

Object-Oriented Pattern-Based Language Implementation

*Xiaoqing WU, *Barrett R. BRYANT, ***Marjan MERNIK

*Department of Computer and Information Sciences, University of Alabama at Birmingham, USA

Email: {wuxi, bryant, mernik}@cis.uab.edu

**Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia

Email: marjan.mernik@uni-mb.si

SUMMARY

Formal methods are often used for programming language description as they can specify syntax and semantics precisely and unambiguously. However, their popularity is offset by the poor reusability and extendibility when applied to non-toy programming language design. One cause of this problem is that classical formal methods lack modularity. Meanwhile there are always needs for informal constructs for semantic analysis, and there is no simple and precise way to specify informal constructs by formal specification, which makes the formal specification too complicated to understand. To address the aforementioned problems with modern software engineering technology, we combine object-oriented Two-Level Grammar with Java to modularize language components and apply design patterns to achieve the modularity and implement the informal constructs in a proper way.

Keywords: design patterns, object-oriented technology, programming language implementation, two-level grammar.

1. INTRODUCTION

The advantages of using formal methods for programming language definition are well known as they can be used to specify syntax and semantics in a precise and unambiguous manner and offer the possibility of automatically constructing compilers or interpreters [1]. However, despite obvious advantages, most widely used formal methods such as attribute grammars, axiomatic semantics, operational semantics and denotational semantics [2] are yet to gain popularity and wide application due to the poor readability, reusability and extendibility [3]. There are several factors for this, and the following two are most critical:

- Traditional formal specification for language implementation lacks modularity [3]. The different phases of interpreter or compiler implementation (e.g. lexical analysis, syntax analysis and semantics analysis) are always tangled together, and the specification for real programming languages is always very large and complex. However, the traditional formal methods lack mechanisms to encapsulate the language components for tight cohesion inside a module and loose coupling between modules.
- Formal specification for language implementation lacks abstraction. The semantics of a programming language are diverse, which hinders specification by pure formal methods. Many mathematics-based formal specifications do not provide a strong library mechanism and I/O capabilities, which make them quite complicated to address *low-level* semantics implementation and hard for user comprehension, therefore the specification is hard to be reused even though they are well

modularized. On the other hand, the general purpose programming languages (GPL) such as Java offer an abundant library of classes and can be directly used with ease.

In order to address these two issues by providing more readability, extendibility and reusability for programming language specifications, we apply *object-oriented* technology and *design patterns* [4] on formal specifications and GPL Java to design a framework for automatic parser generation and semantics implementation.

In this framework, we use *Two-Level Grammar* (TLG) [5] as an object-oriented formal method to properly encapsulate the entwining lexical/syntax rules and abstract semantics of each grammar symbol into a class, and use Java, a GPL, to address the semantics implementation details and obtain the interpreter for the desired language. Therefore, we maximize the automatic code generation capacity of formal specification to precisely specify syntax and semantics, and utilize the massive library of classes in programming languages such as Java to avoid overly complicated use of formal methods. As a result, reuse or extending the language can be easily achieved by rewriting or extending the terminal symbol classes.

The reminder of this paper is structured as follows. Section 2 introduces TLG specification and the concepts of abstract semantics and concrete semantics. Section 3 presents the overview of the whole framework and some of its salient features. Section 4 details the object-oriented design of language implementation in this framework regarding the *interpreter pattern* [4]. Section 5 presents how we use the *chain-of-responsibility pattern* [4] to separate the formal and informal concerns in semantics analysis and section 6 demonstrates in depth how readability, extendibility

and reusability are obtained with our approach. Section 7 describes the related work. We conclude and suggest future research in section 8.

2. BACKGROUND KNOWLEDGE

2.1. Two-Level grammar specification

TLG (also called W-grammar) was originally developed as a specification language for programming language syntax and semantics and was used to completely specify ALGOL 68 [6]. It has been shown that TLG may be used as an object-oriented formal specification language to be translated into existing GPLs [7]. The name “two-level” comes from the fact that TLG contains two context-free grammars corresponding to the set of type domains and the set of function definitions operating on those domains respectively. The syntax of TLG class definitions is:

```
class Identifier-1.
    [extends Identifier-2, ..., Identifier-n].
    {meta rule and hyper rule declarations}
end class.
```

Identifier-1 is declared to be a class which may inherit from classes Identifier-2, ..., Identifier-n.

The type domain declarations (also called meta rules) have the following form:

```
Id1, ..., Id-m :: DataType1; ...; DataType-n.
```

which means that the union of DataType-1, ..., DataType-n forms the type definition of Id1, ..., Id-m.

The function definitions (also known as hyper rules) have the following forms:

```
function-signature:
    function-call-11, ..., function-call-1j;
    ...;
    function-call-n1, ..., function-call-nj.
```

The function body on the right side of ‘:’ specifies the rules of the left hand side function signature. Symbol ‘;’ is used in the right hand side to delimit multiple rules which share the same function signature on the left hand side. For more details on the TLG specification language see [5].

In this framework, we rewrite the TLG keyword **class** as **terminal** and **nonterminal**, and use the following TLG notations for different constructs in the class for each grammar symbol:

- Meta-level keyword **Lexeme** for lexical rules
- Meta-level keyword **Syntax** for syntax rules
- Hyper-level keyword **semantics** for semantics

2.2. Abstract semantics and concrete semantics

One distinguishing feature of our approach is that we separate *abstract semantics* and *concrete semantics* in compiler design, using formal specification and GPL to handle them respectively. In this paper, *abstract semantics* refers to the semantics of a nonterminal that are used to describe the composition of this nonterminal by other grammar symbols. This kind of semantics can be easily specified by formal specification such as TLG. For example, if a *program* is composed by *declarations* and *statements*, then the semantics for *program* can be specified in TLG as:

```
nonterminal Program.
    //Syntax definition
    semantics :
        Declarations,
        Statements.
end nonterminal.
```

which means that the semantics of *Program* is simply composed by the semantics of *Declarations* and *Statements*.

On the other hand, *concrete semantics* refers to those for which the implementation is very low-level or operating system related, such as the calculation of two complex objects (e.g. two matrices) or any I/O operation. Such semantics is difficult to be specified by formal methods and can make specification quite complex and low-level. However, this semantics is easier to be implemented by GPL directly. So our goal is to separate the *abstract semantics* with *concrete semantics* and to have them specified and implemented by TLG and Java, respectively. We will elaborate this in the following sections.

3. OVERVIEW

Figure 1 provides the control flow of programming language implementation in this framework. Tools are shown in ellipses. Shaded boxes contain generated code. Arrows denote control flow. To describe a language, the user specifies the lexical, syntactic rules and abstract semantics for each grammar symbol (terminal or nonterminal symbol) with a single TLG class. The framework takes the TLG class file as input, and extracts lexical rules and syntax rules, which will be compiled by the lexer generator JLex [8] and parser generator CUP [9], respectively, to generate the corresponding lexer and parser in Java, respectively. Meanwhile, Java classes and interfaces for nonterminals are generated and class structures (class names, method signatures, etc.) for terminals are generated into two separated files. Users can later add Java code for concrete semantics analysis

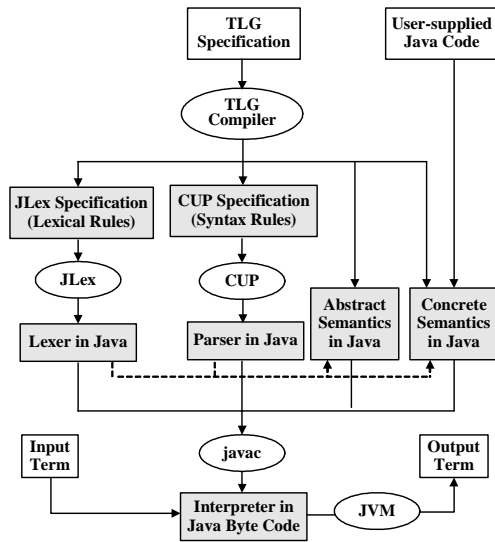


Fig. 1 Language implementation overview.

into the second file, which is the only file users need to manage in the programming language level. Once the lexer, parser and semantics in Java are compiled together (using javac), an interpreter in Java byte code is produced. The relationship among lexer, parser and semantics Java classes is as follows: the parser takes tokens produced by the lexer as input, creates semantic objects of Java classes, and builds Abstract Syntax Tree (AST) by calling the construction methods of these objects.

4. OBJECT-ORIENTED MODULARIZATION

```

program ::= declaration-list statement-list
declaration-list ::= declaration | declaration-list declaration
declaration ::= id "=" integer-list ";"
integer-list ::= integer "," integer-list | integer
statement-list ::= statement-list | statement-list statement ";"
statement ::= assignment-statement | print-statement
assignment-statement ::= id "=" expression
print-statement ::= "print" expression
expression ::= term | binary-expression | unary-expression
binary-expression ::= expression binary-operator expression
unary-expression ::= unary-operator term
term ::= id | integer | parentheses-expression
parentheses-expression ::= "(" expression ")"
binary-operator ::= "+" | "-" | "*" | "/"
unary-operator ::= "+" | "-"

```

Fig. 2 The context-free grammar of Sam.

To design the TLG specification in this framework, we apply the *interpreter pattern* [4], treating each grammar symbol as a class. For illustrative purposes, we will explore how the framework models grammars based on a sample language named Sam, which is a very simple language for specifying computations involving integer arithmetic only. Figure 2 is the context-free grammar of the Sam language. Symbols in bold

stand for terminals, in which quoted strings and characters stand for keywords/meta-symbols/operators, integer and id stand for integer values and identifiers, respectively. The other symbols are nonterminals.

Nonterminal symbol classes: Each nonterminal symbol must have an associated class. For each production rule in the form of $R ::= R_1 R_2 \dots R_n$, we create a class for the left-hand-side (LHS) nonterminal R , and specify the syntax rule using the TLG keyword **Syntax** followed by the right-hand-side (RHS) of the production $R_1 R_2 \dots R_n$. The syntax will not only help direct the grammar specification in CUP but also generate constructor methods of each Java class to store the instance variables of $R_1 R_2 \dots R_n$. For example, the TLG class for nonterminal *Binary_expression*:

```

nonterminal Binary_expression.
Syntax ::= Expression1 Binary_operator
           Expression2.
           //semantics analysis
end nonterminal.

```

will generate the following constructor in the Java class:

```

class Binary_expression{
    Expression expression1;
    Binary_operator binary_operator;
    Expression expression2;

    Binary_expression(
        Expression expression1,
        Binary_operator binary_operator,
        Expression expression2){
        this.expression1=expression1;
        this.binary_operator = binary_operator;
        this.expression2=expression2;
    }
    //semantic analysis
}

```

The semantics of R is represented by the keyword **semantics**, followed by the semantics operations of $R_1 R_2 \dots R_n$, and in the generated Java code, semantics implementation is obtained by applying method *semantics()* iteratively on the instance variables representing $R_1 R_2 \dots R_n$. For example, the semantics for nonterminal *program* in Sam will be composed by the semantics of nonterminal *declaration-list* and *statement-list*. However, the nonterminals can directly delegate the responsibilities of implementing *concrete semantics* to terminals as well, as described in the next section.

Notice that if a nonterminal is the LHS of several different productions, then all the corresponding productions should be *unit productions* [10], i.e.

only one RHS variable in the production (if there exists a non-unit production for this nonterminal, we can easily eliminate it by rewriting the original grammar). We make the LHS variable as a super class (i.e. *interface* in Java), with each RHS variable as its subclass. Since interfaces can't be initialized in Java, all the semantics of this super class will be completely implemented by its subclasses. This technique reduces the number of the generated AST nodes and provides a proper level of abstraction for those LHS nonterminals, as illustrated in Figure 3, where the AST of a print statement "print a" is presented with shaded boxes represent the actual AST nodes.

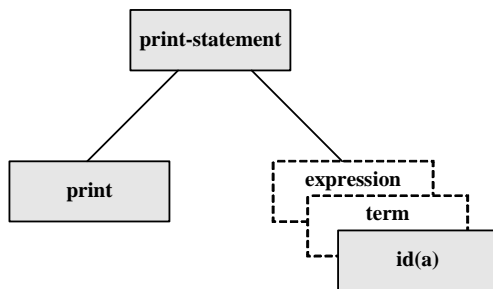


Fig. 3 The AST for a print statement.

Terminal symbol classes: Each terminal symbol in a grammar may have its own class. The lexical rule for each terminal is defined in its own class using keyword **Lexeme** followed by the quoted regular expression of the symbol. However, to avoid creating too many terminal classes, we can only specify the lexeme of those *non-trivial* tokens using a class. Here *non-trivial* tokens refer to the semantics-related tokens such as Identifier, Integer, Operator, etc. *Trivial* tokens are those that have no significant semantics contributions, such as meta-symbols, whose lexeme can be specified in the syntax of its parent class. The semantics interface associated with terminal symbols is also introduced in the terminal class followed by hyper-level keyword **semantics**. A corresponding Java class interface is generated, into which user can add concrete Java code directly. For example, the binary operator "+" in Sam will have the following TLG class:

```

terminal PLUS.
  Lexeme :: "+".
  semantics with Expression1 and Expression2.
end terminal.

```

Notes: The generated Java class of this TLG class will contain an interface of a method for *concrete semantics* implementation in Java, with Expression1 and Expression2 as parameters (their types are both *Expression*).

The use of the interpreter pattern has the following two benefits: firstly, it is easy to change and extend the grammar. As each grammar is composed by a number of terminals and nonterminals, the designer can always modify the grammar by class manipulation or extend the grammar using inheritance. Secondly, implementing the grammar becomes much easier. In our specification, each AST node is represented by a TLG class. The semantics part is easy to write node by node and the generation of the corresponding Java objects can be automated with a parser generator, such as CUP. Besides the above two benefits, we also make some adaptation on the basis of the sample approach introduced in [4]: first, instead of using recursive-descent parsing [10], we reuse the lexer and parser generator components JLex and CUP to generate a bottom-up parser, and then traverse the generated abstract syntax tree to implement semantics. Thus we leverage the LALR (1) parsing power of bottom-up parsing and the natural traversal property of top-down semantic analysis. Secondly, we create classes for nonterminals and terminals in contrast to the approach in [4] of creating classes for productions. Therefore, we can delegate *concrete semantics* of the nonterminals to terminal classes to separate the formal and informal concerns of semantic analysis and we only need to add Java code to terminal classes, which is in a separated file. This actually solves the major drawback for the interpreter pattern, namely that too many classes are to be managed by the user.

5. SEPARATION OF FORMAL AND INFORMAL SEMANTICS

As described before, the Java codes generated from TLG can be used to build the AST by the calls to the constructor methods. This tree is built during parsing and the calls to constructors are embedded as the *action codes* [9] following each production of the CUP file. For instance, the production and action code for *print-statement* in CUP is as below:

```

....
print_statement ::=
  PRINT : PRINT expression : expression
  {
    RESULT = new
      Print_statement (PRINT , expression);
  };
...

```

In some interpreter generation approaches such as SableCC [11] and JJForester [12], once the AST is built, semantic actions will be added to every AST node and the interpreter or compiler is implemented by iterative traversal of this tree. However, this kind of method tangles the abstract semantics and

concrete semantics together and breaks the formal property of the AST. As a result, the syntax grammar is bounded by embedding the semantic actions and hard to be extended or reused.

Another drawback of the traditional method is that the formal specification of those *concrete semantics* is very low-level and hard to read. For example, in a grammar production for doing I/O operations, the specification should be used to implement the input or output with the environment, which is operating system related; in an expression for addition calculation, specification may be used to deal with calculating the sum of two expression values. It is not hard for a formal specification to handle addition of two integers, however, the specification will be quite complicated when facing the addition of two matrices unless some additional functions are pre-defined in the formal specification on demand. This hampers the designer in reusing any implementation components of an existing language as they are bounded by low-level domain-related details. For example, the designer of a matrices calculator can take no benefits from the existing implementation of an integer calculator although they share quite a few syntax productions.

In order to address these problems, we apply the *chain of responsibility* design pattern [4] in the AST to recursively throw the responsibilities of implementing *concrete semantics* from the upper nodes to the lower nodes, until they reach the leaf nodes, i.e., nodes for terminal symbols. The applicability of this delegation method is explored below. Given a program written in a certain language, each *concrete semantics* operation is actually represented and distinguished from each other by at least one terminal symbol. For example,

an I/O operation is indicated by terminal “print” and a requirement of addition is expressed by terminal “+”. Since each semantic action is represented by a terminal such as “print” or “+”, it is applicable for the *concrete semantics* operating on nonterminal nodes to finally find a terminal node to delegate the analysis responsibility. Even if no such terminal node can be found or the path between the nonterminal node and the terminal node is too long, we can introduce a *dummy terminal*, which has no lexeme at all and is only used to delegate the *concrete semantics*. This idea is actually similar to the well know mechanism of inserting markers in the attribute grammar [10].

Figure 4 is the partial UML diagram of the generated Java classes. Since we separated the concrete semantics with abstract semantics, we keep all the middle nodes (nodes for nonterminals) abstract and formal, and leave the concrete and informal semantics implementation to terminal nodes. For example, the TLG classes for nonterminal *print-statement* and terminal “print” in Sam can be as following:

```

nonterminal print_statement.
  Syntax :: PRINT Expression.
  semantics : PRINT with Expression.
end nonterminal.

```

Notes: nonterminal *print-statement* delegates the *concrete semantics* to terminal *PRINT* with *Expression* as the parameter.

```

terminal PRINT.
  Lexeme :: “print”.
  semantics with Expression.
end terminal.

```

Notes: The generated Java class of this TLG class will contain an interface of a method for *concrete semantics* implementation in Java, with *Expression* as the parameter of the method.

Now the user only needs to add concrete semantics into all the generated terminal classes (represented by gray boxes in Figure 4), using the full-featured operation library of Java. Continuing with the above example, the completed Java class for *PRINT* is:

```

class PRINT{
  public void semantics(Expression expression){
    System.out.println(
      ((Integer)expression).intValue());
  }
}

```

If we want to modify the language to make it handle matrix computation instead of integer computation, we need to make some adaptation to the semantics since there are different calculation

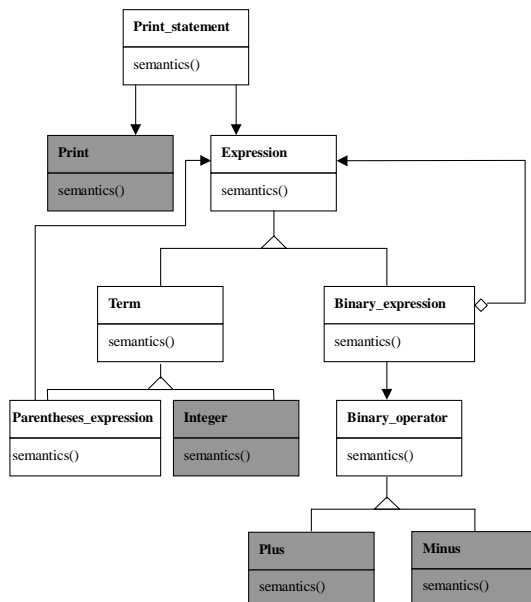


Fig. 4 Partial UML diagram of the generated Java classes.

methods and I/O strategies applied to integers and matrices. In our approach, we only need to rewrite the terminal Java classes to achieve this adaptation. As in Figure 4, we only need to rewrite the Java classes of leaf nodes represented by the gray-boxes, e.g. *Print*, *Integer* and *Plus* with the middle nodes intact. In the case of terminal class *Print*, the new semantics class could be as below:

```
class PRINT{
    public void semantics(Expression expression){
        DecimalFormat fmt =
            new DecimalFormat("0.##");
        System.out.println();
        for(int i=0;i<matrix.getRowNum();i++){
            for(intj=0;j<matrix.getColumnNum();j++){
                System.out.print(fmt.format(
                    matrix.getFloat(i,j).floatValue())+" \t");
            }
            System.out.println();
        }
        System.out.println();
    }
}
```

In our real implementation of this language, we utilized lots of existing Java APIs, such as *ArrayList* to store the matrix value and we used Java applets for polished output of the matrices.

6. SIGNIFICANCE

With the help of *T-Clipse* [13], which is an Integrated Development Environment (IDE) for two-level grammar based on the Eclipse framework [14], we have developed an interpreter for the Sam language. Then we reused the Sam specification to quickly develop a language called BasicM for matrix calculation, as well as reusing the interpreter for Sam to build an interpreter for BasicM. Our experience in developing these two languages show that our approach does improve the readability, extendibility and reusability, as described below:

- **Readability.** The TLG classes embrace a one to one mapping with grammar symbols (except the punctuation such as comma or semicolon).

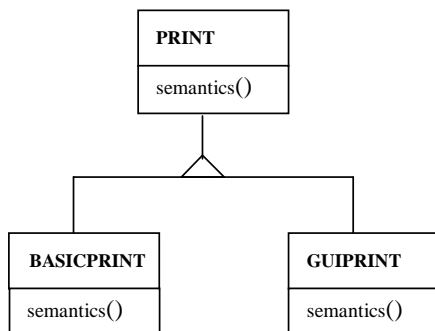


Fig. 5 Extend the output function of Sam.

Each grammar symbol's lexical/syntax rules and semantics are all defined in the same class, which is easy to read. In the TLG level, we only specify the *abstract semantics* in TLG classes, which makes the formal specification concise; in the Java code level, the user only needs to manage the file that contains terminal Java classes. This reflects the *separation of concerns* principle in software engineering.

- **Extendibility.** Adding another output operation in this language (e.g. output to a window instead of the console) can be achieved by make the terminal *PRINT* as a nonterminal, i.e. make it abstract, and let two new terminal classes named *GUIPRINT* and *BASICPRINT* to extend *PRINT* as in Figure 5. Terminal *BASICPRINT* can reuse the semantics component for original terminal *PRINT*. In this manner, to extend the output statements, user only needs to write a semantics class for terminal *GUIPRINT*.
- **Reusability.** Swithing the domain of expressions from integer calculation to matrix calculation can be achieved as below. A new grammar symbol *Matrix* is created, which is composed by some nonterminals and terminals, and replace the *Integer* class in the TLG level to regenerate the lexer, parser and abstract semantics (nonterminal Java classes) for the new language. To maximize the reuse of the concrete semantics components, only the Java classes of new terminals are automatically regenerated, while the Java classes of existing terminals are changed manually, which is the same approach used in JavaCC when regenerating AST node classes[15].

7. RELATED WORK

Many researchers are working on object-oriented modular specifications from which compilers or interpreters can be automatically produced. Java Comiler Compiler (JavaCC) [15] is a Java parser generator written in the Java programming language. JavaCC integrates lexical and grammar specifications into one file to make specification easier to read and maintain. Combined with tree generators such as JJTree [16] or Java Tree Builder [17], it can be used to generate object-oriented interpreter/compiler. JavaCC (together with the tree generator) use the *Visitor pattern* [4] for tree traversal. However, JavaCC cannot handle left recursive grammars since it only generates *recursive-descent* parsers, which are less expressive than LALR(1) parsers. Another drawback of JavaCC is that the *Visitor pattern* is only applicable when the grammar is rarely changed because changing the grammar requires redefining the interface to all

visitors, which is potentially costly [4]. This provides bad reusability for the specifications.

The ASF+SDF Meta-Environment [18] is an environment for the development of language definitions and tools. It combines the syntax definition formalism SDF with the term rewriting language ASF. SDF is supported with Generalized LR (GLR) parsing technology. ASF is a rather pure executable specification language that allows rewrite rules to be written in concrete syntax. However, though ASF is good for the prototyping of language processing systems, it lacks some features to build mature implementations. For instance, ASF does not come with a strong library mechanism, I/O capabilities, or support for generic term traversal [12]. As a major step to alleviate these drawbacks, JJForester [12] was implemented, which combined SDF with the general purpose programming language Java. However, again, it has the same drawback as JavaCC as it uses the *Visitor pattern* for tree traversal.

The LISA system [19] is a tool for automatic language implementation in Java. LISA uses well-known formal methods, such as regular expressions and BNF for lexical and syntax analysis, and use attribute grammar to define semantics. LISA provides reusability and extendibility by integrating the key concepts of object-oriented programming, i.e. templates, multiple inheritance, and object-oriented implementation of semantic domains [3].

Our major distinction with all the above research is as follows:

- As we use TLG to encapsulate the lexical, syntactic rules and semantics of each grammar symbol in a single class based on an object-oriented manner, we provide good modularization for grammar components making them easily extendible and reusable.
- We successfully separate the formal concerns and informal concerns in language implementation, and combine the feature of automatic code generation from formal specification with the massive library of classes in Java, to precisely specify syntax and semantics and minimize complexity in the use of formal methods.
- We separate the parsing from semantics analysis, realizing bottom-up parsing and top-down semantics analysis. The LALR(1) [10] parsing power and the natural property of *recursive descent* semantics analysis are combined together.

An additional benefit in our approach which is not discussed in this paper is the TLG specification's strong computation power compared to other formal methods [20], e.g. TLG can specify the semantics of a loop statement in programming languages while attribute grammar cannot.

8. CONCLUSION & FUTURE WORK

In this paper, with an aim to provide good modularization and abstraction for formal specification in programming language description, Two-Level Grammar is introduced as an object-oriented formal specification language for modeling language components and constructs. Some software design patterns are also applied to help with the organization of the TLG classes and separate the informal concerns from formal concerns in language implementation. Therefore, we provide good modularity, readability, reusability and extendibility for TLG specification while leveraging mature programming language technology such as Java, thereby achieving our research objectives. Therefore, our approach offers a means to take advantage of the synergy between formal methods and general programming languages. The benefits of using our approach have been demonstrated by a sample language.

Besides the *interpreter pattern* and *chain-of-responsibility pattern* we described in the paper, there are other possible patterns that could be applied in this framework. For example, the generated AST is actually an instance of the *Composite pattern* [4], with the terminal classes as *leaf*, and the nonterminal classes as *composite*. Another pattern we are interested to use in the future is *Mediator pattern* [4]. Once the grammar becomes large, it is quite common that *non-local dependency* [21] will appear, which means that the semantics of one AST node is dependent on another node which is contained in another sub-tree, such as the name analysis problem where properties of an identifier use site depends on properties of an identifier declaration site. Attribute grammar uses a propagating method to deliver related attributes through the path of linked nodes. This is obviously inefficient and Hedin has listed four drawbacks of this kind of approach in [22]. Our current practice is to forward the reference of one object to the others by the common ancestor of two node objects, which is similar to Hedin's reference attribute grammar. However, our strategy is still not as efficient as desired and complicates the formal specification somewhat. We found that the *Mediator pattern* is well suited to solve this problem as its applicability is to the situation when a set of objects have to communicate in complex ways and the interdependencies are unstructured and difficult to understand. So, for AST nodes that need to communicate to other ones far away, we could create a *mediator* for them to communicate with. The major challenge is it is hard to design an algorithm for dynamically creating mediators for objects, since the AST is only built dynamically during parsing. We are still working on this.

REFERENCES

- [1] F. G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice Hall, 1981.
- [2] K. Slonneger, B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.
- [3] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. A Reusable Object-Oriented Approach to Formal Specifications of Programming Languages. *L'Objet* Vol. 4, No. 3, pp. 273-306, 1998.
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [5] B. R. Bryant, B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proceedings of 35th Hawaii Intl Conf. System Sciences*, 2002. http://www.hicss.hawaii.edu/HICSS_35/HICSS_papers/PDFdocuments/STDSL01.pdf
- [6] A. van Wijngaarden. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, Vol. 5, pp. 1-236, 1974.
- [7] B. R. Bryant, A. Pan. Formal Specification of Software Systems Using Two-Level Grammar. *Proc. COMPSAC '91, 15th. Intl. Computer Software and Applications Conf.*, pp.155-160, 1991.
- [8] *JLex: Java Lexical Analyzer Generator*. <http://www.cs.princeton.edu/~appel/modern/java/JLex/>
- [9] *CUP: Parser Generator for Java*. <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
- [10] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.
- [11] E. Gagnon. *SableCC, An Object-Oriented Compiler Framework*. Master's thesis, McGill University, Montreal, Quebec, March 1998.
- [12] T. Kuipers, J. Visser. Object-Oriented Tree Traversal with JForester. *Electronic Notes in Theoretical Computer Science*, Vol. 44, 2001.
- [13] B.-S. Lee, X. Wu, F. Cao, S.-H. Liu, W. Zhao, C. Yang, B. R. Bryant, and J. G. Gray. T-Clipse: An Integrated Development Environment for Two-Level Grammar. *Proceedings of OOPSLA03 Workshop on Eclipse Technology eXchange*, 2003.
- [14] Object Technology International, Inc. *Eclipse Platform Technical Overview*, February 2003
- [15] *JavaCC: Java Compiler Compiler*, Sun Microsystems, Inc. <https://javacc.dev.java.net/>
- [16] *Introduction to JJTree*. <http://www.jpaine.org/jjtree.html>
- [17] *JTB: Java Tree Builder* <http://www.cs.purdue.edu/jtb/releasenotes.html>
- [18] M. G. J. van den Brand, J. Heering, P. Klint and P.A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4): 334-368, 2002.
- [19] M. Mernik, V. Žumer., M. Lenič, E. Avdičaušević. Implementation of Multiple Attribute Grammar Inheritance In The Tool LISA. *ACM SIGPLAN Not.*, Vol. 34, No. 6, June 1999, pp. 68-75.
- [20] M. Sintzoff. Existence of van Wijngaarden's Syntax for Every Recursively Enumerable Set, *Ann. Soc. Sci. Bruxelles*, Vol. 2, pp. 115-118, 1967.
- [21] J. T. Boyland. Analyzing Direct Non-Local Dependencies In Attribute Grammars. *Proc. CC '98, International Conference on Compiler Construction*, Springer-Verlag Lecture Notes in Computer Science, Vol. 1383, pp. 31-49, 1998.
- [22] G.. Hedin, Reference Attributed Grammars, in D. Parigot and M. Mernik, eds., *Second Workshop on Attribute Grammars and their Applications*, WAGA'99, Amsterdam, The Netherlands, (1999), 153-172. INRIA Rocquencourt.

BIOGRAPHY

Xiaoqing Wu is a Ph. D. student in the Computer and Information Sciences Department at the University of Alabama at Birmingham. His research is focusing on compiler design, programming languages, formal specification and software engineering.

Barrett R. Bryant is a Professor and the Associate Chair in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). He joined UAB after completing his Ph. D. in computer science at Northwestern University in 1983. He has held various visiting positions at universities and research laboratories since joining UAB. Barrett's primary research focus is in theory and implementation of programming languages, especially formal specification languages, and object-oriented and component-based software technology.

Marjan Mernik received his M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently an associate professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He was a visiting professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham in 2004. His research interests include principles, paradigms, design and implementation of programming languages, compilers, formal methods for programming language description, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.