URL: http://www.elsevier.nl/locate/entcs/volume44.html 25 pages

Object-oriented Tree Traversal with JJForester

Tobias Kuipers [1]

CWI, Department SEN,
PO Box 94079, 1090 GB Amsterdam, The Netherlands

Joost Visser [2]

CWI, Department SEN,
PO Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

We want to use the advanced language processing technology available in the ASF+SDF Meta-Environment in combination with general purpose programming languages. In particular, we want to combine the syntax definition formalism SDF and the associated components that support generalized LR parsing, with the object-oriented language Java. To this end, we implemented JJForester, a tool that generates class structures from SDF grammar definitions. The generated class structures implement a number of *design patterns* to facilitate construction and traversal of parse trees represented by object structures. In a detailed case study, we demonstrate how program analyses and transformations can be constructed with JJForester.

1 Introduction

JJForester is a parser and visitor generator for Java that takes language definitions in the syntax definition formalism SDF [13,23] as input. It generates Java code that facilitates the construction, representation, and manipulation of syntax trees in an object-oriented style. To support *generalized LR parsing* [22,21], JJForester reuses the parsing components of the ASF+SDF Meta-Environment [17].

The ASF+SDF Meta-Environment is an interactive environment for the development of language definitions and tools. It combines the syntax definition formalism SDF with the term rewriting language ASF [2]. SDF is supported with

[1] Email:tobias@acm.org
[2] Email:Joost.Visser@cwi.nl

generalized LR parsing technology. For language-centered software engineering applications, generalized parsing offers many benefits over conventional parsing technology [8]. ASF is a rather pure executable specification language that allows rewrite rules to be written in concrete syntax.

In spite of its many qualities, a number of drawbacks of the ASF+SDF Meta-Environment have been identified over the years. One of these is its unconditional bias towards ASF as programming language. Though ASF was well suited for the *prototyping* of language processing systems, it lacked some features to build mature *implementations*. For instance, ASF does not come with a strong library mechanism, I/O capabilities, or support for generic term traversal. Also, the closed nature of the meta-environment obstructed interoperation with external tools. As a result, for a mature implementation one was forced to abandon the prototype and fall back to conventional parsing technology. Examples are the ToolBus [4], a software interconnection architecture and accompanying language, that has been simulated extensively using the ASF+SDF Meta-Environment, but has been implemented using traditional Lex and Yacc parser technology and a manually coded C program. For Stratego [25], a system for term rewriting with strategies, a simulator has been defined using the ASF+SDF Meta-Environment, but the parser has been hand coded using ML-Yacc and Bison. A compiler for RISLA, an industrially successful domain-specific language for financial products, has been prototyped in the ASF+SDF Meta-Environment and re-implemented in C [7].

To relieve these drawbacks, the Meta-Environment has recently been re-implemented in a component-based fashion [5]. Its components, including the parsing tools, can now be used separately. This paves the way to adding support for alternative programming languages to the Meta-Environment.

As a major step into this direction, we have designed and implemented JJForester. This tool combines SDF with the main stream general purpose programming language Java. Apart from the obvious advantages of object-oriented programming (e.g. data hiding, intuitive modularization, coupling of data and accompanying computation), it also provides language tool builders with the massive library of classes and design patterns that are available for Java. Furthermore, it facilitates a myriad of interconnections with other tools, ranging from database servers to remote procedure calls. Apart from Java code for constructing and representing syntax trees, JJForester generates visitor classes that facilitate generic traversal of these trees.

The paper is structured as follows. Section 2 explains JJForester. We discuss what code it generates, and how this code can be used to construct various kinds of tree traversals. Section 3 provides a case study that demonstrates in depth how a program analyzer (for the Toolbus language) can be constructed using JJForester.

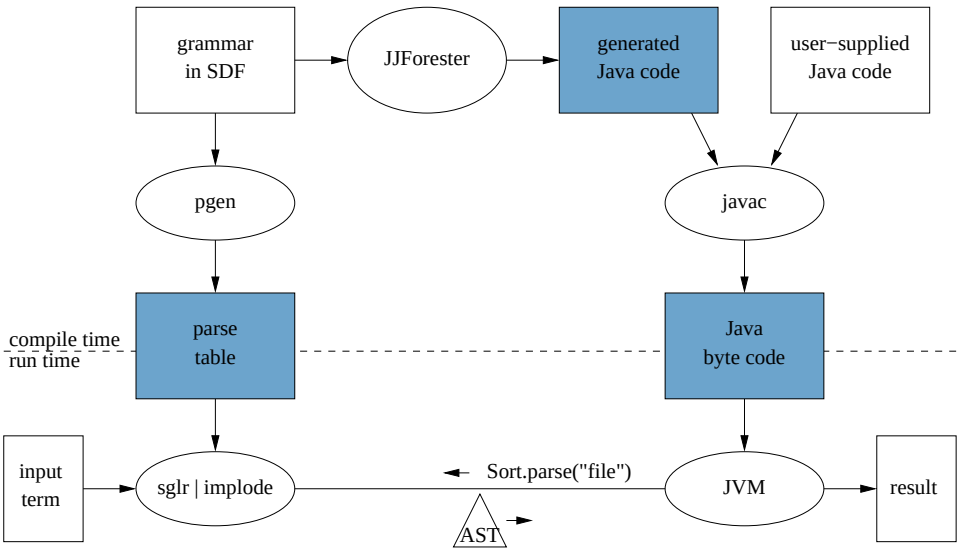

Fig. 1. Global architecture of JJForester. Ellipses are tools. Shaded boxes are generated code.

2 JJForester

JJForester is a parser and visitor generator for Java. Its distinction with respect to existing parser and visitor generators, e.g. Java Tree Builder, is twofold. Firstly, it deploys generalized LR parsing, and allows *unrestricted*, *modular*, and *declarative* syntax definition in SDF (see Section 2.2). These properties are essential in the context of component-based language tool development where grammars are used as *contracts* [15]. Secondly, to cater for a number of reoccuring tree traversal scenarios, it generates variants on the Visitor pattern that allow different traversal strategies. In this section we will give an overview of JJForester. We will give a brief introduction to SDF which is used as its input language. By means of a running example, we will explain what code is generated by JJForester and how to program against the generated code.

2.1 Overview

The global architecture of JJForester is shown in Figure 1. Tools are shown as ellipses. Shaded boxes are generated code. Arrows in the bottom row depict run time events, the other arrows depict compile time events. JJForester takes a grammar defined in SDF as input, and generates Java code. In parallel, the parse table generator PGEN is called to generate a parse table from the grammar. The generated code is compiled together with code supplied by the user. When the resulting byte code is run on a Java Virtual Machine, invocations of *parse* methods will result in calls to the parser SGLR. From a given input term, SGLR produces a parse tree as output. These parse trees are passed through the parse tree implosion tool *implode* to obtain abstract syntax trees.

```
module Expr
exports
  context-free syntax
    Identifier          -> Expr {cons("Var")}
    Expr Expr           -> Expr {cons("Apply"), left}
    "\\" Identifier ":" Type "." Expr
                        -> Expr {cons("Lambda")}
    "(" Expr ")"        -> Expr {bracket}
```

```
module Identifier
exports
  lexical syntax
    [A-Za-z0-9]+ -> Identifier
  lexical restrictions
    Identifier -/- [A-Za-z0-9]
```

```
module Type
exports
  context-free syntax
    Identifier          -> Type {cons("TVar")}
    Type "->" Type      -> Type {cons("Arrow"),right}
    "(" Type ")"        -> Type {bracket}
```

```
module Layout
exports
  lexical syntax
    [\ \t\n] -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Fig. 2. Example SDF grammar.

2.2 SDF

The language definition that JJForester takes as input is written in SDF. In order to explain JJForester, we will give a short introduction to SDF. A complete account of SDF can be found in [13,23].

SDF stands for Syntax Definition Formalism, and it is just that: a formalism to define syntax. SDF allows the definition of lexical and context-free syntax in the same formalism. SDF is a modular formalism; it allows productions to be distributed at will over modules. For instance, mutually dependent productions can appear in different modules, as can different productions for the same non-terminal. This implies, for instance, that a kernel language and its extensions can be defined in different modules. Like extended BNF, SDF offers constructs to define optional symbols and iteration of symbols, but also for separated iteration, alternatives, and more.

Figure 2 shows an example of an SDF grammar. This example grammar gives a modular definition of a tiny lambda calculus-like language with typed lambda functions. Note that the orientation of SDF productions is reversed with respect to BNF notation. The grammar contains two context-free non-terminals, Expr and Type, and two lexical non-terminals, Identifier and LAYOUT. The latter non-terminal is used *implicitly* between all symbols in context-free productions. As the example details, expressions can be variables, applications, or typed lambda abstractions, while types can be type variables or function types.

SDF's expressiveness allows for defining syntax concisely and naturally. SDF's modularity facilitates reuse. SDF's declarativeness makes it easy and retargetable. But the most important strength of SDF is that it is supported by *Generalized LR Parsing*. Generalized parsing removes the restriction to a non-ambiguous subclass of the context-free grammars, such as the LR(k) class. This allows a maximally natural expression of the intended syntax; no more need for 'bending over backwards' to encode the intended grammar in a restricted subclass. Furthermore, generalized parsing leads to better modularity and allows 'as-is' syntax reuse.

As SDF removes any restriction on the class of context-free grammars, the grammars defined with it potentially contain ambiguities. For most applications, these ambiguities need to be resolved. To this end, SDF offers a number of disambiguation constructs. The example of Figure 2 shows four such constructs. The *left* and *right* attributes indicate associativity. The *bracket* attribute indicates that parentheses can be used to disambiguate Exprs and Types. For the lexical non-terminals the longest match rule is explicitly specified by means of *follow restrictions*. Not shown in the example is SDF's notation for relative priorities.

In the example grammar, each context-free production is attributed with a *constructor name*, using the *cons(..)* attribute. Such a grammar with constructor names amounts to a simultaneous definition of concrete and abstract syntax of the language at hand. The *implode* back-end turns concrete parse trees emanated by the parser into more concise abstract syntax trees (ASTs) for further processing. The constructor names defined in the grammar are used to build nodes in the AST. As will become apparent below, JJForester operates on these abstract syntax trees, and thus requires grammars with constructor names. A utility, called *sdf-cons* is available to automatically synthesize these attributes when absent.

SDF is supported by two tools: the parse table generator PGEN, and the scannerless generalized parser SGLR. These tools were originally developed as components of the ASF+SDF Meta-Environment and are now separately available as stand-alone, reusable tools.

2.3 Code generation

From an SDF grammar, JJForester generates the following Java code:

Class structure

For each non-terminal symbol in the grammar, an *abstract* class is generated. For each production in the grammar, a *concrete* class is generated that extends the abstract class corresponding to the result non-terminal of the production. For example, Figure 3 shows a UML diagram of the code that JJForester generates for the grammar in Figure 2. The relationships between the abstract classes *Expr* and *Type*, and their concrete subclasses are known as the Composite pattern.

Lexical non-terminals and productions are treated slightly differently: for each lexical non-terminal a class can be supplied by the user. Otherwise, this lexical non-terminal is replaced by the pre-defined non-terminal `Identifier`, for which a single concrete class is provided by JJForester. This is the case in our example.

When the input grammar, unlike our example, contains complex symbols such as optionals or iterated symbols, additional classes are generated for them as well. The case study will illustrate this.

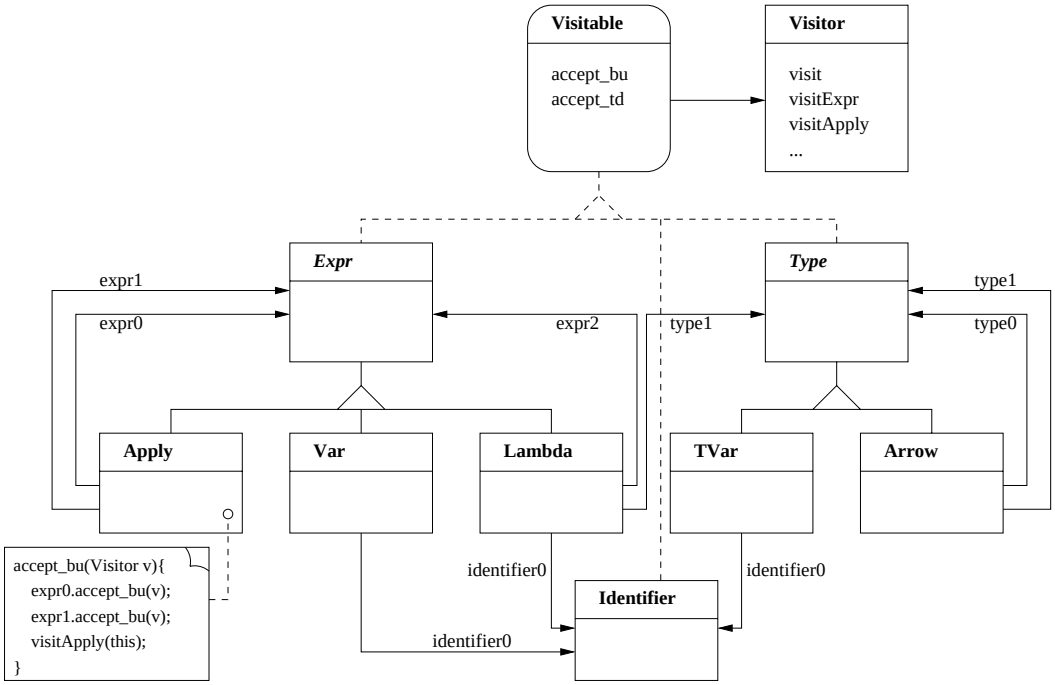

Fig. 3. The UML diagram of the code generated from the grammar in Figure 2.

Parsers

Also, for every non-terminal in the grammar, a parse method is generated for parsing a term (plain text) and constructing a tree (object structure). The actual parsing is done externally by SGLR. The parse method implements the Abstract Factory design pattern; each non-terminal class has a parse method that returns an object of the type of one of the constructors for that non-terminal. Which object gets returned depends on the string that is parsed.

Constructor methods

In the generated classes, constructor methods are generated that build *language-specific* tree nodes from the generic tree that results from the call to the external parser.

Set and get methods

In the generated concrete classes, set and get methods are generated to inspect and modify the fields that represent the subtrees. For example, the Apply class will have `getExpr0` and `setExpr0` methods for its first child.

Accept methods

In the generated concrete classes, several accept methods are generated that take a Visitor object as argument, and apply it to a tree node. Currently, two *iterating* accept methods are generated: `accept_td` and `accept_bu`, for top-down and bottom-up traversal, respectively. For the Apply class, the bottom-up accept method is shown in the Figure 3.

Visitor classes

A Visitor class is generated which contains a visit method for each production and each non-terminal in the grammar. Furthermore, it contains one unqualified visit method which is useful for *generic* refinements (see below). These visit methods are *non-iterating*: they make no calls to accept methods of children to obtain recursion. The default behavior offered by these generated visit methods is simply to do nothing.

Together, the Visitor class and the accept methods in the various concrete classes implement a variant of the Visitor pattern [12], where the responsibility for iteration lies with the accept methods, not with the visit methods. We have chosen this variant for several reasons. First of all, it relieves the programmer who specializes a visitor from reconstructing the iteration behavior in the visit methods he redefines. This makes specializing visitors less involved and less error-prone. In the second place, it allows the iteration behavior (top-down or bottom-up) to be varied. In Section 4.3 we will comment on the possibilities of offering even more control over iteration behavior.

Apart from generating Java code, JJForester calls PGEN to generate a parse table from its input grammar. This table is used by SGLR which is called by the generated parse methods.

2.4 Programming against the generated code

The generated code can be used by a tool builder to construct tree traversals through the following steps:

(i) Refine a visitor class by redefining one or more of its visit methods. As will be explained below, such refinement can be done at various levels of genericity, and in a step-wise fashion.
(ii) Start a traversal with the refined visitor by feeding it to the accept method of a tree node. Different accept methods are available to realize top-down or bottom-up traversals.

This method of programming traversals by refining (generated) visitors provides interesting possibilities for reuse. Firstly, many traversals only need to do something 'interesting' at a limited number of nodes. For these nodes, the programmer needs to supply code, while for all others the behavior of the generated visitor is inherited. Secondly, different traversals often share behavior for a number of nodes. Such common behavior can be captured in an initial refinement, which is then further refined in diverging directions. Unfortunately, Java's lack of multiple inheritance prohibits the converse: construction of a visitor by inheritance from two others (but see Section 4.3 for further discussion). Thirdly, some traversal actions may be specific to nodes with a certain constructor, while other actions are the same for all nodes of the same type (non-terminal), or even for all nodes of any type. As the visitors generated by JJForester allow refinement at each of these levels of specificity, there is

```
public class VarCountVisitor extends Visitor {
    public int counter = 0;
    public void visitVar(Var x) {
        counter++;
    }
    public void visitTVar(TVar x) {
        counter++;
    }
}
```

Fig. 4. Specific refinement: a visitor for counting variables.

no need to repeat the same code for several constructors or types. We will explain these issues through a number of small examples.

Constructor-specific refinement

Figure 4 shows a refinement of the Visitor class which implements a traversal that counts the number of variables occurring in a syntax tree. Both expression variables and type variables are counted.

This refinement extends Visitor with a counter field, and redefines the visit methods for Var and TVar such that the counter is incremented when such nodes are visited. The behavior for all other nodes is inherited from the generated Visitor: do nothing. Note that redefined methods need not restart the recursion behavior by calling an accept method on the children of the current node. The recursion is completely handled by the generated accept methods.

Generic refinement

The refinement in the previous example is specific for particular node constructors. The visitors generated by JJForester additionally allow more generic refinements. Figure 5 shows refinements of the Visitor class that implement a more generic expression counter and a fully generic node counter. Thus, the first visitor counts all expressions, irrespective of their constructor, and the second visitor counts all nodes, irrespective of their type. No code duplication is necessary.

Step-wise refinement

Visitors can be refined in several steps. For our example grammar, two subsequent refinements of the Visitor class are shown in Figure 6. The class GetVarVisitor is a visitor for collecting all variables used in expressions. It is defined by extending the Visitor class with a field `vars` initialized as the empty set of variables, and by redefining the visit method for the Var class to insert each variable it encounters into this set. The GetVarVisitor is further refined into a visitor that collects *free* variables, by additionally redefining

```
public class ExprCountVisitor extends Visitor {
    public int counter = 0;
    public void visitExpr(Expr x) {
        counter++;
    }
}
```

```
public class NodeCountVisitor extends Visitor {
    public int counter = 0;
    public void visit(Object x) {
        counter++;
    }
}
```

Fig. 5. Generic refinement: visitors for counting expressions and nodes.

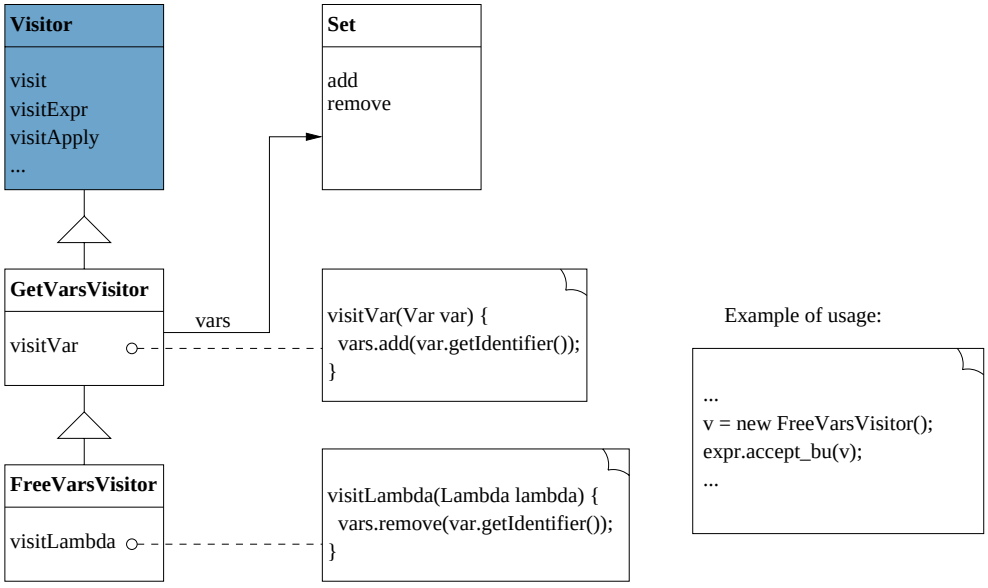

Fig. 6. UML diagram for user code.

the visit method for the Lambda class. This redefined method removes the variables bound by the lambda expression from the current set of variables. Finally, this second visitor can be unleashed on a tree using the `accept_bu` method. This is illustrated by an example of usage in Figure 6.

Note that the visitors in Figures 4 and 5 can be refactored as refinements of a common initial refinement, say CountVisitor, which contains only the field counter.

Of course, our running example does not mean to suggest that Java would be the ideal vehicle for implementing the lambda calculus. Our choice of example was motivated by simplicity and self-containedness. To compare, an

implementation of the lambda calculus in the ASF+SDF Meta-Environment can be found in [10]. In Section 3 we will move into the territory for which JJForester is intended: component-based development of program analyses and transformations for languages of non-trivial size.

2.5 Assessment of expressiveness

To evaluate the expressiveness of JJForester within the domain of language processing, we will assess which program transformation scenarios can be addressed with it. We distinguish three main scenarios:

Analysis A value or property is distilled from a syntax tree. Type-checking is a prime example.

Translation A program is transformed into a program in a different language. Examples include generating code from a specification, and compilation.

Rephrasing A program is transformed into another program, where the source and target language coincide. Examples include normalization and renovation.

For a more elaborate taxonomy of program transformation scenarios, we refer to [24]. The distinction between analysis and translation is not clear-cut. When the value of an analysis is highly structured, especially when it is an expression in another language, the label 'translation' is also appropriate.

The traversal examples discussed above are all tree analyses with simple accumulation in a state. Here, 'simple' accumulation means that the state is a value or collection to which values are added one at a time. This was the case both for the counting and the collecting examples. However, some analyses require more complex ways of combining the results of subtree traversals than simple accumulation. An example is pretty-printing, where literals need to be inserted *between* pretty-printed subtrees. In the case study, a visitor for pretty-printing will demonstrate that JJForester is sufficiently expressive to address such more complex analyses. However, a high degree of reuse of the generated visit methods can currently only be realized for the simple analyses. In the future work section (4.3), we will discuss how such reuse could be realized by generating special visitor subclasses or classes that model updatable many-sorted folds [19].

Translating transformations are also completely covered by JJForester's expressiveness. As in the case of analysis, the degree of reuse of generated visit methods can be very low. Here, however, the cause lies in the nature of translation, because it typically takes every syntactic construct into account. This is not always the case, for instance, when the translation has the character of an analysis with highly structured results. An example is program visualization where only dependencies of a particular kind are shown, e.g. module structures or call graphs.

In the object-oriented setting, a distinction needs to be made between destructive and non-destructive rephrasings. Destructive rephrasings are covered by JJForester. However, as objects can not modify their *self* reference, destructive modifications can only change subtrees and fields of the current node, but they cannot replace the current node by another. Non-destructive rephrasings can be implemented by refining a traversal that clones the input tree. A visitor for tree cloning can be generated, as will be discussed in Section 4.3.

A special case of rephrasing is decoration. Here, the tree itself is traversed, but not modified except for designated attribute fields. Decoration is useful when several traversals are sequenced that need to share information about specific nodes. JJForester does not cover decoration yet.

3 Case study

Now that we have explained the workings of JJForester, we will show how it is used to build a program analyzer for an actual language. In particular, this case study concerns a static analyzer for the ToolBus [4] script language. In Section 3.1 we describe the situation from which a need for a static analyzer emerged. In Section 3.2 the language to be analyzed is briefly explained. Finally, Section 3.3 describes in detail what code needs to be supplied to implement the analyzer.

3.1 The Problem

The ToolBus is a coordination language which implements the idea of a software bus. It allows applications (or *tools*) to be "plugged into" a bus, and to communicate with each other over that bus. Figure 7 gives a schematic overview of the ToolBus. The protocol used for communication between the applications is not fixed, but is programmed through a ToolBus script, or T-script.

A T-script defines one or more processes that run inside the ToolBus in parallel. These processes can communicate with each other, either via synchronous point-to-point communication, or via asynchronous broadcast communication. The processes can direct and activate external components via *adapters*, small pieces of software that translate the ToolBus's remote procedure calls into calls that are native to the particular software component that needs to be activated. Adapters can be compiled into components, but off-the-shelf components can be used, too, as long as they possess some kind of external interface.

Communication between processes inside the ToolBus does not occur over named channels, but through pattern matching on terms. Communication between processes occurs when a term sent by one matches the term that is expected by another. This will be explained in more detail in the next section.

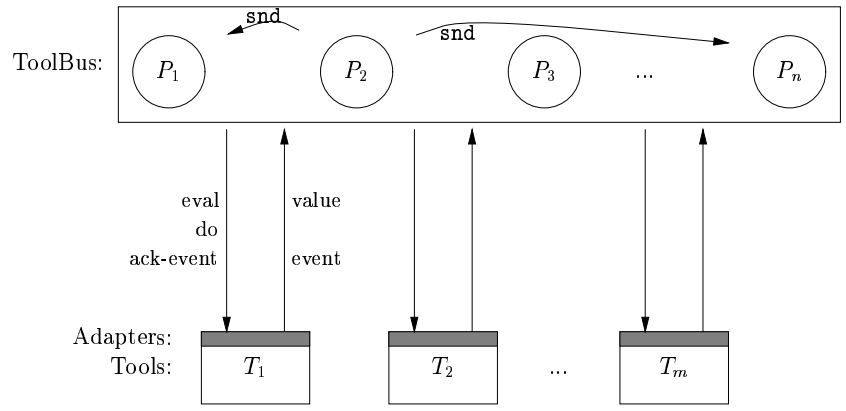

Fig. 7. The Toolbus architecture. Tools are connected to the bus through adapters. Inside the bus, several processes run in parallel. These processes communicate with each other and the adapters according to the protocol defined in a T-script.

This style of communication is powerful, flexible and convenient, but tends to make it hard to pinpoint errors in T-scripts. To support the T-script developer, the ToolBus runtime system provides an interactive visualizer, which shows the communications taking place in a running ToolBus. Though effective, this debugging process is tedious and slow, especially when debugging systems with a large number of processes.

To complement the runtime visualizer, a *static* analysis of T-scripts is needed to support the T-script developer. Static analysis can show that some processes can never communicate with each other, that messages that are sent can never be received (or vice versa), or that two processes that should not communicate with each other may do so anyway. Using JJForester, such a static analyzer is constructed in Section 3.3.

3.2 T-scripts explained

T-scripts are based on ACP (Algebra of Communicating Processes) [1]. They define communication protocols in terms of *actions*, and operations on these actions. We will be mainly concerned with the communication actions, which we will describe below. Apart from these, there are assignment actions, conditional actions and basic arithmetic actions. The action operators include sequential composition ($a.b$), non-deterministic choice ($a + b$), parallel composition ($a \mid\mid b$), and repetition ($a * b$). The full specification of the ToolBus script language can be found in [3].

The T-script language offers actions for communication between processes and tools, and for synchronous and asynchronous communication between processes. For the purposes of this paper we will limit ourselves to the most commonly used *synchronous* actions. These are `snd-msg(T)` and `rec-msg(T)` for sending and receiving messages, respectively. These actions are parameterized with arbitrary data `T`, represented as ATerms [6]. A successful synchronous communication occurs when a term that is sent matches a term

```
process Pump is
let D: int
in
( rec-msg(activate(D?)).
  rec-msg(on).
  snd-msg(report(D))
) *
delta
endlet

process Operator is
let C: int, D: int,
    Payment: int, Amount: int
in
( rec-msg(request(D?,C?)).
  Payment := D.
  snd-msg(schedule(Payment,C)).
  rec-msg(result(D?)).
  Amount := sub(Payment,D).
  snd-msg(remit(Amount))
) *
delta
endlet
```

```
process Customer is
let
C: int, D: int
in
C := process-id.
D := 10.
snd-msg(prepay(D,C)).
rec-msg(okay(C)).
snd-msg(turn-on).
printf(
 "Customer %d using pump\n",
  C).
rec-msg(stop).
rec-msg(change(D?)).
printf(
 "Customer %d got $%d change\n",
  C, D)
endlet
```

```
process GasStation is
let
D: int, C: int
in
( rec-msg(prepay(D?,C?)).
  snd-msg(request(D,C))
||rec-msg(schedule(D?,C?)).
  snd-msg(activate(D)).
  snd-msg(okay(C))
||rec-msg(turn-on).
  snd-msg(on)
||rec-msg(report(D?)).
  snd-msg(stop).
  snd-msg(result(D))
||rec-msg(remit(D?)).
  snd-msg(change(D))
)*
delta
endlet

toolbus(GasStation,Pump,
  Customer,Customer,Operator)
```

Fig. 8. The T-script for the gas station with control process.

that is received. For instance, the closed term `snd-msg(f(a))` can match the closed term `rec-msg(f(a))` or the open term `rec-msg(f(T?))`. At successful communication, variables in the data of the receiving process are instantiated according to the match.

To illustrate, a small example T-script is shown in Figure 8. This example contains only processes. In a more realistic situation these processes would communicate with external tools, for instance to get the input of the initial value, and to actually activate the gas pump. The script's last statement is a mandatory `toolbus(..)` statement, which declares that upon startup the processes GasStation, Pump, Customer and Operator are all started in parallel. The first action of all processes, apart from Customer, is a `rec-msg` action. This means that those processes will block until an appropriate communication is received. The Customer process starts by doing two assignment statements. `process-id` (a built-in variable that contains the identifier of the current process) is assigned to `C`, and 10 to `D`. The first communication action performed by Customer is a `snd-msg` of the term `prepay(D,C)`. This term is received by the GasStation process, which in turn sends the term `request(D,C)` message. This is received by Operator, and so on.

The script writer can use the mechanism of communication through term matching to specify that any one of a number of processes should receive a message, depending on the state they are in, and the sending process does not need to know this. It just sends out a term into the ToolBus, and anyone of the accepting processes can "pick it up". Unfortunately, when incorrect or too general terms are specified, communication will not occur as expected, and the exact cause will be difficult to trace. The static analyzer developed in the next section is intended to solve this problem.

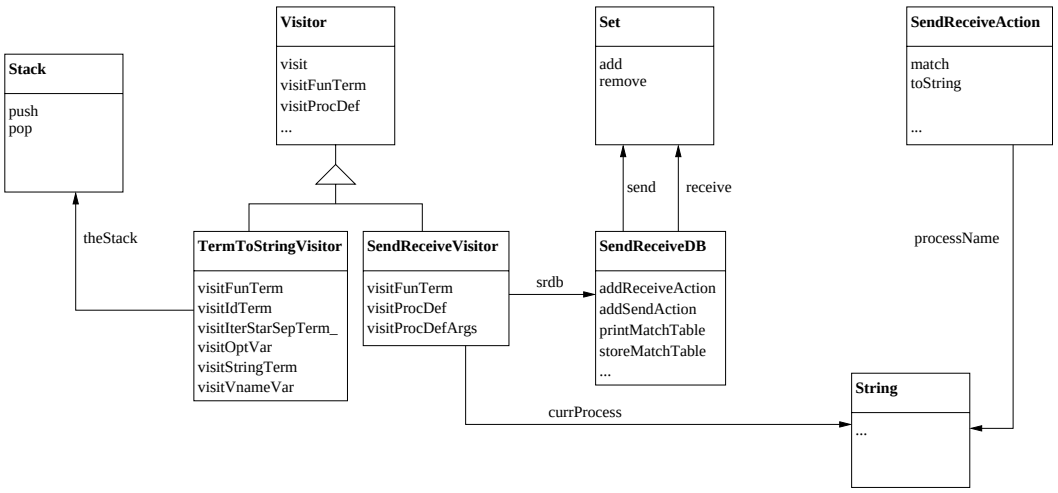

Fig. 9. UML diagram of the ToolBus analyzer.

3.3 Analysis using JJForester

We will first sketch the outlines of the static analysis algorithm that we implemented. It consists of two phases: collection and matching. In the collection phase, *all* send and receive actions in the T-script are collected into a (internal, non-persistent) database. In the matching phase, the send and receive actions in the database are matched to obtain a table of potential matching events, which can either be stored in a file, or in an external, persistent relational database. To visualize this table, we use the back-end tools of a documentation generator we developed earlier (DocGen [11]).

We used JJForester to implement the parsing of T-scripts and the representation and traversal of T-script parse trees. To this end, we ran JJForester on the grammar of the ToolBus [3] which contains 35 non-terminals and 80 productions (both lexical and context-free). From this grammar, JJForester generated 23 non-terminal classes, 64 constructor classes, and 1 visitor class, amounting to a total of 4221 lines of Java code.

We will now explain in detail how we programmed the two phases of the analysis. Figure 9 shows a UML diagram of the implementation.

3.3.1 The collection phase

We implemented the collection phase as a top-down traversal of the syntax tree with a visitor called SendReceiveVisitor. This refinement of the Visitor class has two kinds of state: a database for storing send and receive actions, and a field that indicates the name of the process currently being analyzed. Whenever a term with outermost function symbol snd-msg or rec-msg is encountered, the visitor will add a corresponding action to the database, tagged with the current process name. The current process name is set whenever a

[3] This SDF grammar can be downloaded from the GrammarBase, at http://www.program-transformation.org/gb.

```
context-free syntax
 "process" ProcessName "is" ProcessExpr
      -> ProcessDef  {cons("procDef")}
 "process" ProcessName "(" {VarDecl ","}* ")" "is" ProcessExpr
      -> ProcessDef  {cons("procDefArgs")}
```

Fig. 10. The syntax of process definitions.

```
public void visitProcDef(procDef definition) {
  currProcess = definition.getIdentifier0().toString();
}
public void visitProcDefArgs(procDefArgs definition) {
  currProcess = definition.getIdentifier0().toString();
}
```

Fig. 11. Specialized visit methods to extract process definition names.

process definition is encountered during traversal. Since sends and receives occur only *below* process definition in the parse tree, the top-down traversal strategy guarantees that the current process name field is always correctly set when it is needed to tag an action.

To discover which visit methods need to be redefined in the SendReceiveVisitor, the ToolBus grammar needs to be inspected. To extract process definition names, we need to know which syntactic constructs are used to declare these names. The two relevant productions are shown in Figure 10. So, in order to extract process names, we need to redefine `visitProcDef` and `visitProcDefArgs` in our specialized SendReceiveVisitor. These redefinitions are shown in Figure 11. Whenever the built-in iterator comes across a node in the tree of type `procDef`, it will call our specialized `visitProcDef` with that `procDef` as argument. From the SDF definition in Figure 10 we learn that a `procDef` has two children: a ProcessName and a ProcessExpr. Since ProcessName is a *lexical* non-terminal, and we chose to have JJForester identify all lexical non-terminals with a single type `Identifier`, the Java class `procDef` has a field of type `Identifier` and one of type `ProcessExpr`. Through the `getIdentifier0()` method we get the actual process name which gets converted to a String so it can be assigned to `currProcess`.

Now that we have taken care of extracting process names, we need to address the collection of communication actions. The ToolBus grammar allows for arbitrary terms ('Atoms' in the grammar) as actions. Their syntax is shown in Figure 12.

Thus, send and receive actions are not distinct syntactical constructs, but they are functional terms (`funTerm`s) where the `Id` child has value `snd-msg` or `rec-msg`. Consequently, we need to redefine the `visitFunTerm` method such that it inspects the value of its first child to decide if and how to collect a communication action. Figure 13 shows the redefined method.

```
context-free syntax
  Vname                 -> Var      {cons("vnameVar")}
  Var                   -> GenVar   {cons("var")}
  Var "?"               -> GenVar   {cons("optVar")}
  GenVar                -> Term     {cons("genvarTerm")}
  Id                    -> Term     {cons("idTerm")}
  Id "(" TermList ")"   -> Term     {cons("funTerm")}
  {Term ","}*           -> TermList {cons("termStar")}
  Term                  -> Atom     {cons("termAtom")}
```

Fig. 12. Syntax of relevant ToolBus terms.

```
public void visitFunTerm(funTerm term) {
  SendReceiveAction action =
              new SendReceiveAction(currProcess,
                                    term.getTermlist1());
  if (term.getIdentifier0().equals("\"snd-msg\"")) {
    srdb.addSendAction(action);
  } else if (term.getIdentifier0().equals("\"rec-msg\"")) {
    srdb.addReceiveAction(action);
  }
}
```

Fig. 13. The visit method for send and receive messages.

The visit method starts by constructing a new `SendReceiveAction`. This is an object that contains the term that is being communicated and the process that sends or receives it. The process name is available in the `SendReceiveVisitor` in the field `currProcess`, because it is put there by the `visitProcDef` methods we just described. The term that is being communicated can be selected from the `funTerm` we are currently visiting. From the SDF grammar in Figure 12 it follows that the term is the second child of a `funTerm`, and that it is of type `TermList`. Therefore, the method `getTermlist1` will return it.

The newly constructed action is added to the database as a send action, a receive action, or not at all, depending on the first child of the `funTerm`. This child is of lexical type `Id`, and thus converted to an `Identifier` type in the generated Java classes. The `Identifier` class contains an `equals(String)` method, so we use string comparison to determine whether the current `funTerm` has "snd-msg" or "rec-msg" as its function symbol.

Now that we have built the specialized visitor to perform the collection, we still need to activate it. Before we can activate it, we need to have parsed a T-script, and built a class structure out of the parse tree for the visitor to operate on. This is all done in the `main()` method of the analyzer, as shown in Figure 14. The main method shows how we use the generated parse method for `Tscript` to build a tree of objects. Tscript.parse() takes a filename as an

```
public static void main(String[] args) throws ParseException {
  String inFile = args[0];
  Tscript theScript = Tscript.parse(inFile);
  SendReceiveVisitor srvisitor = new SendReceiveVisitor();
  theScript.accept_td(srvisitor);       // collection phase
  srvisitor.srdb.constructMatchTable(); // matching phase
}
```

Fig. 14. The main() method of the ToolBus analyzer.

argument and tries to parse that file as a Tscript. If it fails it throws a ParseException and displays the location of the parse error. If it succeeds it returns a `Tscript`. We then construct a new `SendReceiveVisitor` as described in the previous section. The `Tscript` is subsequently told to accept this visitor, and, as described in Section 2.4 iterates over all the nodes in the tree and calls the specific visit methods for each node. When the iterator has visited all nodes, the `SendReceiveVisitor` contains a filled `SendReceiveDb`. The results in this database object can then be processed further, in the matching phase. In our case we call the method `constructMatchTable()` which is explained below.

3.3.2 The matching phase

In the matching phase, the send and receive actions collected in the `SendReceiveDb` are matched to construct a table of potential communication events, which is then printed to file or stored in a relational database. We will not discuss the matching itself in great detail, because it is not implemented with a visitor. A visitor implementation would be possible, but clumsy, since two trees need to be traversed simultaneously. Instead it is implemented with nested iteration over the sets of send and receive actions in the database, and simple case discrimination on terms. The result of matching is a table where each row contains the process names and data of a matching send and receive action.

We focus on an aspect of the matching phase where a visitor *does* play a role. When writing the match table to file, the terms (data) it contains need to be pretty-printed, i.e. to be converted to `String`. We implemented this pretty-printer with a bottom-up traversal with the `TermToStringVisitor`. We chose not to use generated `toString` methods of the constructor classes, because using a visitor leaves open the possibility of refining the pretty-print functionality.

Note that pretty-printing a node may involve inserting literals before, in-between, and after its pretty-printed children. In particular, when we have a list of terms, we would like to print a "," between children. To implement this behavior, a visitor with a single `String` field in combination with a top-down or bottom-up accept method does not suffice. If JJForester would generate *iterating* visitors and *non-iterating* accept methods, this complication would not

```
public void visitIterStarSepTerm_(iterStarSepTerm_ terms) {
  Vector v = terms.getTerm0();
  String str = new String();
  for (int i = 0; i < v.size(); i++){
    if (i != 0) {
      str += ",";
    }
    str += (String) theStack.pop();
  }
  theStack.push(str);
}
```

Fig. 15. Converting a list of terms to a string.

arise. Then, literals could be added to the `String` field in between recursive calls.

We overcome this complication by using a visitor with a *stack* of strings as field, in combination with the bottom-up accept method. The visit method for each leaf node pushes the string representation of that leaf on the stack. The visit method for each internal node pops one string off the stack for each of its children, constructs a new string from these, possibly adding literals in between, and pushes the resulting string back on the stack. When the traversal is done, the user can pop the last element off the stack. This element is the string representation of the visited term. Figure 15 shows the visit method in the `TermToStringVisitor` for lists of terms separated by commas[4]. In this method, the Vector containing the term list is retrieved, to get the number of terms in this list. This number of elements is then popped from the stack, and commas are placed between them. Finally the new string is placed back on the stack. In the conclusion we will return to this issue, and discuss alternative and complementary generation schemes that make implementing this kind of functionality more convenient.

After constructing the matching table, the `constructMatchTable` method writes the table to file or stores it in an SQL database, using JDBC (Java Database Connectivity). We used a visualization back-end of the documentation generator DocGen to query the database and generate a *communication* graph. The result of the full analysis of the T-script in Figure 8 is shown in Figure 16.

3.3.3 Evaluation of the case study

We conducted the ToolBus case study to learn about feasibility, productivity, performance, and connectivity issues surrounding JJForester. Below we briefly

[4] The name of the method reflects the fact that this is a visit method for the symbol {`Term ","`}`*`, i.e. the list of zero or more elements of type Term, separated by commas. Because the comma is an illegal character in a Java identifier, it is converted to an underscore in the method name.

Sender		Receiver	
Pump	report(D)	GasStation	report(D?)
GasStation	change(D)	Customer	change(D?)
Customer	prepay(D,C)	GasStation	prepay(D?,C?)
GasStation	okay(C)	Customer	okay(C)
Operator	remit(Amount)	GasStation	remit(D?)
GasStation	result(D)	Operator	result(D?)
GasStation	activate(D)	Pump	activate(D?)
GasStation	stop	Customer	stop
Customer	turn-on	GasStation	turn-on
Operator	schedule(Payment,C)	GasStation	schedule(D?,C?)
GasStation	request(D,C)	Operator	request(D?,C?)
GasStation	on	Pump	on

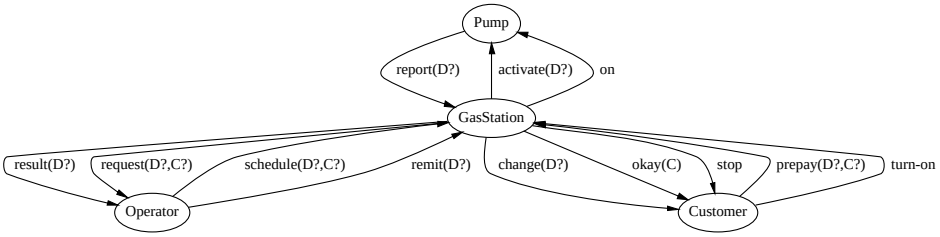

Fig. 16. The analysis results for the input file from Figure 8.

discuss our preliminary conclusions. Apart from the case study reported here, we conducted a case study where an existing Perl component in the documentation generator DocGen was re-implemented in Java, using JJForester. This case study also corroborates our findings.

Feasibility

At first glance, the object-oriented programming paradigm may seem to be ill-suited for language processing applications. Terms, pattern-matching, many-sorted signatures are typically useful for language processing, but are not native to an object-oriented language like Java. More generally, the reference semantics of objects seems to clash with the value semantics of terms in a language. Thus, in spite of Java's many advantages with respect to e.g. portability, maintainability, reuse, its usefulness in language processing is not evident.

The case study, as well as the techniques for coping with traversal scenarios outlined in Section 2, demonstrate that object-oriented programming *can* be applied usefully to language processing problems. In fact, the support offered

by JJForester makes object-oriented language processing not only feasible, but even easy.

Productivity

Recall that the Java code generated by JJForester from the ToolBus grammar amounts to 4221 lines of code. By contrast, the user code we developed to program the T-script analyzer consists of 323 lines. Thus, 93% of the application was generated, while 7% is hand-written.

These figures indicate that the potential for increased development productivity is considerable when using JJForester. Of course, actual productivity gains are highly dependable on which program transformation scenarios need to be addressed (see Section 2.5). The productivity gain is largly attributable to the support for generic traversals.

Components and connectivity

Apart from reuse of generated code, the case study demonstrates reuse of standard Java libraries and of external (non-Java) tools. Examples of such tools are PGEN, SGLR and *implode*, an SQL database, and the visualization back-end of DocGen. Externally, the syntax trees that JJForester operates upon are represented in the common exchange format ATerms. This exchange format was developed in the context of the ASF+SDF Meta-Environment, but has been used in numerous other contexts as well. In [15] we advocated the use of grammars as tree type definitions that fix the interface between language tools. JJForester implements these ideas, and can interact smoothly with tools that do the same. The transformation tool bundle XT [14] contains a variety of such tools.

Performance

To get a first indication of the time and space performance of applications developed with JJForester, we have applied our T-script analyzer to a script of 2479 lines. This script contains about 40 process definitions, and 700 send and receive actions. We used a machine with Mobile Pentium processor, 64Mb of memory, running at 266Mhz. The memory consumption of this experiment did not exceed 6Mb. The runtime was 69 seconds, of which 9 seconds parsing, 55 seconds implosion, and 5 seconds to analyze the syntax tree. A safe conclusion seems to be that the Java code performs acceptably, while the implosion tool needs optimization. Needless to say, larger applications and larger code bases are needed for a good assessment.

4 Concluding remarks

4.1 Contributions

In this paper we set out to combine SDF support of the ASF+SDF Meta-Environment with the general-purpose object-oriented programming language Java. To this end we designed and implemented JJForester, a parser and visitor generator for Java that takes SDF grammars as input. To support generic traversals, JJForester generates non-iterating visitors and iterating accept methods. We discussed techniques for programming against the generated code, and we demonstrated these in detail in a case study. We have assessed the expressivity of our approach in terms of the program-transformation scenarios that can be addressed with it. Based on the case study, we evaluated the approach with respect to productivity, and performance issues.

4.2 Related Work

A number of parser generators, "tree builders", and visitor generators exist for Java. JavaCC is an LL parser generator by Metamata/Sun Microsystems. Its input format is not modular, it allows Java code in semantic actions, and separates parsing from lexical scanning. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The Java Tree Builder (JTB) is another front-end for JavaCC for tree building and visitor generation. JTB generates two iterating (bottom-up) visitors, one with and one without an extra argument in the visit methods to pass objects down the tree. A version of JTB for GJ (Generic Java) exists which takes advantages of type parameters to prevent type casts. Demeter/Java is an implementation of adaptive programming [20] for Java. It extends the Java language with a little (or domain-specific) language to specify traversal strategies, visitor methods, and class diagrams. Again, the underlying parser generator is JavaCC. JJForester's main improvement with respect to these approaches is the support of *generalized* LR parsing. Concerning traversals, JJForester is different from JJTree and JTB, because it generates iterating accept methods rather than iterating visitors. JJForester is less ambitious and more lightweight than Demeter/Java, which is a programming system rather than a code-generator.

ASDL (Abstract Syntax Definition Language [26]) comes with a visitor generator for Java (and other languages). It generates non-iterating visitors and non-iterating accept methods. Thus, traversals are not supported. ASDL does not incorporate parsing or parser generation; it only addresses issues of *abstract* syntax.

In other programming paradigms, work has been done on incorporating support for SDF and traversals. Previously, we combined the SDF support of the ASF+SDF Meta-Environment with the functional programming language Haskell [18]. In this approach, traversal of syntax trees is supported with

updatable, many-sorted folds and fold combinators [19]. Recently, support for generic traversals has been added to the ASF interpreter. These traversals allow concise specification of many-sorted analyses and rephrasing transformations. Stepwise refinement or generic refinement of such traversals is not supported. Stratego [25] is a language for term rewriting with strategies. It offers a suite of primitives that allow programming of (as yet untyped) generic traversals. Stratego natively supports ATerms. It is used extensively in combination with the SDF components of the ASF+SDF Meta-Environment.

4.3 Future Work

Concrete syntax and subtree sharing

Currently, JJForester only supports processing of *abstract* syntax trees. Though the parser SGLR emits full *concrete* parse trees, these are imploded before being consumed by JJForester. For many program transformation problems it is desirable, if not essential, to process concrete syntax trees. A prime example is software renovation, which requires preservation of layout and comments in the source code. The ASF+SDF Meta-Environment supports processing of concrete syntax trees. In order to broaden JJForester's applicability, and to ensure its smooth interoperation with components developed in ASF, we consider adding concrete syntax support.

When concrete syntax is supported, the trees to be processed are significantly larger. To cope with such trees, the ASF+SDF Meta-Environment uses the ATerm library which implements maximal subtree sharing. As a Java implementation of the ATerm library is available, subtree sharing support could be added to JJForester. We would like to investigate the repercusions of such a change to tree representation for the expressiveness and performance of JJForester.

Decoration and aspect-orientation

Adding a Decoration field to all generated classes would make it possible to store intermediate results inside the object structure inbetween visits. This way, a first visitor could calculate some data and store it in thee object structure, and then a second visitor could "harvest" these data and perform some additional calculation on them.

More generally, we would like to experiment with aspect-oriented techniques [16] to customize or adapt generated code. Adding decoration fields to generated classes would be an instance of such customization.

Object-oriented folds and strategies

As pointed out in Sections 2.5 and 3.3.3, not all transformation scenarios are elegantly expressible with our generated visitors. A possible remedy would be to generate additional instances of the visitor class for specific purposes. In particular, visitors for unparsing, pretty-printing, and equality checking

could be generated. Also, the generated visitors could offer additional refinable methods, such as `visitBefore` and `visitAfter`. Another option is to generate iterating visitors as well as non-iterating ones. Several of these possibilities have been explored in the context of the related systems discussed above. Instead of the visitor class, an object-oriented variation on updatable many-sorted folds could be generated. The main difference with the visitor pattern would be that the arguments of visit functions are not (only) the current node, but its children, and only a bottom-up accept method would be available. More experience is needed to establish which of these options would best suit our application domains.

The Visitor pattern, both in the variant offered by JJForester, where iteration is in the accept methods, and in the more common variant where iteration is in the visit methods, is severely limited in the amount of *control* that the user has over traversal behaviour. Generation of classes and methods to support folding would enrich the traversal repertoire, but only in a limited way. To obtain *full* control over traversal behaviour, we intend to transpose concepts from *strategic rewriting*, as embodied by Stratego and the rewriting calculus [9], to the object-oriented setting. In a nutshell the approach comes down to the following. Instead of doing iteration either in visit or accept methods, iteration would be done in neither. Instead, a small set of traversal combinators can be generated for each grammar, in the form of well-chosen refinements of the Visitor class. These traversal combinators would be direct translations of the strategy combinators in the aforementioned rewriting languages. For instance, the sequence combinator $a; b$ can be modelled as a visitor with two fields of type Visitor, and visit methods that apply these two argument visitors one after another. Using such combinators, the programmer can *program* generic traversal strategies instead of merely selecting one from a fixed set. As an additional benefit, such combinators would remove the need for multiple inheritance for combining visitors. We intend to broaden JJForester's generation scheme to generate traversal combinators, and to explore programming techniques with these.

Availability

JJForester is free software, distributed as open source under the GPL license. It can be downloaded from http://www.jjforester.org.

Acknowledgements

We would like to thank Arie van Deursen for his earlier work on building visitors for structures derived from SDF, and the discussions about this work. Ralf Lämmel and Paul Klint provided us with useful comments on a draft version.

References

[1] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Clarendon Press, Oxford, 1995.

[2] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in coöperation with Addison-Wesley, 1989.

[3] J. A. Bergstra and P. Klint. The ToolBus: a component interconnection architecture. Technical Report P9408, University of Amsterdam, Programming Research Group, 1994. Available from http://www.science.uva.nl/research/prog/reports/reports.html.

[4] J. A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[5] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. Submitted for publication, 2000.

[6] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[7] M. G. J. van den Brand et al. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[8] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117. IEEE, 1998.

[9] Horatiu Cirstea and Claude Kirchner. Introduction to the rewriting calculus. Rapport de recherche 3818, INRIA, December 1999.

[10] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[11] A. van Deursen and Tobias Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[14] M. de Jonge, Eelco Visser, and Joost Visser. XT: a bundle of program transformation tools. *Submitted for publication*, 2000.

[15] M. de Jonge and Joost Visser. Grammars as contracts. In *Generative and Component-based Software Engineering (GCSE)*, Erfurt, Germany, October 2000. CD-ROM Proceedings. To be published in Lecture Notes in Computer Science (LNCS), Springer.

[16] Gregor Kiczales, John Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997.

[17] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[18] Jan Kort, Ralf Lämmel, and Joost Visser. Functional transformation systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, September 2000.

[19] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[20] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[21] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[22] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1985.

[23] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

[24] E. Visser et al. The online survey of program transformation. http://www.program-transformation.org/survey.html.

[25] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[26] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–28, Berkeley, CA, October 15–17 1997. USENIX Association.