Object-Oriented Technology

ICSE 97

Abstract Syntax from Concrete Syntax

David S. Wile

University of Southern California / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA USA
wile@isi.edu
(310) 822-1511

ABSTRACT

Modern Software Engineering practice advocates the development of domain-specific specification languages to characterize formally the idioms of discourse and jargon of specific problem domains. With poorly-understood domains it is best to construct an abstract syntax to characterize the domain concepts and abstractions before developing a concrete syntax. Often, however, a good concrete syntax exists a priori: sometimes in sophisticated formal languages characterizing (often mathematical) domains but more often in miniature, legacy-code languages, sorely in need of reverse engineering. In such cases, it is necessary to derive an appropriate abstract syntax – or its first cousin, an object-oriented model – from the concrete syntax. This report describes a transformation process that produces a good abstract representation from a low-level concrete syntax specification.

Keywords
Abstract syntax, concrete syntax, domain-specific languages, program transformation, grammars, object-oriented models, reverse engineering.

Sponsor
Defense Advanced Research Projects Agency under contract number F30602-93-C-0240.

ICSE 97 Boston MA USA

ABSTRACT MODELS

In recent years, programming language design and implementation technology have evolved into useful tools for software engineers. The development of Fourth Generation Languages and, more generally, *domain-specific specification languages* to characterize formally the idioms of discourse and jargon of specific problem domains has lead to productivity increases of as much as two orders of magnitude [1, 2, 5, 8]. Hence, language design and implementation techniques have a new-found relevance to software engineering practice.

Many modern tools have been developed for language processing to aid the software engineer in analyzing, simulating, measuring and synthesizing programs. These in turn are implemented using attribute grammars, recursive descent techniques, abstract interpretation, partial evaluation, etc. These tools rely on an abstract syntax representation of the domain-specific language, a representation that just captures the fundamental underlying concepts in the language, stripped of its "syntactic sugar." Such an abstract representation is the goal for producers of *object-oriented models* as well [3], and the two are invariably closely related [7].

Syntax-directed systems like the Cornell Synthesizer Generator [11], Mentor [6], and Gandalf [9] advocate designing the abstract syntax together with the concrete syntax; the Booch method also advocates finding the key abstractions of a domain early in the process. However, there are common situations where a language already exists in concrete form, but an abstract syntax has not yet been designed for it. In order to use modern language processing tools on such a language it is desirable to convert the existing concrete syntax into an appropriate abstract syntax.

One situation where such conversion in necessary is when a problem domain already has a significant amount of formalism ingrained, and all that is desired is to characterize the existing intuitive concepts in an abstract way. Reverse engineering is another arena in which adapting existing concrete syntax is desirable. Many "dusty decks" have unfathomable YACC code characterizing corporation jargon and everyday practice. If these are to be adapted, for example, to modern object-oriented techniques, some way to convert the concrete syntax into an abstract syntax is needed, e.g., into a set of C++ class declarations.

Present-day technologies in actual use in software engineering practice for developing concrete syntax and mapping it into an abstract syntax are rather primitive. YACC appears to be the tool of choice for specifying concrete syntax with an accompanying program to map it into abstract syntax. (Actually, there is little discipline here – programmers can do all kinds of semantic manipulations during the parse, perhaps circumventing the invention of an abstract syntax altogether!)

Hence, the technique presented in this report can be used as a step in the disciplined development of support tools for language manipulation. The process of converting from concrete to abstract actually is quite straightforward. One can argue that much of the technique presented below actually represents a way to *improve* existing abstract syntaxes. Hence, as a side-benefit, this paper can be used to examine existing abstract models for clumsiness of expression.

The process is described in enough detail that software engineers can apply the process manually and produce good results. The process is indeed heuristic. However, with a little "advice" from the software engineer, a fully automatic version of the process can be implemented to convert concrete syntax into a *tasteful* abstract syntax design. Indications of how this was done in the author's language processing system, Popart, are presented below. Examples from the concrete syntax for CORBA's IDL [10] are used to illustrate the process.

RELATIONSHIPS BETWEEN ABSTRACT AND CONCRETE SYNTAX

In order to discuss the relationships between concrete and abstract syntax, languages for expressing each of them are necessary. YACC's sublanguage for expressing concrete syntax is characterized by production definitions, comprised of a *production name,* followed by the colon symbol, followed by a set of *alternatives* separated by "|"s.[1] Each alternative is a sequence of *nonterminals* (other production names) or *lexical class denotations.* For example,

```
type_property_list:
        LRPAR opt_extent_spec
        opt_key_spec RRPAR

interface_header:
        INTERFACE identifier
        opt_inheritance_spec
                opt_type_property_list

opt_inheritance_spec:
        /* no inheritance specifier */  |
        COLON inheritance_spec

inheritance_spec:
        scoped_name | scoped_name
                COMMA  inheritance_spec

interface:  interface_dcl   |  forward_dcl
```

describe the nonterminals, *type_property_list, interface_header, opt_inheritance_spec, inheritance_spec* and *interface.* The *interface_header* is defined to be the word "interface" (from the lexeme class with the same name) followed by an *identifer* followed by an *opt_inheritance_spec* followed by an *opt_type_property_list.* The *opt_inheritance_spec* is either the empty string or a colon (from the lexeme class, COLON) followed by an *inheritance_spec,* i.e. it is optionally this latter alternative. (The names "_opt" and "_list" are not formal parts of the grammar description language, but rather helpful mnemonics provided in the names. One cannot, of course, count on this having

[1] Interspersed with code executed during the parse, which we ignore here.

been adhered to in the description of arbitrary concrete syntaxes.) And an *interface* is simply an *interface_dcl* or a *forward_dcl*.

The goal of an abstract syntax is to describe the structural essence of a language [6]. Syntax trees are *operators* – describing the important concepts in a language -- applied to *typed operands* – describing the important components associated with the concept. Each operand is named with the *role* it plays in the concept. Trees are classified in a tree of types known as *phyla* – describing inheritance relationships between concepts. Abstract syntax trees terminate in a prespecified set of primitive types, such as *identifier* and *integer*. Here we add the ability to reference *list of phylum* and *optional phylum* as well. For example,

operator type_property_list :
(es|***optional*** extent_spec ,
ks|***optional*** key_spec)

operator interface_header :
(id|identifier ,
inheritance | ***optional list of*** scoped_name,
pl|***optional*** type_property_list)

phylum interface :=
interface_dcl ***union*** forward_dcl

describes the *type_property_list* operator and the *interface_header* operator. The latter has operands named *id*, *inheritance*, and *pl*, whose types are *identifier*, *list of scoped_name*, and *type_property_list*, respectively. Notice that *inheritance* identifies the role of the *scoped_names* not otherwise inferable from the specification. An *interface* is a phylum with subphyla *interface_dcl* and *forward_dcl*. These latter may either be operators or phyla themselves. Any particular abstract syntax representation will need to implement these concepts in an appropriate representation (e.g. *lists* in C++ might use a *List* template).

The relationship between this abstract syntax tree example and the concrete syntax above illustrates several of the concerns when mapping between the two. First, many of the mappings will be straightforward. The *interface* nonterminal became a phylum in the abstract syntax. However, there are cases where extra lexical elements make this less obvious, as will be illustrated below. The *interface_header* operator is *nearly* the obvious mapping, wherein nonterminals referenced on the right hand side of the production become operands of the operator named by the nonterminal on the left hand side of the production. The only deviation here is that what was an *opt_inheritance_spec* has been expanded in the definition of *interface_header*.

Such substitution or "unfolding" is used frequently in converting from the concrete to the abstract. Several instances of it occur in this example. Notice that all references to nonterminals beginning with *opt_* have been removed. Also recursive structures like *inheritance_spec* have turned into lists. Probably the only arbitrary relationship between the two regards the naming of the operand slots.

Abstract Syntax Induced by the Concrete Syntax

For nearly 20 years I have been developing a syntax-based system, called Popart, to be used in prototyping languages and their semantics rapidly [12]. A distinguishing feature of the system is that the language for describing the concrete syntax – an extension of BNF called "WBNF" – induces the abstract syntax entirely automatically. There are two reasons that this abstract syntax is suitable. First, concrete syntactic structures support the higher level abstract syntax notions mentioned above, such as optionality and lists. Second, a simple annotation can be placed on a production to influence the abstract syntax that is derived.

This is quite unlike the existing syntax-manipulating systems mentioned above, which are based on the abstract syntax, with explicit mapping between abstract and concrete required. One of the implications of including high level constructs in WBNF is that there are many ways to represent the same concrete syntax in the language. Hence, the language designer is trading simplicity of expression for additional structure in the abstract syntax tree. Fortunately, there is a set of transformations that preserves the concrete syntax but varies the induced abstract syntax. These enable the transformation of a concrete syntax in

YACC into a "tasteful" abstract syntax definition for the same language in C++.

The conventions used in WBNF grammars are:

- concatenation:
  block := declarations statements;
- alternation:
  declaration := function | procedure;
- constant lexical items:
  plusop := '+ | '-;
- optionality:
  int := 'integer identifier { 'in range };
- "Kleene plus":
  statements := statement + ;
- lists with separators:
  funcall := identifier '(expr ^ ', ');
- precedence:
  expr := < ('+ | '-), ('* | '/), '^ > primitive;
- nesting:
  'declare ('integer | 'real) variable;
- variable lexical items:
  LEXEME <| alphanumeric;

Each production composed from these constructs can be converted automatically to abstract syntax declarations. Generally, conversion is a recursive process (denoted *T*) over the pattern elements on the production's right hand side, generating operands with appropriate roles and types for the operator named by the nonterminal on the left hand side of the production. However, sometimes the translation needs to be modified to produce subphyla where operands would naturally be produced by *T*. This is indicated by the grammar designer with the symbol "||" at the end of the WBNF production.

That is,

T [<production-name> := <pattern>;]

is

operator <production-name> :
(*T* [<pattern>])

However, alternation of nonterminals

<production-name> :=
<nonterminal -1> |...| <nonterminal -n>;

is almost always represented compactly, viz.

<production-name> :=
<nonterminal -1>|...|<nonterminal -n> ||
;

T of this will produce the abstract syntax declaration:

phylum <production-name> :=
<nonterminal -1> ***union***
... ***union*** <nonterminal -n>

The following covers most cases:

T [<pattern-1> | ... |<pattern-n>] ➔
T [<pattern-1>],...,*T* [<pattern-n>]

T [<pattern-1> ... <pattern-n>] ➔
T [<pattern-1>],...,*T* [<pattern-n>]

T [<nonterminal> +] ➔
<role name> | ***list of*** <nonterminal>

T [<nonterminal> ^ <constant>] ➔
<role -name> | ***list of*** <nonterminal>

T [{<pattern>}] ➔ ***optional*** * *T* [<pattern>]

T [<nonterminal>] ➔
<role- name> | <nonterminal>

T [<constant>] ➔

(Here, "*" indicates application of ***optional*** to all phyla resulting from application of *T* to the pattern.)

Hence, *T* translates

interface_header := ' INTERFACE identifier
{ ' : scoped_name ^ ' , }
{ type_property_list } ;

into:

operator interface_header :
(identifier | Identifier ,
scoped_name |***optional list of***
Scoped_name,
type_property_list|***optional***
Type_property_list)

Actually, in WBNF one can annotate occurrences of nonterminals to specify role names:

unary_expr := { unary_operator#op}
primary_expr#ex;

generates:

operator unary_expr :
(op | unary_operator, ex | primary_expr)

Generally, constant lexical items are ignored by *T*, except when the entire right hand side of a production consists of constants. Then an abstract syntax representation for the enumerated type is generated:

plusop := '+ | '-;

generates[2]:

operator plusop : (lexeme|{PLUS,MINUS})

To summarize, the translation takes each nonterminal that has been indicated as "compact" and turns that into a phylum declaration with the right hand side types as subtypes. Each nonterminal that is not compact becomes an operator with operands having the types of the nonterminals on the right hand side. Role names should be generated that describe the relationship of the operand to the operator. If the nonterminal is embedded in an optional clause, a corresponding construct should be introduced on the right hand side; similarly for lists.

However, there are a few more special cases where the translation is not so straightforward. For example, with

unary_expr := { unary_operator#op}
primary_expr#ex;

most *unary_exprs* will not involve a *unary_operator*. Whether one wants the extra indirection in the abstract syntax is a matter of taste. Often, instead, this will be represented as a phylum with name *unary_expr* with two subtypes: *primary_expr* and *rep_unary_expr*, where the latter operator is defined:

operator rep_unary_expr : (op | unary_ operator, ex | primary_expr)

phylum unary_expr : primary_expr ***union*** rep_unary_expr

Again, WBNF uses the "||" to force this interpretation in the induced abstract syntax, viz.

unary_expr := { unary_operator#op}
primary_expr#ex || ;

A HEURISTIC CONVERSION PROCESS

The heuristic process involved in transforming from the YACC syntax to C++ is:

- Convert YACC to WBNF;
- Distribute concatenation across alternation and remove excessive nesting;
- Reduce special left-recursive patterns;
  - Introduce iteration;
  - Introduce binary operators (precedence);
  - Rewrite as simple alternatives;
- Unfold certain iterative and optional definitions arrived at above, from bottom to top;
- Cull out productions that can represent phyla;
- Distinguish duplicate labels in remaining productions;
- Determine which constants should be folded into new productions;
- Replace constants with references to productions containing them;
- Remove unused productions
- Convert to C++ class declarations.

The process will be illustrated by examples. First consider an example portion of a YACC grammar:

model: specification

specification: definition
| definition specification

definition:
type_dcl SEMI |
const_dcl SEMI |
except_dcl SEMI |
interface SEMI |
module SEMI

The first stage of the algorithm is to convert this to the following WBNF specification:

[2] An enumerated phylum is induced.

```
model := specification ;

specification := definition
                 | definition specification ;

definition :=
        type_dcl '; | const_dcl '; | except_dcl '; |
          interface '; | module '; ;
```

This is arrived at by simply substituting all the lexical constants with the appropriate constant. (This is unnecessary as long as we keep track of the fact that a reference is to a constant lexical class rather than to another nonterminal, but the examples are easier to read this way.) Replace ":" by ":=" and add a semicolon to the right of each production. Trivial.

It is important to emphasize that the process of transforming the resulting WBNF productions maintains the concrete syntax as invariant while improving the abstract syntax.

Because the process both removes existing nonterminals and introduces new ones (eventually phyla and operators), important nonterminals must be protected from removal. So the usage of nonterminal names in the grammar is analyzed next to determine which productions cannot be reached by productions other than themselves. In YACC, this will only be the distinguished start symbol but other grammars may allow multiple entry points. It is important not to discard these in the conversion process. Other important nonterminals may be "evident" to a software engineer as well. Throughout the conversion process, these should not be discarded, even if the algorithm suggests doing so.

The next step of the process is motivated by the fact that a phylum should not have multiple operands representing the same abstract entity. For example, the *definition* in *specification* above plays the same role semantically in either alternative. The distribution transformation takes alternatives and variously groups them or turns them into options. For example,

```
specification := definition
                 | definition specification ;
```

will be turned into

```
specification := definition { specification } ;
```

Similarly,

```
definition := type_dcl '; | const_dcl '; |
              except_dcl '; | interface '; |
              module '; ;
```

will be turned into[3]

```
definition :=
        ( type_dcl | const_dcl | except_dcl
        | interface | module ) '; ;
```

The next phase is to introduce WBNF's iterative constructs into the grammar. For example,

```
key_list :=  key |  key ', key_list ;
```

will turn into

```
key_list := key ^ ' , ;
```

This is accomplished by looking for particular recursive patterns in the definitions. This introduction is warranted in terms of the abstract syntax that will be produced because the field (*key* in this case) will contain a list (or array) of elements. Precedence and the forms of recursion that WBNF permits are then introduced.

Precedence introduction into the WBNF corresponds to recognizing certain constants from the grammar as infix operators. For example, the productions:

```
add_expr :=  mult_expr
           |  add_expr '+ mult_expr
           |  add_expr '- mult_expr ;

mult_expr := unary_expr
           |  mult_expr '* unary_expr
           | mult_expr '/ unary_expr
           | mult_expr '% unary_expr ;
```

use a stylized pattern of left recursion that would be expressed in WBNF:

```
add_expr := < ( '+ | '-) , ( '* | '/ | '%) >
                unary_expr || ;
```

It is worth mentioning that this must be converted to abstract syntax in a fashion similar to the way the unary operator was above, viz. by introducing a *rep_add_expr* class:

[3] Actually, a phylum should be introduced here: the semicolon should have no effect on the induced abstract syntax. Hence, a "| |" should be added to the end of the production.

```
operator
  rep_add _expr :
      (operator|(PLUS,MINUS,STAR,
                SLASH,PERCENT},
      left | add_expr,
      right | add_expr)
phylum add_expr :
      unary_expr union rep_add _expr
```

In tastefully constructed abstract syntax we rarely have a type with only one field, which itself holds a list of elements. Similarly, whenever the entire right hand side of a production is optional, it is more common to put the optionality with the situation in which it is optional. Hence,

```
interface_header :=
      'INTERFACE identifier
        opt_inheritance_spec
        opt_type_property_list ;

opt_inheritance_spec := { ': inheritance_spec } ;

inheritance_spec := scoped_name ^ ', ;

opt_type_property_list := { type_property_list }
      ;
```

should be turned into:

```
interface_header :=
      ' INTERFACE identifier
      { ' : scoped_name ^ ' , }
      { type_property_list } ;
```

So the algorithm first determines which productions have right hand sides that are simply options or iterations. The definitions are unfolded in a bottom-up fashion. For example, the *inheritance_spec* disappeared in the example above, for just this reason[4].

Two more transformations arise directly from considerations on how WBNF induces an abstract syntax. First, although some fields in the original syntax that have the same name and occur on the right hand side of a production mean the same field, some do not – in particular, multiple occurrences in the same alternative. Such fields must be named uniquely while still conveying the abstract type. In WBNF, such names are indicated by using a "#" sign followed by an identifier. Hence, the algorithm will turn:

```
inverse_traversal_path := identifier ' :: identifier
```

into:

```
inverse_traversal_path :=
      identifier#A ' :: identifier#B ;
```

Of course, a person can invent more mnemonic names, such as:

```
inverse_traversal_path :=
      identifier#source ' :: identifier#target ;
```

Another detail concerning the relationship between concrete and abstract syntaxes is that alternations of constants in the concrete syntax often correspond to discriminators in the abstract syntax. Hence, our next concern is to find alternations of constants (and optional constants) occurring within the tree and to break them out explicitly into their own productions. Thus

```
attr_dcl :=
      { 'READONLY }
      ' ATTRIBUTE domain_type identifier
            { fixed_array_size } ;
```

would occasion the introduction of

```
attr_dcl :=
      { con_2 }
      ' ATTRIBUTE domain_type identifier
      { fixed_array_size } ;

con_2 := ' READONLY ;
```

Again, better names for introduced productions can be chosen by people cognizant of the context. Sometimes these constants need not be separated out; they really are "syntactic sugar," in which case the process should not introduce the new production.

Finally, all that is left to do is to eliminate the productions that are not used any more, i.e., those no longer reachable from the original top level nonterminals, and convert the WBNF to an appropriate language for expressing abstract syntax.

Implementation Details

The abstract syntax produced by Popart is actually produced in Common Lisp. However, a simple

[4] Again, not all unfoldings performed in the process are necessarily desirable; some productions may have mnemonic value and should not be removed.

translator from WBNF has been written that produces C++ in the following straightforward way. The phyla become virtual class declarations. The operators are normal class declarations. Both have their immediate superphyla as superclasses. The only issues involve the uses of lists and optional elements. The former is easily handled by using a *List* template. The latter is similarly solved by using a pointer in the slot, which, when *null*, indicates an empty slot. The only problematic issue is that virtual classes (actually any class hierarchy) must be typed by a pointer. Hence, an operand cannot be both *required* and, say, an *expression*. Of course, methods could be introduced to protect the syntax, but if tools such as parsers and transformers manipulate it, this may be unnecessary.

This heuristic process has been implemented in the Popart system, running in Common Lisp. Further details are available in [13].

The automatic conversion of Corba's IDL into abstract syntax was especially rewarding, since the grammar is sizable and previously, we had used *ad hoc* methods for accomplishing the translation. The version produced by the tool differed in name selection and in that the automated version took advantage of some opportunities missed in the (arduous) hand translation process.

RELATED WORK

Abstract syntax has been around for decades; one can easily argue that Lisp was the first instantiation! In the late 1970s the Mentor group proselytized the concept [6]. About the same time recognition that languages are algebras (with abstract syntax signatures) was recognized by the ADJ-group [4]. All syntax-directed tool suites are founded on abstract syntax: Gandalf [9] and the Cornell Synthesizer Generator [11], for example. Its analog in object classes has been recognized by Lieberherr [7]. He also shows how conversion between LL1 languages and abstract syntax is possible. However, the most effort relating the two has been spent on printing abstract syntax trees in concrete syntax (one of many possible such syntaxes). I am not aware of anyone attempting to solve the problem addressed here.

CONCLUSIONS

There are three possible uses for the algorithm proposed above:

- Converting existing concrete syntax by hand to an abstract syntax;
- Implementing an algorithm to do such a conversion;
- Analyzing modern concrete syntax proposals for appropriateness in modern BNF variants.

The use of YACC as the starting point for the conversion process is quite arbitrary: any syntax specification that is less rich than Popart's will do.

This last use has been especially fruitful as a teaching aid for the construction of appropriate concrete syntax designs for Popart. The clean-up portion of the algorithm is routinely run over new grammars to check for sloppy designs.

ACKNOWLEDGEMENTS

I would like to thank Neil Goldman at ISI for stimulating me to design this process. His hand-conversion of IDL suggested new heuristics during its development. I would also like to thank the two ICSE referees for their substantive comments.

REFERENCES

1. Balzer, R., Feather, M., Goldman, N., and Wile, D. "Domain Specific Notations for Command and Control Message Passing", *Internal report:* USC/Information Sciences Institute, Marina del Rey, CA (1994).

2. Batory, D. "Scalable Software Libraries," *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering* (1993), pp. 191-197.

3. Booch, G.*Object-Oriented Analysis and Design.*Benjamin/Cummings Publishing. (1994)

4. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. "Initial Algebra Semantics and Continuous Algebras," *Journal of the ACM* 24:1(1977), pp. 68-95.

5. Kieburtz, R., Bellegarde, F., Bell, J. , Hook, J., Lewis, J., Oliva, D. Sheard, T., Walton, T, and Zhou, T. "Calculating Software Generators from Solution Specifications," *Technical Report # CS/E-94-032B of the Oregon Graduate Center* (1994).

6. Donzeau-Gouge, V., Kahn, G., Lang, B., and Mélèse, B. "Document Structure and Modularity in Mentor," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Symposium on Practical Software Development Environments* (1984), pp. 141-148.

7. Lieberherr, K. *Adaptive Object--Oriented Software: The Demeter Method with Propagation Patterns,* PWS Publishing Company, Boston (1996)

8. Levy, L. S. *Taming the Tiger: Software Engineering and Software Economics,* Springer-Verlag: New York (1987).

9 . Notkin, D., Ellison, R. J., Staudt, B. J., Kaiser, G. E., Kant, E., Habermann, A. N., Ambriola, V., and Montangero, C. Special issue on the GANDALF project, *Journal of Systems and Software* 5, 2(May 1985).

10. The Object Management Group, *The Common Object Request Broker: Architecture and Specification,* OMG Document # 91.12.1 Digital Equipment, HP, Hyperdesk, NCR, Object Design, Sunsoft (1992).

11. Reps, T. W., and Teitelbaum, T. *The Synthesizer Generator.* Springer-Verlag, New York (1988).

12. Wile, D. S. *Popart: Producers of Parsers and Related Tools,* Reference Manual. USC/Information Sciences Institute, Marina del Rey, CA (1993).

13. Wile, D. S. "Toward a Calculus for Abstract Syntax Trees," in *Proceedings of a Workshop on Algorithmic Languages and Calculi* IFIP's TC2, Strasbourg (Feb., 1997). (To appear.)