T-Clipse: an Integrated Development Environment for Two-Level Grammar

Beum-Seuk Lee, Xiaoqing Wu, Fei Cao, Shih-hsi Liu, Wei Zhao, Chunmin Yang,
Barrett R. Bryant, Jeffrey G. Gray

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
{leebs, wuxi, caof, liush, zhaow, yangc, bryant, gray}@cis.uab.edu

Abstract

T-Clipse is an Integrated Development Environment (IDE) using Eclipse developed for Two-Level Grammar (TLG), a high-level formal specification language. In our research, given a programming language, TLG is used to specify the syntax and semantics of the language to generate an interpreter for that language. This IDE provides a user-friendly environment for editing and navigation of TLG source codes, as well as parsing and error tracing at the source code level and interpreter generation for programming languages. To build this IDE, we leveraged the extension power of plug-ins in Eclipse as well as a TLG parser pre-built from an existing lexical analyzer generator and parser generator.

1   Introduction to TLG

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 [7]. It has been shown that TLG could be used as an executable specification language [3], not only for programming language design and implementation but also for more general software systems, such as natural language interfaces and database and knowledge base systems [2]. Recently, it has been shown that TLG may be used as an object-oriented requirements specification language and serve as the basis for conversion from requirements expressed in natural language into a formal specification [5]. 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.

The syntax of TLG class definitions is:

```
class Identifier-1
  [extends Identifier-2, …, Identifier-n].
    {instance variable and method
      declarations}
end class [Identifier-1].
```

`Identifier-1` is declared to be a class which inherits from classes `Identifier-2, …, Identifier-n`. Note that the `extends` clause is optional so a class need not inherit from any other class. The type domain declarations (also called meta rules) have the following form:

```
Identifier-1, …, Identifier-m ::
  DataType1; …; DataType-n.
```

which means that the union of `DataType-1, …, DataType-n` forms the type definition of `Identifier-1, …, Identifier-m`.

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

```
function-signature.

function-signature:
  function-call-1, …, function-call-n.

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. There may be no rules at all, as in the first case. ';' 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 [1].

Current practice for processing TLG has remained in a console based environment. Editing TLG will leverage a generic editor such as notepad, and then a parsing command is issued from the command line. This unfriendly environment makes it error-prone at coding time and time-

consuming at debugging timing. As a result, this greatly impacts the popularity of TLG in spite of its computational power and flexibility in its syntax as a formal specification language. In order to take full advantage of TLG as aforementioned, we have developed tool support for TLG, *T-Clipse*, which is an IDE for TLG developed for Eclipse. This development environment has integrated all the common functionalities pertaining to use of TLG as a formal specification language.

2 Project Approach

2.1 TLG Specification for Interpreter Generation

Given a programming language (let's call it Foo), the syntax and semantics of Foo is specified in TLG. This TLG specification of Foo is parsed by TLG parser to construct a symbol table. Given the symbol table, the Foo interpreter is generated according to the specification in TLG and an application code in Foo is used as an input to the generated interpreter. Figure 1 illustrates this process.
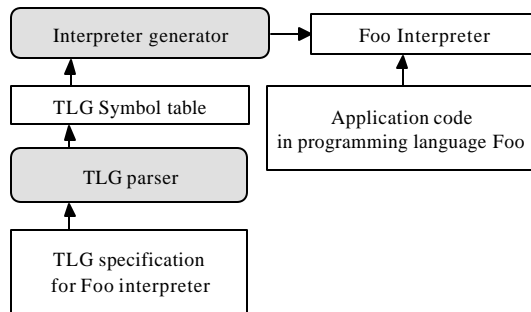

Figure 1: Interpreter Generation

To carry out this process in a user-friendly manner, Eclipse [1] is chosen to build a TLG IDE. Eclipse is a Java IDE offering a platform for building and integrating application development tools delivered via plug-ins.

2.2 T-Clipse Environment

T-Clipse is a natural outcome of the marriage between Eclipse and TLG. By using the Eclipse PDE (Plug-in Development Environment), we can extend the existing plug-ins instead of building a TLG IDE from scratch. Figure 2 gives an overview of the modules of T-Clipse. The directional arrows denote the processing order and the dependencies among the modules.
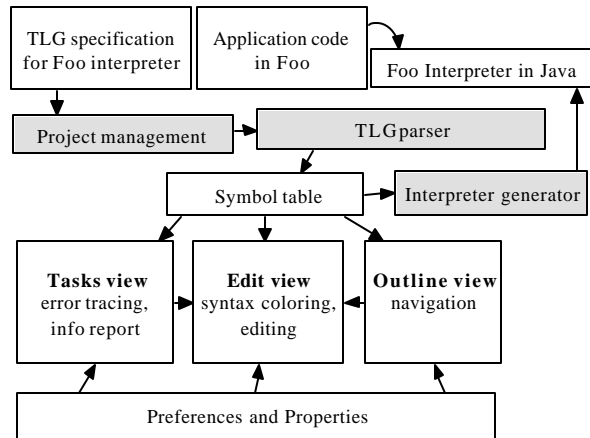

Figure 2: Modules of T-Clipse

The TLG project management module handles the loading of TLG source files from the file system as well as editing and saving of files in the editor view. The TLG plug-in will parse and type-check the TLG file whenever the file is loaded, changed, or saved in the editing environment. The symbol table is generated after the parsing phase, which contains all the parsed information of TLG source code such as class names, their parents classes, meta rules, hyper rules, syntax errors, the locations of lexemes and appropriate syntax tree. Other modules such as the tasks view and the outline view, as well as the interpreter generator, refer to this symbol table to retrieve the related information, which is further detailed in the following.

We used JLex, a lexical analyzer generator, and CUP[2], a parser generator, to build a TLG parser. There are two key features for the parser that are noteworthy. First, this parser supports an error recovery mechanism. Once a syntax error is identified for a class, the parser will try to recover it and continue to parse the remaining classes. Second, this parsing is applied dynamically. In other words, when changes are made in the editor area, the parser will parse it again immediately, and the code coloring section and outline view will be updated according to the changes.

---

[1] http://www.eclipse.org

[2] http://www.cs.princeton.edu/~appel/modern/java

The output of parsing, which is stored in a symbol table, constitutes the kernel information, which is vital for other modules to work with:

- **Outline View**: Using this information, the structure of the source code is captured and appropriately displayed in the outline view. Also, the editor subscribes as a listener to the outline view for the *selection change* event. Once notified with this event, the editor will highlight the corresponding part of the code to the outline view selection using the character position information derived from the *selection change* event.
- **Tasks View**: The error information collected from parsing the TLG code is reported in the tasks view. For each parsing error, a marker will be created using such parsing information as error location, and these markers will be used to pinpoint the errors in the TLG source code by clicking the corresponding tasks view.
- **Syntax Coloring**: TLG, unlike other high level programming languages, has very natural-language like syntax for its function signatures. This flexible syntax, giving a good readability to humans, also calls for highlighting of class names and variables to distinguish these from other semantically insignificant lexemes which are used only for readability. In addition, according to the TLG convention, a type name followed by the word "List" or by a natural number is considered, and thus highlighted, as a variable. Given the class and variable information from the syntax parsing, the class names and variables are highlighted with the specified color in the preferences. The syntax coloring for the source code in the editor is dynamically rendered. When the content of the current active view has changed (by user keyboard input or mouse input), the parser for TLG is called and a check is done to see if there have been any changes in the rule definition. If there are some changes in the rule definitions, the outline view is updated accordingly and the keyword stack is updated. Otherwise, only the coloring of the current view is updated accordingly.

A TLG perspective has also been created, along with a TLG project wizard and preferences for TLG coloring. When the TLG color preferences have been changed, the color rules defined for each type of token (variables, keywords, constants) are updated accordingly and the color changes are also reflected in all the open views. Our approach in building this TLG IDE, ensuring user-friendliness while preserving the conventional Eclipse interface, is strictly enforced.

A partial TLG specification is shown below, which defines the syntax and semantics of PAM (Pluggable Authentication Module) language, based upon Pam [6].

```
class Series.

  Syntax ::
    Statement; Series ';' Statement.

  semantics with Configuration :
    isSyntax Statement,
    Statement semantics with
      Configuration;
    isSyntax Series ';' Statement,
    Configuration1 :=
      Series semantics with Configuration,
    Statement semantics with
      Configuration1.

end class Series.


class Statement.

  InputFile, OutputFile :: File.

  Syntax :: IOStatement; AssignStatement;
    IfThenStatement; ToStatement;
    WhileStatement; IfThenElseStatement.

  extractConfigurationComponents :
    Store := Configuration get Store,
    InputFile :=
      Configuration get InputFile,
    OutputFile :=
      Configuration get OutputFile.

  semantics with Configuration :
    defined in subclass.

end class Statement.
```

Because of the space limitation, the generated JLex and CUP files are omitted in this paper. The semantics implementation file for the interpreter, which contains the corresponding semantics for this TLG is as follows.

```
class Series {

  Statement statement;
  Series series;


  boolean isSyntaxStatement = false;
  boolean isSyntaxSeriesStatement = false;
```

```
Series (Statement statement) {
  isSyntaxStatement = true;
  this . statement = statement;
  series = null;
}

Series (Series series,
    Statement statement) {
  isSyntaxSeriesStatement = true;
  this . series = series;
  this . statement = statement;
}

void semanticsWithConfiguration
    (Configuration configuration) {
  if (isSyntaxStatement)
    statement .
      semanticsWithConfiguration
        (configuration);
  else if (isSyntaxSeriesStatement) {
    series . semanticsWithConfiguration
      (configuration);
    statement .
      semanticsWithConfiguration
        (configuration);
  }
}
}

abstract class Statement {

  Store store = new Store();
  TLGList inputFile = new TLGList();
  TLGList outputFile = new TLGList();
```

```
void extractConfigurationComponents
    (Configuration configuration) {
  store = configuration . getStore();
  inputFile =
    configuration . getInputFile();
  outputFile =
    configuration . getOutputFile();
}

abstract void semanticsWithConfiguration
  (Configuration configuration);
}
```

3 Implementation Status and Future Work

Most of the major features as presented above have been implemented. This includes the TLG project, TLG prospective, TLG preferences, Outline View, Tasks View, and Syntax Coloring.

Figure 3 is a screen shot of the T-Clipse environment and the TLG file being edited is for the PAM language.

The main challenge during the development of T-Clipse comes from the fact that Eclipse plug-in development was new to us at the very beginning and there was a learning curve involved.

Also Eclipse itself has some inconveniences which are unexpected, e.g., we had to rewrite some of the JFace source code to implement the dy-
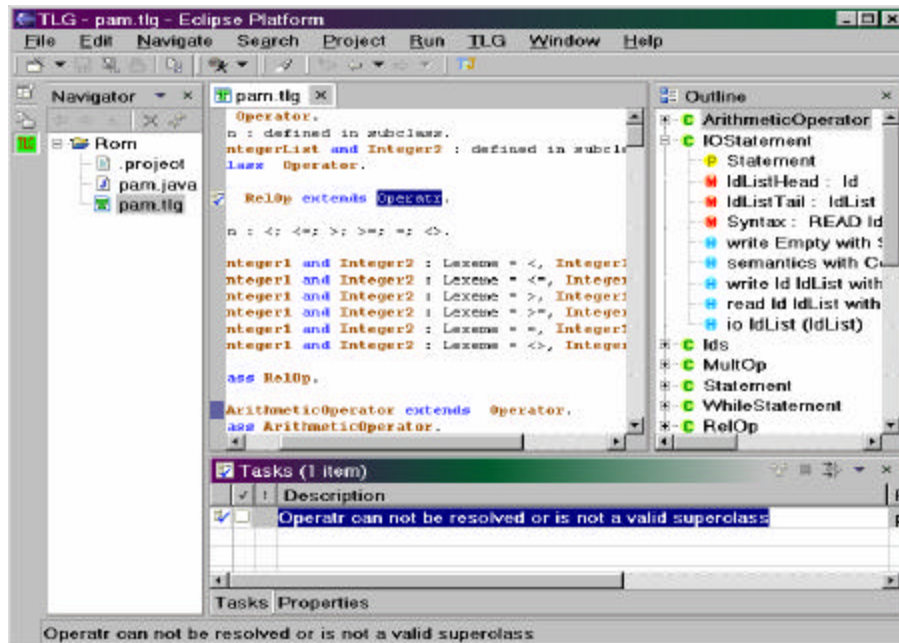

Figure 3: T-Clipse Environment

namic syntax coloring for TLG data types. Additionally, the documentation of Eclipse is not adequate: we had to use the contextual Java IDE help to traverse the API and determine the right function calls.

Currently we are leveraging a TLG parser pre-generated outside the T-Clipse environment. In order to provide the flexibility for processing TLG source code, we plan to integrate the editor and compiler for JLex and CUP into the current T-Clipse environment so that the T-Clipse user can not only edit the TLG code, but also get control on how TLG code itself can be interpreted. The interpreter generation in the current version of T-Clipse is immature and requires further improvement.

Because the TLG is a language-independent specification language, we plan to specify interpreters for various languages, which aligns with the vision of generative programming [4]. By using TLG as an intermediate formal specification language to specify an interpreter for a given programming language, we can leverage the full power of TLG. Along this line, we can create an extension point to our Eclipse project for different programming languages.

4   Conclusion

Eclipse is a nice environment to build an IDE for any type of programming language. T-Clipse takes full advantage of the facilities that Eclipse has offered, such as Perspective, View (including task view, outline view), Editor, Dialog (including preferences), Resource, and Marker.

The contribution of this project is that we have provided a promising language development tool for the Eclipse plug-in family. Not only T-Clipse itself, but also the language implementations it generates may take benefit from Eclipse or make themselves as plug-ins to be extended. This widens the application field and provides more flexibility for Eclipse development.

Our experience with T-Clipse is adequate enough to make us well-positioned to move onto the next level of T-Clipse development or other language IDE development. For more updated information, please visit the website of our Software Composition and Modeling Laboratory at: http://www.cis.uab.edu/info/Eclipse/.

About the Authors

Beum-Seuk Lee received his Ph. D. degree in computer science in August, 2003, at UAB and is
now with the University of Bristol, UK, as an artificial intelligence researcher. Xiaoqing Wu, Fei Cao, Shih-hsi Liu, Wei Zhao, and Chunmin Yang are Ph. D. students and Dr. Barrett Bryant and Dr. Jeff Gray are faculty members in the Computer and Information Sciences Department at UAB.

References

[1] B. R. Bryant, B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Intl Conf. System Sciences,* 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.

[2] B. R. Bryant, A. Pan. Two-Level Grammar: A Functional/Logic Query Language for Database and Knowledge-Base Systems. *Proc LPAR '92, 1992 Intl Conf. Logic Programming and Automated Reasoning,* pp. 78-83, 1992.

[3] 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.

[4] K. Czarnecki, U.W. Eisenecker: *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000.

[5] B.-S. Lee, B. R. Bryant. Contextual Natural Language Processing and DAML for Understanding Software Requirements Specifications. *Proc. COLING 2002, 19th Intl Conf. on Computational Linguistics,* pp. 516-522, 2002.

[6] F. G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice Hall, 1981.

[7] A. van Wijngaarden. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, vol. 5, pp. 1-236, 1974.