Centrum voor Wiskunde en Informatica

Grammars as Contracts

M. de Jonge, J.M.W. Visser

Software Engineering (SEN)

SEN-R0112 May 31, 2001

Report SEN-R0112
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Grammars as Contracts

Merijn de Jonge
email: Merijn.de.Jonge@cwi.nl

Joost Visser
email: Joost.Visser@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
Component-based development of language tools stands in need of meta-tool support. This support can be offered by generation of code – libraries or full-fledged components – from syntax definitions. We develop a comprehensive architecture for such syntax-driven meta-tooling in which grammars serve as contracts between components. This architecture addresses exchange and processing both of full parse trees and of abstract syntax trees, and it caters for the integration of generated parse and pretty-print components with tree processing components.

We discuss an instantiation of the architecture for the syntax definition formalism SDF, integrating both existing and newly developed meta-tools that support SDF. The ATerm format is adopted as exchange format. This instantiation gives special attention to adaptability, scalability, reusability, and maintainability issues surrounding language tool development.

1998 ACM Computing Classification System: D.2.2, D.2.3, D.2.4, D.2.9, D.2.11, D.2.12, D.2.13
Keywords and Phrases: Grammars, contracts, components, languages, tool construction, exchange format, code generation
Note: This work appeared in the proceedings of the Second International Conference on Generative and Component-Based Software Engineering (GCSE'00)

1. Introduction

A need exists for meta-tools supporting component-based construction of language tools. Language-oriented software engineering areas such as development of domain-specific languages (DSLs), language engineering, and automatic software renovation (ASR) pose challenges to tool-developers with respect to adaptability, scalability, and maintainability of the tool development process. These challenges call for methods and tools that facilitate reuse. One such method is component-based construction of language tools, and this method needs to be supported by appropriate meta-tooling to be viable.

Component-based construction of language tools can be supported by meta-tools that generate code – subroutine libraries or full-fledged components – from syntax definitions. Figure 1 shows a global architecture for such meta-tooling. The bold arrows depict meta-tools, and the grey ellipses depict generated code. From a syntax definition, a parse component and a pretty-print component are generated that take input terms into trees and vice versa. From the same syntax definition a library is generated for each supported programming language, which is imported by components that operate on these trees. One such component is depicted at the bottom of the picture (more would clutter the picture). Several of these components, possibly developed in different programming languages can interoperate seamlessly, since the imported exchange code is generated from the same syntax definition.

In this paper we will refine the global architecture of Figure 1 into a comprehensive architecture for syntax-driven meta-tooling. This architecture embodies the idea that grammars can serve as contracts governing all

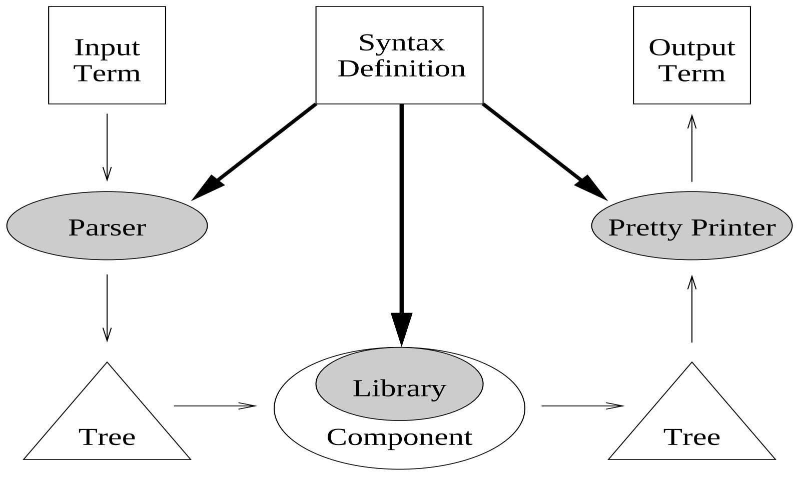

Figure 1: Architecture for meta-tool support for component based language tool development. Bold arrows are meta-tools. Grey ellipses are generated code.

exchange of syntax trees between components and that representation and exchange of these trees should be supported by a common exchange format. An instantiation of this architecture is available as part of the Transformation Tools package XT.

The paper is structured as follows. In Sections 2, 3, and 4 we will develop several perspectives on the architecture. For each perspective we will make an inventory of meta-languages and meta-tools and formulate requirements on these languages and tools. We will discuss how we instantiated this architecture: by adopting or developing specific languages and tools meeting these requirements. In Section 5 we will combine the various perspectives thus developed into a comprehensive architecture. Applications of the presented meta-tooling will be described in Section 6. Sections 7, and 8 contain a discussion of related work and a summary of our contributions.

2. Concrete syntax definition and meta-tooling

One aspect of meta-tooling for component based language tool development concerns the generation of code from *concrete* syntax definitions (grammars). Figure 2 shows the basic architecture of such tooling. Given a concrete syntax definition, parse and pretty-print components are generated by a parser generator and a pretty-printer generator, respectively. Furthermore, library code is generated, which is imported by tool components (Figure 2 shows no more than a single component to prevent clutter). These components use the generated library code to represent parse trees (i.e. *concrete* syntax trees), read, process, and write them. Thus, the grammar serves as an interface description for these components, since it describes the form of the trees that are exchanged.

A key feature of this approach is that meta-tools such as pretty-printer and parser generators are assumed to operate on the same input grammar. The reason for this is that having multiple grammars for these purposes introduces enormous maintenance costs in application areas with large, rapidly changing grammars. A grammar serving as interface definition enables smooth interoperation between parse components, pretty-print components and tree processing components. In fact, we want grammars to serve as contracts governing all exchange of trees between components, and having several contracts specifying the same agreement is a recipe for disagreement.

Note that our architecture deviates from existing meta-tools in the respect that we assume full parse trees can be produced by parsers and consumed by pretty-printers, not just abstract syntax trees (ASTs). These parse trees contain not only semantically relevant information, as do ASTs, but they additionally contain nodes representing literals, layout, and comments. The reason for allowing such concrete syntax information in trees

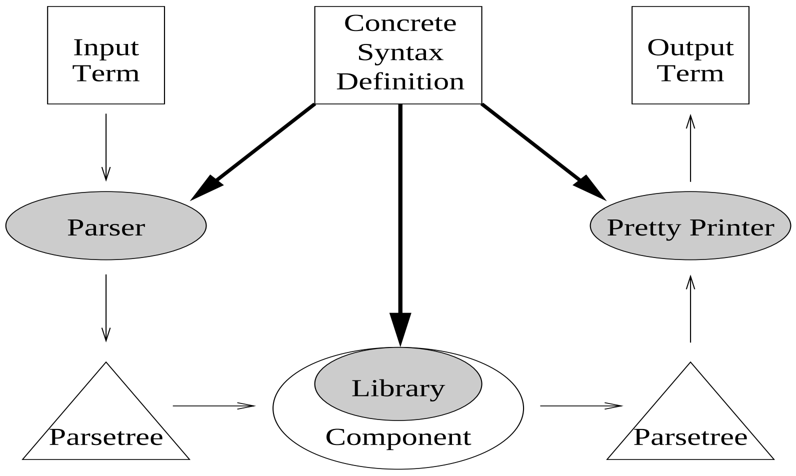

Figure 2: Architecture for *concrete syntax* meta-tools. The concrete syntax definition serves as contract between components. Components that import generated library code interoperate with each other and with generated parsers and pretty-printers by exchanging parse trees adhering to the contractual grammar.

is that many applications, e.g. software renovation, require preservation of layout and comments during tree transformation.

2.1 Concrete syntax definition

In order to satisfy our adaptability, scalability and maintainability demands, the concrete syntax definition formalism must satisfy a number of criteria. The syntax definition formalism must have powerful support for modularity and reuse. It must be possible to extend languages without changing the grammar for the base language. This is essential, because each change to a grammar on which tooling is based potentially leads to a modification avalanche. Also, the grammar language must be purely declarative. If not, its reusability for different purposes is compromised.

In our instantiation of the meta-tool architecture, the central role of concrete syntax definition language is fulfilled by the Syntax Definition Formalism SDF [11]. Figure 3 shows an example of an SDF grammar. This example definition contains lexical and context-free syntax definitions distributed over a number of modules. Note that the orientation of productions is flipped with respect to BNF notation.

SDF offers powerful modularization features. Notably, it allows modules to be mutually dependent, and it allows alternatives of the same non-terminal to be spread across multiple modules. For instance, the syntax of a kernel language and the syntaxes of its extensions can be defined in separate modules. Also, mutually dependent non-terminals can be defined in separate modules. Renamings and parameterized modules further facilitate syntax reuse.

SDF is a highly expressive syntax definition formalism. Apart from symbol iteration constructors, with or without separators, it provides notation for optional symbols, sequences of symbols, optional symbols, and more. These notations for building compound symbols can be arbitrarily nested. SDF is not limited to a subclass of context-free grammars, such as LR or LL grammars. Since the full class of context-free syntaxes, as opposed to any of its proper subclasses, is closed under composition (combining two context-free grammars will always produce a grammar that is context-free as well), this absence of restrictions is essential to obtain true modular syntax definition, and “as-is” syntax reuse.

SDF offers disambiguation constructs, such as associativity annotations and relative production priorities, that are decoupled from constructs for syntax definition itself. As a result, disambiguation and syntax definition are not tangled in grammars. This is beneficial for syntax definition reuse. Also, SDF grammars are purely declarative, ensuring their reusability for other purposes besides parsing (e.g. code generation, pretty-printing).

SDF offers the ability to control the shape of parse trees. The alias construct (see module *Def* in Figure 3)

```
definition
module Exp
exports
  context-free syntax
    Identifier                        → Exp {cons(var)}
    Identifier "(" {Exp ","}* ")" → Exp {cons(fcall)}
    "(" Exp ")"                       → Exp {bracket}

module Let
exports
  context-free syntax
    let Defs in Exp  → Exp {cons(let)}
    Exp where Defs → Exp {cons(where)}

module Def
exports
  aliases
    {( Identifier "=" Exp ) ","}+ → Defs

module Main
imports Exp Let Def

exports
  sorts Exp
  lexical syntax
    [\␣\t\n] → LAYOUT
  context-free restrictions
    LAYOUT?      -/- [\␣\t\n]
```

Figure 3: An example SDF grammar.

allows auxiliary names for complex sorts to be introduced without affecting the shape of parse trees or abstract syntax trees. Aliases are resolved by a normalization phase during parser generation, and they do not introduce auxiliary nodes.

2.2 Concrete meta-tooling

Parsing SDF is supported by *generalized* LR parser generation [15]. In contrast to plain LR parsing, generalized LR parsing is able to deal with (local) ambiguities and thereby removes any restrictions on the context-free grammars. A detailed argument that explains how the properties of GLR parsing contribute to meeting the scalability and maintainability demands of language-centered application areas can be found in [7]. The meta-tooling used for parsing in our architecture consist of a parse table generator, and a generic parse component, called `sglr` , which parses terms using these tables, and generates parse trees [16].

Parse tree representation In our architecture instantiation, the parse trees produced from generated parsers are represented in the SDF parse tree format, called AsFix [16]. AsFix trees contain all information about the parsed term, including layout and comments. As a consequence, the exact input term can always be reconstructed, and during tree processing layout and comments can be preserved. This is essential in the application area of software renovation.

Full AsFix trees rapidly grow large and become inefficient to represent and exchange. It is therefore of vital importance to have an efficient representation for AsFix trees available. Moreover, component based software development requires a uniform exchange format to share data (including parse trees) between components. The ATerm format is a term representation suitable as exchange format for which an efficient representation exists. Therefore AsFix trees are encoded as ATerms to obtain space efficient exchangeable parse trees ([5] reports compression rates of over 90 percent). In Section 3.2 we will discuss tree representation using ATerms in more detail.

Pretty-printing We use GPP, a generic pretty-printing toolset that has been defined in [13]. This set of meta-tools provides the generation of customizable pretty-printers for arbitrary languages defined in SDF. The layout of a language is expressed in terms of pretty-print rules which are defined in an ordered sequence of pretty-print tables. The ordering of tables allows customization by overruling existing formatting rules.

The standard distribution of GPP contains a formatter which operates on AsFix parse trees and supports comment preservation. An additional formatter which operates on ASTs is distributed as part of XT.

Since GPP is an open system which can be extended and adapted easily, support for new output formats (in

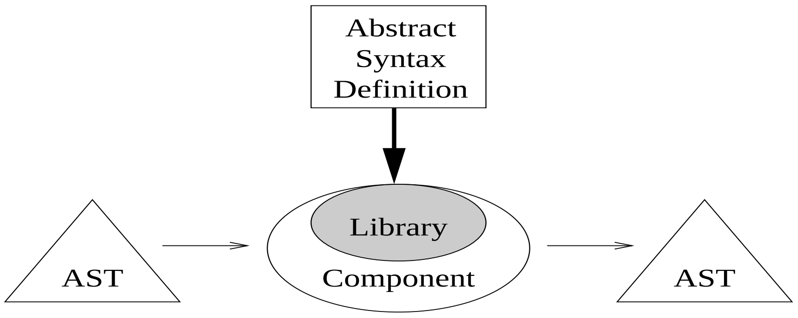

Figure 4: Architecture for *abstract syntax* meta-tools. The abstract syntax definition, prescribing tree structure, serves as a contract between tree processing components.

addition to plain text, LaTeX, and HTML which are supported by default) and language specific formatters can be incorporated with little effort.

3. Abstract syntax definition and meta-tooling

A second aspect of meta-tooling for component based language tool development concerns the generation of code from *abstract* syntax definitions. Figure 4 shows the architecture of such tooling. Given an abstract syntax definition, library code is generated, which is used to represent and manipulate ASTs. The abstract syntax definition language serves as an interface description language for AST components. In other words, abstract syntax definitions serve as tree type definitions (analogous to XML's document type definitions).

3.1 Abstract syntax definition

For the specification of abstract syntax we have defined a subset of SDF, which we call AbstractSDF. AbstractSDF was obtained from SDF simply by omitting all constructs specific to the definition of *concrete* syntax. Thus, AbstractSDF allows only productions specifying prefix syntax, and it contains no disambiguation constructs or constructs for specifying lexical syntax. AbstractSDF inherits the powerful modularity features of SDF, as well as the high expressiveness concerning arbitrarily nested compound sorts. Figure 5 shows an example of an AbstractSDF definition.

The need to define separate concrete syntax and abstract syntax definitions would cause a maintenance problem. Therefore, the concrete syntax definition can be annotated with abstract syntax directives from which an AbstractSDF definition can be generated (see Section 3.3 below). These abstract syntax directives consist of optional constructor annotations for context-free productions (the “cons” attributes in Figure 3) which specify the names of the corresponding abstract syntax productions.

3.2 Abstract syntax tree representation

In order to meet our scalability demands, we will require a tree representation format that provides the possibility of efficient storage and exchange. However, we do not want a tree format that has an efficient binary instantiation only, since this makes all tooling necessarily dependent on routines for binary encoding. Having a human readable instantiation keeps the system open to the accommodation of components for which such routines are not (yet) available. Finally, we want the typing of trees to be *optional*, in order not to preempt integration with typeless, generic components. For instance, a generic tree viewer should be able to read the intermediate trees without explicit knowledge of their types.

ASTs are therefore represented in the ATerm format, which is a generic format for representing annotated trees. In [5] a 2-level API is defined for ATerms. This API hides a space efficient binary representation of ATerms (BAF) behind interface functions for building, traversing and inspecting ATerms. The binary representation format is based on maximal subtree sharing. Apart from the binary representation, a plain, human-readable representation is available.

```
definition
module Exp
exports
  syntax
    "var" ( Identifier )            → Exp
    "fcall" ( Identifier, Exp* ) → Exp

module Let
exports
  syntax
    "let" ( Defs, Exp )       → Exp
    "where" ( Exp, Defs ) → Exp

module Def
exports
  aliases
    ( Identifier Exp )+ → Defs

module Main
imports Exp Let Def
```

Figure 5: Generated AbstractSDF definition.

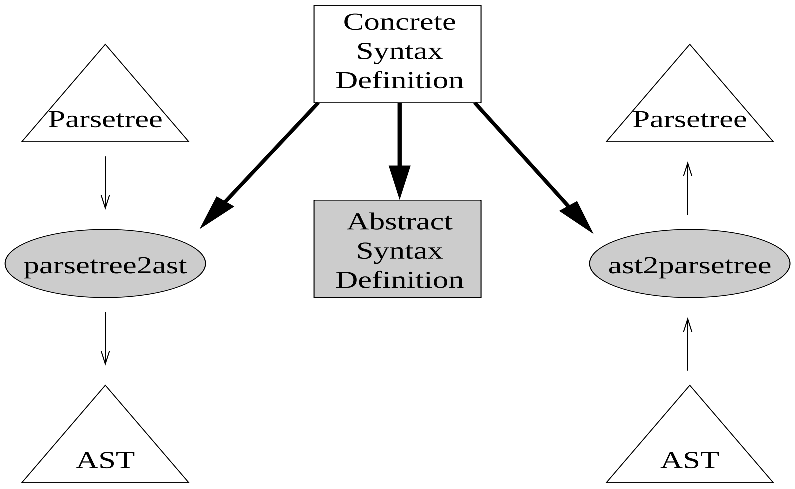

Figure 6: Architecture for meta-tools linking abstract to concrete syntax. The abstract syntax definition is now generated from the concrete syntax definition.

AbstractSDF definitions can be used as type definitions for ATerms by language tool components. In particular, the AbstractSDF definition of the parse tree formalism AsFix serves as a type definition for parse trees (See Section 2). The AbstractSDF definition of Figure 5 defines the type of ASTs representing expressions. Thus, the ATerm format provides a generic (type-less) tree format, on which AbstractSDF provides a typed view.

3.3 Abstract from concrete syntax

The connection between the abstract syntax meta-tooling and the concrete syntax meta-tooling can be provided by three meta-tools, which are depicted in Figure 6. Central in this picture is a meta-tool that derives an abstract syntax definition from a concrete syntax definition. The two accompanying meta-tools generate tools for converting full parse trees into ASTs and vice versa. Evidently, these ASTs should correspond to the abstract syntax definition which has been generated from the concrete syntax definition to which the parse trees correspond.

An abstract syntax definition is obtained from a grammar in two steps. Firstly, concrete syntax productions are optionally annotated with prefix constructor names. To derive these constructor names automatically, the meta-tool `sdfcons` has been implemented. This tool basically collects keywords and non-terminal names from productions and applies some heuristics to synthesize nice names from these. Non-unique constructors are made unique by adding primes or qualifying with non-terminal names. By manually supplying some seed constructor names, users can steer the operation of `sdfcons`, which is useful for languages which sparsely contain keywords.

Secondly, the annotated grammar is fed into the meta-tool `sdf2asdf`, yielding an AbstractSDF definition. For instance, the AbstractSDF definition in Figure 5 was obtained from the SDF definition in Figure 3. This transformation basically throws out literals, and replaces mixfix productions by prefix productions, using the associated constructor name.

Together with the abstract syntax definition, the converters `parsetree2ast` and `ast2parsetree` which translate between parse trees and ASTs are generated. Note that the first converter removes layout and comment information, while the second inserts *empty* layout and comments.

Note that the high expressiveness of SDF and AbstractSDF, and their close correspondence are key factors for the feasibility of generating abstract from concrete syntax. Standard, Yacc-like concrete syntax definition languages are not satisfactory in this respect. Since their expressiveness is low, and LR restrictions require non-natural language descriptions, generating abstract syntax from these languages would result in awkwardly structured ASTs, which burden the component programmers.

4. GENERATING LIBRARY CODE

In this section we will discuss the generation of library code (see Figures 2 and 4). Our language tool development architecture contains code generators for several languages and consequently allows components to be developed in different languages. Since ATerms are used as uniform exchange format, components implemented in different programming languages can be connected to each other.

4.1 Targeting C

For the programming language C an efficient ATerm implementation exists as a separate library. This implementation consists of an API which hides the efficient binary representation of ATerms based on maximal sharing and provides functions to access, manipulate, traverse, and exchange ATerms.

The availability of the ATerm library allows generic language components to be implemented in C which can perform low-level operations on arbitrary parse trees as well as on abstract syntax trees.

A more high-level access to parse trees is provided by the code generator `asdf2c` which, when passed an abstract syntax definition, produces a library of match and build functions. These functions allow easy manipulation of parse trees without having to know the exact structure of parse trees. These high-level functions are type-preserving with respect to the AbstractSDF definition.

4.2 Targeting Java

Also for the Java programming language an implementation of the ATerm API exists which allows Java programs to operate on parse trees and abstract syntax trees. As yet, there is no code generator for Java available to provide high level access and traversals of trees similar to the other supported programming languages. Such a code generator has been designed and is being developed. It will represent syntax trees as object trees, and tree traversals will be supported by generated libraries of refinable visitors.

4.3 Targeting Stratego

Our initial interest was to apply our meta-tooling to program transformation problems, such as automatic software renovation. For this reason we selected the transformational programming language Stratego [17] as the first target of code generation. Stratego offers powerful tree traversal primitives, as well as advanced features such as separation of pattern-matching and scope, which allows pattern-matching at arbitrary tree depths. Furthermore, Stratego has built-in support for reading and writing ATerms. Stratego also offers a notion of pseudo-constructors, called *overlays*, that can be used to operate on full parse trees using a simple AST interface.

Two meta-tools support the generation of Stratego libraries from syntax descriptions. The library for AST processing is generated by `asdf2stratego` from an AbstractSDF definition. The library for combined parse tree and AST processing is generated by `sdf2stratego` from an SDF grammar. The latter library subsumes the former.

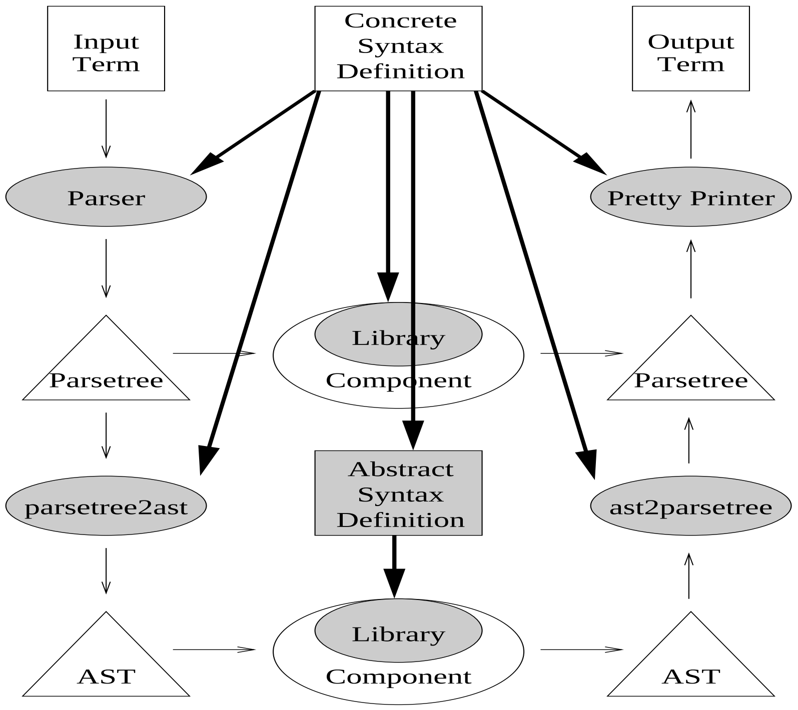

Figure 7: Complete meta-tooling architecture. The grammar serves as the contract governing all tree exchange.

The Stratego code generation allows programming on parse trees as if they were ASTs. Underneath such AST-style manipulations, parse trees are processed in which hidden layout and literal information is preserved during transformation. This style of programming can be mixed freely with programming directly on parse trees. Since Stratego has native ATerm support, there is no need for generating library code for reading and writing trees.

4.4 Targeting Haskell

Work has also been done on targeting Haskell. Code generated in this case is of various kinds. Firstly, datatypes are generated to represent parse trees and ASTs. These datatypes are quite similar to the signatures generated for Stratego. Secondly, code is generated for reading ATerm representations into these Haskell datatypes and writing them to ATerms. Finally, full-fledged transformation frameworks consisting of (monadic) paramorphisms and corresponding algebras are generated to facilitate purely functional transformational programming. The reader is referred to [14] for details and for a software renovation case study.

Note that not only general purpose programming languages of various paradigms can be fitted into our architecture, but also more specialized, possibly very high-level languages. An attribute grammar system, for instance, would be a convenient tool to program certain tree transformation components.

5. A COMPREHENSIVE ARCHITECTURE

Combining the partial architectures of the foregoing subsections leads to the complete architecture in Figure 7. This figure can be viewed as a refinement of our first general architecture in Figure 1, which does not differentiate between concrete and abstract syntax, or between parse trees and ASTs.

The refined picture shows that all generated code (libraries and components), and the abstract syntax definition stem from the same source: the grammar. Thus, this grammar serves as the single contract that governs the structure of all trees that are exchanged. In other words, all component interfaces are defined in a single location: the grammar. (When several languages are involved, there are of course equally many grammars.) This single contract approach eliminates many maintenance headaches during component based development. Of course, careful grammar version management is needed when maintenance due to language changes is not carried out for all components at once.

5.1 Grammar version management

Any change to a grammar, no matter how small, potentially breaks all tools that depend on it. Thus, sharing grammars between tools or between tool components, which is a crucial feature of our architecture, is potentially at odds with grammar *change*. To pacify grammar change and grammar sharing, grammar management is needed.

To facilitate grammar version management, we established a *Grammar Base*, in which grammars are stored. Furthermore, we subjected the stored grammars to simple schemes of grammar version numbers and grammar maturity levels.

To allow tool builders to unequivocally identify the grammars they are building their tool on, each grammar in the Grammar Base is given a name and a version number. To give tool builders an indication of the maturity of the grammars they are using to build their tools upon, all grammars in the Grammar Base are labeled with a maturity level. We distinguish the following levels:

volatile	The grammar is still under development.
stable	The grammar will only be subject to minor changes due to bug fixing.
immutable	The grammar will never change.

Normally, a grammar will begin its life cycle at maturity level *volatile*. To build extensive tooling on such a grammar is unwise, since grammar changes are to be expected that will break this tooling. Once confidence in the correctness of the grammar has grown, usually through a combination of testing, bench-marking, and code inspection, it becomes eligible for maturity level *stable*. At this point, only very local changes are still allowed on the grammar, usually to fix minor bugs. Tool-builders can safely rely on stable grammars without risking that their tools will break due to grammar changes. Only a few grammars will make it to level *immutable*. This happens for instance when a grammar is published, and thus becomes a fixed point of reference. If the need for changes arises in grammars that are stable or immutable, a *new* grammar (possibly the same grammar with a new version number) will be initiated instead of changing the grammar itself.

5.2 Connecting components

The connectivity to different programming languages allows components to be developed in the programming language of choice. The use of ATerms for the representation of data allows easy and efficient exchange of data between different components and it enables the composition of new and existing components to form advanced language tools.

Exchange between components and the composition of components is supported in several ways. First, components can be combined using standard scripting techniques and data can be exchanged by means of files. Secondly, the uniform data representation allows for a sequential composition of components in which Unix pipes are used to exchange data from one component to another. Finally, the ToolBus [3] architecture can be used to connect components and define the communication between them. This architecture resembles a hardware communication bus to which individual components can be connected. Communication between components only takes place over the bus and is formalized in terms of Process Algebra [1].

6. APPLICATIONS

Only preliminary experience is available about actually applying the meta-tooling presented in the previous sections. We will present a selection of such experiences.

To start with, the meta-tooling has been applied for its own development, and for the development of some other meta-tools that it is bundled with in the Transformation Tools package XT. These bootstrap flavored applications include the generation of an abstract syntax definition for the parse tree format AsFix from the grammar of SDF. From this abstract syntax definition, a modular Stratego library for transforming AsFix trees was generated and used for the implementation of some AsFix normalization components. Also, the tools `sdf2stratego`, `sdfcons`, `asdf2stratego`, `sdf2asdf`, and many more meta-tools were implemented by parsing, AST processing in one or more components, and pretty-printing.

Apart from SDF and AbstractSDF, the domain specific languages BOX (for generic formatting), and BENCH (for generating benchmark reports), have been implemented with syntax-driven meta-tooling. In the BOX implementation, a grammar for pretty-print tables was built by reusing the SDF grammar and the BOX grammar. New BOX components were implemented in Stratego and connected to existing BOX components programmed in other languages.

The generated transformation frameworks for Haskell are being applied to software renovation problems. In [14], a COBOL renovation application is reported. It involves parsing according to a COBOL grammar, applying a number of function transformers to solve a data expansion problem, and unparsing the transformed parse trees. The functional transformers have been constructed by refining a transformation framework generated from the COBOL grammar. Application to the development of documentation generators [10] has commenced.

7. RELATED WORK

Syntax-driven meta-tools for language tool development are ubiquitous, but rarely do they address a combination of features such as those addressed in this paper. We will briefly discuss a selection of approaches some of which attain a degree of integration of various features.

- Parser generators such as Yacc [12] and JavaCC are meta-tools that generate parsers from syntax definitions. Compared with SDF and `sglr` , they offer poor support for *modular* syntax definition, their input languages are not sufficiently declarative to be reusable for the generation of other components than parsers, and they do not generally target more than a single programming language.

- The language SYN [4] combines notations for specifying parsers, pretty-printers and abstract syntax in a single language. However, the underlying parser generator is limited to LALR(1), in order to have both parse trees and ASTs, users need to construct two grammars, and code the mapping between trees by hand. Moreover, the expressiveness of the language is much smaller than the expressiveness of SDF, and the language is not modular. Consequently, SYN and its underlying system can not meet our adaptability, scalability and maintainability requirements.

- Wile [20] describes derivation of abstract syntax from concrete syntax. Like us he uses a syntax description formalism more expressive than Yacc's BNF notation in order to avoid warped ASTs. Additionally, he provides a procedure for transforming a Yacc-style grammar into a more "tasteful" grammar. His BNF extension allows annotations that steer the mapping with the same effect as SDF's aliases. He does not discuss automatic name synthesis.

- AsdlGen [19] provides the most comprehensive approach we are aware of to syntax-driven support of component-based language tools. It generates library code for various programming languages from abstract syntax definitions. It offers ASDL as abstract syntax definition formalism, and *pickles* as space-efficient exchange format. It offers no support for dealing with concrete syntax and full parse trees.

  AsdlGen targets more languages than our architecture instantiation does at the moment. The choice of target languages, including C and Java, has presumably motivated some restrictions on the expressiveness of ASDL. ASDL lacks certain modularity features, compared to AbstractSDF: no mutually dependent modules, and all alternatives for a non-terminal must be grouped together. Furthermore, ASDL is much less expressive. It does not allow nesting of complex symbols, it has a very limited range of symbol constructors, and it does not provide module renamings or parameterized modules.

  Unlike ATerms, the exchange format that comes with ASDL is always typed, thus obstructing integration with generic components. In fact, the compression scheme of ASDL relies on the typedness of the trees. The rate of compression is significantly smaller than for ATerms [5]. Furthermore, pickles have a binary form only.

- The DTD notation of XML [8] is an alternative formalism in which abstract syntax can be defined. Tools such as HaXML [18] generate code from DTDs. HaXML offers support both for type-based and for generic transformations on XML documents, using Haskell as programming language. Other languages are not targeted. Concrete syntax support is not integrated.

  XML is originally intended as mark-up language, not to represent abstract syntax. As a result, the language contains a number of inappropriate constructs, and some awkward irregularities from an abstract syntax point of view. XML also has some desirable features, currently not offered by AbstractSDF, such as annotations, and inclusion of DTDs (abstract syntax definitions) in documents (abstract terms).

- Many elements of our instantiation of the architecture for syntax-driven component-based language tool development were originally developed in the context of the ASF+SDF *Meta-Environment* [2, 11, 9]. This is an integrated language development environment which offers SDF as syntax definition formalism and the term rewriting language ASF as programming language. Programming takes place directly on concrete syntax, thus hiding parse trees from the programmers view. Programming, debugging, parsing, rewriting and pretty-printing functionality are all offered via a single interactive user interface. Meta-tooling has been developed to generate ASF-modules for term traversal from SDF definitions [6].

  The ASF+SDF Meta-Environment offers a single programming language (ASF), programming on abstract syntax is not supported. Support for component-based development is (currently) limited to gluing compiled ASF programs that read and write flat terms.

  To provide support for component-based tool development, we have adopted the SDF, AsFix, and ATerm formats from the ASF+SDF Meta-Environment as well as the parse table generator for SDF, the parser `sglr`, and the ATerm library. To these we have added the meta-tooling required to complete the instantiation of the architecture of Figure 7. In future, some of these meta-tools might be integrated into the Meta-Environment.

8. CONTRIBUTIONS

We have presented a comprehensive architecture for syntax-driven meta-tooling that supports component based language tool development. This architecture embodies the vision that grammars can serve as contracts between components under the condition that the syntax definition formalism is sufficiently expressive and the meta-tools supporting this formalism are sufficiently powerful. We have presented our instantiation of such an architecture based on the syntax formalism SDF. SDF and the tools supporting it have agreeable properties with respect to modularity, expressiveness, and efficiency, which allow them to meet scalability and maintainability demands of application areas such as software renovation and domain-specific language implementation. We have shown how abstract syntax definitions can be obtained from grammars. We discussed the meta-tooling which generates library code for a variety of programming languages from concrete and abstract syntax definitions. Components that are constructed with these libraries can interoperate by exchanging ATerms that represent trees.

Acknowledgments The authors thank Arie van Deursen and Eelco Visser for valuable discussions.

References

1. J. Baeten and W. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
2. J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.
3. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.
4. R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.
5. 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.
6. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE, 1997.
7. 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.
8. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.
9. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.
10. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.
11. 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.
12. S. C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
13. M. de Jonge. A Pretty-Printer for Every Occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000).* University of Wollongong, Australia, 2000.
14. J. Kort, R. Lämmel, and J. Visser. Functional transformation systems. In *Proceedings of the*

9th International Workshop on Functional and Logic Programming, Sept. 2000.

15. J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.
16. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.
17. E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 1999.
18. M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *International Conference on Functional Programming (ICFP'99), Paris, France, ACM SIGPLAN*, Sept. 1999.
19. 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, Oct. 15–17 1997. USENIX Association.
20. D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 472–480, Berlin - Heidelberg - New York, May 1997. Springer.