Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2005-19

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Development of a Semantics-aware Editor for TTCN-3 as an Eclipse Plug-in

Jochen Kemnade

am Institut für

Informatik

Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

21. September 2005

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 21. September 2005

Bachelor's Thesis

Development of a Semantics-aware Editor for TTCN-3 as an Eclipse Plug-in

Jochen Kemnade

September 21, 2005

supervised by Prof. Dr. Grabowski
Software Engineering for Distributed Systems Group
Institute for Informatics
Georg-August-University Göttingen

Abstract

This thesis describes the process of extending a given parser for the TTCN-3 language by the ability to perform a basic semantic analysis on a given file and integrating it into a plug-in for the eclipse platform. It deals with the principles of parsing and semantic checking as well as showing the basic concept of developing a dedicated editor for a formal language. The integration of the parser into the eclipse platform and strategy of verifying variable assignments are presented in detail. Among others, the plug-in features syntactic highlighting, the annotation of errors in TTCN-3 and the validation of variable assignments including nested expressions.

Zusammenfassung

Diese Arbeit beschreibt die Erweiterung eines bestehenden Parsers für TTCN-3 um grundlegende Funktionalität im Bereich der semantischen Analyse und seine Integration in eine Programmerweiterung für die Eclipse Plattform. Neben den Grundlagen des Parsens und der semantischen Überprüfung wird das Vorgehen beim Entwickeln eines dedizierten Editors für eine formale Sprache dargestellt. Die Einbindung des Parsers in Eclipse und die Vorgehensweise beim Validieren von Variablenzuweisungen werden ausführlich beschrieben. Die Erweiterung unterstützt unter Anderem das Hervorheben von Schlüsselwörtern, das Anzeigen von Fehlern in TTCN-3-Dateien und die Validierung von Variablenzuweisungen inklusive geschachtelter Ausdrücke.

Contents

1 Introduction

Testing is an important part in the process of developing software. The importance of having one's applications tested grows with the failsafe performance one wants the application to assure. Especially, that applies to widespread systems like protocols and common software. The Test and Test Control Notation's third edition (TTCN-3) is an intuitive and flexible language by means of which tests can be specified.
Furthermore, the eclipse platform is known to be a powerful environment for software development and editing source code. The platform is very extensible, hence its functionality can be upgraded, for instance to have another language supported, by developing a plug-in.
This thesis describes the process of combining both of those tools in a dedicated editor for TTCN-3 as a plug-in to the eclipse platform. This extension should base on a given parser [17], that was already capable of syntactic checking and which was to be extended by the capability of basic semantic analysis.
The development's objective was to provide an editor for TTCN-3 offering syntax highlighting of keywords, comments and strings and the annotation of errors. The latter relates to syntax on the one hand and semantics on the other. As for the semantic analysis, the parser should check the validity of variable assignments regarding type compatibility and visibility.
Chapter 2 of this thesis provides a brief overview of the TTCN-3 language and the concepts of parsers and describes the procedure of integrating additional functionality into the eclipse platform. The 3$^{\text{rd}}$ chapter deals with the demands on the plug-in's functionality and explains the basic approach to semantic checking. The actual implementation is exposed in chapter 4, which contains explanations about the editor's features and elaborately demonstrates how the semantic checking was realized. Chapter 5 gives some remarks on the installation of the plug-in and an outlook on which features are not yet implemented. Finally, the thesis is concluded by the 6$^{\text{th}}$ chapter, which summarizes its results.

2 Foundations

2.1 TTCN-3

TTCN-3 [12] is the Test and Test Control Notation in its third edition, which was published by the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union's Telecommunication Standardization Bureau (ITU-T) in 2001. TTCN is a text-based language for specifying tests for a wide range of applications compromising protocols, software modules and APIs. Due to its versatility, the Test and Test Control Notation has been used to write tests for many widespread applications like GSM, DECT, ATM or IPv6.
By its third version, TTCN has become more and more similar to modern programming languages, hence it uses common concepts like global and local variables, assignments, loops and methods, and is, therefore, easily both learnable and usable.
In addition to the textual form, TTCN-3 also offers a tabular and a graphical presentation format, both of which are used to represent tests in a more intuitive and easily readable form, but as the scope of this thesis is semantic analysis of TTCN-3 code, which has to be performed on the textual representation, it only deals with the core language.
The communication between the test system and the components, that are to be tested, is abstracted using ports, which can then be connected and accessed inside a test.
A TTCN-3 module consists of a definitions part and an optional control part. One or more modules result in a test.
In the module definition part, one may declare one's module's parameters, which are similar to global variables in other programming languages, define one's own data types and functions and import definitions from other modules. Also the concept of functions is common to other languages. They may have parameters, a return type and local variables. Besides the general functions, that are

mainly used to divide one's tests' code into units, there are two special types of functions in TTCN-3, namely altsteps and testcases. Altsteps are used to structure alternative behavior, whereas testcases are the portions that control the test sequence. Functions and Altsteps *may* and testcases *must* specify the type they use by a `runs on <typename>` clause.
The module control part, which may also define its own local variables, executes the testcases specified in the definitions part.

2.2 Analyzing source code

When one writes a text, one uses a spell-checker to make sure that it does not contain any typing errors. Afterwards, one could also run a grammar checker to check for missing commas or wrongly used tenses. One wants everything to be written correctly. The same applies if one has written the text in a programming language, it is called source code, the spell-checker is termed lexer and the grammar checker is a parser.

2.2.1 Parsers

Simply spoken, a parser is a program, that tries to transform a text input into a tree structure, which can, in the case of a formal language, later on be used for the semantic checking. A typical parser comprises two separate parts – the lexer and the actual parser.
The lexer converts a given character stream (a text) into a stream of tokens, which represent the basic units of an expression, like "subject" or "predicate" in the case of a natural language. The mathematical expression $2 + 3 \cdot 5$ would for example be transformed into a sequence of tokens like `Number AddOp Number MultOp Number`. Among others, the lexer's task is to verify a given input's validity with respect to a specific language. For example, a lexer for binary strings would report a lexical error processing `01 121 1011`, because the symbol `2` does not belong to the set of permitted characters in a binary expression.

The generated token stream is now passed to the parser, which checks its syntactic correctness verifying if the tokens form a valid expression. Reconsidering mathematical expressions, the term $2++$ would be transformed into `Number AddOp AddOp` by the lexer. This input would make the parser report an error message like "Unexpected token. Expecting number, found operator", because it expects, an addition operator to be surrounded by a number on each side.
If the given token stream is deemed to be valid, the parser transforms it into a so called parse tree. The parse tree for $2 + 3 \cdot 5$ for instance would look similar to the one shown in figure 2.1. This tree can then be evaluated and transformed

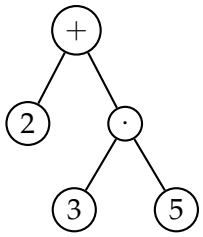

Figure 2.1: The parse tree for a simple mathematical expression

by a tree parser or tree walker. Maintaining the example of mathematical expressions, the tree walker's task could be the computation of the expression's result. Figure 2.2 shows how the lexer, the parser and the tree walker interact.

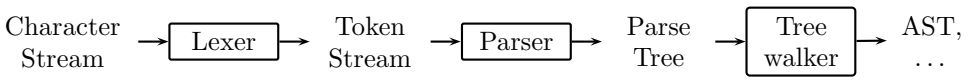

Figure 2.2: Relationship of lexer, parser and tree walker

More information on parsing can be obtained from [10, 15].

2.2.2 Parser generators and ANTLR

When it comes to analyzing high-level programming languages, the parsers' classes gain in complexity and size and writing those classes by hand becomes less and less feasible. Also, as the process of lexing and parsing is very much the same for different programming languages, the resulting lexer and parser classes have a lot of commonalities. These are even increased by the fact that many programming languages are comparable in their structures and have a similar syntax.
Noticing those facts, programmers tried to automate the process of lexer and parser building and they began to develop generator tools like Lex & Yacc [8] and JavaCC [6] just to mention a few. Those tools mostly take some kind of Backus-Naur Form (BNF) as their input and generate source code in common programming languages like C, C++, Java or Python. This procedure saves a lot of time, as there usually exists a BNF for every language, one could want to develop a parser for. The parser generator that was used to generate the lexer, parser and tree walker for the TTCN-3 plug-in is called ANTLR [1] and is being developed by Terrence Parr since 1989.

2.2.3 Semantic Analysis

If the lexer and the parser process a given input file without any errors, the resulting parse tree represents a *syntactically* correct file. But high level languages, programming languages in particular, also have to satisfy *semantic* constraints. For instance the TTCN-3 expression `var integer i := "foo";` is a variable declaration, which is syntactically correct as it consists of the `var` keyword, a type, a variable identifier and an (optional) assignment of a value, but obviously the assignment it is not valid.
As formal languages like TTCN-3 are deterministic and well-defined, they are comparatively easy to check for semantic correctness, which does not apply to natural languages, which tend to contain inconsistencies and ambiguous constructs.

2.3 Eclipse

Eclipse [4] has become popular as an Integrated Development Environment (IDE) for Java development, although the Java editing extension is just one of many plug-ins which are available for eclipse. Figure 2.3 shows the broad

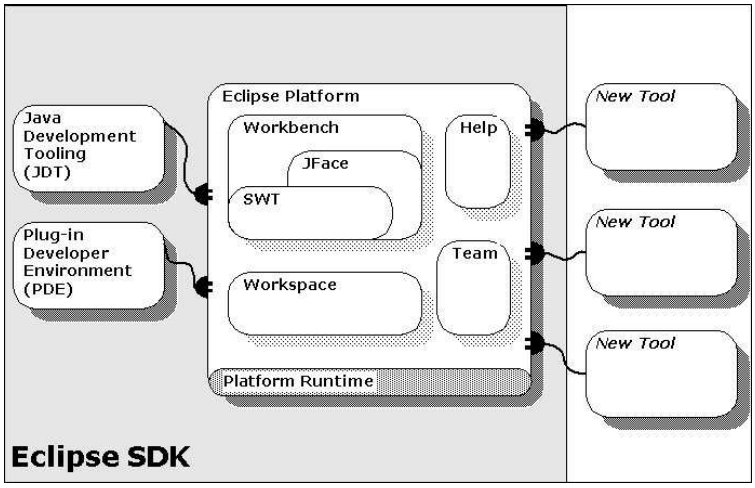

Figure 2.3: Eclipse's architecture
source: `http://help.eclipse.org/help31/topic/org.eclipse.platform.doc.isv/guide/arch.htm`

outline of its architecture. The core platform is extensible by a growing variety of plug-ins providing the ability to make eclipse a powerful tool for conveniently editing sources in a wide range of programming or markup languages, accessing version control repositories or developing extensive applications. There are also plug-ins that provide completely different functionality, which is not at all related to programming, like creating charts or playing music.

2.3.1 Eclipse Plug-ins

Virtually every part of eclipse that the user interacts with, such as editors for different programming languages, can be considered a plug-in. The platform only offers basic functionality such as opening and saving files or the marginal resource perspective.
As eclipse is a framework, it offers an extensive collection of predefined parts, hence developing plug-ins for the platform primarily means extending a set of those classes with one's own code. The TTCN-3 plug-in's main editor is derived from the `org.eclipse.ui.texteditor.AbstractDecoratedTextEditor` template class for instance.
The core platform provides several extension points to which one can attach one's plug-in's classes.
The `plugin.xml` file contains all the information about the plug-in, such as the name, the version number and instructions on which spot the plug-in is extending the eclipse platform. Figure 2.4 shows an excerpt from that file,

```
1  <extension point="org.eclipse.ui.editors">
2    <editor
3      class="ttcn3.editors.TTCN3MultiPageEditor"
4      contributorClass="ttcn3.editors.TTCN3ActionContributor"
5      default="true"
6      extensions="ttcn3"
7      icon="icons/ttcn3.gif"
8      id="ttcn3.editors.TTCN3Editor"
9      name="TTCN-3 Editor">
10   </editor>
11 </extension>
```

Figure 2.4: Defining an extension

which states, that the plug-in's main `TTCN3MultiPageEditor` class is integrated into the `org.eclipse.ui.editors` extension point, stating that this should be the default editor for files with a `.ttcn3` extension and specifying the location of an icon to indicate those files. The `id` can be used to reference the `TTCN3MultiPageEditor` from other parts of the `plugin.xml` file.

2.3.2 Eclipse's plug-in development environment

The most comfortable way of developing an eclipse plug-in is to utilize the Plug-in Development Environment (PDE), which is incidentally also a plug-in itself. The PDE incorporates several wizards for setting up the plug-in's basic structure letting one choose between some predefined templates for editors, toolbars or dialogs.
Another important feature, that is offered by the PDE, is the possibility to test and debug one's own plug-ins by running and using them within a second instance of eclipse that runs inside the first one. As eclipse has the capability of *hot swapping*, changes to the plug-in's code are directly affecting the running platform.
Furthermore, the plug-in's settings can be controlled using special dialogs instead of having to write the `plugin.xml` file by hand.

3 Analysis and Design

3.1 Eclipse plug-in

As the editor was developed as an eclipse plug-in, it was desirable to take advantage of some of the capabilities that make eclipse so comfortable and powerful.
One of the most important things an editor for formal languages should provide is syntactic highlighting, because that makes it easier to read the code and therefore makes it easier to edit it. So the editor should highlight the language's keywords, comments and strings in different adjustable colors. Another helpful feature is the annotations, that eclipse shows to mark the line, where an error has been found. By creating those conspicuous markers, it becomes almost impossible to overlook an error in one's code.
Finally, the outline view, which always shows a summary of the editor's content, is very useful for staying aware of what the present file contains. For a TTCN-3 file, the outline should display a tree structure, showing the file's modules as root elements, each of which should accordingly embed its definition and control parts and the functions, testcases and altsteps in its underlying nodes.

3.2 Semantic Analysis

The semantic analysis is to be performed on the parse tree, so it has do be done by the tree walker, which, therefore, has to be able to validate the source code by examining the parse tree and checking things like variable assignments.

3.2.1 Variable declaration and assignments

A simple variable assignment already needs a lot of checks, for instance the variable has to be declared, must be of the appropriate type, and needs to be accessible at the location, where the assignment is made.
In figure 3.1 the example variable declaration statement from page 10 is revived, however this time it is split into both a declaration and an assignment statement inside a function of a sample TTCN-3 module.

```
1 module assignmentTest
2 {
3   function bar(){
4     var integer i;
5     i:="foo";
6   }
7 }
```

Figure 3.1: A simple TTCN-3 module

After the parser has built the parse tree, the tree walker examines this tree one node at a time, so when it reaches the assignment expression, it has already visited the declaration. To recognize that the *charstring* `"foo"` cannot be assigned to the *integer* variable `i`, the tree walker has to store the information, that `i` is declared as an integer. Respectively this applies to every variable and every global parameter used throughout a TTCN-3 module. This information is kept in a table, which is being attached to the root node of the scope that contains the declaration.
In the example shown in figure 3.2, the tree walker first reaches the declaration's node and adds an entry like "i : integer" to the table. As the tree parser gets to the assignment, it looks up the variable `i` in the scope's symbol table, where it finds the information, that `i` is of type integer and, as `i` is being assigned a charstring value, reports a semantic error.
In the case that the table does not contain an entry for the variable, that is being referenced, that either means, that the variable is undefined or that it was declared in a different scope and therefore is contained by a different symbol table.

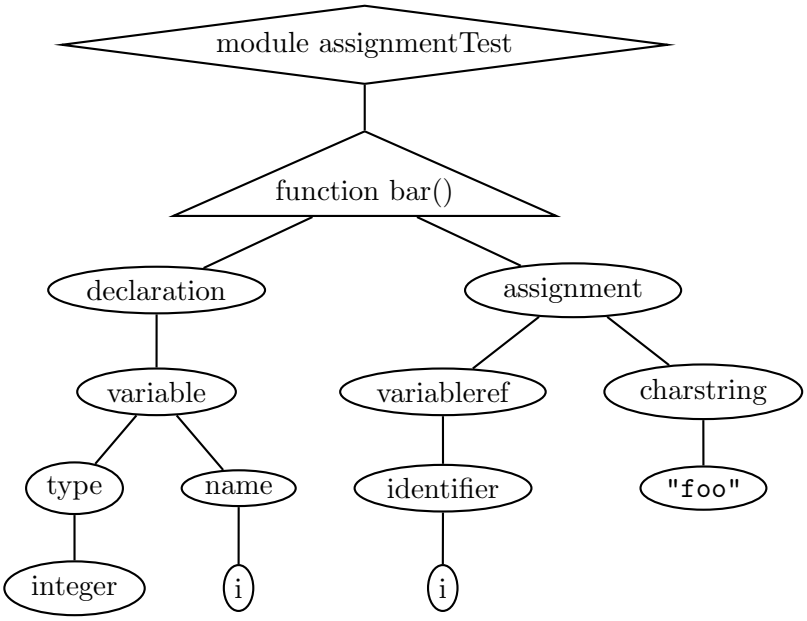

Figure 3.2: The simplified parse tree for the assignmentTest module

3.2.2 Variable scopes

Most modern programming languages contain the concept of (variable) scopes, so does TTCN-3. A typical TTCN-3 module consists of multiple scope levels and, therefore, its resulting parse tree contains several symbol tables. Apart from the functions, there are also scopes for altsteps and testcases, the module control part and for blocks of statements like loops.

```
1 module assignmentTest2
2 {
3   modulepar{
4     integer i;
5   }
6   function bar(){
7     i:="foo";
8   }
9 }
```

Figure 3.3: A module containing a scopes spanning assignment

Figure 3.3 shows a function that assigns a value to a variable, which was declared as a global parameter of the containing module. This time, the tree walker does not find an entry for `i` in the function's symbol table, so it has to ascend the tree looking for an overlying scope – in the example that would be the module itself – and search its table for an entry about the variable `i`.

3.2.3 Registering functions and types

Besides the data types TTCN-3 offers, there is the possibility to create one's own types. So the tree parser also has to store information on which types have been defined inside a module. As types are globally available throughout a module, there is only one additional table needed for all of the module's self-defined types.
Another set of information has to be kept about the module's functions, for a variable can also be assigned the return value of a function or altstep, so each module has a third table, in which the tree walker stores its operations' names and return values.

4 Implementation

The plug-in was developed in three basic steps, the first of which was the implementation of a core editor part, which only focused on keyword highlighting and usual editors' behavior, whereafter the existing TTCN-3 parser was integrated into the plug-in, providing additional functionality such as syntactic checking including creating error markers and displaying an overview of a file's content in the outline page. Finally, the ability to perform semantic analysis was added to the tree walker.
A UML class diagram of the plug-in's basic structure and its most important classes is depicted in figure 4.1. The `ttcn3.core.parser` package contains the parser classes including a specialized node type's class out of which the parse tree is built, namely the `LocationAST`. The package `ttcn3.editors` contains those classes that constitute the editor. The `TTCN3ReconcilingStrategy` class from that package associates the editor and the parser. The `ttcn3.editors.outline` is responsible for creating an overview page for the file that is being edited. The following sections contain ample descriptions about how those classes working and how they collaborate.

4.1 The editor

The plug-in's main class is the `TTCN3Editor`, that is derived from the `org.eclipse.ui.texteditor.AbstractDecoratedTextEditor`, which already provides most of the functionality, that one expects a typical developer's editor to have. This includes basic tasks like copying, text searching and replacing and extends in line numbering and resource markers. Those given features were extended by the TTCN-3 specific parts.

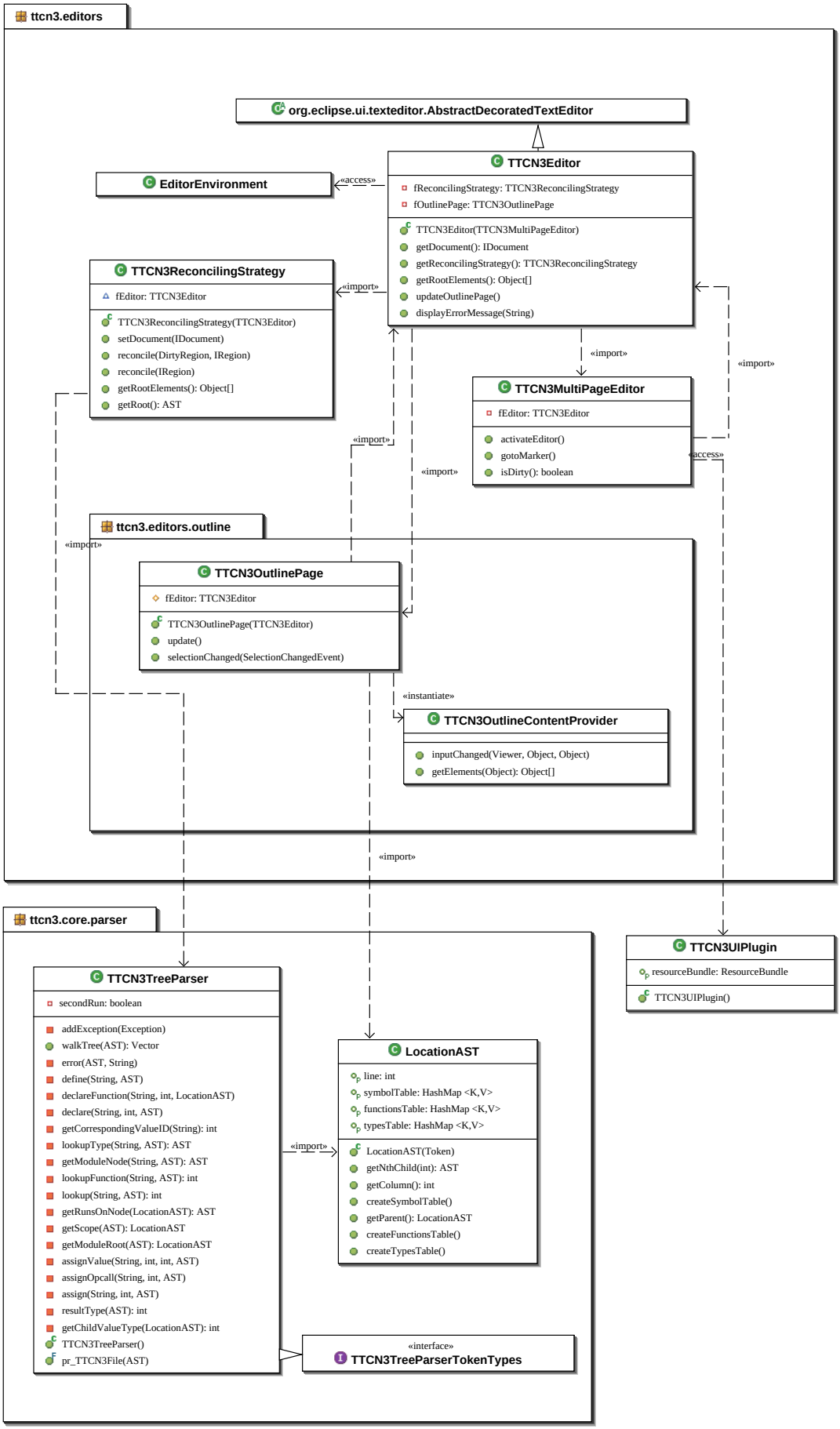

Figure 4.1: A class diagram of the plug-in's main structure

4.1.1 Syntax highlighting

The syntax highlighting consists of keywords highlighting on the one hand, as well as comments and strings highlighting on the other. The highlighting of keywords is taken care of by the `TTCN3CodeScanner`, that extends the predefined `org.eclipse.jface.text.rules.RuleBasedScanner` class. This class is basically a lexer, that transforms the words into a stream of "keyword" and "other" tokens, by means of which the words are assigned a specific appearance. The rules, which the `RuleBasedScanner` needs to distinguish between whitespaces, keywords and ordinary text, are created within the scanner's constructor, which

```
1  public TTCN3CodeScanner(
2      TTCN3ColorProvider aColorProvider) {
3    IToken keyword =
4        new Token(new TextAttribute(aColorProvider
5            .getColor("keyword"), null, SWT.BOLD));
6    IToken other = new Token(new TextAttribute(aColorProvider
7        .getColor("default")));
8    List rules = new ArrayList();
9    rules.add(new WhitespaceRule(
10       new TTCN3WhitespaceDetector()));
11   WordRule wordRule =
12       new WordRule(new TTCN3WordDetector(), other);
13   for (int i = 0; i < TTCN3_KEYWORDS.length; i++) {
14     wordRule.addWord(TTCN3_KEYWORDS[i], keyword);
15   }
16   rules.add(wordRule);
17   IRule[] result = new IRule[rules.size()];
18   rules.toArray(result);
19   setRules(result);
20 }
```

Figure 4.2: Defining the rules for the syntax highlighting

is shown in figure 4.2. Those are a `WhitespaceRule` and a `WordRule` both of which are provided by the `org.eclipse.jface.text.rules` package and which are instantiated in lines 9–12.
The `WhitespaceRule`, which is used to separate text into words to transform

them into tokens, is initialized with a `TTCN3WhitespaceDetector`, which comprises the solitary method `isWhitespace (char aChar)`. The `WordRule` is instantiated by passing a `TTCN3WordDetector` and a default token that is to be returned, if none of its rules matches. The `TTCN3WordDetector` provides the `isWordStart (char aChar)` and the `isWordPart (char aChar)` methods, which are used to check, whether the given character can belong to a TTCN-3 keyword at all. As for TTCN-3, all keywords consist of alphabetic characters, so both methods return the result of the predefined method `java.lang.Character.isLetter(aChar)`.
For each of the words, that are to be highlighted by the editor, an according entry is registered in the word rule (lines 13–15). Namely that is one entry for each of the keywords of the TTCN-3 language, which are specified in the `TTCN3_KEYWORDS` array. The scanner is instructed to return the "keyword" token for all of them. Finally the rules are applied to the scanner in line 19.
The highlighting of comments and strings, however, is taken care of by the `TTCN3PartitionScanner`, which divides the text into partitions using rules as well. This scanner distinguishes those partitions using of a `SingleLineRule` for strings, a `MultiLineRule` for multi line and an `EndOfLineRule` for single line comments. The former rules are specified by giving their start and end sequences, for example `/*` and `*/` delimit a multi line comment whereas strings are delimited by quotation marks at both ends. The `EndOfLineRule` matches from the given start sequence (`//`) to the end of the line. The colors, that are to be used to indicate the particular parts of the code, are managed by the `TTCN3ColorProvider`, that stores them in a `HashMap`, and can be adjusted in the TTCN-3 editor's preferences page. The assignmentTest2 module (figure 3.3) looks figure 4.3 in the plug-in.

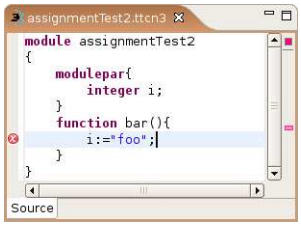

Figure 4.3: The plug-ins's editor showing the assignmentTest2 module

4.2 Embedding the parser

To make the eclipse platform aware of the TTCN-3 language to check a file for errors and to show resource markers at the appropriate locations in particular, the existing TTCN-3 parser was integrated into the plug-in. As ANTLR can be set to output Java code, the generated files could just be merged into the `ttcn3.core.parser` package.

4.2.1 Parsing the editor's content

The eclipse platform offers a concept for integrating a parser into one's plug-in. The editor has to provide a class, that contains instructions on how the parsing is to be performed. That class extends the `org.eclipse.jface.text.reconciler.IReconcilingStrategy` interface and therefore has to implement several methods which are called depending on whether the whole document or only a specific region should be reconciled. When eclipse notices, that the editor's content has changed, it queries the editor for its associated reconciling strategy and the platform calls this strategy's appropriate method.
The TTCN-3 plug-in always checks the the editor's complete input on a change, hence the `TTCN3ReconcilingStrategy` relays the call of any of those methods to its private `parse()` function, which processes the whole document and which is partially given in figure 4.4.
In lines 8–10 the method initializes a lexer object with the editor's input and a parser object, that works on the lexer. The file is processed by a call to the parser's `pr_TTCN3File()` method in line 12, that reads the tokens from the lexer one after another and builds the parse tree. The method can throw a `RecognitionException` in the case that a syntactic error was detected or a `TokenStreamException` that indicates that there were problems reading the file. In that case, an error message is displayed in the plug-in's status bar by the `log` function in line 16. The tree is then retrieved by calling the parser's `getAST()` method in lines 20–21 and can finally be passed to the tree walker, that examines it with respect to semantics. The parser throws a single exception in the case of an error, whereas the tree parser returns a vector of exceptions which represent the semantic errors which were detected in the file. In both cases, the exceptions are saved in a vector, that will be used to create the markers (line 28).

```
1  private void parse() {
2    TTCN3Parser parser = null;
3    TTCN3Lexer lexer = null;
4    TTCN3TreeParser treeParser = null;
5    Vector exceptions = new Vector();
6    AST root = null;
7    if (fEditor.getDocument().getLength() > 0) {
8      lexer = new TTCN3Lexer(
9          new StringReader(fEditor.getDocument().get()));
10     parser = new TTCN3Parser(lexer);
11     try {
12       parser.pr_TTCN3File();
13     } catch (RecognitionException e) {
14       exceptions.add(e);
15     } catch (TokenStreamException e) {
16       TTCN3UIPlugin.log(e);
17     }
18     if (exceptions.isEmpty()) {
19       treeParser = new TTCN3TreeParser();
20       Vector semanticExceptions = treeParser
21           .walkTree(parser.getAST());
22       if (semanticExceptions != null) {
23         exceptions.addAll(semanticExceptions);
24       }
25     }
26     root = parser.getAST();
27     if (!(exceptions.isEmpty())) {
28       createMarkers(exceptions);
29     }
30   }
31 }
```

Figure 4.4: The parser is linked to the plug-in

4.2.2 Marking errors

That vector is passed to the `createMarkers (Vector exceptions)` method, which, for each of the vector's exception elements, invokes the `createMarker (IFile aFile, RecognitionException anException)` function (figure 4.5) together with a handle to the file, that is being edited. This method ac-

```
1  private void createMarker(IFile aFile,
2      RecognitionException anException) {
3    Map markerAttributes = new Hashtable();
4    markerAttributes.put(IMarker.SEVERITY, new Integer(
5        IMarker.SEVERITY_ERROR));
6    markerAttributes.put(IMarker.PRIORITY, new Integer(
7        IMarker.PRIORITY_HIGH));
8    MarkerUtilities.setMessage(markerAttributes,
9        anException.getMessage());
10   MarkerUtilities.setLineNumber(markerAttributes,
11       anException.getLine());
12   try {
13     MarkerUtilities.createMarker(aFile, markerAttributes,
14         IMarker.PROBLEM);
15   } catch (CoreException e) {
16   TTCN3UIPlugin.log(e);
17   }
18 }
```

Figure 4.5: Marking a parse error in a file

quires the error's data from the exception and creates a set of attributes for the marker that is to be created in lines 3–11. Afterwards it makes use of the `org.eclipse.ui.texteditor.MarkerUtilities` class to have an error marker with the appropriate settings created for the given resource in lines 12–15 again logging a potential `CoreException` to the plug-in's status bar (line 16).
The markers are shown as annotations in the editor window as shown in figure 4.3 and as problems in the problem view which is depicted in figure 4.6.

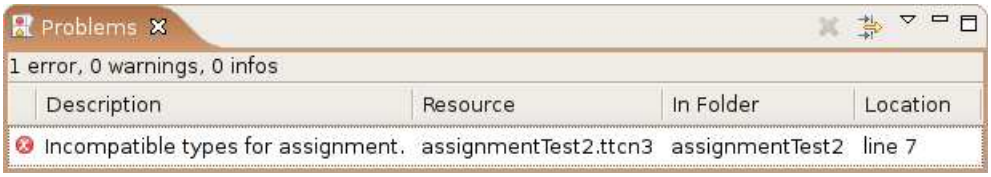

Figure 4.6: The problem view for the assignmentTest2 module

4.3 Outline view

Integrating the parser into the plug-in was also necessary for the creation of the outline page, which is generated out of the parse tree. For the assignmentTest2

Figure 4.7: The outline view for the assignmentTest2 module

module the outline view would look like shown in figure 4.7. The package `ttcn3.editors.outline`, which takes care of the outline view for a TTCN-3 file's contents, comprises three parts.
The `TTCN3OutlineContentProvider`, which extends the `org.eclipse.jface.viewers.ITreeContentProvider` template class, is responsible for determining the file's most important structural units, which were described in section 3.1, by recursively examining the parse tree.

For each node, the `hasChildren (Object anElement)` function checks, whether `anElement` has children that are of a type, which is to be listed in the outline view. The `getChildren (Object anElement)` function, shown in figure 4.8

```
1  public Object[] getChildren( Object anElement ) {
2    Vector theChildren = new Vector();
3    if (anElement instanceof AST
4        && ((AST) anElement).getFirstChild() != null) {
5      AST aChild = ((AST) anElement).getFirstChild();
6      do {
7        int type = aChild.getType();
8        if (type == TTCN3ParserTokenTypes.ModuleControlPart
9            || type == TTCN3ParserTokenTypes
10               .ModuleDefinitionsPart
11           || type == TTCN3ParserTokenTypes.TTCN3Module) {
12         theChildren.add(aChild);
13       } else if (type ==
14           TTCN3ParserTokenTypes.ModuleDefinition) {
15         int subtype = aChild.getFirstChild().getType();
16         if (isFunctionDef(subtype)) {
17           theChildren.add(aChild);
18         }
19       }
20     } while ((aChild = aChild.getNextSibling()) != null);
21   }
22   return theChildren.toArray();
23 }
```

Figure 4.8: Determining which nodes are to be shown in a file's outline view

checks the type of each of `anElement`'s children and returns the ones that are to be taken into account for the outline page's creation. The loop in lines 6–20 iterates over every child adding them to the set of considered nodes if they represent modules or their control or definition parts (lines 8–12). As for module definitions, which are the parent nodes for testcases, altsteps and functions among others, the tree parser determines its child's type in line 15 and, if it represents one of the function types (line 16), the appropriate module definition is also added to the set in line 17. Finally, the set is returned in line 22.

The outline page's elements' labels are set by the `TTCN3OutlineLabelProvider`, which specifies both the labels' text and its icon that is to be shown in the outline view. Those can be queried calling the `getImage (Object anElement)` or the `getText (Object anElement)` function respectively. The image depends on the node's type, being a special one for testcases, altsteps or functions and a default one for modules and their definition and control parts. The text is either the name for methods, or the textual representation of the node's type. Finally, the `TTCN3OutlinePage`, that is derived from the abstract `org.eclipse.ui.views.contentoutline.ContentOutlinePage`, links the provider classes to the outline page's tree viewer. Furthermore, it takes care of setting the editor's focus to a specific part of its content, if the corresponding label is clicked in the outline view.

4.4 Semantic analysis

As already mentioned in section 2.2.1, all the semantic checking is done by the tree parser, so its class has to be extended by several methods, which are called, when the tree walker reaches a node, that is relevant for the semantics of a TTCN-3 file. Those methods are stated in the tree parser's grammar file's action section, which is unalteredly transferred into the generated Java code. The functions are called from within the tree parser's definition, that is specified in the file `TTCN3TreeParser.g` and from which figure 4.9 shows an excerpt.

```
1  pr_SingleVarInstance : {
2    declare( ((LocationAST)_t).getNthChild(2).getText(),
3           ((LocationAST)_t).getParent().getParent()
4               .getNthChild(3).getType(),
5           _t);}
6    #( SingleVarInstance
7      pr_Identifier ( pr_ArrayDef )? (
8        {assign(((LocationAST)_t).getParent().getNthChild(2)
9              .getText(),
10             ((LocationAST)_t).getNthChild(2).getType(), _t );
11       }
12     pr_Expression )? )
13 ;
```

Figure 4.9: Function calls invoked when parsing a variable instantiation

During the process of semantic checking the tree has to be traversed twice. This is necessary because of the possibility to import specifications from other modules, which could be defined in a subsequent section of the file[1]. As that section would not have been processed by the tree parser when it reaches a node in a module that uses definitions from the imported one, it would not be equipped with the necessary tables so far. So the tree parser processes the tree one time to build up the symbols', functions' and types' tables and once again to check the variable assignments.

[1] In fact, the TTCN-3 standard does not regulate how modules are to be organized in files. Hence, the editor allows several modules to be placed inside a single file

4.4.1 Declaration of variables, types and operations

At the first pass, a symbol table is created for every scope's root node and the modules' root nodes are also provided with a functions and a types table. Each of those tables is of type `java.util.HashMap`. Furthermore all the modules' variable declarations are registered in the corresponding scope's symbol table. In the parse tree, that is built by the TTCN-3 parser, a variable declaration's root node is a `SingleVarInstance`. Figure 4.9 shows the part of the tree walker's grammar file, which contains the instructions on how to process a node of that type.

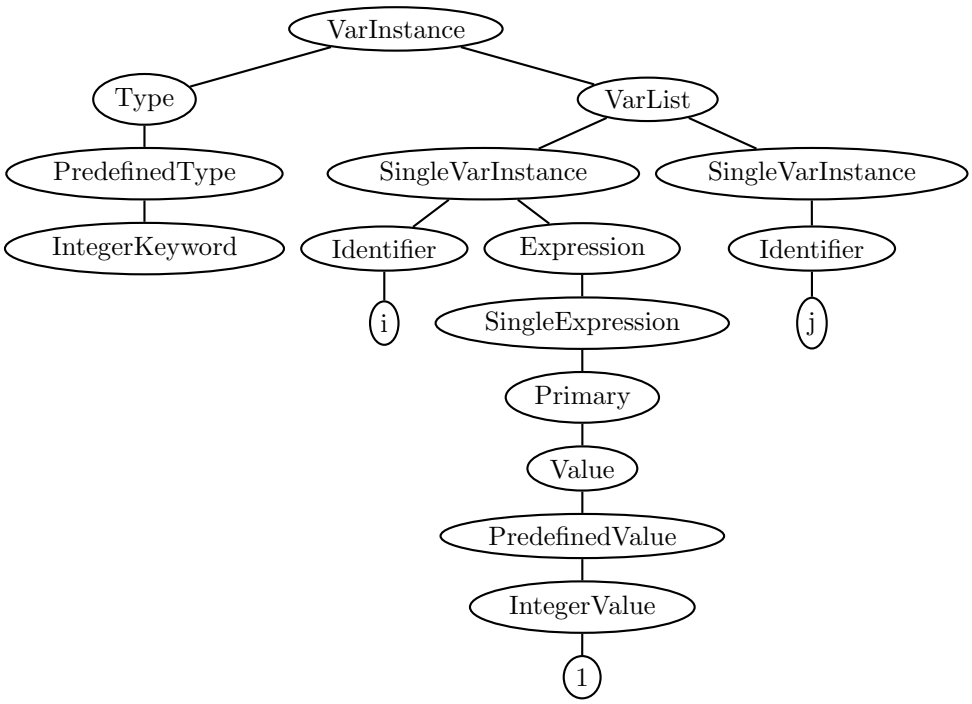

Figure 4.10: The parse tree for variable declarations

The parse tree for the variable declaration `var integer i:=1, j;`, as depicted in figure 4.10, would be processed in three steps, two for the declaration of the variables and one for the assignment.

As for the declarations, the function `declare (String tokenName, int type, AST anAST)` is called, which is shown in figure 4.11. The required arguments

```
1  private void declare( String tokenName, int type,
2      AST anAST ) {
3    if (secondRun) return;
4    LocationAST scope = getScope(anAST);
5    int previousDefinition = lookup (tokenName, anAST);
6    if (previousDefinition != EOF){
7      error( anAST,
8          "Duplicate definition of \""+tokenName+"\"" );
9    }
10   if (type == TypeReference
11       && anAST.getType() == SingleVarInstance){
12     String typeName = ((LocationAST)anAST).getParent()
13         .getParent().getNthChild(5).getText();
14     AST theType = lookupType ( typeName, anAST);
15     if (theType.getType() == SubTypeDef){
16       type = ((LocationAST)theType).getNthChild(3)
17           .getType();
18     }
19   }
20   String typeName = TTCN3TreeParserTokenTypes.class
21       .getFields()[type-2].getName();
22   if (typeName.endsWith("Keyword")) {
23     type = getCorrespondingValueID(typeName);
24   }
25   scope.getSymbolTable().put(tokenName, new Integer(type));
26 }
```

Figure 4.11: Registering a variable in the symbol table

for the method are retrieved from the parse tree by the tree walker as shown in figure 4.9. Since TTCN-3 does not allow the declaration of multiple variables with the same name, `tokenName` is first verified not yet to be declared by trying to find it in the parse tree by using the `lookup` function in line 5. If the variable has not been introduced before, it is registered in its enclosing scope's symbol table. In the case that the variable's type is derived from a predefined one, it

is registered to be of that supertype (lines 15–18).
The method `getCorrespondingValueID (String typeName)` in line 23 is required, because of the different nodes created for a type's keyword and a value of corresponding type, for instance "IntegerKeyword" and "IntegerValue" like in the example on page 29. The method uses Java Reflection [9] in lines 20–21 to determine the value of the correlating type's token constant from the `TTCN3TreeParserTokenTypes` interface.
Type and operation declarations are treated analogously by making use of the module's types or operations tables respectively. After the first pass, the parse

```
1  module simpleModule
2  {
3    modulepar{
4      integer i;
5    }
6    function foo() return integer{
7      var integer j := i;
8      return j;
9    }
10 }
```

Figure 4.12: A TTCN-3 module with a parameter and a function

tree for the module shown in figure 4.12 would be extended by a symbol and a functions table for the module node and a symbol table for the function. The resulting nodes with their tables are depicted in figure 4.13.

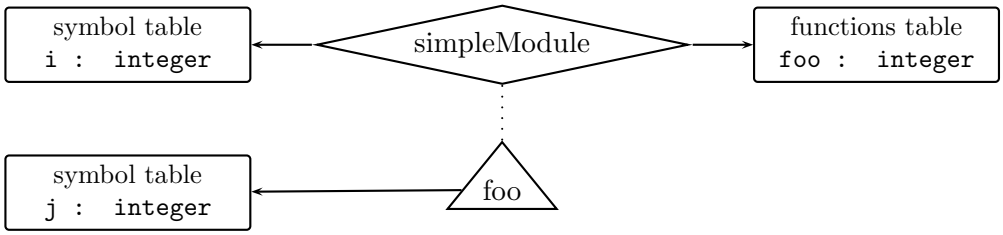

Figure 4.13: A module and a function node with their tables

4.4.2 Assignments

During the second pass, the tree walker validates every variable assignment. The particular variable is looked up in the scope's symbol table by the `lookup (String tokenName, AST anAST)` function (figure 4.14). The auxiliary method `getScope (AST anAST)` (line 2) returns the root node of `anAST`'s enclosing scope. At first, this node's symbol table is checked for an entry for the variable called `tokenName` and, if there is one, its according type is returned (lines 3–7). If the table does not contain an entry for this variable, the further procedure depends on the scope's type.
If the scope belongs to a testcase or a function or altstep that runs on a specific component type, which is then referenced by a `runs on <typename>` statement, the method may as well use variables, which are declared in that component type. In this case, the tree walker makes use of the method `getRunsOnNode (AST anAST)` in line 3, which returns the node, that contains the information on which type the specific method is to be executed. The type's name is then looked up in the module's types table (lines 21–22) and, if it has been defined, the tree walker proceeds with querying the referenced type's definition's symbol table for an entry about that variable (lines 28–29). Otherwise, the tree parser recursively continues its search upwards the tree (line 31) until either it finds an entry or it ends up at the file's root node.
Assuming that the specific variable was declared previously, the tree parser calls the `assign (String tokenName, int type, AST anAST)` method, that is used to distinguish whether the assignment is a value or variable reference or a function reference. In the case of either reference, the tree parser calls the `lookup` function for a variable or the `lookupFunction` for a function reference and deals with the return value as if a value of that type was directly to be assigned to the variable. The tree walker now needs to check, if the assigned value's type matches the variable's type by calling the `assignValue` function, which is partially given in figure 4.15.
This method compares the variable's type with the type of the expression that is to be assigned and returns in line 7, if they match. Timers need a special treatment here, because the parser does not distinguish between ordinary float values and those, that are being assigned to a timer. So, for a timer, the method would be called with `type` being `FloatValue` and `previousDefinition` being `TimerValue`. This case is handled in lines 3–6. If the assigned value is a vari-

```
1  int lookup( String tokenName, AST anAST ) {
2    LocationAST scope = getScope(anAST);
3    if (scope.getSymbolTable()!= null
4        && scope.getSymbolTable().containsKey(tokenName) ){
5      return ((Integer)scope.getSymbolTable()
6          .get(tokenName)).intValue();
7    }
8    if ( scope.getParent() != null){
9      AST runsOnNode = null;
10     switch (scope.getType()){
11       case FunctionDef:
12       case AltstepDef:
13         runsOnNode = getRunsOnNode(scope);
14         if (runsOnNode == null) {
15           return lookup(tokenName, scope.getParent());
16         }
17       case TestcaseDef:
18         runsOnNode = getRunsOnNode(scope);
19         String runsOnName = ((LocationAST)runsOnNode)
20             .getNthChild(3).getText();
21         AST runsOnTargetNode =
22             lookupType(runsOnName,getModuleRoot(anAST));
23         if (runsOnTargetNode == null){
24           error(anAST, "Type " + runsOnName
25               + " could not be found");
26           break;
27         }
28         return lookup(tokenName,
29             runsOnTargetNode.getFirstChild());
30       default:
31         return lookup(tokenName, scope.getParent());
32     }
33   }
34   return EOF;
35 }
```

Figure 4.14: Looking up a variable declaration

```
1  private void assignValue( String tokenName, int type,
2    int previousDefinition, AST anAST) {
3    if (previousDefinition == TimerValue){
4      assignValue(tokenName, type, FloatValue, anAST);
5      return;
6    }
7    if (type == previousDefinition) return;
8    else if (type == EnumeratedValue){
9      String referencedVariable = ((LocationAST)anAST)
10         .getNthChild(7).getText();
11     int referencedVariableType =
12         lookup(referencedVariable, anAST);
13     if (referencedVariableType == EOF){
14       error(anAST, "Variable "+ referencedVariable
15           +" was not found");
16       return;
17     }
18     assignValue( tokenName, referencedVariableType,
19         previousDefinition, anAST);
20     return;
21   }
22   error( anAST, "Incompatible types for assignment. Found "
23       + TTCN3TreeParserTokenTypes.class
24           .getFields()[type-2].getName() +", expecting "
25       + TTCN3TreeParserTokenTypes.class
26           .getFields()[previousDefinition-2].getName());
27 }
```

Figure 4.15: Verifying the assignment of a value to a variable

able reference, its type is looked up in lines 11–12 and, if it was declared, the `assignValue` calls itself giving the referenced variable's type as the type that is to be assigned (lines 18–19).

4.4.3 Assignment of nested expressions

Of course, not only can one assign simple values to variables, but also nested statements like `i:=1*1;` are possible. Figure 4.16 depicts a parse tree similar to

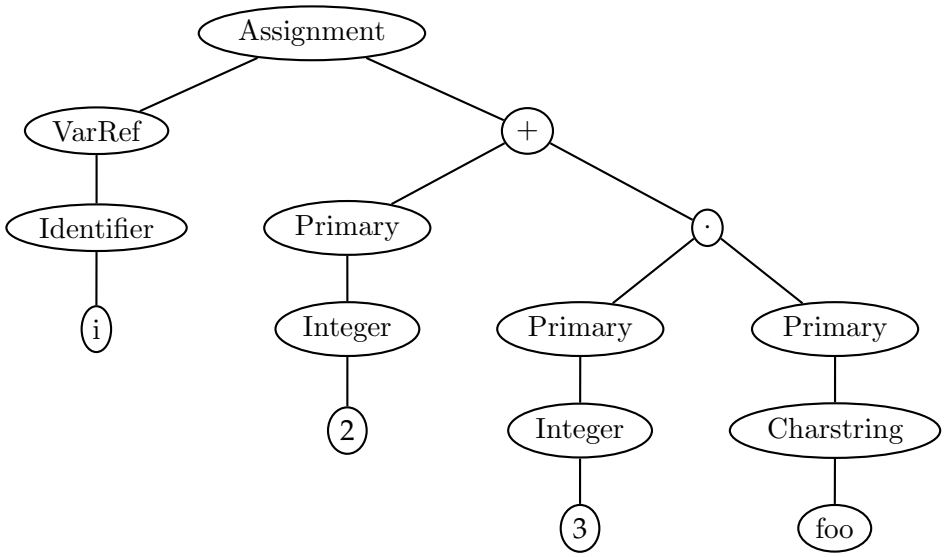

Figure 4.16: The simplified parse tree for an assignment of a nested expression

the one that would be built for the statement `i:=2+3*"foo"`. To have the tree walker process those nested statements, another auxiliary method is needed to determine the type of the value that is yielded by computing the expression and then proceed the same way as if a single value was assigned.

```
1  private int resultType (AST anAST){
2    int leftType = leftChild.getType();
3    int rightType = rightChild.getType();
4    if (leftType != Primary){
5      leftType = resultType (leftChild);
6    }
7    if (rightType != Primary){
8      rightType = resultType (rightChild);
9    }
10   if (leftType == EOF || rightType == EOF) return EOF;
11   if (leftType == Primary) leftType =
12       getChildValueType(leftChild);
13   if (rightType == Primary) rightType =
14       getChildValueType(rightChild);
15   if (leftType == EOF || rightType == EOF) return EOF;
16   switch (anAST.getType()){
17     case PLUS:
18     case MINUS:
19     case STAR:
20     case SLASH:
21       if (leftType == rightType){
22         if (leftType == IntegerValue
23             || leftType == FloatValue){
24           return leftType;
25         }
26       }
27       error (anAST,
28           "Incompatible types for arithmetic operation");
29       break;
30   }
31   return EOF;
32 }
```

Figure 4.17: Computing a nested expression's result's type

The method `resultType (AST anAST)`, which is partially shown in figure 4.17, returns the type of the value resulting from concatenating `anAST`'s children with the operator that is specified in the root node. For each child, the type is deter-

mined, depending on its kind, by a recursive call of the `resultType` function in lines 4–9 for a child that represents a subexpression or by calling the auxiliary `getChildValueType` method (lines 11–14) that returns the type of a "Primary" node's child, which again can either be a definite value or a variable or function reference.
Once the operands' types are determined, the operation's result type is computed depending on the operator. The excerpt shown in figure 4.17 deals with arithmetic operators, that can be applied to either integer or float type operands and which returns a result of the respective type (lines 21–26). Besides the arithmetic terms, the `resultType` function can also process modulo, shift and rotate operations.

5 Remarks on the plug-in

5.1 Installation

To install the TTCN-3 plug-in, just unpack the given ZIP file. For a system-wide installation, extract it into the eclipse root directory. On linux systems that is supposed to be `/usr/share/eclipse` for example. For a per-user installation, use the user's appropriate eclipse directory (mostly `~/.eclipse/eclipse`) instead.

5.1.1 Requirements

Besides the eclipse platform itself, the plug-in requires the ANTLR plug-in for eclipse to be installed. That plug-in and information on how to install it can be obtained from the `http://antlreclipse.sourceforge.net` web site.

5.2 Limitations

The plug-in's semantic checking abilities are limited to variable assignments so far. The following list gives some examples of what is not being supported yet.

- type checking on comparisons
  e.g. `if (4 == true) {...}`
- importing from modules in other files
- checking return types of functions
- checking assignments to be in the proper range for subtypes
  ```
  type integer NegativeInt (-infinity ..  -1);
  var NegativeInt := 0;
  ```
- forbid overwriting of constants

6 Conclusion

By the steps depicted in the preceding chapters, the aims, that were set in the introduction, were achieved: The plug-in's editor integrates with the platform providing syntax highlighting in customizable colors, which is one of the most important features easing the editing of source code. Besides that, the plug-in shows an overview of the file being worked on in the outline page thus allowing fast navigation between the most important parts of a file. Furthermore, it shows parse errors as annotations in the editor's vertical annotation ruler as well as in the problem view, both of which conspicuously indicate errors throughout a file.
As for the semantic analysis, the plug-in detects the use of undeclared or inaccessible variables, assignments of values of an unsuitable type and misused operands in arithmetic, modulo, shift and rotate expressions. In addition, it supports importing variables and type definitions from other modules in the *same file*.
The plug-in utilizes the powerful environment of the eclipse platform and extends it by the capability to deal with source code in the TTCN-3 language. As a result it provides a comfortable way to edit TTCN-3 files making the task of test specification more convenient and less error-prone.

Bibliography

[1] ANTLR parser generator.
`http://www.antlr.org`.

[2] ANTLR plugin for eclipse.
`http://antlreclipse.sourceforge.net`.

[3] Eclipse platform API specification.
`http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/`.

[4] Eclipse.org website.
`http://eclipse.org`.

[5] Hosting your own language in eclipse.
`http://www.awprofessional.com/articles/article.asp?p=370625`.

[6] Javacc project home.
`https://javacc.dev.java.net`.

[7] JSEditor eclipse plug-in.
`http://jseditor.sourceforge.net`.

[8] The lex & yacc page.
`http://dinosaur.compilertools.net`.

[9] The reflection API.
`http://java.sun.com/docs/books/tutorial/reflect`.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann.
Compilerbau, volume 1.
Oldenbourg Wissenschaftsverlag GmbH, second edition, 1999.

[11] Berthold Daum.
Java-Entwicklung mit Eclipse 3.
dpunkt.verlag GmbH, second edition, 2004.

[12] ETSI European Standard (ES) 201 873-1 V3.1.1 (2005-06).
The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.
European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Recommendation Z.140, `http://www.ttcn3.org/doc/es_20187301v030101p.pdf`.

[13] Jens Grabowski and Michael Schmitt.
TTCN-3 - Eine Sprache für die Spezifikation und Implementierung von Testfällen.
'at - Automatisierungstechnik', Oldenbourg Verlag, 3/2002:A5–A8, 2002.

[14] Jens Grabowski and Andreas Ulrich.
An Introduction to TTCN-3.
Tutorial, Proceedings of 'The TTCN-3 User Conference' at the European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), May 3-5, 2004, May 2004.

[15] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen.
Modern Compiler Design.
John Wiley & Sons Ltd, 2000.

[16] Stefan Hendrata.
Standardisiertes Testen mit TTCN-3.
automotive, 09-10/2004:64–65, 2004.

[17] Wei Zhao.
Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR.
Bachelor's thesis, Georg-August-Universität Göttingen, 2005.

List of Figures

Acronyms

ANTLR Another Tool for Language Recognition

API Application Programming Interface

ATM Asynchronous Transfer Mode

BNF Backus-Naur Form

DECT Digital European / Enhanced Cordless Telephone

ETSI European Telecommunications Standards Institute

GSM Global System for Mobile Communication

IPv6 Internet Protocol, Version 6

ITU-T International Telecommunication Union's Telecommunication Standardization Bureau

IDE Integrated Development Environment

PDE Plug-in Development Environment

TTCN Test and Test Control Notation