DeFacto: Language-Parametric Fact Extraction from Source Code

H.J.S. Basten and P. Klint

Centrum Wiskunde & Informatica, P.O. Box 94079,
NL-1090 GB Amsterdam, The Netherlands
H.J.S.Basten@cwi.nl, P.Klint@cwi.nl

Abstract. Extracting facts from software source code forms the foundation for any software analysis. Experience shows, however, that extracting facts from programs written in a wide range of programming and application languages is labour-intensive and error-prone. We present DeFacto, a new technique for fact extraction. It amounts to annotating the context-free grammar of a language of interest with *fact* annotations that describe how to extract elementary facts for language elements such as, for instance, a declaration or use of a variable, a procedure or method call, or control flow statements. Once the elementary facts have been extracted, we use relational techniques to further enrich them and to perform the actual software analysis.
We motivate and describe our approach, sketch a prototype implementation and assess it using various examples. A comparison with other fact extraction methods indicates that our fact extraction descriptions are considerably smaller than those of competing methods.

1 Introduction

A call graph extractor for programs written in the C language extracts (caller, callee) pairs from the C source code. It contains knowledge about the syntax of C (in particular about procedure declarations and procedure calls), and about the desired format of the output pairs. Since call graph extraction is relevant for many programming languages and there are many similar extraction tasks, it is wasteful to implement them over and over again for each language; it is better to take a generic approach in which the language in question and the properties to be extracted are parameters of a generic extraction tool. There are many and diverse applications of such a generic fact extraction tool: ranging from collecting relevant metrics for quality control during development or managing software portfolios to deeper forms of analysis for the purpose of spotting defects, finding security breaches, validating resource allocation, or performing complete software renovations.

A general workflow for *language-parametric* software analysis is shown in Figure 1. Starting point are *Syntax Rules*, *Fact Extraction Rules*, and *Analysis Rules*. *Syntax Rules* describe the syntax of the system or source code to be analyzed. In a typical case this will be the grammar of C, C++, Java or

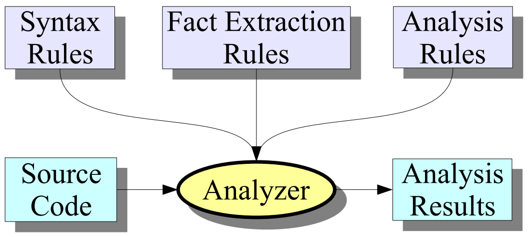

Fig. 1. Global workflow of fact extraction and source code analysis

Cobol possibly combined with the syntax rules for some embedded or application languages. *Fact Extraction Rules* describe what elementary facts have to be extracted from the source code. This may, for example, cover the extraction of variable definitions and uses, and the extraction of the control flow graph. Observe that these extraction rules are closely tied to the context-free grammar and differ per language. *Analysis Rules* describe the actual software analysis to be performed and express the desired operations on the facts, e.g., checking the compatibility of certain source code elements or determining the reachability of a certain part of the code. The *Analyzer* reads the source code and extracts *Facts*, and then produces *Analysis Results* guided by the *Analysis Rules*. Analysis Rules have a weaker link with a programming language and may in some cases even be completely language-agnostic. The analysis of multi-language systems usually requires different sets of fact extraction rules for each language, but only one set of analysis rules.

In this paper we explore the approach just sketched in more detail. The emphasis will be on fact extraction, since experience shows that extracting facts from programs written in a wide range of programming and application languages is labour-intensive and error-prone. Although we will use relational methods for processing facts, the approach as presented here works for other paradigms as well.

The main contributions of this work are an explicit design and prototype implementation of a language-parametric fact extraction method.

1.1 Related Research

Lexical analysis The mother and father of fact extraction techniques are probably Lex [25], a scanner generator, and AWK [1], a language intended for fact extraction from textual records and report generation. Lex is intended to read a file character-by-character and produce output when certain regular expressions (for identifiers, floating point constants, keywords) are recognized. AWK reads its input line-by-line and regular expression matches are applied to each line to extract facts. User-defined actions (in particular print statements) can be associated with each successful match. This approach based on regular expressions is in wide use for solving many problems such as data collection, data mining, fact extraction, consistency checking, and system administration. This same approach is used in languages like Perl, Python, and Ruby. The regular expressions used in an actual analysis are language-dependent. Although the lexical approach works very well for ad hoc tasks, it cannot deal with nested language constructs and in the long turn, lexical extractor become a maintenance burden.

Murphy and Notkin have specialized the AWK-approach for the domain of fact extraction from source code [30]. The key idea is to extend the expressivity of regular expressions by adding context information, in such a way that, for instance, the begin and end of a procedure declaration can be recognized. This approach has, for instance, been used for call graph extraction [31] but becomes cumbersome when more complex context information has to be taken into account such as scope information, variable qualification, or nested language constructs. This suggests using grammar-based approaches.

Compiler instrumentation Another line of research is the explicit instrumentation of existing compilers with fact extraction capabilities. Examples are: the GNU C compiler GCC [13], the CPPX C++ compiler [5], and the Columbus C/C++ analysis framework [12]. The Rigi system [29] provides several fixed fact extractors for a number of languages. The extracted facts are represented as tuples (see below). The CodeSurfer [14] source code analysis tool extracts a standard collection of facts that can be further analyzed with built-in tools or user-defined programs written in Scheme. In all these cases the programming language as well as the set of extracted facts are fixed thus limiting the range of problems that can be solved.

Grammar-based approaches A more general approach is to instrument the grammar of a language of interest with fact extraction directives and to automatically generate a fact extractor. This generator-based approach is supported by tools like Yacc, ANTLR, ASF+SDF Meta-Environment, and various attribute grammar systems [20, 33, 10]. Our approach is an extension of the Syntax Definition Formalism SDF [16] and has been implemented as part of the ASF+SDF Meta-Environment [4]. Its fact extraction can be seen as a very light-weight attribute grammar system that only uses synthesized attributes. In attribute grammar systems the further processing of facts is done using attribute equations that define the values of synthesized and inherited attributes. Elementary facts can be described by synthesized attributes and are propagated through the syntax tree using inherited attributes. Analysis results are ultimately obtained as synthesized attributes of the root of the syntax tree. In our case, the further processing of elementary facts is done by using relational techniques.

Queries and Relations Although extracted facts can be processed with many computational techniques, we focus here on relational techniques. Relational processing of extracted facts has a long history. A unifying view is to consider the syntax tree itself as “facts” and to represent it as a relation. This idea is already quite old. For instance, Linton [27] proposes to represent all syntactic as well as semantic aspects of a program as relations and to use SQL to query them. He encountered two large problems: the lack of expressiveness of SQL (notably the lack of transitive closures) and poor performance. Recent investigations [3, 15] into efficient evaluation of relational query languages show more promising results.

In Rigi [29], a tuple format (RSF) is introduced to represent relations and a language (RCL) to manipulate them. The more elaborate GXL format is described in [18]. In [35] a *source code algebra* is described that can be used to express relational queries on source text. Relational algebra is used in GROK [17], Relation Manipulation Language (RML) [3], .QL [7] and Relation Partition Algebra (RPA) [11] to represent basic facts about software systems and to query them. In GUPRO [9] graphs are used to represent programs and to query them. Relations have also been proposed for software manufacture [24], software knowledge management [28], and program slicing [19]. Vankov [38] has explored the relational formulation of program slicing for different languages. His observation is also that the fact extraction phase is the major stumbling block.

In [2] set constraints are used for program analysis and type inference. More recently, we have carried out promising experiments in which the relational approach is applied to problems in software analysis [22, 23] and feature analysis [37]. These experiments confirm the relevance and urgency of the research direction sketched in this paper. A formalization of fact extraction is proposed in [26].

Another approach is proposed by de Moor [6] and uses path expressions on the syntax tree to extract program facts and formulate queries on them. This approach builds on the work of Paige [34] and attempts to solve a classic problem: how to incrementally update extracted program facts (relations) after the application of a program transformation.

To conclude this brief overview, we mention one example of work that considers program analysis from the perspective of the meta-model that is used for representing extracted data. In [36] the observation is made that the meta-model needs adaptation for every analysis and proposes a method to achieve this.

1.2 Plan of the Paper

We will now first describe our approach (Section 2) and a prototype implementation (Section 3). Next we validate our approach by comparing it with other methods (Section 4) and we conclude with a discussion of our results (Section 5).

2 Description of our Approach

In this section we will describe our fact extraction approach, called DeFacto, and show how it fits into a relational analysis process.

2.1 Requirements

Before we embark on a description of our method, we briefly summarize our requirements. The method should be:

- *language-parametric*, i.e., parametrized with the programming language(s) from which the facts are to be extracted;
- *fact-parametric*, i.e., it should be easy to extract different sets of facts for the same language;

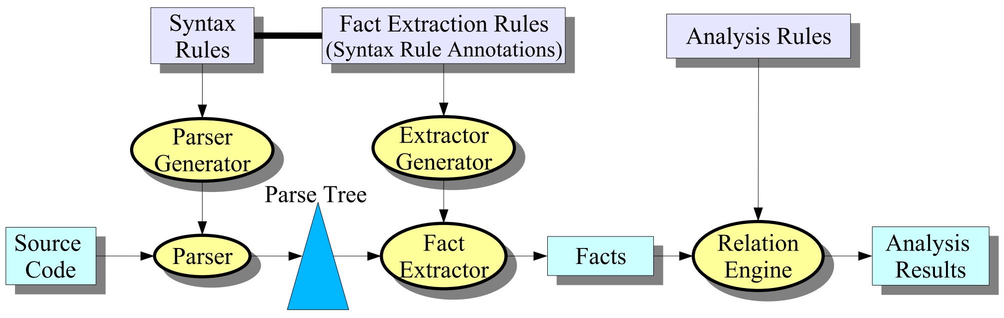

Fig. 2. Global workflow of the envisaged approach

- *local* regarding extracting facts for specific syntax rules;
- *global* when it comes to using the facts for performing analysis;
- *independent* from any specific analysis model;
- *succinct* and should have a high notional efficiency;
- completely *declarative*;
- *modular*, i.e., it should be possible to combine different sets of fact extraction rules;
- *disjoint* from the grammar so that no grammar modifications are necessary when adding fact extraction rules.

2.2 Approach

As indicated above, the main contribution of this paper is a design for a language-parametric fact extraction method. To show how it can be used to accommodate (relational) analysis, we describe the whole process from source code to analysis results. Figure 2 shows a global overview of this process.

As syntax rules we take a context free grammar of the subject system's language. The grammar's productions are instrumented with *fact annotations*, which declare the facts that are to be extracted from the system's source code. We define a fact as a relation between source text elements. These elements are substrings of the text, identified by the nodes in the text's parse tree that yield them. For instance a `declared` relation between two `Statement` nodes of the use and the declaration of a variable. With a *Relational Engine* the extracted facts are further processed and used to produce analysis results. We will discuss these steps in the following sections.

2.3 Fact Extraction with DeFacto

Fact Annotations The fact extraction process takes as input a parse tree and a set of fact annotations to the grammar's production rules. The annotations declare relations between nodes of the parse tree, which identify source code elements. More precisely, a fact annotation describes relation tuples that should be created when its production rule appears in a parse tree node. This can be arbitrary n-ary tuples, consisting of the node itself, its parent, or its children. Multiple annotations can contribute tuples to the same relation.

As an example, consider the following production rule[1] for a variable declaration like, for instance, `int Counter;` or `char[100] buffer;`:

```
Type Identifier ";" -> Statement
```

A fact extraction annotation can be added to this production rule as follows:

```
Type Identifier ";" -> Statement {
    fact(typeOf, Identifier, Type)
}
```

The `fact` annotation will result in a binary relation `typeOf` between the nodes of all declared variables and their types.

In general, a fact annotation with $n + 1$ arguments declares an n-ary relation. The first argument always contains the name of the relation. The others indicate the parse tree nodes to create the relation tuples with, by referring to the production rule elements that will match these nodes. These elements are referenced using their nonterminal name, possibly followed by a number to distinguish multiple elements of the same nonterminal. List elements are postfixed by `-list` and optionals by `-opt`. The keyword `parent` refers to the parent node of the node that corresponds to the annotated production rule.

Annotation Functions Special functions can be used to deal with the parse tree structures that lists and optionals can generate. For instance, if the above production is modified to allow the declaration of multiple variables within one statement we get:

```
Type {Identifier ","}+ ";" -> Statement {
    fact(typeOf, each(Identifier-list), Type)
}
```

Here, the use of the `each` function will extend the `typeOf` relation with a tuple for each identifier in the list. Every tuple consists of an identifier and its type. In general, each function or reference to a production rule element will yield a set (or relation). The final tuples are constructed by combining these sets using Cartesian products. Empty lists or optionals thus result in an empty set of extracted tuples.

Table 1 shows all functions that can be used in fact annotations. The functions `first`, `last` and `each` give access to list elements. The function `next` is, for instance, useful to extract the control flow of a list of statements and `index` can be useful to extract, for instance, the order of a function's parameters.

A function can take an arbitrary number of production rule elements as arguments. The nodes corresponding to these elements are combined into a single list before the function is evaluated. The order of the production rule elements specifies the order in which their nodes should be concatenated.

[1] Production rules are in SDF notation, so the left and right hand sides are switched when compared to BNF notation.

Function	Description
`first()`	First element of a list.
`last()`	Last element of a list.
`each()`	The set of all elements of a list.
`next()`	Create a binary relation between each two succeeding elements of a list.
`index()`	Create a binary relation of type (int, node) that relates each element in a list to its index.

Table 1. Functions that can be used in fact annotations.

As an example, consider the Java constructor body in which the (optional) invocation of the super class constructor must be done first. This can be described by the syntax rule:

```
"{" SuperConstructorInvocation? Statement* "}" -> ConstructorBody
```

To calculate the control flow of the constructor we need the order of its contained statements. Because the SuperConstructorInvocation is optional and the list of regular statements can also be empty, various combinations of statements are possible. By combining all existing statements into a single list, the statement order can be extracted with only one annotation using the `next` function:

```
"{" SuperConstructorInvocation? Statement* "}" -> ConstructorBody {
    fact(succ, next(SuperConstructorInvocation-opt, Statement-list))
}
```

This results in tuples of succeeding statements to be added to the `succ` relation, only if two or more (constructor invocation) statements exist.

Selection Annotations Sometimes however, the annotation functions might not be sufficient to extract all desired facts. This is the case when, depending on the presence or absence of nodes for a list or optional nonterminal, different facts should be extracted, but the nodes of this list or optional are not needed. In these situations the *selection annotations* `if-empty` and `if-not-empty` can be used. They take as first argument a reference to a list or optional nonterminal and as second and optionally third argument a set of annotations. If one or more parse tree nodes exist that match the first argument, the first set of annotations is evaluated, and otherwise the second set (if specified). Multiple annotations can be nested this way.

For instance, suppose the above example of a declaration statement is modified such that variables can also be declared static. If we want to extract a set (unary relation) of all static variables, this can be done as follows:

```
Static? Type Identifier ";" -> Statement {
    if-not-empty(Static-opt, [ fact(static, Identifier) ] )
}
```

Additional relations Apart from the relations indicated with fact annotations, we also extract relations that contain additional information about each extracted node. These are binary relations that link each node to its nonterminal type, source code location (filename + coordinates) and yielded substring. Injection chains are extracted as a single node that has multiple types. This way not every injection production has to be annotated. The resulting relations also become more compact, which requires less complex analysis rules.

2.4 Decoupling Extraction Rules from Grammar Rules

Different facts are needed for different analysis purposes. Some facts are common to most analyses; use-def relations, call relations, and the control flow graph are common examples. Other facts are highly specialized and are seldomly used. For instance, calls to specific functions for memory management or locking in order to search for memory leaks or locking problems.

It is obvious that adding all possible fact extraction rules to one grammar will make it completely unreadable. We need some form of decoupling between grammar rule and fact extraction rules. It is also clear that some form of modularization is needed to enable the modular composition of fact extraction rules.

Our solution is to use an approach that is reminiscent of aspect-oriented programming. The fact extraction rules are declared separately from the grammar, in combinable modules. Grammar rules have a name[2] and fact extraction rules refer to the name of the grammar rule to which they are attached. Analysis rules define the facts they need and when the analysis is performed, all desired fact extraction rules are woven into the grammar and used for fact extraction. This weaving approach is well-known in the attribute grammar community and was first proposed in [8].

2.5 Relational Analysis

Fact annotations only allow the declarations of *local* relations, i.e., relations between a parse tree node and its immediate children, siblings or parent. However this is not sufficient for most fact extraction applications. For instance, the declaration and uses of a local variable can be an arbitrary number of statements apart and are typically in different branches of the parse tree.

In the analysis phase that follows fact extraction we allow the creation of relations between arbitrary parts of the programs. The extracted parse tree nodes and relations do not have to form a tree anymore. They can now be seen as (possibly disconnected) graphs, in which each node represents a source text element. Based on these extracted relations, new relations can be calculated and analyzed. Both for this enrichment of facts and for the analysis itself, we use RSCRIPT, which is explained below.

The focus of fact annotations is thus local: extracting individual tuples from one syntax rule. We now shift to a more global view on the facts.

[2] Currently, we use the constructor attribute `cons` of SDF rules for this purpose.

2.6 RSCRIPT at a glance

RSCRIPT is a typed language based on relational calculus. It has some standard elementary datatypes (booleans, integers, strings) and a non-standard one: source code locations that contain a file name and text coordinates to uniquely describe a source text fragment. As composite datatypes RSCRIPT provides sets, tuples (with optionally named elements), and relations. Functions may have type parameters to make them more generic and reusable. A comprehensive set of operators and library functions is available on the built-in datatypes ranging from the standard set operations and subset generation to the manipulation of relations by taking transitive closure, inversion, domain and range restrictions and the like. The library also provide various functions (e.g., conditional reachability) that enable the manipulation of relations as graphs.

Suppose the following facts have been extracted from given source code and are represented by the relation `Calls`:

```
type proc = str
rel[proc , proc] Calls = {
    <"a", "b">, <"b", "c">, <"b", "d">,
    <"d", "c">, <"d", "e">, <"f", "e">,
    <"f", "g">, <"g", "e">}.
```

The user-defined type `proc` is an abbreviation for strings and improves both readability and modifiability of the RSCRIPT code. Each tuple represents a call between two procedures. The *top* of a relation contains those left-hand sides of tuples in a relation that do not occur in any right-hand side. When a relation is viewed as a graph, its top corresponds to the root nodes of that graph. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
set[proc] entryPoints = top(Calls)
```

In this case, `entryPoints` is equal to `{"a", "f"}`. In other words, procedures `"a"` and `"f"` are the entry points of this application.

We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation:

```
rel[proc, proc] closureCalls = Calls+
```

We know now the entry points for this application (`"a"` and `"f"`) and the indirect call relations. Combining this information, we can determine which procedures are called from each entry point. This is done by taking the *right image* of `closureCalls`. The right image operator determines all right-hand sides of tuples that have a given value as left-hand side:

```
set[proc] calledFromA = closureCalls["a"]
```

yields `{"b", "c", "d", "e"}` and

```
set[proc] calledFromF = closureCalls["f"]
```

yields `{"e", "g"}`. Applying this simple computation to a realistic call graph makes a good case for the expressive power and conciseness achieved in this description. In a real situation, additional information will also be included in the relation, e.g., the source code location where each procedure declaration and each call occurs.

Another feature of RSCRIPT that is relevant for this paper are the *equations*, i.e., sets of mutually recursive equations that are solved by fixed point iteration. They are typically used to define sets of dataflow equations and depend on the fact that the underlying data form a lattice.

3 A Prototype Implementation

We briefly describe a prototype implementation of our approach. With this prototype we have created two specifications for the extraction of the control flow graph (CFG) of Pico and Java programs.

3.1 Description

The prototype consists of two parts: a fact extractor and an RSCRIPT interpreter. Both are written in ASF+SDF [21, 4].

DEFACTO *Fact extractor* The fact extractor extracts the relevant nodes and fact relations from a given parse tree, according to a grammar and fact annotations. We currently use two tree traversals to achieve this. The first identifies all nodes that should be extracted. Each node is given a unique identifier and its non-terminal type, source location and text representation are stored. In the second traversal the actual fact relations are created. Each node with an annotated production rule is visited and its annotations are evaluated. The resulting relation tuples are stored in an intermediate relational format, called RSTORE, that is supported by the RSCRIPT interpreter. It is used to define initial values of variables in the RSCRIPT (e.g., extracted facts) and to output the values of the variables after execution of the script (e.g., analysis results). An RSTORE consists of (name, type, value) triples.

RSCRIPT *interpreter* The RSCRIPT interpreter takes an RSCRIPT specification and an RSTORE as input. A typical RSCRIPT specification contains relational expressions that declare new relations, based on the contents of the relations in the given RSTORE. The interpreter calculates these declared relations, and outputs them again in RSTORE format. Since the program is written is ASF+SDF, sets and relations are internally represented as lists.

3.2 Pico Control Flow Graph Extraction

As a first experiment we have written a specification to extract the control flow graph from Pico programs. Pico is a toy language that features only three types of statements: assignment, if-then-else and while loop. The specification consists of 13 fact annotations and only 1 RSCRIPT expression. The CFG is constructed as follows. For each statement we extract the local `IN`, `OUT` and `SUCC` relations. The `SUCC` relation links each statement to its succeeding statement(s). The `IN` and `OUT` relations link each statement to its first, respectively, last substatement. For instance, the syntax rule for the while statement is:

```
"while" Exp "do" {Statement ";"}* "od" -> Statement
```

It is annotated as follows:

```
"while" Exp "do" {Statement ";"}* "od" -> Statement {
    fact(IN, Statement, Exp),
    fact(SUCC, next(Exp, Statement-list, Exp)),
    fact(OUT, Statement, Exp)
}
```

The three extracted relations are then combined into a single graph containing only the atomic (non compound) statements, with the following RSCRIPT expression:

```
rel[node, node] basicCFG = { <N1, N4> | <node N2, node N3> : SUCC,
    node N1 : reachBottom(N2, OUT), node N4 : reachBottom(N3, IN) }
```

Where `reachBottom` is a built-in function that returns all leaf nodes of a binary relation (graph) that are reachable from a specific node. If the graph does not contain this node, the node is returned instead.

3.3 Java Control Flow Graph Extraction

After the small Pico experiment we applied our approach to a more elaborate case: the extraction of the intraprocedural control flow graph from Java programs. We wrote a DEFACTO and an RSCRIPT specification for this task, with the main purpose of comparing them (see Section 4.2) with the JastAdd specification described in [32]. We tried to resemble the output of the JastAdd extractor as close as possible.

Our specifications construct a CFG between the statements of Java methods. We first build a basic CFG containing the local order of statements, in the same way as the Pico CFG extraction described above. After that, the control flow graphs of statements with non-local behaviour (`return`, `break`, `continue`, `throw`, `catch`, `finally`) are added.

Fact annotations are used to extract information relevant for the control flow of these statements. For instance, the labels of break and continue statements, thrown expressions, and links between try, catch and finally blocks. This information is then used to modify the basic control flow graph. For each return, break, continue and throw statement we add edges that visit the statements of relevant enclosing catch and finally blocks. Then their initial successor edges are removed.

The specifications contain 68 fact annotations and 21 RSCRIPT statements, which together take up only 118 lines of code. More detailed statistics are described in section 4.

4 Experimental Validation

It is now time to compare our earlier extraction examples. In Section 4.1 we discuss an implementation in ASF+SDF of the Pico case (see Section 3.2). In Section 4.2 we discuss an implementation in JastAdd of the Java case (see Section 3.3).

4.1 Comparison with ASF+SDF

Conceptual Comparison
ASF+SDF is based on two concepts *user-definable syntax* and *conditional equations.* The user-definable syntax is provided by SDF and allows defining functions with arbitrary syntactic notation. This enables, for instance, the use of concrete syntax when defining analysis and transformation functions as opposed to defining a separate abstract syntax and accessing syntax trees via a functional interface. Conditional equations (based on ASF) provide the meaning of each function and are implemented by way of rewriting of parse trees.

Fact extraction with ASF+SDF is typically done by rewriting source code into facts, and collecting them with traversal functions. Variables have to be declared that can be used inside equations to match on source code terms. These equations typically contain patterns that resemble the production rules of the used grammar. In our approach we make use of implicit variable declaration and matching, and implicit tree traversal. We also do not need to repeat production rules, because we directly annotate them. However, ASF+SDF can match different levels of a parse tree in a single equation, which we cannot.

Pico control flow extraction using ASF+SDF
CFG extraction for Pico as described earlier in Section 3.2 can be defined in ASF+SDF by defining an extraction function `cflow` that maps language constructs to triples of type `<IN, SUCC, OUT>`. For each construct, a conditional equation has to be written that extracts facts from it and transforms these facts into a triple.

Extraction for statement sequences is done with the following conditional equation:

```
[cfg-1] <In1, Succ1, Out1> := cflow(Stat),
        <In2, Succ2, Out2> := cflow(Stats)
        =================================
        cflow(Stat ; Stats) =
        < In1,
          union(Succ1, product(Out1, In2), Succ2),
          Out2 >
```

The function `cflow` is applied to the first statement `Stat`, and then to the remaining statements `Stats`. The two resulting triples are combined using relational operators to produce the triple for the complete sequence.

Extraction for while statements follows a similar pattern:

DeFacto + Rscript	
Fact extraction rules	
Fact annotations	11
Unique relations	3
Lines of code	*11*
Analysis rules	
Relation expressions	1
Lines of code	*2*
Totals	
Statements	**12**
Lines of code	**13**

Asf+Sdf	
SDF	
Function definitions	2
Variable declarations	10
Lines of code	*17*
ASF	
Equations	6
Lines of code	*31*
Totals	
Statements	**18**
Lines of code	**48**

Table 2. Statistics of Pico Control Flow Graph extraction specifications.

```
[cfg-3] <In, Succ, Out> := cflow(Stats),
        Control := <unparse-to-string(Exp), get-location(Exp)>
        ======================================================
        cflow(while Exp do Stats od) =
        < {Control},
          union(product({Control}, In), Succ, product(Out, {Control})),
          {Control} >
```

The text as well as the source code location of the expression are explicitly saved in the extracted facts. Observe here (as well as in the previous equation) the use of concrete syntax in the argument of `cflow`. The text `while Exp do Stats0 od` matches a while statement and binds the variables `Exp` and `Stats0`.

Comparing the two CFG specifications
Using these and similar equations, leads to a simple fact extractor that can be characterized by the statistics shown in Table 2. Comparing the Asf+Sdf version with our approach one can observe that the latter is shorter and that the fact extraction rules are simpler since our fact annotations have built-in functionality for building subgraphs, while this has to be spelled out in detail in the Asf+Sdf version. The behaviour of our fact annotations can actually be accurately described by relational expressions as occur inside the Asf+Sdf equations shown above.

The Asf+Sdf version and the approach described in this paper both use Sdf and do not need a specification for a separate abstract syntax.

4.2 Comparison with JastAdd

Conceptual Comparison
We have already pointed out that there is some similarity between our fact extraction rules and synthesized attributes in attribute grammars. Therefore we compare our method also with JastAdd [10], a modern attribute grammar system. The global workflow in such a system is shown in Figure 3. Given syntax rules and a definition of the desired abstract syntax tree, a parser generator

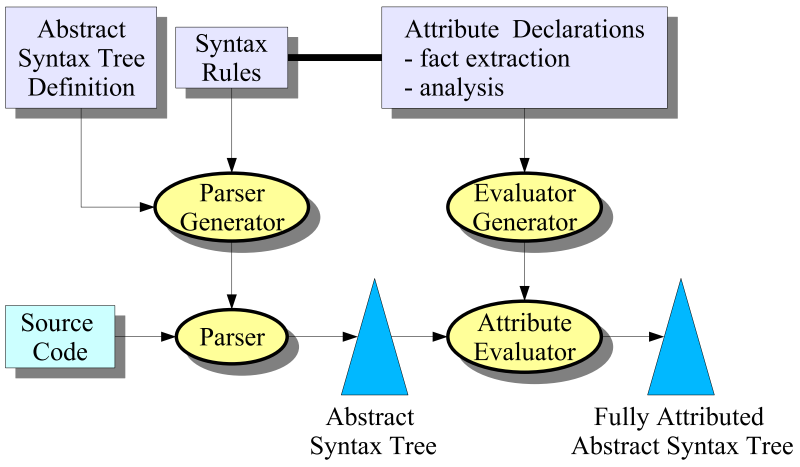

Fig. 3. Architecture of attribute-based approach

produces a parser that can transform source code into an abstract syntax tree. *Attribute Declarations* define the further processing of the tree; we focus here on fact extraction and analysis. Given the attribute definitions, an attribute evaluator is generated that repeatedly visits tree nodes until all attribute values have been computed. The primary mechanisms in any attribute grammar system are:

– *synthesized attributes*: values that are propagated from the leaves of the tree to its root.
– *inherited attributes*: values that are propagated from the root to the leaves.
– *attribute equations* define the correlation between synthesized and inherited attributes.

Due to the interplay of these mechanisms, information can be propagated between arbitrary nodes in the tree. Synthesized attributes play a dual role: for the upward propagation of facts that directly occur in the tree, and for the upward propagation of analysis results. This makes it hard to identify a boundary between pure fact extraction and the further processing of these facts. JastAdd adds to this several other mechanisms: circular attributes, collection attributes, and reference attributes, see [10] for further details.

The definitional methods used in both approaches are summarized in Table 3. The following observations can be made:

– Since we use SDF, we work on the parse tree and do not need a definition of the abstract syntax, which mostly duplicates the information in the grammar and doubles the size of the definition.
– After the extraction phase we employ a global scope on the extracted facts, so no code is needed for propagating information through an AST.
– In the concept of attribute grammars the calculation of facts is scattered across different nonterminal equations, while in our approach the global scope on extracted facts allows for an arbitrary separation of concerns.

Definition	DEFACTO + RSCRIPT	**JastAdd**
Syntax	SDF	Any Java based parser grammar
Abstract Syntax Tree	Not needed, uses Parse Trees	AST definition + Java actions in syntax definition
Fact extraction	Modular fact extraction rules (annotation of syntax rules)	Synthesized attributes, Inherited attributes,
Analysis	RSCRIPT (relational expressions and fixed point equations)	Attribute equations, Circular attributes, Java code

Table 3. Comparison with JastAdd

- The fixed point equations in RSCRIPT and the circular attributes in JastAdd are used for the same purpose: propagating information through the (potentially circular) control flow graph. We use the equations for reachability calculations.
- JastAdd uses Java code for AST construction as well as for attribute definitions. This gives the benefits of flexibility and tool support, but at the cost of longer specifications.
- Our approach uses less (and we, perhaps subjectively, believe simpler) definitional mechanisms, which are completely declarative. We use a grammar, fact extraction rules, and RSCRIPT while JastAdd uses a grammar, an AST definition, attribute definitions, and Java code.

Java control flow extraction using JastAdd

In [32] an implementation of intraprocedural flow analysis of Java is described, which mainly consists of CFG extraction. Here we compare its CFG extraction part to our own specification described earlier in Section 3.3.

The JastAdd CFG specification declares a `succ` attribute on statement nodes, which holds each statement's succeeding statements. Its calculation can roughly be divided into two parts: calculation of the "local" CFG and the "non-local" CFG, just like in our specification. The local CFG is stored in two helper attributes called `following` and `first`. The `following` attribute links each statement to its directly following statements. The `first` attribute contains each statement's first substatement. The following example shows the equations that define these attributes for block statements:

```
eq Block.first() = getNumStmt() > 0 ?
    SmallSet.empty().union(getStmt(0).first()) : following();
eq Block.getStmt(int i).following() = i == getNumStmt() - 1 ?
    following() : SmallSet.empty().union(getStmt(i + 1).first());
```

These attributes are similar to the (shorter) `IN` and `SUCC` annotations in our specification:

```
"{" BlockStatement* "}" -> Block {
    fact(IN, Block, first(BlockStatement-list)),
    fact(SUCC, next(BlockStatement-list)),
    fact(OUT, Block, last(BlockStatement-list))
}
```

DeFacto + Rscript	
Fact extraction rules	
Fact annotations	68
Selection annotations	0
Unique relations	14
Lines of code	*72*
Analysis rules	
Relation expressions	19
Function definitions	2
Lines of code	*46*
Totals	
Statements	**89**
Lines of code	**118**

JastAdd	
Analysis rules	
Synthesized attr. decl.	8
Inherited attr. decl.	15
Collection attr. decl.	1
Unique attributes	17
Equations (syn)	27
Equations (inh)	47
Contributions	1
Lines of Java code	*186*
Totals	
Statements	**99**[1]
Lines of code	**287**

[1] Excluding Java statements

Table 4. Statistics of Java Control Flow Graph extraction specifications.

Based on these helper attributes the `succ` attribute values are defined, which hold the entire CFG. This also includes the more elaborate control flow structures of the `return`, `break`, `continue` and `throw` statements. Due to the local nature of attribute grammars, equations can only define the `succ` attribute one edge at a time. This means that for control flow structures that pass multiple AST nodes, each node has to contribute his own outgoing edges. If multiple control flow structures pass a node, the equations on that node have to handle all these structures. For instance, the control flow of a `return` statement has to pass all `finally` blocks of enclosing `try` blocks, before exiting the function. The equations on `return` statements have to look for enclosing `try-finally` blocks, and the equations on `finally` blocks have to look for contained `return` statements. Similar constructs are required for `break`, `continue` and `throw` statements.

In our specification we calculate these non local structures at a single point in the code. For each `return` statement we construct a relation containing a path through all relevant `finally` blocks, with the following steps:

1. From a binary relation holding the scope hierarchy (consisting of blocks and `for` statements) we select the path from the root to the scope that immediately encloses the `return` statement.
2. This path is reversed, such that it leads from the `return` statement upwards.
3. From the path we extract a new path consisting only of `try` blocks that have a `finally` block.
4. We replace the `try` blocks with the internal control flow of their `finally` blocks.

The resulting relation is then added to the basic control flow graph in one go. Here we see the benefit of our global analysis approach, where we can operate on entire relations instead of only individual edges.

	DeFacto + Rscript			JastAdd			
Extraction	68 / 61%	Fact annos	68	–		
		Selection annos	0			
Propagation	–			58 / 45%	Syn. attrs + eqs	14
					Inh. attrs + eqs	44
Helper stats	11 / 25%	Relation exprs	10	22 / 32%	Syn. attrs + eqs	4
		Function defs	3		Inh. attrs + eqs	18
Calculation	10 / 14%	Relation exprs	9	19 / 23%	Syn. attrs + eqs	17
		Function defs	0		Coll. attrs + contr.	2

Table 5. Statement statistics of Java Control Flow Graph extraction specifications

Comparing the two CFG specifications
Since both methods use different conceptual entities, it is non-trivial to make a quantitative comparison between them. Our best effort is shown in Tables 4 and 5. In Table 4, we give general metrics about the occurrence of "statements" (fact annotation, attribute equation, relational expression and the like) in both methods. Not surprisingly, the fact annotation is the dominating statement type in our approach. In JastAdd this are attribute equations. Our approach is less than half the size when measured in lines of code. The large number of lines of Java code in the JastAdd case is remarkable.

In Table 5 we classify statements per task: extraction, propagation, auxiliary statements, and calculation. For each statement type, we give a count and the percentage of the lines of code used up by that statement type. There is an interesting resemblance between our fact extraction rules and the propagation statements of the JastAdd specification. These propagation statements are used to "deliver" to each AST node information needed to calculate the analysis results. Interestingly, the propagated information contains no calculation results, but only facts that are immediately derivable from the AST structure. Our fact annotations also select facts from the parse tree structure, without doing any calculations. In both specifications the fact extraction and propagation take up the majority of the statements.

It is also striking that both methods need only a small fragment of their lines of code for the actual analysis 14% (Our method) versus 23% (JastAdd).

Based on these observations we conclude that both methods are largely comparable, that our method is more succinct and does not need inline Java code. We also stress that we only make a comparison of the concepts in both methods and do not yet—given the prototype state of our implementation–compare their execution efficiency.

5 Conclusions

We have presented a new technique for language-parametric fact extraction called DeFacto. We briefly review how well our approach satisfies the requirements given in Section 2.1.

The method is certainly *language-parametric* and *fact-parametric* since it starts with a grammar and fact extraction annotations.

Fact extraction annotations are attached to a single syntax rule and result in the extraction of *local* facts from parse tree fragments. Our method does *global* relational processing of these facts to produce analysis results.

Since arbitrary fact annotations can be added to the grammar, it is *independent* from any preconceived analysis model and is fully general. The method is *succinct* and its notational efficiency has been demonstrated by comparison with other methods.

The method is *declarative* and *modular* by design and the annotations can be kept *disjoint* from the grammar in order to enable arbitrary combinations of annotations with the grammar. Observe that this solves the problem of meta-model modification in a completely different manner than proposed in [36].

The requirements we started with have indeed been met.

We have also presented a prototype implementation that is sufficient to assess the expressive power of our approach. One observation is that the intermediate RSTORE format makes it possible to completely decouple fact extraction from analysis. We have already made clear that the focus of the prototype was not on performance. Several obvious enhancements of the fact extractor can be made. A larger challenge is the efficient implementation of the relational calculator but many known techniques can be applied here. An efficient implementation is clearly one of the next things on our agenda.

Our prototype is built upon SDF, but our technique does not rely on a specific grammar formalism or parser. Also, for the processing of the extracted facts, other methods could be used as well, ranging from Prolog to Java. We intend to explore how our method can be embedded in other analysis and transformation frameworks.

The overall insight of this paper is that a clear distinction between language-parametric fact extraction and fact analysis is feasible and promising.

Acknowledgements

We appreciate discussions with Jurgen Vinju and Tijs van der Storm on the topic of relational analysis. We also thank Magiel Bruntink for his feedback on this paper. Jeroen Arnoldus kindly provided his SDFWEAVER program to us.

References

1. A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk - a pattern scanning and processing language. *Software–Practice and Experience*, 9(4):267–280, 79.
2. A. Aiken. Set constraints: Results, applications, and future directions. In *Second International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, 1994.
3. D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137+, 2005.

4. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
5. The CPPX home page. See `http://swag.uwaterloo.ca/~cppx/aboutCPPX.html`, Visited July, 2008.
6. O. de Moor, D. Lacey, and E. van Wyk. Universal regular path queries. *Higher-order and symbolic computation*, 16:15–35, 2003.
7. O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. Keynote address: .ql for source code analysis. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 3–16, Washington, DC, USA, 2007. IEEE Computer Society.
8. G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.
9. J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - generic understanding of programs. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
10. T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007.
11. L.M.G. Feijs, R. Krikhaar, and R.C. Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, april 1998.
12. R. Ferenc, I. Siket, and T. Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.
13. The GCC home page. See `http://gcc.gnu.org/`, Visited July, 2008.
14. GrammaTech. Codesurfer. See `http://www.grammatech.com/products/codesurfer/`, Visited July 2008.
15. E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In David Thomas, editor, *Proceedings of the European Conference on Object-Oriented Programming*, 2006.
16. 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.
17. R.C. Holt. Binary relational algebra applied to software architecture. CSRI 345, University of Toronto, march 1996.
18. R.C. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 162–171. IEEE Computer Society, 2000.
19. D. Jackson and E. Rollins. A new model of program dependences for reverse engineering. In *Proc. SIGSOFT Conf. on Foundations of Software Engineering*, pages 2–10, 1994.
20. M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 209–222, 1990.
21. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.
22. P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

23. P. Klint. Using rscript for software analysis. In *Proceedings of Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, June 2008. To appear.
24. D.A. Lamb. Relations in software manufacture. Technical Report 1990-292, Queen's University School of Computing, Kingston Ontario, 1991.
25. M.E. Lesk. Lex - a lexical analyzer generator. Technical Report CS TR 39, Bell Labs, 1975.
26. Y. Lin and R.C. Holt. Formalizing fact extraction. In *ATEM 2003: First International Workshop on Meta-Models and Schemas for Reverse Engineering*, Victoria BC, November 13 2003.
27. M. A. Linton. Implementing relational views of programs. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, 1984.
28. B. Meyer. The software knowledge base. In *Proceedings of the 8th international conference on Software engineering*, pages 158–165. IEEE Computer Society Press, 1985.
29. H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 10)*, pages 80–86, April 1988.
30. G.C. Murphy and D. Notkin. Lightweight source model extraction. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 116–127, New York, NY, USA, 1995. ACM Press.
31. G.C. Murphy, D. Notkin, W.G. Griswold, and E.S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.
32. E. Nilsson-Nyman, T. Ekman, G. Hedin, and E. Magnusson. Declarative intraprocedural flow analysis of java source code. In *Proceedings of 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, 2008.
33. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
34. R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Joint 6th International Conference on Programming Language Implementation and Logic Programming (PLILP) and 4th International Conference on Algebraic and Logic Programming (ALP)*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1991.
35. S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.
36. D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An extensible meta-model for program analysis. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 380–390, Philadelphia, USA, 2006. IEEE Computer Society.
37. T. van der Storm. Variability and component composition. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, June 2004.
38. I. Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005. See `www.cwi.nl/~paulk/theses/Vankov.pdf`.