Layout is a special nonterminal in SDF named `LAYOUT`. A symbol for this nonterminal is inserted between the symbols of productions in a context-free syntax section to allow layout between the symbols there. The following lexical syntax section defines layout (`LAYOUT`), identifiers (`Id`), and integer constants (`IntConst`).

```
lexical syntax
  [A-Za-z][A-Za-z0-9]* -> Id
  [0-9]+               -> IntConst
  [\r\n\t\ ]           -> LAYOUT
```

These lexical sorts can be used in a context-free syntax section to define the more complex constructs of a language. The context-free syntax of the example language consists of variables, integer literals, arithmetic operations,

method invocations and conditional expressions. Note that the definition of expressions is concise, without the hurdles of introducing additional nonterminals to handle priority and associativity of operators.

```
context-free syntax
  Id          -> Exp {cons("Var")}
  IntConst    -> Exp {cons("Int")}

  Exp "+" Exp -> Exp {cons("Plus")}
  Exp "-" Exp -> Exp {cons("Min")}
  Exp "*" Exp -> Exp {cons("Mul")}
  Exp "/" Exp -> Exp {cons("Div")}

  Exp "." Id  "(" {Exp ","}* ")" -> Exp {cons("Call")}
  "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
```

The productions are annotated with a constructor name (`cons("...")`). These constructor names are used to construct an abstract syntax tree from the parse tree that results from parsing an input text. We use the ATerm format [van den Brand et al. 2000] for the representation of parse and abstract syntax trees. The ATerm format is somewhat comparable to XML, but is more structured. It supports lists, tuples, integers, string literals, and of course constructor applications. For example, the input `4 + 1.get(5)` is represented by the ATerm:

```
Plus(Int("4"), Call(Var("1"), "get", [Int("5")]))
```

Disambiguation

SDF supports the full class of context-free grammars, including ambiguous grammars. The implementation of SGLR is able to produce a parse forest of all possible parse trees, but obviously we would like to define what parse trees are preferred over others. For the purpose of disambiguation SDF allows separate disambiguation filters instead of hacking the syntax definition itself into a non-ambiguous form. Separate priority definitions, follow restrictions, reject, avoid, and prefer filters can be used to disambiguate a syntax definition [van den Brand et al. 2002].

The syntax definition above is highly ambiguous for several reasons. First of all there is an *associativity* problem for the binary operators. The input `1 + 2 + 3` can be parsed as either

```
Plus(..., Plus(...,...))   or   Plus(Plus(..., ...), ...)
```

Since there is more than one possible interpretation, parsing the input will result in a parse forest. The ambiguous phrases are represented by `amb` nodes in the parse tree. Production rules in SDF can be annotated with `right`, `left`, `assoc` or `non-assoc` to define the associativity of an operator. In this case the `+` operator is left associative, therefore the new production rule is:

```
Exp "+" Exp -> Exp {left, cons("Plus")}
```

Another problem is the *priority* of operators. The input `1 + 2 * 3` can be parsed as

```
Plus(..., Mul(..., ...))   or   Mul(Plus(..., ...), ...)
```

A related problem is the input `4 + x.get()`. Although it might seem apparent that the method `get` must be invoked on the variable `x`, this is not the only possible parse. The input can be parsed as

```
Plus(..., Call(..., ..., ...))   or   Call(Plus(..., ...), ..., ...)
```

These ambiguities can be solved by defining priorities of production rules. Priorities in SDF are defined *relatively* and not by defining priority levels. The priority of `*` is usually higher then the priority of `+` and the priority of the method call is higher than all operators in our example language. Since such priorities are not properties of a single production, there is a separate section in an SDF module for defining priorities. Production rules in priority definitions can also be grouped, which means that their priority is equal. In this case the associativity of such a group should be defined. If `-` and `+` are in the same priority group, then the associativity of the group determines how for example the input `1 - 2 + 3` is parsed. The resulting priority definition is:

```
context-free priorities
    Exp "." Id  "(" {Exp ","}* ")" -> Exp
  > {left:
      Exp "/" Exp -> Exp
      Exp "*" Exp -> Exp
    }
  > {left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp
    }
```

These priorities solve all ambiguity problems in the example. There is still one problem left, but it is not an ambiguity problem in this example. The syntax definition is too liberal: identifiers are not recognized in a greedy way, which means that the input `if 1 then aelse b` is valid. Also, the syntax definition does not forbid `if 1 thena else b`, since the restriction that keywords cannot be immediately followed by an identifier character has not been expressed. The syntax definition must exclude these two input texts by recognizing identifiers and integer literals in a greedy way and by disallowing a keyword to be followed by an identifier character. The longest match policy is implicit in most parser generators, especially in those with a separate scanner. In SDF a longest match restriction can be defined in the syntax definition itself by using a *follow restriction*. A follow restriction `A -/- CC` forbids that the nonterminal `A` is followed by a character from the character class `CC`.

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
  "if" "then" "else" -/- [A-Za-z0-9]
```

2.4.2 *The Importance of Modularity*

Modularity is essential for the definition of the syntax of language embeddings. To make the embedding of the syntax of a language A in a host language B concise and maintainable, it must be possible to develop the syntax

definitions of language A and B independently from each other. Modularity in the definition of language embeddings becomes even more important if more than one language needs to be embedded. For example, XML and SQL, XPath and XML, Java and XML. Defining these embeddings in a single syntax definition is unacceptable from the point of view of clarity, maintenance, and reusability.

Most syntax definition techniques that are used in practice are limited to some subset of the context-free grammars, since they target a parser generator that applies a certain restricted parsing algorithm. This is illustrated by the fact that most syntax definition languages are coupled with a parser generator implementation. Depending on the parsing algorithm that the generated parser is using, the syntax definition is restricted to a certain subclass of the context-free grammars, such as LALR for YACC, CUP and SableCC, or LL for ANTLR and JavaCC. Restricting syntax definitions to some proper subclass of context-free grammars is to a certain extent acceptable for generating parsers from monolithic syntax definitions of single programming languages, but it is not for combining programming languages, for example in the embedding of languages in host languages.

An interesting formal result is that there is no proper subclass of the context-free grammars that is closed under union. As a consequence, grammar languages that are restricted to some proper subset of the context-free languages cannot be modular since the combination of two, for example LR, syntax definitions is not guaranteed to be in this subset. Therefore, SDF supports the full class of context-free grammars.

Supporting the full class of context-free grammars in a syntax definition formalism introduces some challenges to the applied parsing technology. Parsers and parser generators that allow the full class of context-free grammars apply Generalized LR (GLR) [Tomita 1985, Rekers 1992] or Earley [Earley 1970] parsing. The GLR parsing algorithm maintains multiple LR parsing states in parallel. At phrases where the parser cannot choose from multiple production rules to apply, it forks the current parser. The forked parsers meet again if a token in the input is to be parsed with the same nonterminal.

The modularity features of SDF are comprehensive. Since using modular syntax definition in SDF introduces not a single awkwardness, defining syntax definitions in a modular way is even encouraged. Renamings make the combination *and* isolation of syntax definitions very flexible as explained in Section 2.3.2. SDF is modular to the core.

2.4.3 *The Importance of Scannerless Parsing*

Most parsers apply a separate scanner for lexical analysis. The purpose of this lexical analysis phase is to break up the input into tokens, such as identifiers, numbers, layout, and specific keywords (e.g. `if`, `switch`, `try`, `catch`). That is, the lexical analysis phase decides what kind of token a lexeme is. A separate lexical analysis phase allows the parser to operate on the list of lexemes, ignoring their actual contents. The lexical syntax of a language is

usually specified by regular expressions. The scanner applies finite automata to recognize the tokens specified by these regular expressions.

Context of Tokens Scanners that do not interact with the parser cannot consider the context of a sequence of characters in the decision which token a sequence represents. This is essentially the reason for having reserved keywords, which cannot be used as identifiers in the language. The separate lexical analyzer is also the reason for not allowing nested block comments (`/* /* */ */`) and for disallowing `a -- b` to be parsed as `a - (-b)`. Thus, the lexical analyzer decides what token a list of characters represents, without considering the context of the tokens in the input stream.

This may be reasonable for parsing inputs written in a single language, in particular since this language can be designed with these restrictions in mind. Yet, it is more problematic when the input consists of a combination of languages. By not considering the context of a sequence of characters in the input stream, it is impossible to assign it different tokens, since this ultimately depends on the context of the lexeme. As regards embedded languages, the scanner cannot assign different token types to the same lexeme in embedded code and in host code. This is a serious problem if the lexical syntax of the embedded language is entirely different from the lexical syntax of the host language. It is not a problem if the embedded language is the same language as the host language (for example Java embedded in Java), since the lexical syntax is in this case not different in the embedded code fragments. However, embedding XML in Java is more challenging, since the lexical syntax of XML is completely different from Java. In this case the lexical analyzer must consider the context of a sequence of characters before deciding what kind of token it is.

Preserve Structure of Lexemes A scanner usually passes lexemes as an unstructured atomic value to the parser. The internal structure of the tokens is thus not preserved. This is a pity since tokens such as floating point and String literals can have quite a complex structure. The string literal `"Hello\r\n world!"` will typically result in the token `"\"Hello\\r\\n world!\""`. The semantic tools that operates on parse (or abstract syntax) trees must analyze the structure of the tokens again to determine its meaning. However, preservation of the structure of a lexeme is essential if it must be possible to escape to the metalanguage *inside* a token. We have already seen such escapes in our examples, namely the escape in XML attribute values and Java string literals (see Section 2.3.3). Parsing with a separate scanner does not support such structured lexemes, since lexemes are passed as a plain sequence of characters to the parser. In *scannerless parsing* the structure of lexemes can be preserved. For example, in our Java tools the string literal above is represented as:

```
String([
  Chars("Hello"), NamedEscape(114), NamedEscape(110), Chars(" world!")
])
```

The preservation of lexical structure is mostly a matter of convenience if parsing inputs of single programming languages. However, for the embedding

of a language in a host language, the preservation of lexical structure is extremely useful, if not a requirement. For example, the following XML attribute contains an escape to the meta level inside an XML attribute:

```
<a href="http://www.<% s %>.org">Stratego/XT</a>
```

In scannerless parsing the structure of the attribute value can be preserved. In the embedding of XML in Java, the attribute is represented as:

```
Attribute(
  QName(None, "href")
, DoubleQuoted([
    Literal("http://www.")
  , Literal(FromExpr(ExprName(Id("s"))))
  , Literal(".org")
  ])
)
```

Note that preserving the structure of lexical syntax is related to the context of tokens. The content of a string literal is in a sense an embedded language. Scanning string literals could result in separate tokens of the lexical syntax of this language. This requires the scanner to know about the context of a sequence of characters. After all, the lexical syntax of this 'embedded' string literal language is completely different from the lexical syntax of the host language.

Solutions Of course these problems can all be solved in hand-written scanners. Such a scanner can interact with the parser in arbitrary ways. However, writing efficient parsers and scanners by hand that can handle ambiguous input streams is very difficult. Writing scanners and parsers by hand is in general not an option if languages need to be combined.

The solution is to discard the separate lexical analysis phase completely. The parser directly operates on the characters of the input. In syntax definitions for scannerless parsers the lexical syntax and context-free syntax can be specified in a single syntax definition formalism. Hence, the full syntax of a language is specified in a single definition. All information regarding the disambiguation can be in the same syntax definition and there are no implicit disambiguation rules based on the parser technology that is applied. Since the parser operates on individual characters the context-free analysis of the parser can be used to determine the types of individual characters. Thus, the context of the character is taken into account. If a certain token is not expected at a certain location in the input, then it will not be considered. Parsing without a separate scanner is called scannerless, a term that was coined in [Salomon & Cormack 1989]. This solution combined with the GLR algorithm is known as Scannerless Generalized LR parsing [Visser 1997a]. The parser generator for SDF (pgen) generates a parse table for a Scannerless Generalized LR parser (sglr). The implementation is available at [ASF+SDF MetaEnv, SDF Website].

2.5 PREVIOUS WORK

SDF [Heering et al. 1989] was originally designed for use as a general syntax definition formalism. However, through its implementation it was closely tied to the algebraic specification formalism ASF+SDF [van Deursen et al. 1996], which is supported by the ASF+SDF Meta-Environment [van den Brand et al. 2001]. Redesign and reimplementation as SDF2 [Visser 1997b, van den Brand et al. 2002] has made the language available for use outside the Meta-Environment. SDF2 is also distributed as part of the Stratego/XT bundle of program transformation tools [Visser 2004, de Jonge et al. 2001]. Syntax definition in SDF2 is limited to *context-free grammars*. This is a limitation for languages with context-sensitive syntax such as that of Haskell (offside rule). However, in the setting of embedded concrete syntax, in which small fragments are used and not all context is always available, any parsing technique will have a hard time. The combination of SDF with ASF was the first to use SDF for the definition of embedded syntax.

Stratego [Visser et al. 1998, Visser 2004, Stratego Website] is a language for the implementation of program transformation based on rewrite rules under control of programmable rewriting strategies. Rewrite rules transform first-order terms. In [Visser 2002] an extension of Stratego with concrete syntax for the program fragments manipulated by its rewrite rules is introduced. The paper presents a single *generic* assimilation transformation, mapping term representations of abstract syntax trees into Stratego terms, similar to the assimilation of Java programs to ATerms in Section 2.3.4. The paper also gives an outline of a general architecture for extension of an arbitrary host language with concrete syntax.

In [Fischer & Visser 2004] the approach is extended to Prolog as a metalanguage, mainly to provide concrete syntax for the schemas of the AutoBayes program synthesis engine. Since Prolog is also a term-based language the assimilation is defined in a similar generic way.

The contributions of this chapter are the extension of the approach to an object-oriented host language, the targeting of specific APIs by the assimilation transformation, and the specific extensions of Java for code generation, XML generation, and user interface construction. The design of SWUL and its embedding in Java is a nice by-product of this project.

2.6 RELATED WORK

There have been many other approaches to make programming languages syntactically extensible. We give an overview of such approaches and discuss their relation to MetaBorg.

2.6.1 *Extensible Syntax*

The extensible syntax [Cardelli et al. 1994] applied in the implementation of $F_{<:}$ [Cardelli 1993] was the first step into incremental syntax definition by

extending and restricting a small core language. The definition of syntax extensions and the assimilation into the host language are in the source code of the host language. Syntax extensions can be local (`syntax ... in ... end`) or defined at top-level. The syntax can be defined incrementally by updating, extending, or adding production rules to an existing grammar. The syntax extensions include a rewriting to the host language by constructor applications or action definitions that rewrite the syntax directly to the target language.

The implementation of extensible syntax is based on LL parsing, which makes the syntax definition cumbersome and not modular, since the class of LL grammars is not closed under union and concatenation. It would be interesting to develop a comparable inline extensible syntax mechanism based on the more powerful parsing technique of scannerless generalized LR parsing, which was not mature enough when extensible syntax was developed. The latest work in this direction is the extensible generalized LR parser Dypgen [Onzon 2007] for Objective Caml. Dypgen supports extensible syntax, but the lexical syntax of the base language cannot be extended. Also, modifying the syntax triggers a reconstruction of the full parse table. Chapter 6 addresses this issue using *parse table composition*. Integration of parse table composition in a runtime extensible parser is future work.

Although the implementation of user-definable syntax in this work is very flexible (despite the LL parsing), we do not believe that there is a need for defining *ad-hoc* syntax extensions so powerful. Rather, in practice syntax extensions are carefully designed in a separate role. Developing appropriate domain-specific syntax extensions of a host language is thus a separate role and does not necessarily need to be performed in the host language. The implementation can be provided in separate tools. The additional advantage of these separate tools is that the metaprogrammer can choose the most appropriate programming language for implementing an assimilation. The MetaBorg method aids the implementation of these tools by providing powerful and generic tools for parsing and assimilation into the host language.

2.6.2 *Harmonia's Blender*

Blender [Begel & Graham 2004], developed in the Harmonia project, targets parsing inputs that cannot be designed to be non-ambiguous. Begel and Graham [Begel & Graham 2004] thoroughly analyze the problems in handling ambiguous inputs, especially embedded languages. Blender generates from Flex- and Bison-like definitions a lexical analyzer, parse tables and class definitions for representing abstract syntax trees in C++.

Blender applies GLR parsing with an incremental lexical analyzer that is forked together with the LR parsers in the GLR algorithm. The scanner is thus not completely separated from the parser, but parsing is not scannerless. If multiple lexical types can be assigned to a token, then the scanner reports all possible interpretations to the parser. For each interpretation of a token a parser and scanner is forked. This approach was also used by the implementation of SDF before the introduction of scannerless parsing [Heer-

ing et al. 1989]. The approach was abandoned because of huge numbers of forks and the increased complexity of the algorithm when trying to reduce the number of forks. In addition, the use of context-free grammars instead of regular grammars greatly increases expressivity, allowing nested comments and anti-quotation within lexical syntax, for example.

GLR parsing where the scanner is forked together with LR parsers should solve most issues with using a separate scanner. We have not been able to investigate whether this is really a performance improvement over Scannerless Generalized LR parsing, since no benchmarks are provided and Blender is not available for download. The main argument for not using scannerless parsing in the Harmonia project is to facilitate interactive environments using incremental parsing. In our view parse trees of scannerless parsing can be persisted as well and be reused to reparse just the edited parts. Begel and Graham suggest this as well and remark that the size of the parse trees might be a problem. However, the efficient ATerm format [van den Brand et al. 2000] has already proven to be successful in reducing the size of parse trees through maximal sharing.

Syntax definitions of Blender are modular, thus languages can be developed and maintained independently. Unfortunately, the syntax definition formalism used in Blender is much less concise than SDF. The lexical analyzer applies longest match and order-based matching. Although this is appropriate for most programming languages, we believe that implicit disambiguation by the parser should be avoided. Order-based selection is especially problematic if the lexical syntax is defined in several modules, as is typically the case when defining the syntax of an embedded language. The built-in preference for the longest match and tokens defined first, requires insight in the algorithms applied by the lexical analyzer generator. This makes debugging of the syntax definition more complex and requires studying traces of the parser. The declarative disambiguation methods of SDF allow the complete definition of the syntax of a language, independent from the tools applied to the definition.

2.6.3 *Jakarta Tool Suite (JTS)*

The Jakarta Tool Suite (JTS) [Batory et al. 1998] is a set of tools for extending Java with domain specific constructs. The main tools of JTS are Jak, which allows metaprogramming for Java in Java using the Java concrete syntax, and Bali, a frontend of JavaCC [JavaCC Website]. JTS targets the implementation of component-based generators, called GenVoca generators.

The Jak language of JTS supports the generation of Java programs by using the concrete syntax of Java in Java. Code fragments are embedded in tree constructors (for example `exp{...}exp`). From these fragments it is possible to escape to the meta language by an escape such as `$exp(...)`. The Jak language is comparable to our JavaJava language, but since our embedding is entirely user-definable and easy to extend, JavaJava supports a wider range of quotations (tree constructors) and anti-quotations (escapes). In Java-

JAVA the explicit declaration of nonterminals in tree constructors and escapes is not required if the nonterminal of code fragments can be determined by the parser. This makes code fragments much more concise. We would like to remove even more nonterminal declarations by disambiguating the code fragments in an extended type checker for Java. This future work is discussed later. In addition to anti-quotations, JAVAJAVA also supports *metavariables* which is a very compact embedding of variables from the meta language in the generated code fragments. Both solutions guarantee the syntactic correctness of the code fragments and construct an abstract syntax tree representation of the generated code.

Jak also features a matching facility. We have not implemented this yet in JAVAJAVA, because there is no easy mapping of match statements to existing APIs or language constructs in Java. The Tom pattern matching compiler [Moreau et al. 2003] could be used to implement support for patterns in concrete syntax.

JTS Bali is a tool for generating tools for extensions of Java. Bali is a front-end of JavaCC. Bali generates abstract syntax tree definitions and a parser by passing a grammar to JavaCC. Relying on JavaCC parsing technology introduces a lot of problems. Syntax definitions are much less concise compared to SDF and even to other available parser tools. JavaCC is an LL(k) parser generator, which implies that grammars are not composable and have to be manipulated to solve ambiguities. This parsing technology is appropriate for parsing languages that are designed not to be ambiguous, but in a toolkit for defining language extensions it is a major issue, as has been discussed in Section 2.4.

Bali composes grammars by combining the lexical and grammar rules. This merging is not guaranteed to succeed since LL(k) grammars are not composable. Bali uses heuristics to combine lexical rules such that the best results are produced in the common cases. Lexical scanners prefer the longest match and select rules that apply by order. Therefore, keyword rules are put before the more general rules, such as for identifiers. This implies that keywords for the embedded language may cancel use of those keywords as identifiers in the host language. The syntax definition and parsing techniques of SDF are much easier to use and can be applied for arbitrary embeddings. Because of the parser technology which Bali relies on, Bali will not be able to handle syntactic extensions that have an entirely different lexical syntax. SDF has no problem such embeddings, as discussed in Section 2.4.

2.6.4 *Syntax Macros*

Syntax macros [Leavenworth 1966] define syntactic abstractions over code fragments. Syntax macros usually operate on abstract syntax tree representations of programs. A syntax macro accepts abstract syntax tree arguments and generates a new abstract syntax tree that replaces the syntax macro invocation. To ensure the production of syntactically correct programs, the resulting

abstract syntax tree of a syntax macro invocation must be of the same type as the macro invocation.

In all implementations of syntax macros, a macro invocation must start with a unique macro delimiter, which is usually an identifier consisting of letters. This identifier indicates the syntax macro that is invoked. Such a macro delimiter is required since the syntax of macro invocations is defined in the input file itself. Some syntax macro implementations allow overloading of these identifiers to refer to different syntax macro definitions using the same identifier. The fixed syntax for a macro invocation is in many cases acceptable. The fixed syntax for invocations can even result in attractive, data-like, programs (called a Very Domain-Specific Language, VDSL, in [Brabrand & Schwartzbach 2002]).

However, the fixed invocation syntax of syntax macros is a limited extension of the syntax of the host language. The embedding of a domain-specific language in MetaBorg can be an arbitrary context-free language. The expressivity of our extensions is illustrated by the embedding of Java, XML, a tuple syntax, and SWUL. MetaBorg also allows an embedded language to use an entirely different lexical syntax, which is not the case for syntax macros, where lexical analysis of the macros is performed by the scanner of the base language. In particular, this feature of MetaBorg is illustrated by the embedding of XML in Java.

The main difference between syntax macro systems is the expressiveness of argument definitions. The most limited systems only allow a fixed number of arguments, usually with a syntax comparable to procedure calls. The most expressive systems are *MS*^{2} [Weise & Crew 1993], which allows regular languages, and *Dylan* [Shalit 1996] and *Metamorphic Syntax Macros* [Brabrand & Schwartzbach 2002], which allow context-free languages. The nonterminals appearing in actual arguments of macros in Dylan, are not represented as abstract syntax trees, but as a list of tokens. In [Brabrand & Schwartzbach 2002] the user-defined nonterminals are represented as an abstract syntax tree. These user-defined nonterminals are however required to have a fixed associated nonterminal in the host language. The MetaBorg method releases this restriction by completely separating the assimilation into the host language from the syntax definition of language extensions.

Implementations of syntax macros also differ in the expressivity of the rewriting to the base language in the syntax macro definitions, called assimilation in MetaBorg. Usually a syntax macro definition consists of the definition of the invocation syntax, which is some language over terminals and nonterminals of the host language. The arguments of the syntax macro can be used in the definition of code to be produced by a macro invocation. In [Leavenworth 1966] conditional syntax macros were already proposed, which increases the expressivity of the rewriting language. In the programmable syntax macros of *MS*^{2} [Weise & Crew 1993] the macro language is a small extension of the host programming language. This extension is targeted at rewriting syntactic extensions to the base language. *C++ templates* are well-known to be Turing complete, but this expressivity is not based on rewriting trees, but

on constant folding.

The Java Syntactic Extender (JSE) [Bachrach & Playford 2001] macro system is mainly inspired by Dylan. The expressiveness of the syntax that can be introduced is limited. As in most macro systems, a macro identifier is required in invocations and the parser used by JSE is not extensible. Rather, a source file is parsed by a fixed parser to a 'skeleton syntax tree' (SST). The SST is a lexical representation of a Java source file, but it is somewhat more structured than a plain sequence of tokens. JSE macros are implemented in Java and operate on the SST, i.e. lexical syntax. Hence, the source code fragments that are manipulated by JSE are not syntactically structured and JSE does not guarantee that the fragments are syntactically correct. Processing of the input arguments is done by pattern matching, or by arbitrary procedural code. Therefore, JSE supports any syntax extensions that can be represented in the SST, but it is not possible to introduce syntax that does not conform to the lexical syntax of Java. To improve error reporting, JSE tries to maintain the original source code location in the generated code. The source location will be lost if arbitrary manipulations of the SST are performed.

For a more extensive survey of the properties of various systems supporting syntax macros, see [Brabrand & Schwartzbach 2002].

2.6.5 *Lexical Macros*

Lexical macro languages, such as CPP [CPP Manual] and M4 [GNU M4 Website], replace characters or tokens in an input file by a sequence of tokens defined in the macro definition. Macro definitions consist of an identifier of the macro, a fixed number of arguments, and a sequence of characters and references to the arguments of the macro. Lexical macros provide abbreviations at the lexical level of a language. Lexical macro languages do not have any knowledge of the structure of the input file and the context of a macro invocation. This may be considered an advantage, since these macro languages can thus be used for input files of arbitrary languages. On the other hand this is an disadvantage since the macro language cannot use language specific knowledge to make the substitution more reliable. Since the lexical syntax must identify parts of the input that form a macro invocation, the invocation syntax is not liberal enough to allow the extension of languages with domain-specific notations. Using lexical macros is error-prone since the result of the macros can interfere with the context of the invocation and is thus parsed entirely differently than might be expected. Lexical macros are therefore usually loaded with parentheses to enforce correct parsing.

2.6.6 *Metafront*

Metafront [Brabrand et al. 2003] was designed as a more general solution to the parsing issues experienced in metamorphic syntax macros [Brabrand & Schwartzbach 2002]. Metafront applies a novel parsing algorithm: *specificity parsing*. Metafront extends the formalism of context-free grammars with a

separate set of regular terminal languages. The lexical syntax of a language is thus still defined separately from the context-free syntax, but there is not a separate scanner and parser tool. However, Metafront is not a scannerless parser in the sense of not tokenizing the input stream at all. It uses a separate scanner for tokenizing the input file. If at a certain point in the input stream multiple options out of the set of terminal languages are available, then the most specific one is chosen. Specificity parsing thus has a built-in preference for the longest token. Specificity parsing in Metafront does not return to a choicepoint in the input stream; it immediately commits the choice for a certain token. Left recursion is not allowed in syntax definitions to ensure termination. Metafront adds a form of lookahead called *attractors* to solve ambiguities if there is not a most specific token. An attractor specifies after how many successful tokens for a certain production this alternative must be chosen. Attractors are also used to reject certain alternatives.

Because of the attractor disambiguation construct, syntax definitions in Metafront are less concise then syntax definitions in SDF2. Left recursion has to be removed, which is unfortunate. However, Metafront syntax definitions are indeed modular, unlike other parser generators that do not use Earley or GLR. The main goal of Metafront is not to sacrifice performance when there is a need for extensible syntax definitions. Since the current implementation is a prototype that interprets language definitions, it is difficult to determine whether specificity parsing will in practice have significantly better performance than Scannerless Generalized LR parsing.

Metafront also features a transformation language. Metafront transformations are guaranteed to terminate and will transform syntactically correct input to syntactically correct output. It is unclear to us why termination guarantees are important in this context. Guaranteeing termination sacrifices the expressivity and abstraction facilities of the transformation language and we prefer a more expressive language over termination guarantees. In the Stratego/XT project we have developed many complex transformations, in particular program optimizations. Testing transformations has in practice proven to be sufficient to see whether a transformation terminates or not. It would be interesting to separate Metafront in a parsing and transformation component and experiment with the performance of specificity parsing.

2.7 FUTURE WORK

2.7.1 *Application Domains*

There are many opportunities for applying MetaBorg to create domain-specific extensions of Java and other host languages. We summarize some of our ideas to give more insight into the scope of concrete syntax for objects.

In *linguistic reflection* a program can generate new programs at runtime and execute these generated programs as part of its own execution [Kirby et al. 1998]. Linguistic reflective programs are program generators and as such the generator is concerned with the syntactic and semantic correctness of the

generated programs. In [Kirby et al. 1998] programs are generated by constructing a textual representation in strings, which is unclear and error-prone. The authors suggest as further work to generate code using an abstract syntax and place the challenge to have the generator code look as the generated language would normally be written. Our solution for embedding languages using MetaBorg makes this possible by embedding a concrete syntax for the abstract syntax of a programming language.

Linguistic reflection tools targeting a specific platform might even target standardized abstract syntax trees such as the CodeDOM of .NET. These abstract syntax trees can then be accepted directly by a compiler or interpreter, which makes an expensive and error-prone intermediate textual form unnecessary. It might even be possible to directly compile embedded fragments to Java bytecode or the .NET Intermediate Language[5], for example by defining a concrete syntax for the objects used by an existing byte code construction library, such as the Byte Code Engineering Library (BCEL) [BCEL Website].

JJTraveler [Visser 2001] is a visitor combinator framework for Java. Visitor combinators can be used to compose a tree traversal rewriting functionality from basic visitor combinators. The set of basic combinators of JJTraveler is inspired by the strategy primitives of Stratego, a strategic program transformation language where rewriting rules are applied according to user-definable rewriting strategies. However, JJTraveler visitor construction is somewhat verbose compared to the concise notation for strategies in Stratego. In essence JJTraveler provides classes for all strategy operators of Stratego. The embedding of concrete syntax for these strategy operators would be an interesting application of MetaBorg.

Doug Lea's *java.util.concurrent* library introduces useful abstractions for concurrent programming. This library provides utility classes for concurrent programming developed in [Lea 2000]. Other programming languages already have these abstractions over the low-level facilities for concurrent programming as built-in constructs. Domain-specific notations for these abstractions can be developed with MetaBorg as an extension of the Java language.

XQuery is a query language developed by the W3C. It enjoys wide support by researchers and the industry. JSR 225 aims at standardizing an API for evaluating XQuery operations. XQueries are however even less attractive for inclusion in String literals than XPath and SQL statements since the queries tend to be larger and span multiple lines. Obviously, this language needs to be embedded in Java to make static syntactic verification possible and embedding of values from the host language easier and more secure.

2.7.2 *Open Compilers*

Open compilers can improve the integration of MetaBorg based embedded languages in the compilation process. Assimilators can be arbitrarily complex and in some cases they will need to have more detailed semantic information

[5] Indeed, the new open compiler approach of Kats [Kats 2007] is based on embedding bytecode syntax in Java to facilitate code generation and the implementation of language extensions.

on the input program than a structured representation of the input program in an abstract syntax tree provides. We already experience some problems where assimilators needed to know the type of expressions.

Improved integration of the assimilators in the compiler will provide the required information to the assimilator and will improve the analysis that can be performed on programs with a reasonable amount of work. Application of assimilators in different phases of an open compiler (e.g. parsing, semantic analysis, and translation to intermediate code) will improve error reporting since the aspects of assimilation into the host language can be implemented separately for each phase by extending components of the open compiler.

A useful experiment of extending the components of an open compiler is delayed disambiguation in the input programs. In many cases the embedding of a language requires disambiguation because the fragments of the embedded language can be parsed as more than one nonterminal. In the embedding of Java in Java this disambiguation is performed by optionally prefixing the fragments with `exp` and `bstm` to indicate the nonterminal of the fragment. In the embedding of XML in Java the nonterminal of anti-quotations has to be defined since the anti-quotation might refer to character data as well as content. When this explicit disambiguation is required is not always clear and excessive use of disambiguations makes the code less concise.

An interesting solution to this problem is to solve the ambiguities by means of interaction with a type-checker. From a fragment such as `Expr e = expr ⟦ 1 + 2 ⟧` it is easy to see that there is unnecessary duplication of type declarations. If the type-checker of the host language can be extended, then during type-checking the appropriate alternative from a set of ambiguous abstract syntax trees can be chosen. By preserving the ambiguities until the type checking phase they can be handled at this point. This requires the parser to be able to report all possible parses, which SGLR supports. In Chapter 3 we discuss the method of a disambiguating type-checker in detail.

2.8 CONCLUSIONS

Programming language design these days often is about selecting candidates for tasteful inclusion in the core language. Design patterns and application libraries are among the candidates for assimilation into the host language. Extending existing programming languages with new features that used to be provided by libraries has several advantages. The core language can provide an attractive syntax for these features, and tools such as type-checkers, refactoring tools, IDEs and debuggers will (need to) have built-in knowledge of the extensions.

Cω (first called Xen [Meijer & Schulte 2003b, Meijer & Schulte 2003a]) lifts the area of data-access to the language level. To this end, it makes the type system of the host language more general and extends the language of 'object literals' with an alternative XML syntax. Cω provides languages features for iterating, collecting, filtering, and applying functions over data structures. The Xtatic language [Xtatic], a follow-up of XDuce [Hosoya & Pierce 2000], extends

an existing programming language with XML specific support as well, but uses a translation into the host language, which is C#.

However, the growth of a language with more domain-specific constructs is limited by the general application area of the language. Extensions of the language can potentially be applied in *all* code written in this language. Therefore very domain-specific notation will never be accepted to the core language itself. The kitchen sink is only interesting for kitchen related applications. Hence, language extensions tend to abstract over control-flow rather than over data. A domain-specific concrete syntax for objects will usually not be lifted to the host language, since objects are much more application specific than routines.

Syntax macros take domain-specific extension to the other extreme by allowing the programmer to extend the host language within the program itself. This is a very lightweight extension of the language, but is restricted by a host of technical problems. Our MetaBorg method is somewhere in between designing a new language and ad-hoc extensions with syntax macros. Since extensions need to be defined carefully by a domain expert, domain-specific notations are introduced in a separate role. To conclude, MetaBorg is more flexible than Cω in that it does not target a specific domain and MetaBorg allows arbitrary extensions, as opposed to syntax macros. We expect that extensions will be implemented in MetaBorg or similar toolkits before they are assimilated into the host language for good. For those extensions that will never make it there, MetaBorg provides the way to still make them available to application programmers.

ACKNOWLEDGMENTS

Many people have contributed to the development of Stratego/XT. Merijn de Jonge developed the pretty-printing tools of Stratego/XT. Niels Janssen, Rob Vermaas and Jonne van Wijngaarden contributed to the development of XML support. SDF is maintained and further developed at the Centrum voor Wiskunde en Informatica (CWI) and the Eindhoven University of Technology (TU/e) by Mark van den Brand, Giorgios Robert Economopoulos, and Jurgen Vinju. We thank Eelco Dolstra, Jeff Gray, Merijn de Jonge, Martijn Vermaat and the anonymous reviewers of OOPSLA 2004 for providing useful feedback on an earlier version of this chapter. Finally, we thank the users of Stratego/XT and the MetaBorg method for their inspiration.