



LUND UNIVERSITY

Eclipse Plugin for Bluespec System Verilog

Zipfel, Tobias

2008

[Link to publication](#)

Citation for published version (APA):

Zipfel, T. (2008). *Eclipse Plugin for Bluespec System Verilog*. Lund University.

Total number of authors:

1

General rights

Unless other specific re-use rights are stated the following general rights apply:

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: <https://creativecommons.org/licenses/>

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Lund University
Department of Computer Science

– Project Report –

Bluespec SystemVerilog Eclipse Environment

Tobias Zipfel

VT 08

No.: ES-St-0034

Bluespec SystemVerilog (BSV) is a declarative hardware description language based on a synthesizable subset of SystemVerilog. Developing with BSV means so far to use scripts for existing editors, which enable highlighting and seldom support of the BSV compiler. The absence of a real IDE for BSV makes writing programs an inconvenient task.

This project tries to improve this by providing an BSV Eclipse plugin. Besides code highlighting, it includes project management, and the error feedback from the BSV compiler. To avoid unnecessary and time consuming compiler runs, an BSV parser, which is generated with JastAdd, is also provided. By this means, it is possible to parse source files while editing. The parser supports the user immediately with error feedback, before the compiler is started, and with a concrete syntax tree, which is displayed in the Eclipse outline view. Additionally, the whole build process can be automated with eclipse. In this way, the compiler run time is reduced significantly, which enables the developer to spend more time on programming.

Contents

1	Introduction	5
1.1	Purpose	5
1.2	Bluespec SystemVerilog	6
1.3	Project build-up	6
2	Background	8
2.1	Abstract Syntax Tree	8
2.2	Grammars	8
2.2.1	CAGs	9
2.2.2	RAGs	10
2.2.3	ReRAGs	11
3	Tools	12
3.1	JastAdd	12
3.1.1	JFlex - a Java Scanner Generator	12
3.1.2	Beaver - a Java Parser Generator	13
3.1.3	JastAdd builds AST Java classes	14
3.2	AstViewer - a Python script	16
3.2.1	Command line options	18
3.2.2	Known limitations	19
3.3	Project configuration with Eclipse	19
4	A Parser for BSV	22
4.1	AST and grammar definition	22
4.1.1	A goal for the parser	22
4.1.2	Common nodes	23
4.1.3	Package definition	25
4.1.4	Conditional statements	26
4.1.5	System call tasks	27

4.1.6	Types	28
4.1.7	Type synonym definitions	29
4.2	Attribute definitions and rewriting rules	31
4.2.1	Collection of imports and exports	31
4.2.2	Replacing missing package names	32
4.2.3	Unbound type variable check	32
4.3	Error collection	34
5	BlueSVEP	35
5.1	Existing sources	35
5.1.1	Features provided by interfaces and abstract classes	35
5.1.2	Features provided by attributes	37
5.2	Features	38
5.2.1	Editor features	38
5.2.2	Workbench features	42
5.3	Installation instructions	46
6	Conclusion and outlook	47
	Bibliography	52

1 Introduction

This chapter gives a short introduction to the project, as well as to the hardware description language Bluespec SystemVerilog (BSV) from Bluespec Inc. [1].

1.1 Purpose

The aim of this project is to ease writing code in BSV and managing BSV projects with a **BlueSpec SystemVerilog Eclipse Plugin** (BlueSVEP).

So far in writing BSV programs one depends on editors like Emacs, Vim or Jedit. All support syntax highlighting and Emacs has even a few more options. These features are based on configuration files, which are available from Bluespec.

A feature in Emacs is for example the linking of compiler errors to the source code after a compilation was started. Consider the case that the first error is in the uppermost package, which is compiled as a last step. With this long running compilation the error search will be slowed up. Even if nothing else has changed and only the uppermost package has to be recompiled, running the compiler takes time. Additionally the compiler stops at the first error, which requires one run for each error. To prevent these unnecessary compiler runs, the source code has to be parsed before or even while typing.

Using Eclipse as a tool framework, makes other valuable plugins such as Subclipse, a SVN plugin, or CDT, a C++ IDE, available in just one software. Other benefits are the many ways to enhance the plugin using the existing extensions points from Eclipse or to extend the basic functionality from the JastAdd core plugin (see sec. 5.1). Possible enhancements are, just to name a few, to display the syntax tree of the current program in the outline view, to search object references in the source code, to fold certain source code parts, or to refactor the source code.

Furthermore, the plugin can act as a precompiler for the BSV compiler to introduce own language constructs or to alter the source code before compiling in a desired way. It is also feasible to integrate the different command line tools, which come from Bluespec, into the Eclipse workbench, so they can be easily accessed. The auto-build functionality of Eclipse has also to be mentioned, which can start time consuming builds in the back-

ground. This leads to a more controlled build environment and prevent the programmer searching for error resulting from inconsistent build states.

1.2 Bluespec SystemVerilog

BSV is a declarative hardware description language based on a synthesizable subset of SystemVerilog [2]. It extends SystemVerilog by rules for state transitions, which can express concurrency easier. BSV also supports polymorphism and first-class objects, constructs that are usable without restrictions in the program, whereby code reusability is increased.

1.3 Project build-up

The project requires carrying out several steps. First, extracting the grammar for BSV, generate a parser for it and writing a plugin to use this parser. There are two steps necessary before the plugin implementation can begin. Both steps are made clearer in fig. 1.1.

As a first step, the BSV language description [2] was used to extract the BSV syntax words and literal definitions. With this information one can create an input file for JFlex (see sec. 3.1.1) to generate a scanner. The scanner reads a source file and replaces matched strings with special marks (tokens), which are further analyzed by a parser.

This parser is generated with JastAdd (see sec. 3.1) in a second step. For this purpose two things have to be done in parallel. Modeling an abstract syntax tree (AST) is one part(see sec. 2.1) and the other one is building-up a grammar description. The grammar uses the nodes specified in the AST to create an concrete syntax tree (CST) object for the BSV program. The necessary information is also taken from [2]. Additional attributes, defining equations and methods are described in the `.jrag` and `.jadd` files. These files are the input for JastAdd, which generates the parser and all Java classes corresponding to the AST nodes. JastAdd can in principle work with any Java parser generator. For this project Beaver is chosen (see 3.1.2) to deal with this task. See chapter 3 for more detailed description on this steps and the tools used.

The generated Java classes, the scanner and the parser can now be used in BlueSVEP to parse BSV programs. Besides parsing while typing, BlueSVEP provides syntax highlighting, project management and an explorer view. The plugin itself is based on a core plugin from JastAdd, which provides a general basis to build on (see sec. 5.1).

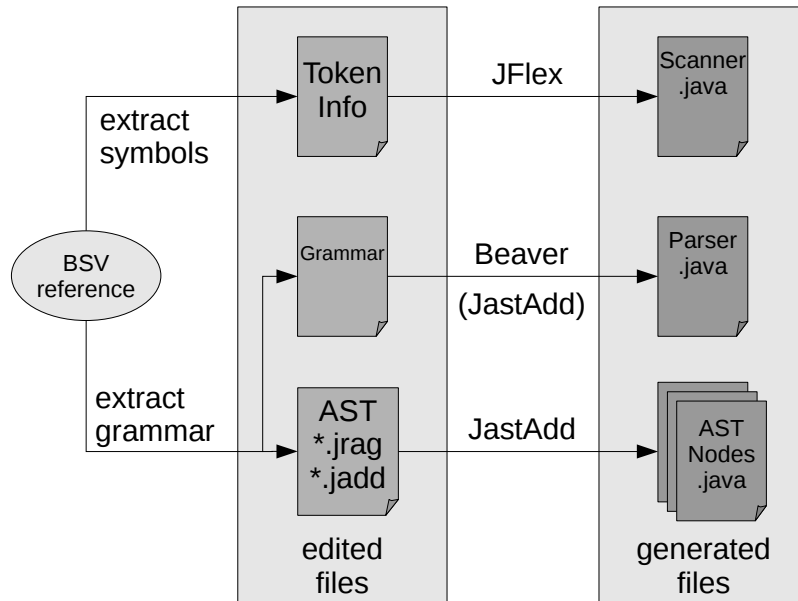


Figure 1.1: Project build-up

The report is organized as follows:

The next chapter gives a rough overview about rewritable attribute grammars. These are used for error checking, for type analysis, or to easily add functionality to the plugin provided by the core plugin.

Chapter 3 describes some of the tools used for code generation (JFlex, Beaver, and JastAdd), the software development environment Eclipse and the `AstViewer.py` Python script, that was created during the project. This script reads a AST file and generates a graphical representation of the tree structure.

The subject of chapter 4 is the generated parser and the AST model. The model is presented in sections corresponding to the chapters in [2]. In order to use the parser together with the plugin, the errors have to be collected and returned. The error collection method is therefore presented to make clear how the interaction between the parser and the plugin is built up.

Chapter 5 describes the plugin itself. The parts used from the JastAdd core plugin and the implemented features are described here in detail. A short installation instruction is given as well.

Finally, chapter 6 presents our conclusions and an outlook.

2 Background

This chapter introduces basic knowledge about how to represent a program in a tree structure with nodes and edges. The tree structure can be improved by adding node attributes, which save values or reference information to other nodes, or transformed by applying specific node re-writing rules to reorganize the tree.

2.1 Abstract Syntax Tree

In general an abstract syntax tree (AST) is a directed tree with finite set of labeled nodes. ASTs are used to express programs of a language in a structured way. Each language construct is presented as one of these nodes, called operators, which can have several children, called operands [3]. Building up an AST requires a grammar providing rules to produce the non-terminals, which are the root and the inner nodes. The leaf nodes, called terminals, correspond to variables and constants.

2.2 Grammars

To show the differences between various grammars, we use a small subset of the BSV language. This sub-language allows a root package node to have multiple type definitions, variable declarations and variable assignments. Type definitions consist just of a name, declarations have a type and a name, and assignments have a name and a value. The following grammar description is given in Backus-Naur form:

Listing 2.1: Grammar for a subset of the BSV language

```
1 Package ::= package string ; {BlockStmt} endpackage
   BlockStmt ::= <TypeDef> | <VarDecl> | <VarAssign>
3 TypeDef ::= typedef string string;
   VarDecl ::= string string ;
5 VarAssign ::= string = string ;
```

2.2.1 Canonical Attribute Grammars

A grammar is considered context-free if the left side of each production rules consists of only one non-terminal symbol and if these rules can be applied in any context. Canonical attribute grammars (CAGs) were first presented in [4] and are in general a composition of context-free grammars with attribute definitions for non-terminals. Attributes can either be synthesized or inherited. Synthesized attributes provide information up to the parent nodes and inherited attributes are used to push information down to child nodes.

An AST generated using the CAG given in listing 2.1 looks like fig. 2.1.

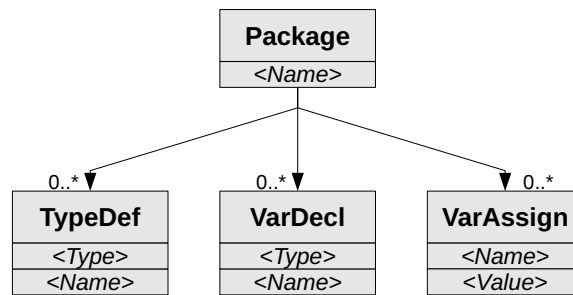


Figure 2.1: The AST for the BSV sub-language.

Listing 2.2 is used to generate an example of a CST. The code represents a correct program based on the sub-language from listing 2.1. The CST therefore looks very similar to fig. 2.1, except from the multiplicities and attribute values (see fig. 2.2). After parsing, every textual information provided is stored as values in node attributes.

Listing 2.2: Example program based on the sub-language

```

1 package Foo;
   typedef int MyInt;
3  MyInt number;
   number = 11;
5 endpackage
  
```

Storing only values and no references leads to multiple tree accesses if further information about the program is desired. This is an obvious drawback of CAGs. For example, starting with the variable assignment in line 4, it would take two more tree accesses every time to get the type definition node. This is not a problem for this short example

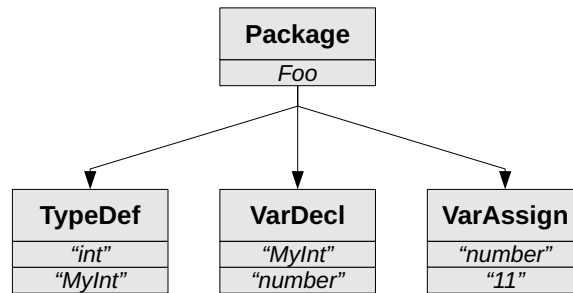


Figure 2.2: concrete syntax tree for the example program in listing 2.2

program, but it would become a time consuming task for a large and complex program with many nodes. As mentioned in [5] another uninspired solution to this would be to replicate the all necessary information where it is needed. This would lead to very complex attributes and complicate possible extensions to the grammar.

2.2.2 Reference Attribute Grammars

Reference attribute grammars (RAGs) are presented in [5] and address the drawbacks of CAGs. The introduced references (a special kind of attributes) allow a straightforward information retrieval from arbitrarily far away nodes. The concrete syntax tree (CST) for the example Program is shown in fig. 2.3 with dashed arrows as references.

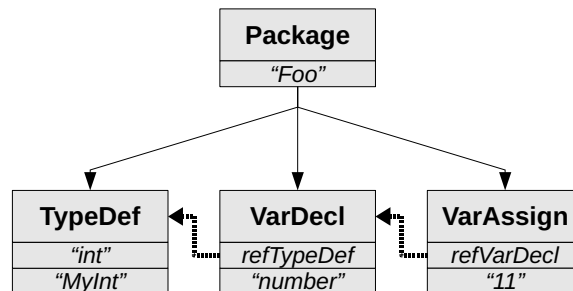


Figure 2.3: CST produced by a RAG.

This extension facilitate immensely type analysis and name analysis. A type analysis of the variable assignment in line 4 leads directly to the VarDecl node and from here to the TypeDef node.

2.2.3 Rewritable Reference Attribute Grammars

As described in [6], rewritable reference attribute grammars (ReRAGs) are an enhancement of RAGs. They add the ability to rewrite the AST nodes under condition dependent rules. Rewrites are triggered as soon as the nodes are accessed. This helps preventing a time consuming task and separates the workload into smaller pieces. To demonstrate a simple rewrite we use the following rule:

Listing 2.3: Rewrite rule with condition

```
1 rewrite VarAssign {  
    when (getType().equals(Type.NUMBER))  
3   to IntAssigning new IntAssign(getValue().toInteger());  
}
```

After applying this to the AST in fig. 1.1 the right VarAssign node is replaced by a new IntAssign node. An another rule could instead replace it by an UndeclaredVar node, if the variable in line 4 was not declared before. Rewrites could also be used for refactoring when, for instance, inner members are extracted to a upper level. For more concrete examples refer to [6].

3 Tools

This chapter presents the tools involved in the creation process of the scanner, parser and plugin. The general project build-up was already mentioned in chapter 1.3. Therefore the following sections provide just a detailed description of the respective steps.

3.1 JastAdd

JastAdd is an Java-based open source compiler compiler (see [7]). It is capable of generating a scanner, parser and compiler. Additionally it generates Java classes for all defined AST nodes. For this project a compiler is not of interest, as the BSV compiler from Bluespec is used. However, all other features are needed. As a first step, one can separate the writing of an input file from the scanner generation.

3.1.1 JFlex - a Java Scanner Generator

JastAdd uses JFlex (see [8]) to generate the scanner, which later provides the parser with the necessary token information. Tokens are in general syntax constructs, which match a specified string. The following listing (3.1) gives therefore an idea.

Listing 3.1: Section from the JFlex input file

```
...  
2 "package"    { return sym(Terminals.PACKAGE); }  
  "endpackage" { return sym(Terminals.ENDPACKAGE); }  
4 "("         { return sym(Terminals.LPAREN); }  
...
```

The method call `sym(Terminals.PACKAGE)` in line 2 will create a new symbol object. This object holds information about the line, the column and the type of syntax word, which is a start of a package in this case. With this information the parser can distinguish between syntax words and identifiers or comments.

3.1.2 Beaver - a Java Parser Generator

Beaver needs an input file with a grammar description (see [9] for details) to generate a the Java parser class. The grammar has to be expressed in the Extended Backus-Naur form, as presented in the following example, in listing 3.2.

Listing 3.2: Section from the Beaver input file

```

...
2 PackageDef packageDef =
    PACKAGE identifier SEMICOLON packageblock ENDPACKAGE endName
4   {:
    checkEndName(identifier, endName, PACKAGE);
6   return new PackageDef(identifier.getName(), packageblock);
    :};
8 ...

```

The first expression in line 2 represents the Java class `PackageDef` and the second the symbol `packageDef` used in the grammar. After the equal sign follows the production rule for this symbol.

The rule consists of terminal and non-terminal symbols. Terminals are given in capitals, like `PACKAGE` or `SEMICOLON`, are placed by the scanner (see line 2 and 3 in listing 3.1). Non-terminal symbols, like `identifier` or `packageblock`, lead to other production rules also defined in the grammar.

The lines between the `{: ... :}` are Java code, that is executed when the rule is applied. In the case of line 5, the method call `checkEndName()` compares the `endName` symbol with the `identifier` symbol to guaranty equal strings. Finally, a new `PackageDef` object is returned in the next line.

In addition to the grammar description, more information is necessary. Beaver has to know the tokens, otherwise produced by the scanner. Also the relations between grammar symbols and Java classes have to be set. `JastAdd` takes care of these last two tasks, which leaves nearly only the grammar design to deal with.

To choose more exactly between the rules while parsing, it is essential to fix the precedence of operators. This is done by listing them in the right order in the input file, otherwise Beaver would choose an order of rules by itself. This produces normally very many warnings and possibly does not lead to the desired behavior.

Another important task is error handling and gathering. This can be accomplished by overloading some methods inherited from Beaver classes or adding new ones. For this

purpose one can use the predefined syntax word `%embed {: ... :}` wherein all extra Java code have to be placed. This code and the grammar form the input file for JastAdd, which adds type and terminal symbol information. The outcome of this together with the precedence of operators build up the input file required by Beaver.

3.1.3 JastAdd builds AST Java classes

JastAdd generates Java classes from an AST definition. It also adds equations, attributes, rewrites, methods and fields to this classes defined in aspects. This information should be separate in two general file-types, to keep a certain clarity. The first type has the extension `.ast` for the AST definitions and the latter contains the aspect definitions and ends with `.jrag` or `.jadd`.

The following sections briefly describe the syntax for these files. Further information can be found in the reference manual [7].

The AST definition file

The AST definition file describes for each node, its base class, its children and attributes. An section of such a file is shown in listing 3.3.

Listing 3.3: Section from the AST definition file

```

...
2 abstract BlockStmt;
  abstract Def: BlockStmt ::= <DefID:String>;
4 PackageDef: Def ::= Body:PackageBlock;
  PackageBlock: Block ::= Exports Imports BlockStmt*;
6 ...
  IfStmt: CondStmt ::= CondPredicate TrueStmt:BlockStmt [FalseStmt:BlockStmt];
8 ...

```

It is possible to declare abstract classes, like in line 2, which unify common behavior.

The syntax at line 4 is used to inherit from such and other classes. In general, it starts with the new class name, followed by a colon and the base class. In this case, the class `PackageDef` inherits the attribute `<DefID:String>` from `Def` and defines the child node of type `PackageBlock` by itself.

The generated Java classes provide methods to access these defined child nodes and attributes. In line 4, for the child node named `Body`, the corresponding get method is

named `getBody()`. If no name is denoted, the method is named by `get` plus the type-name. The `get` method name of the inherited attribute `<DefIDE:String>` is therefore `getDefIDE()`.

In line 5, the character `*` is used after the child node `BlockStmt`. This means that a `PackageBlock` object can contain zero or more children of type `BlockStmt`. Internally this is stored with the `List` class, which handles the objects of this type like a vector.

Optional child nodes can be expressed with surrounding braces. The nodes are then encapsulated by an object of type `Opt` and the generated class has a method to check for existence. For instance, the class `IfStmt` in line 7 has a method named `hasFalseStmt()`, which returns the Boolean value `true` if the object has such a child.

Another node type is declared between slashes (`/ NodeName /`). These nodes are not generated by a parser. Instead, they are treated like attributes, which have to be defined by an equation. Therefore they are called non-terminal attributes (NTAs). How attributes and their equations are defined is shown in the following section.

The aspect definition file

This file type can contain attributes, equations and rewrites belonging to a certain aspect. In this case it should be saved with the extension `.jrag`. For instance, an aspect addressing type analysis and containing respective attributes and equations would be stored with such an extension.

Attributes are differentiated into two kinds. Synthesized attributes propagate information upwards to parent nodes while inherited attributes enable leaf nodes to access information from a parent node.

Synthesized attributes are used for example to extract type information or a printable name from certain child nodes. They are defined with the following syntax:

```
syn AttributeType ClassName.attributeName();
```

All synthesized attributes have to be defined by this class and by all its subclasses, if the class is abstract. Subclasses can also overwrite the definition from super classes. The syntax for the defining equation follows this pattern:

```
eq ClassName.attributeName() = Java-expression;
```

The use of inherited attributes is not as intuitive as the use of synthesized attributes. In this case attribute inheritance has nothing to do with class inheritance. The inheritance relates here to the AST structure. An inherited attribute declaration in a child node must be defined by an equation in all nodes that can contain children of this node type.

This is used for example in sec. 4.2.3 to provide information only accessible from the parent node. The syntax therefore is shown in the following:

```
inh ChildClassName.attributeName();
eq ParentClassName.getChildClassName().attributeName() = Java-expression;
```

A convenient feature of JastAdd is the automated copying of inherited attribute equations. An equation defined in a parent node is valid for all nodes between this parent and the child that declares the inherited attribute. To make this attribute visible in any other child node it has to be declared as inherited attribute.

All Java expressions can also be replaced by method bodies. Instead of:

```
... = Java-expression;
```

one can write:

```
{ ...; return Java-expression; }.
```

If an aspect simply consists of class fields or methods, the file should end with `.jadd`. An aspect covering all `toString()` methods, which are used to print out the actual AST structure, is an example for this case.

The general syntax for this looks as follows:

```
AccessType ReturnType ClassName.methodName(parameter list){ ... }
```

for methods and

```
AccessType Type ClassName.fieldName = InitialValue;
```

for fields.

3.2 AstViewer - a Python script

While constructing the AST definition file, one can get easily lost in the textual description of big amounts of nodes with all their inheritances and parent-child relations. It is much more convenient to have a visual representation of the AST definition.

A Python script was therefore written to reveal a possible reuse of existing nodes and to get a better overview. It needs an `.ast` file as input. The output is a `dot` file [10] and an image displaying the relations between the nodes with regard to the used command line options.

Two programs are used to generate the images. The first one is `dot`, which is used in most cases, and the second one is `neato`. The latter is invoked if `dot` fails or if the option `'-oC'` is used. The difference between `dot` and `neato` is that `dot` tries to generate hierarchical tree structures while `neato` tries to place the nodes corresponding to the lowest energy configuration concerning the system of nodes and their relations. The

relations between the nodes are treated here as springs. The length of these springs can be adjusted with the '-SL=X.X,X.X' option.

To present these different command line options, the following AST definition is used:

Listing 3.4: Example AST definition file

```

Program ::= Class;
2 Class ::= BlockStmt*;
  abstract BlockStmt;
4 abstract Decl: BlockStmt ::= <DeclID:String>;
  TypeDecl: Decl;
6 VarDecl: Decl ::= <varName:String>;
  VarAssign: BlockStmt ::= <VarName:String> <Value:String>;

```

Similar to an UML class diagram, the nodes are displayed as boxes with arrows to superclasses and children. Abstract classes have a gray shape and an italic font.

The output image format depends on the `-T` option, which corresponds to the same option for `dot`. Without this options the standard is `png`. All images for this report used the `-Tps2` option to generate postscript files. These are converted to `pdf` in a next step.

Without any options figure 3.1 shows the image for listing 3.4.

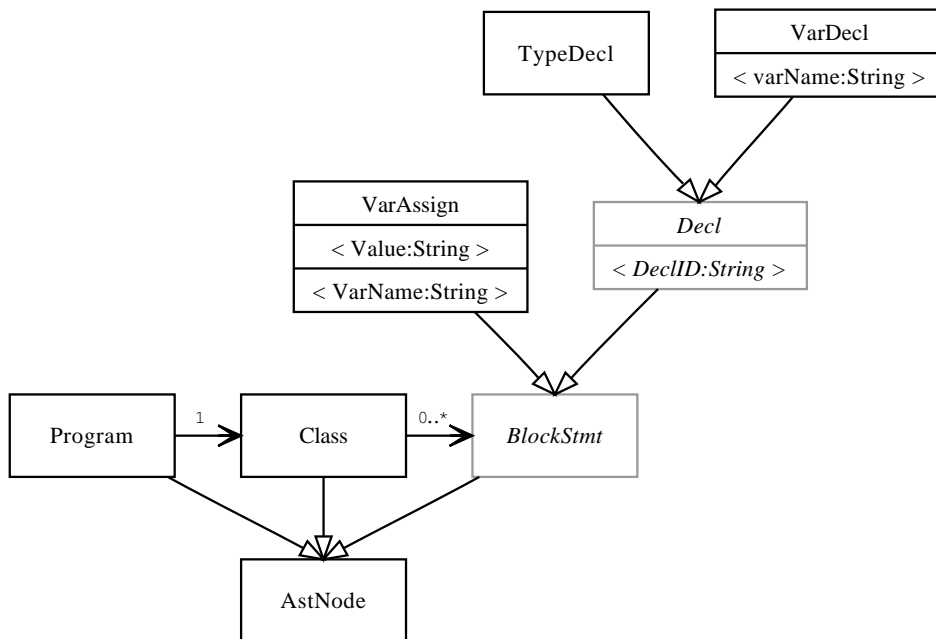


Figure 3.1: Output without options.

3.2.1 Command line options

The general invoke command has this form:

```
python AstViewer.py [options] fileName.ast
```

In the following each option is presented with a short description and, if necessary, a resulting image.

- oI** only inheritance dependencies are generated. See fig. [3.2](#).
- oC** only child dependencies are generated. Instead of generating images with `dot` `neato` is used. This will result in smaller images. See fig. [3.3](#).
- iC** children are included into the node shape. See fig. [3.4](#).
- iCI** children from superclasses are included into the node shape. See fig. [3.5](#).
- S** generates a `dot` file and an image for each section. To create a section one has to use the following pattern:

```
// [number|number.number] SectionName
```

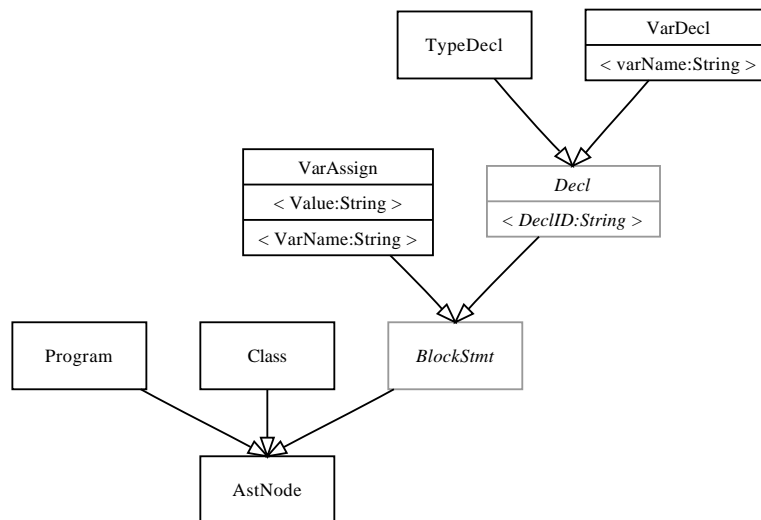
followed by the nodes belonging to this section.
- SL=X.X,X.X** set the spring length for `neato`. The first value is for inheritance relations, second for child relations. This will affect how close the nodes are placed.
- Tformat** sets the output image format. Corresponds to the same `dot` option.
Examples for *format* are: png, gif, ps2, ...
Refer to the `dot` manual for a complete list [\[10\]](#).

-**noImpL** disables the imprint of generation options on the image.

Multiple options can be used at the same time. Exceptions from this are:

- Only one of '-oI' or '-oC' can be used in the same invocation.
- If only '-iCI' is used, '-iC' is set automatically.

In most cases, the best results are produced if the '-oI' option is used. Otherwise a large amount of parent-child relations will degrade the overview and `dot` will produce larger images to place the nodes correctly.

Figure 3.2: Output with `-oI`.

3.2.2 Known limitations

It is not always possible to generate images for long AST definition files with many nodes, inheritance and child relations with `dot`. In this case, `AstViewer` tries to generate the image with `neato` instead. If a hierarchical node placement is nevertheless desired, using the `'-oI'` option together with `'-iC'` or `'iCI'` is recommended. Another solution could also be the use of the `'-S'` option to generate smaller images.

3.3 Project configuration with Eclipse

Eclipse is a well known integrated development environment (IDE) (see [11]). It is highly adaptable through the plugin interface and supports mainly the Java language. However, other languages are supported by plugins. The CDT plugin for C/C++ is just one example. Besides language specific plugins, there are several plugins heading towards a more general functionality, such as the Subclipse plugin for SVN integration.

Eclipse provides a project management especially suited for plugin development. For clarity, the project is organized in two of these plugin projects. The basic scanner and parser files form one project, called `BSVFrontend`. `BlueSVEP` is configured in a separated project. In this way, it is possible to develop and test the scanner and parser independently from `BlueSVEP`. Another reason is to use the bare parser on single input files or strings without the need of `BlueSVEP`.

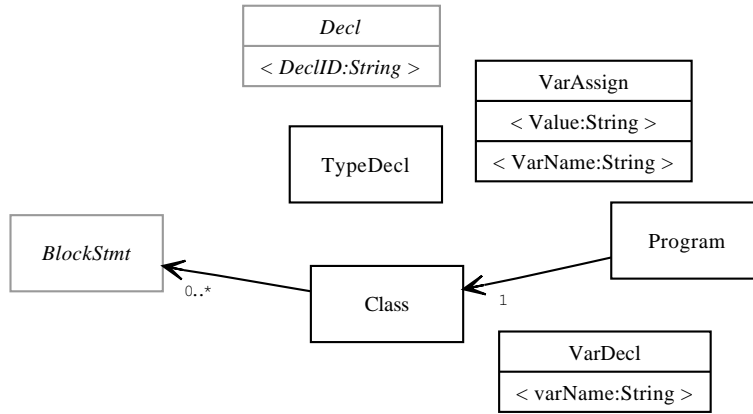


Figure 3.3: Output with -oC.

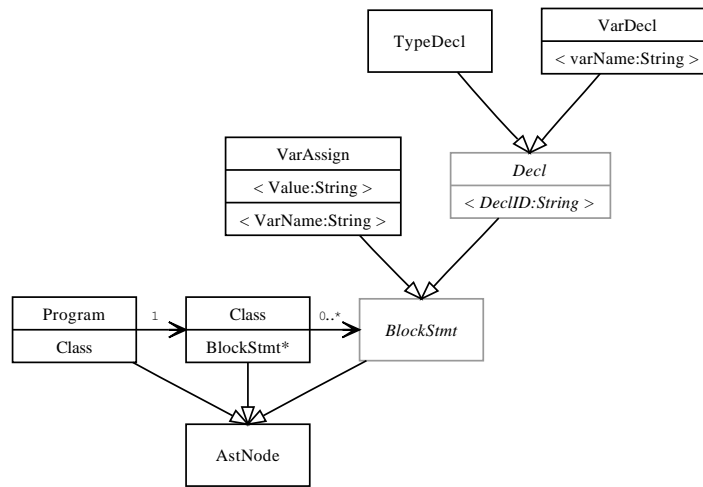


Figure 3.4: Output with -iC.

Within the BSVFrontend plugin, the input files for the parser generation are separated corresponding to the chapters in [2]. This is easier than to orient oneself in just one long file. These files are concatenated in the generation process to a single input file. The concatenation and other steps are run by an Ant XML build file in both projects. This guarantees the generation process to go on in a controlled and repeatable manner.

The BlueSVEP plugin has no runtime dependency to BSVFrontend, but it needs the basic files for scanner and parser for its own generation process. In this process a special preamble section is added to these files, which reflect project specific package structure and includes. To enable the JastAdd core plugin functionality, specific '.jrag' files are also added for the AST node generation.

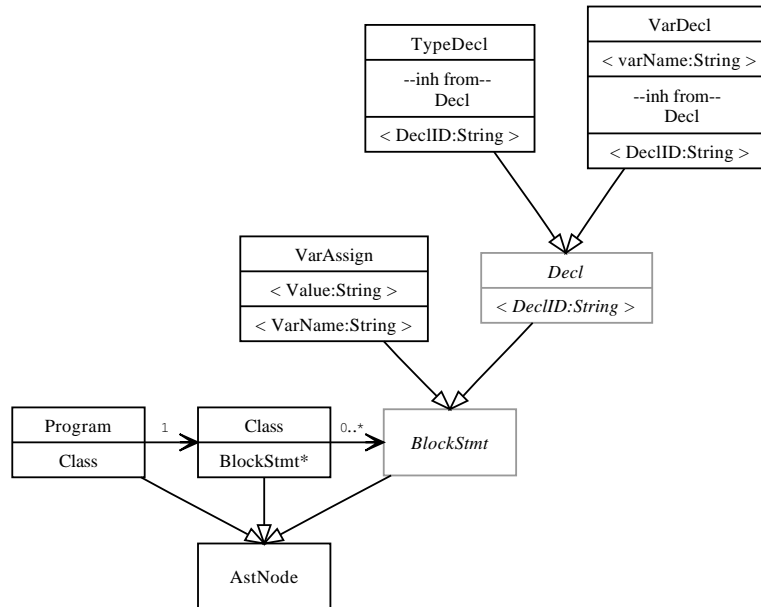


Figure 3.5: Output with -iCI.

Debugging is a very essential part of implementation. Therefore the log4j logger [12] is used to have more control over system printouts. For backup reasons and to provide version control, all projects are organized in a Subversion repository.

4 A Parser for Bluespec SystemVerilog

This chapter concerns the implementation aspects of the BSV Parser, which are presented in excerpts in the following sections.

Generating the parser for BSV requires an AST and grammar definition. As described in the former chapter, the generated parser uses the Java classes from the AST definition.

The outcome of the parser is the concrete syntax tree (CST) for the parsed source file. Feedback is given in form of errors deriving from different parsing steps.

4.1 AST and grammar definition

In the following, the AST and the corresponding grammar description are presented for several sections. The AST images are sometimes reduced to the essential nodes or separated into several parts to keep a certain clarity. Consider that the nodes in the AST represent Java classes. The terms 'node' and 'class' are therefore used in an interchangeable manner.

4.1.1 A goal for the parser

The parser generated by Beaver returns a root node, when calling the parse method. This node contains the whole CST for the parsed source file. A single BSV file can contain several package definitions. Figure 4.1 presents the AST for `Program`.

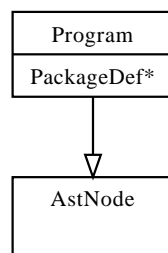


Figure 4.1: Parser goal: `Program`

4.1.2 Common nodes

These nodes are used in several other sections. The first part is shown in figure 4.2 includes `Identifier`, `Blocks` and `BlockStmt`.

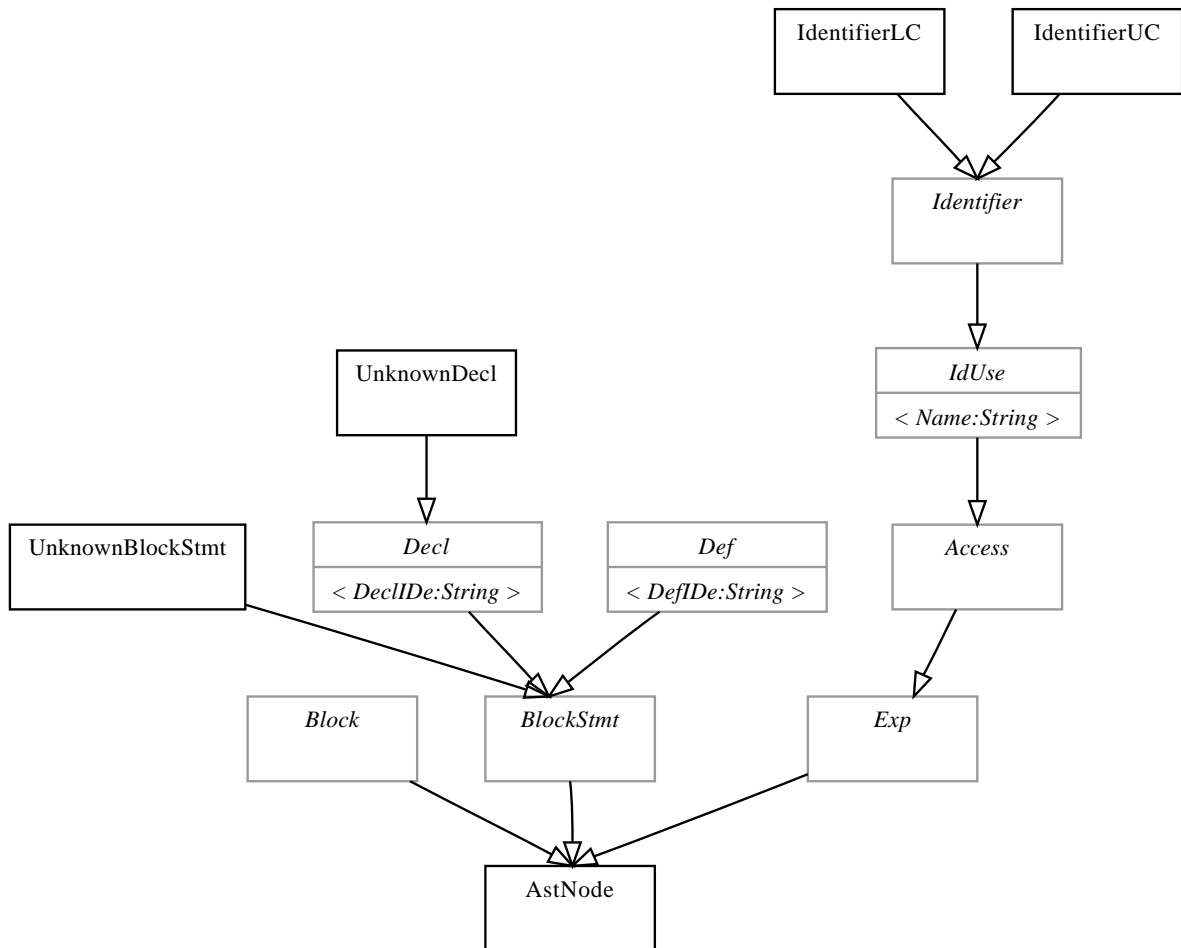


Figure 4.2: Common nodes, 1st part

- The abstract class `Identifier` has two sub classes `IdentifierLC`, for lower case identifiers, and `IdentifierUC`, for upper case identifiers. These are needed, as BSV requires to differentiate upper and lower case in some language constructs.
- The abstract class `Block` is the base class for several other classes such as `PackageBlock` or `FunctionBlock`.
- The abstract class `BlockStmt` is the base class for all kind of statements, that can occur inside a block.

The second part (see fig. 4.3) shows the abstract class `Member` with its subclasses. These are used for example by parts of a module or interface definition, which inherit from a suitable subclass of `Member`.

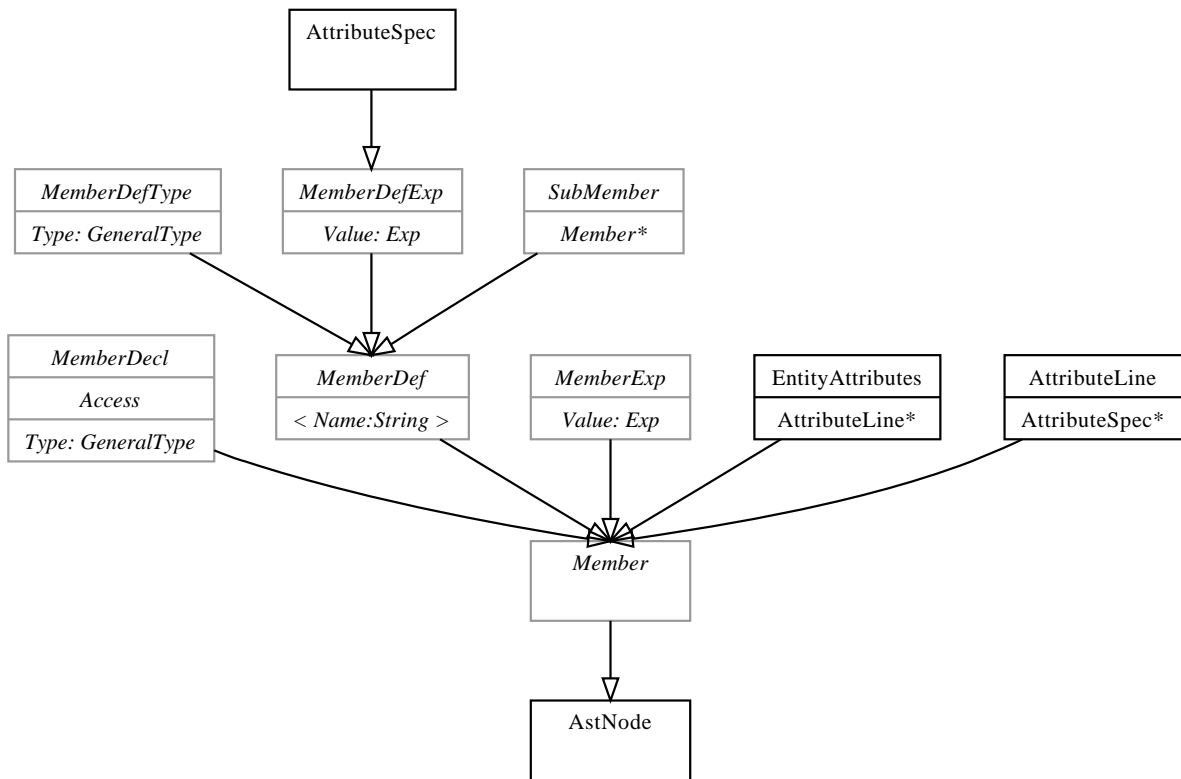


Figure 4.3: Common nodes, 2nd part

For instance, the classes `EntityAttributes`, `AttributeLine` and `AttributeSpec` are used before several language constructs in the following way:

```
(* AttributeA=20, AttributeB=true *)
(* AttributeC *)
module/interface/...
```

The grammar for `AttributeLine` (see listing 4.1) is taken as an example of how multiple children are implemented in the grammar. In line 3 the actual attribute is parsed. The `assignValue_opt` symbol is either nothing or a equal sign followed by an expression. The `attributeSpec` symbol can be written as comma separated list, according to line 8. To solve the repetition problem, the symbol definition for `attributeSpec_list` contains itself. Notice, that this list only contains the attribute for one line. Multiple lines require an additional production for another list, which is not shown here. This list is saved in objects of type `EntityAttributes`.

Listing 4.1: Grammar for `AttributeLine`

```

AttributeSpec attributeSpec =
2   identifier assignValue_opt
   {: return new AttributeSpec(identifier.getName(), assignValue_opt); :} ;
4
List attributeSpec_list =
6   attributeSpec
   {: return new List().add(attributeSpec); :}
8 | attributeSpec_list COMMA attributeSpec
   {: return attributeSpec_list.add(attributeSpec); :} ;

```

4.1.3 Package definition

A package definition is the uppermost language construct inside a BSV file. It consists of a block containing imports and exports and several block statements (see fig. 4.4).

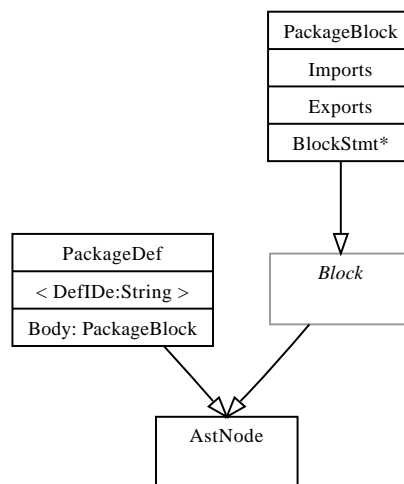


Figure 4.4: Package definition, 1st part

Listing 4.2 presents the possible subclasses/productions, that are allowed inside a package. In this manner, the AST needs only to handle abstract superclasses such as `BlockStmt`. The grammar description defines then a more detailed set of subclasses.

Listing 4.2: Grammar for package statements

```

BlockStmt packageStmt =
2   attrModuleDef | attrIFDecl
   | typeDef

```

```

4 | varDecl | scalarAssign
  | functionDef;

```

Imports and exports are not treated like normal block statements as they can only appear at the beginning of a package definition. Figure 4.5 shows that all classes related to imports and exports inherited from `Member`, either directly or through the abstract class `MemberDef`.

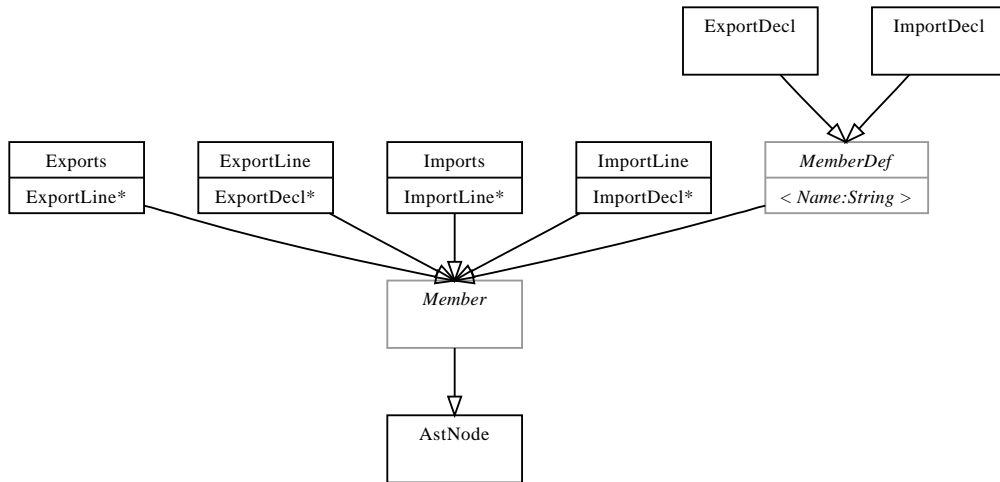


Figure 4.5: Package definition, 2nd part

The question arises why a package definition has extra classes to handle multiple import and exports lines. It would also be possible to have list of `ImportsLine/ExportLine` directly in `PackageDef`, such as for `BlockStmt`.

The reason therefore is founded in the 'content outline view' feature from the core plugin, which is presented in sec. 5.1. Every element, that should have a outline node with a name and icon, have to be a single node and not a list of nodes.

4.1.4 Conditional statements

Conditional statements can contain optional parts. In figure 4.6, the class `CaseStmt` has an optional child of type `DefaultItem` and the class `IfStmt` has the optional `BlockStmt` with the name `FalseStmt`.

For the latter, listing 4.3 shows how an optional child is implemented in the grammar. An `if` statement can be placed in different contexts. To be able to allow only proper statements, there has to be a specific `IfStmt` production rule for every context.

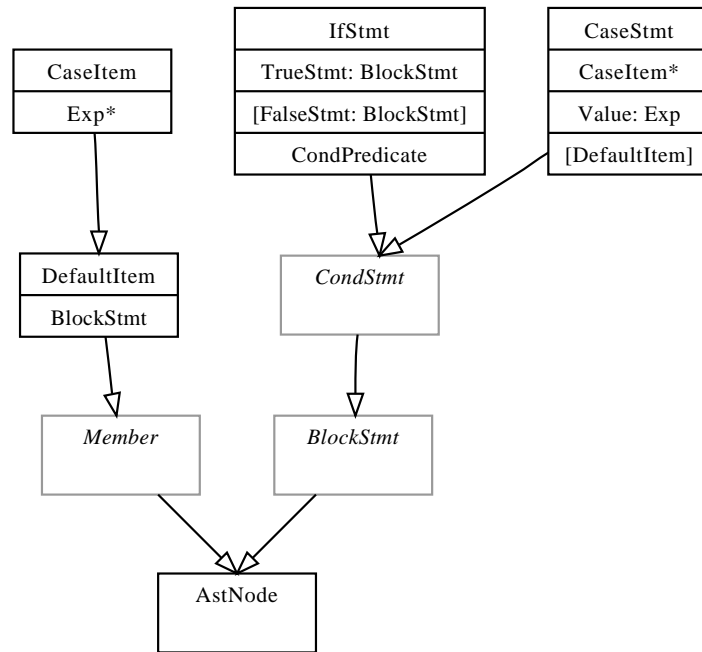


Figure 4.6: Conditional statements

Listing 4.3: Grammar for IfStmt

```

IfStmt functionIf =
2   IF LPAREN condPredicate RPAREN functionBlockStmt functionElse
   {: return new IfStmt(condPredicate, functionBlockStmt, functionElse); :};
4
Opt functionElse =
6   {: return new Opt(); :}
   | ELSE functionBlockStmt {: return new Opt(functionBlockStmt); :};

```

Optional children lead to a generated `boolean` method, that can be used to check if the parent has a certain child node. For `IfStmt` this method is called with `hasFalseStmt()`.

4.1.5 System call tasks

System calls in BSV are used, for example, to write and read files or to display variable values. Every call starts with a `$` followed directly by a name.

A context sensitive scanning is applied for system call tasks. After recognizing the initial dollar sign, the scanner switches to another context. In this `SYSTEMCALL` context,

only the predefined system calls are accepted. Listing 4.4 shows the relevant sections from the scanner file.

Listing 4.4: Scanner rules for system tasks.

```

SystemCallStart = "$"
2 ...
{SystemCallStart} { yybegin(SYSTEMCALL); }
4 ...
<SYSTEMCALL> {
6 "fopen"   { yybegin(YINITIAL); return sym(Terminals.SC_FOPEN); }
  "fclose"  { yybegin(YINITIAL); return sym(Terminals.SC_FCLOSE); }
8 ...
}

```

After a system call was identified, the scanner switches back to the initial state.

4.1.6 Types

Many constructs require type information. Variable declarations, methods or functions are just a few examples. The following grammar (listing 4.5) in the Backus-Naur form describes how a type can be build up. The resulting AST is shown in figure 4.7. The original grammar for types in [2] could not be used, as it parses more than the BSV compiler accepts.

Listing 4.5: Type grammar

```

1  type   ::= typePrimary

3  typePrimary ::= typeId [ # ( innerType {, innerType } ) ]
   | bit [typeNat : typeNat]
5   | int

7  innerType ::= typePrimary
   | typeNat
9   | typeVar

11 typeIde ::= Identifier
   typeVar ::= identifier
13 typeNat ::= decDigits

```

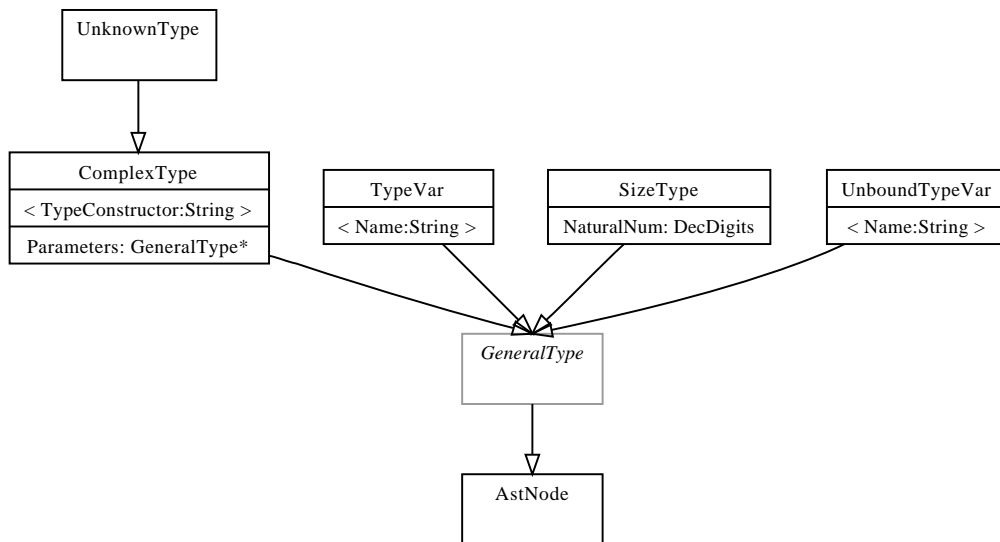


Figure 4.7: Types

In BSV it is possible to use type variables to increase the code reusability. E.g. a type can be expressed as `List#(a)`, which is a list of variables of type `a`. A type synonym definition, which is presented in the following section, is an example how type variables are used.

4.1.7 Type synonym definitions

A convenient method to define own types is the type synonym language construct. The corresponding grammar is presented in listing 4.6 and the AST in figure 4.8.

Type synonyms allow definitions such as `typedef bit [7:0] Byte;` or `typedef Tuple3#(a, a, b) Triple#(type a, type b);`.

The first part after `typedef` is a type and the next part is the new type name, optionally followed by a list of type formals.

Listing 4.6: Type synonym grammar

```

TypeDefSynonym typeDefSynonym =
2   TYPEDEF complexType identifierUC typeFormals SEMICOLON
   {: return new TypeDefSynonym(identifierUC.getName(),
4   typeFormals, complexType); :};

6 List typeFormals = ...
...

```

```

8 TypeFormal typeFormal =
  TYPE identifierLC {: return new TypeFormal(identifierLC.getName()); :}
10 | NUMERIC TYPE identifierLC {: return new TypeFormal(identifierLC.getName()); :};

```

The class `TypeDefSynonym` has a `ComplexType` and a list of `TypeFormal` children. Each type variable used inside `ComplexType` has to be specified by a type formal form this list. Otherwise the BSV compiler will return with an error.

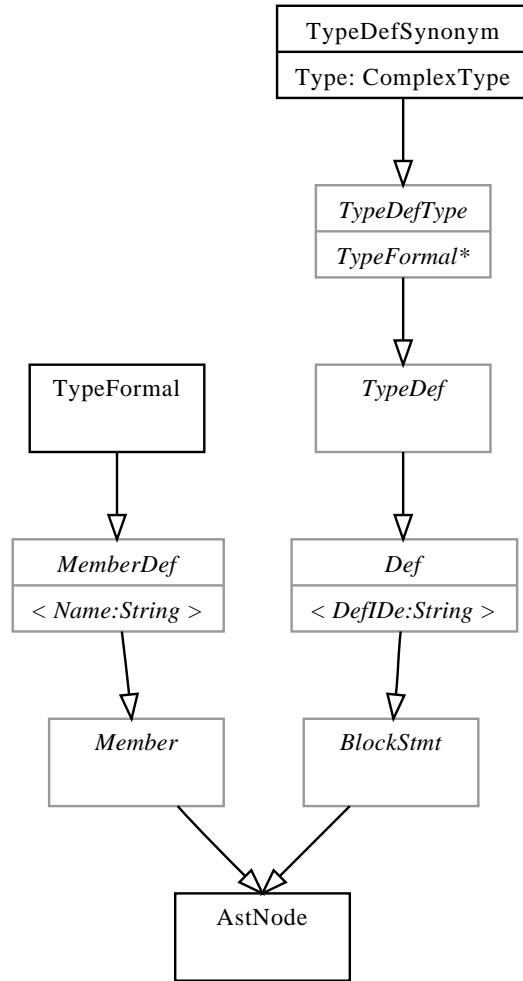


Figure 4.8: Type synonym definitions

This kind of errors have to be checked before time consuming compiler runs are started. The attributes and rewrite rule presented in section 4.2.3 cover this task.

4.2 Attribute definitions and rewriting rules

Possible uses of attributes and rewrites are nearly endless. To give an impression how they can be applied, the following sub-section present examples implemented in the BSV parser and BlueSVEP files.

4.2.1 Collection of imports and exports

For further type and name analysis it is useful to have access to imports and exports as lists. Corresponding to figure 4.5, e.g. imports can be distributed into several lists. One list for each line and one list containing all 'single' line lists. To prevent to go through this structure every time, the import and export declarations should be stored in a new list.

A convenient feature of JastAdd is the definitions of 'self-filling' collections. Listing 4.7 presents therefore the attribute definitions for imports.

Listing 4.7: Collection of imports

```

1 inh PackageDef ImportDecl.currentPackage();
2
3 eq PackageDef.getBody().currentPackage() = this;
4
5 syn lazy HashSet PackageDef.getAllImports() = allImports();
6 coll HashSet PackageDef.allImports() [new HashSet()] with add;
  ImportDecl contributes this to PackageDef.allImports() for currentPackage();

```

The inherited attribute in line 1 and its defining equation in line 3 is used to have a reference to the surrounding package.

The JastAdd syntax word `lazy` in line 5 sets the synthesized attribute `getAllImports()` to be calculated once when it is accessed the first time. The result is then cached for following accesses. Its definition returns the collection from line 6. This collection attribute is defined starting with `coll`, followed by the type, the owning class and a name. The value between `[]` is the initialization data and the name after `with` defines the method, that is used to add items to the collection.

In line 7 finally, the objects of class `ImportDecl` contribute themselves to the `HashSet` from line 6 for the package reference defined in line 3.

4.2.2 Replacing missing package names

The package name is normally given in the surrounding 'package-endpackage' construct. Nevertheless the BSV compiler also accepts files with missing package definitions, that start directly with imports and exports. This would result in packages nodes titled with `$NO_PACKAGE_NAME` in the eclipse outline view.

To set the package name, a rewrite rule is used. In the condition of this rule, a simple equality test identifies those packages and resets the name. As no new object has to be created, the updated object is returned. The rewrite rule is shown in listing 4.8.

Listing 4.8: Rewrite PackageDef

```

rewrite PackageDef {
2  when (getDefIDe().equals(BSVParser.Constants.NO_PACKAGE_NAME))
    to PackageDef{
4      setDefIDe(getProgram().getSrcPath().removeFileExtension().lastSegment());
      return this;
6  }
}

```

The rewriting is triggered once, when the `PackageDef` object is accessed through a `get()` method the first time and a second time after the modified object was returned. The second rewrite attempt fails, as the name changed to the last segment of the source path. Then the object is returned to the caller of the `get()` method.

4.2.3 Unbound type variable check

Some BSV constructs allow the definition of type variables (e.g. interface declarations, structs, tagged unions and type synonyms. The latter is used as an example to present the unbound type variable check). Listing 4.9 presents the required attribute definitions.

Listing 4.9: Attributes for unbound type var check

```

inh boolean TypeVar.checkUnbound(TypeVar typeVar);
2
eq TypeDefSynonym.getType().checkUnbound(TypeVar typeVar) =
4  findTypeFormal(typeVar.getName());
6
syn boolean TypeDefType.findTypeFormal(String name) {
  for(int i = 0; i < getNumTypeFormal(); i++){

```

```

8   if(getTypeFormal(i).getName().equals(name)){
      return true;
10  }
      }
12  return false;
    }

```

The inherited attribute in line 1 have to be defined by every possible parent of `TypeVar`. This has to be done even for classes that will never have a child of that concrete type, according to their productions rules. For example, every subclass of `MemberDefType` (see fig. 4.3) or their parent nodes have to define this inherited attribute as `MemberDefType` has a `GeneralType` child. For nodes where a check for unbound type variable makes no sense, the equation returns `true`.

The class `TypeDefSynonym` has a `ComplexType` child, that can contain type variables (see fig. 4.7). The equation in line 3 sets the attribute to the return value from the synthesized attribute `findTypeFormal(String)`.

This attribute is defined in line 6 for `TypeDefType`. This is a superclass of `TypeDefSynonym` (see fig. 4.8). If the argument is found in the type formals, it returns `true`, otherwise `false`.

The `checkUnbound(TypeVar)` attribute is used in the following rule (listing 4.10). It rewrites a `TypeVar` to an `UnboundTypeVar` if the attribute value equals false.

Listing 4.10: Rewrite `TypeVar`

```

rewrite TypeVar {
2  when (checkUnbound(this) == false)
      to UnboundTypeVar{
4      UnboundTypeVar newNode = new UnboundTypeVar(getName());
      newNode.setLocation(this);
6      return newNode;
      }
8 }

```

An error has to be generated, after this rewrite were successfully applied. The next section presents how the error generation and collection works and answers why line 5 is necessary.

4.3 Error collection

As described in the introduction to this chapter, the errors derive from different parsing steps. Each step produces specific kind of errors, which are linked with the causing symbol:

1. The scanner tries to insert symbol tokens for the parser, which can produce **lexical** errors.
2. The parser builds a CST of the code, which can contain **syntactic** errors.
3. The type, name and other code analysis possibly leads to **semantic** errors.

Calling the parse method from the BSV parser adds lexical and syntactic errors to the returned `Program` object. The collection of semantic errors has to be started by calling the method `collectErrors()` on this object. Listing 4.11 presents the implementation.

Listing 4.11: Collect errors method from `ErrorCheck.jadd`

```

public void ASTNode.collectErrors() {
2  typeAnalysis();
   for(int i = 0; i < getNumChild(); i++) {
4   getChild(i).collectErrors();
   }
6 }

```

The `getChild(i)` method call in line 4 triggers all rewrites before `collectErrors()` is called on that object. According to listing 4.12 the type analysis method of class `UnboundTypeVar` calls the error method. This method uses the objects location within the source file to generate an error with all necessary information to place an error marker.

Listing 4.12: Type analysis method from `TypeAnalysis.jadd`

```

public void ASTNode.typeAnalysis() {}
2
public void UnboundTypeVar.typeAnalysis(){
4  error("Unbound type variable: " + getName());
}

```

This is the reason for the method call `setLocation(this)` in line 5 in listing 4.10. Otherwise this information would be lost and the error marker could not be placed at the right position.

5 BlueSVEP - A BSV Eclipse plugin

This chapter introduces the relevant sources from the JastAdd core plugin and how they are used in the **BlueSpec SystemVerilog Eclipse Plugin** (BlueSVEP). The resulting inherited features from the core plugin and the additional features implemented in BlueSVEP are shown in the next section. The chapter concludes with short installation instructions for BlueSVEP.

During the implementation process, the book “Eclipse - Building Commercial-Quality Plug-ins” [13] is used as reference as well as many Eclipse corner articles from [11] and the Eclipse Wiki [14].

5.1 Existing sources - The JastAdd core plugin

The core plugin provides basic implementations for several Eclipse features. Adding implementations of abstract methods or attribute equation definitions makes these features available. The JastAdd Java plugin, which is also based on the core plugin, is used as a reference project. Parts of the core plugin are presented in the following sub sections.

5.1.1 Features provided by interfaces and abstract classes

Besides inheriting from certain JastAdd core plugin classes and implementing abstract methods, the `plugin.xml` file has to reflect the features provided by the plugin with the corresponding extension points.

Model provider

BlueSVEP uses the `org.jastadd.plugin.model.JastAddModelProvider` extension point from the core plugin, which requires a class to provide and build the model for a BSV source file. This class, named `BlueSVEPModel`, has to be specified in the `plugin.xml` file and inherits from the abstract `JastAddModel` class.

The `BlueSVEPModel` class updates the model, provides a file reference for each CST node, and returns the CST node corresponding to a certain position in the editor.

Updating the model means to parse the BSV source file and build the CST. This CST is then stored for each file in an array to save unnecessary recomputations, if a model update is triggered and the content is unchanged. Parsing errors are tagged on the files with markers, which are defined at the `org.eclipse.core.resources.markers` extension point.

A file reference is necessary, for example, when the user selects a node in the outline view and the editor jumps to the specific position in the text.

To provide hover information in the text editor, the position in the file has to be linked with a certain node. This is done by the core plugin and requires only little adaption in the `BlueSVEPModel` class.

Editor

The existing abstract `JastAddEditor` class provides already most of the functionality. The `BlueSVEPEditor` class provides therefore only the editor id, which is also used in the `plugin.xml` at the `org.eclipse.ui.editors` extension point. An additional class, named `BlueSVEPEditorConfiguration`, enables access to the `BlueSVEPScanner` class, which handles the syntax word highlighting.

Builder and nature

For this part, the Java Java plugin cannot be used as a reference. It applies the `JastAddBuilder` class from the core plugin, which complicates a direct control over the build process.

Therefore the class `BlueSVEPBuilder` is implemented for a better integration of the BSV compiler. The builder has a build method, which is called every time a full or incremental build was triggered. By default the BSV compiler starts only when no parse errors appear. The `BlueSVEP` builder requires also a nature, which connects it to a project.

The builder is defined at the `org.eclipse.core.resources.builders` extension point and the nature at `org.eclipse.core.resources.natures`.

Views and perspective

The core plugin provides already a extensively implemented basis for the explorer and the navigator view. The explorer requires a bit more adjustment as it reflects the source code errors on the corresponding CST nodes. Both views are defined at the

`org.eclipse.ui.views` extension point. The explorer is the standard view in the BlueSVEP perspective, which takes the most configuration from the `JastAddPerspectiveFactory` super-class. The BlueSVEP perspective adds only a new wizard shortcut for BlueSVEP projects and it is defined at the `org.eclipse.ui.perspectives` extension point.

5.1.2 Features provided by attributes

Some features depend on AST node attributes and are defined in `.jrag` files. The presented features adjust the behavior in the editor and the content outline view.

Folding in the editor

Folding markers can only be placed at specified nodes and they are set in the `BlueSVEPFolding` file. If a node should have a fold marker the following equation, in this case for `PackageDef`, has to return true.

```
eq PackageDef.hasFolding() = true;
```

To achieve the desired folding markers placement it is sometimes necessary to change the grammar a bit. For example, `List` objects have to be encapsulated into another node to provide the proper folding of all list items as shown in the following code line for `Exports`.

```
Exports exports = exports_list_opt {: return new Exports(exports_list_opt); :};
```

It is also important not to place a foldable AST node as a first node in a rule, if the rule is for a foldable AST node itself. This leads to misplaced folding markers. E.g. the definition for a `moduleDef` has to be separated in the following rules (listing 5.1). The `hasFolding()` equations for `entityAttributes` and `moduleDef` both return true.

Listing 5.1: Grammar for `ModuleDef`

```
ModuleDefWithAttributes attrModuleDef =
2  entityAttributes moduleDef
   {: return new ModuleDefWithAttributes(entityAttributes, moduleDef); :};
4
ModuleDef moduleDef =
6  MODULE identifierLC ...
```

If the `moduleDef` rule would also include `entityAttributes`, the folding marker for `moduleDef` would be placed at the same position as the folding marker for `entityAttributes`.

Hover comments in the editor

Hover comments are used in the standard Eclipse Java editor to display Javadoc for the element selected. In BlueSVEP, the comment text is specified in the `BlueSVEPHoverComments` file. It only includes a general comment style.

Content outline view

Eclipse provides a content outline view, which presents the CST for Java classes. The core plugin already provides the necessary base structure to populate the content outline view with a CST for the used model. Three equations have to be defined for each node, which should appear in the view. These equations are defined in the `BlueSVEPContentOutline` file. Listing 5.2 presents the three equations for `ModuleDef`. The first equation in line 1 sets the visibility in the view. The default value is `false`. The two next equations set the display name and image.

Listing 5.2: Equations for content outline view elements

```
... eq ModuleDef.showInContentOutline() = true;
2   eq ModuleDef.contentOutlineLabel() = getDefIDe();
   eq ModuleDef.contentOutlineImage() =
4   Activator.getImage("icons/outline/Module.gif"); ...
```

The core plugin implementation iterates through all children and displays only visible nodes. It displays thereby also visible child nodes of a non-visible child.

5.2 Features

In the following, editor and workbench features are described and presented with screen shots.

5.2.1 Editor features

These features add new functionality to the editor.

Syntax highlighting

Syntax highlighting significantly improves the code readability. Additionally it is an easy method to check for misspelled syntax words, which prevents the programmer from

finding these errors one after another with several compiler runs. Besides the BSV syntax words, some predefined types, such as `Bit` or `Action`, are also highlighted. Figure 5.1 presents some example code lines.

```

(* always_ready *)
interface OPB_Slave_if;
  // outputs
  (* result = "Sln_DBus" *) method Word sln_DBus();
  (* result = "Sln_errAck" *) method Bool sln_errAck();
  (* result = "Sln_retry" *) method Bool sln_retry();
  (* result = "Sln_toutSup" *) method Bool sln_toutSup();
  (* result = "Sln_xferAck" *) method Bool sln_xferAck();

  // inputs
  (* always_enabled *)
  (* prefix = "OPB" *)
  method Action opb(
    (* port = "DBus" *) Word dBus,
    (* port = "ABus" *) Word aBus,
    (* port = "BE" *) Bit#(4) be,
    (* port = "RNW" *) Bool rnw,
    (* port = "select" *) Bool sel,
    (* port = "seqAddr" *) Bool seqAddr );
endinterface

```

Figure 5.1: Syntax highlighting for syntax words, strings, comments and normal text.

The highlighting colors can be set via the BlueSVEP preferences. The corresponding preference page is shown in figure 5.2. The changed colors will be displayed after the editor was reopened.

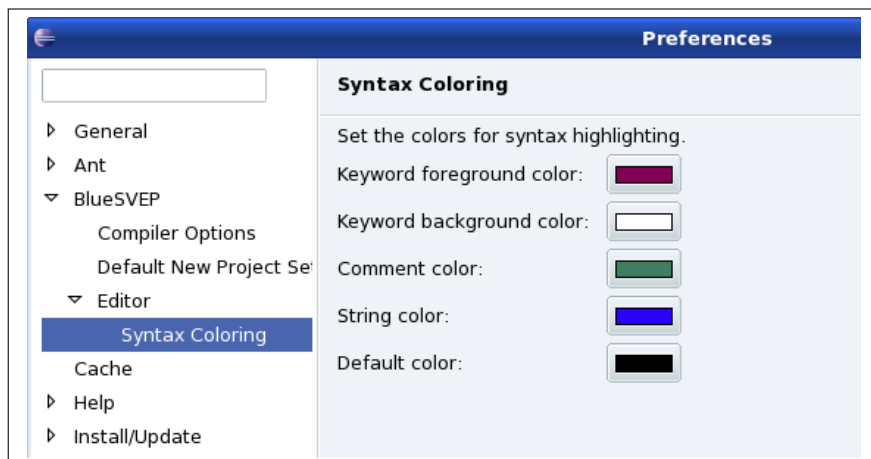


Figure 5.2: Preferences for syntax highlighting

Folding

A foldable element is marked with a minus sign to fold it and with a plus sign to unfold it. Figure 5.3 presents some folded elements.

```

184
185*interface OPB_SBridge;[]
188
189 // just connect a master to a slave
190 (* always_ready *)
191*module mkDummyOPB#(OPB_Master_if m,
211
212*module mkDummy_MultipleSlavesOPB#(C
256
257*typedef struct {[]
261

```

Figure 5.3: Folding markers

Folding is enabled for the following nodes:

- Package, module, method, and function definitions
- Interface declarations
- Multiple export and import lines
- Multiple attribute lines
- Type definitions, structs, tagged unions and enumerations
- Begin-end, action, action value blocks
- Rules

Hover comments

Presenting further information about the element under the cursor can be very helpful. The BlueSVEP plugin displays the start/end line and column of the node, followed by the its type, which is presented in figure 5.4. The yellow hover comment states the node type as `InterfaceDecl`.

```

10
11 interface SystemMemory;
12     method Byte readByte(WordAddress wa);
13     method Word readWord(WordAddress wa);
14     method Action writeWord(WordAddress wa, Word w);
15 endinterface

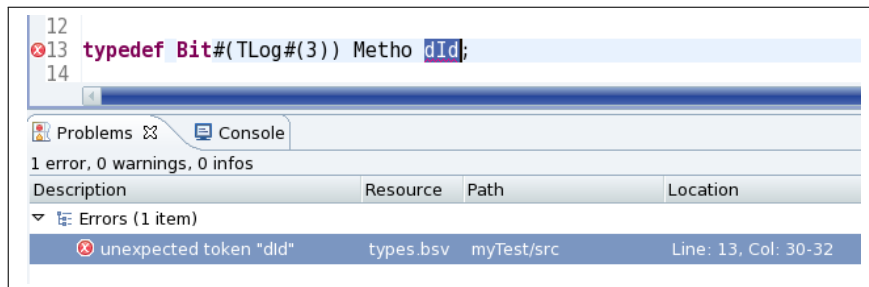
```

Figure 5.4: Hover comment for an `InterfaceDecl` object

It is also possible to display other information. For example the `doc` attribute described in chapter 13.6 in [2] could be used for this purpose.

Source code errors

While typing source code, the programmer is nearly immediately informed about errors through the parser feedback. After the code changed, the parsing is started again. Figure 5.5 presents an error resulting from a misplaced white-space character.



```

12
13 typedef Bit#(TLog#(3)) Metho dId;
14

```

Description	Resource	Path	Location
unexpected token "dId"	types.bsv	myTest/src	Line: 13, Col: 30-32

Figure 5.5: Syntactic error from parsing.

The parser will treat every unrecognized token as an error. In the case that the parser is not able to recover from an error, the parsing will stop at that line until the programmer corrects it. To improve the parser error recovery, the file `errorProductions.parser` adds some production rules, which are included in the generated parser. The rule for `ImportLine` is exemplified presented in the following:

```
ImportLine importLine = IMPORT error SEMICOLON {: return new ImportLine(); :};
```

The syntax word `error` is predefined in the Beaver specification and used as “fallback position”.

Even if the parser returns without an error, there could still be some semantic errors, which can only be found through code analysis. Such errors are presented in figure 5.6.

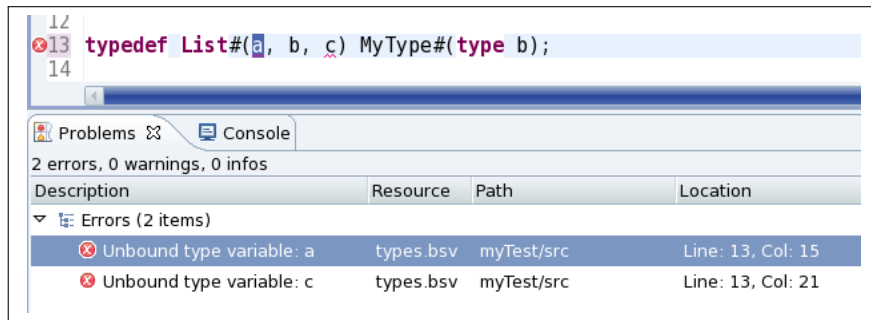


Figure 5.6: Semantic error from unbound type variable analysis.

The faulty code line has two errors, as the type variables `a` and `c` are not specified by the new type synonym. The correct line should end as follows:

```
MyType#(type a, type b, type c).
```

5.2.2 Workbench features

This section presents all features, which are not mainly focused on the editor.

Project management

BlueSVEP provides a BSV specific project type. When a new project wizard is started, it is possible to choose the project name and the source and output folder. The wizard page is shown in figure 5.7.

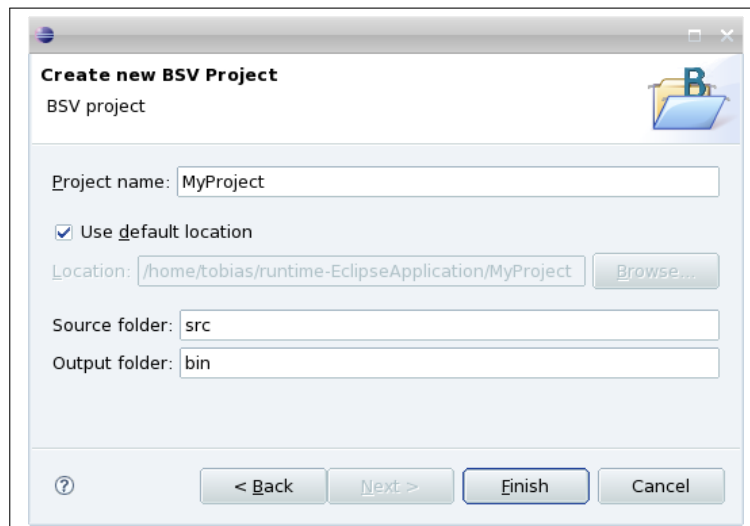


Figure 5.7: New Project wizard.

The source and output folder are changeable after the project creation. To do this, select the “Properties” item from the project’s context menu and choose the “Project Settings” tab (fig. 5.8).

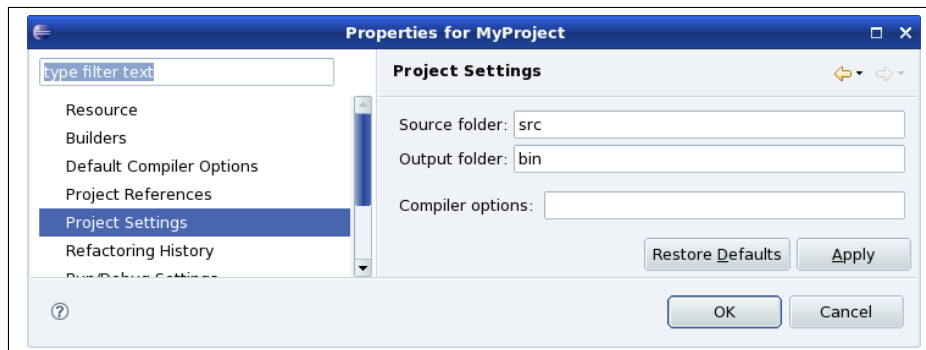


Figure 5.8: Project specific settings.

It is possible to change the wizard default source and output folder in the BlueSVEP preferences as well.

Build process

The Eclipse build process is triggered automatically or manually.

Automatic means that shortly after the programmer stopped typing, Eclipse runs the project’s builders. As a result, the `build` method from the `BlueSVEPBuilder` class is invoked, which starts the parsing and places the respective markers. In case there are no parser errors in any file, or if parser errors are ignored, the BSC compiler is started. If more than one file is changed, the BSC compiler is started as soon as all files are saved.

Manually triggered builds give the programmer more control over the time, when the BSC compiler should be started. Before parsing and compiling is started, the programmer is asked to save any unsaved resources.

In both cases, a main BSV file has to be set before the BSC compiler can be started. The context menu of BSV files provides the item “Set as main BSV file”. After a main file is selected, it is presented with a different icon (🔧) in the explorer view (see `ObjectAccess.bsv` in fig. 5.12).

To provide access to all BSC compiler features, it is possible to set further command line options through preference pages. These options can be set globally and/or project specific, whereas project specific options override global ones. Figure 5.9 presents the preference page for global compiler options.

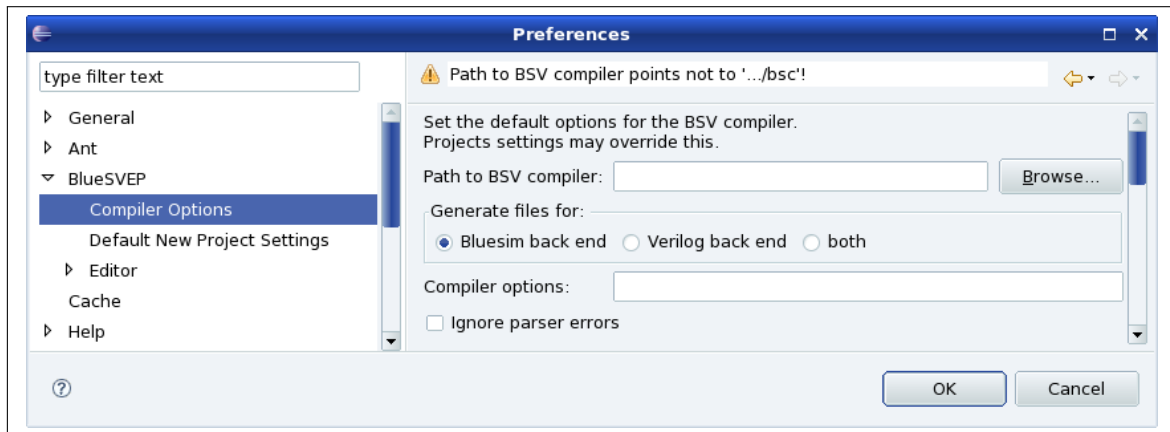


Figure 5.9: Global compiler settings.

In the case that the BSV compiler was successfully started, the feedback is displayed in a console view, which is presented in figure 5.10.

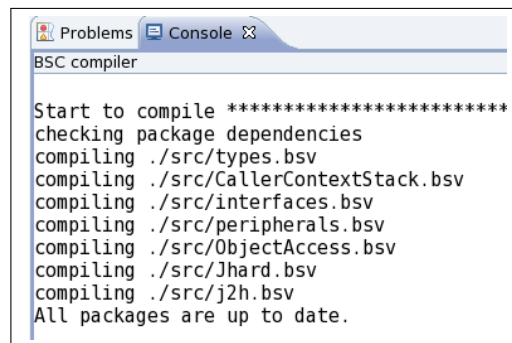


Figure 5.10: BSC compiler feedback.

Content outline view

Going through the source code is much more easier with an outline of syntax elements. Therefore BlueSVEP provides a content outline view for several types. An example is shown in figure 5.11.

The CST is the result from parsing and includes every node, which is set to be visible in the view. To navigate between these elements, it is possible to select an element of interest to reset the editor's focus to the respective line and column of that element. The supported types and the used icons are presented in table 5.1.

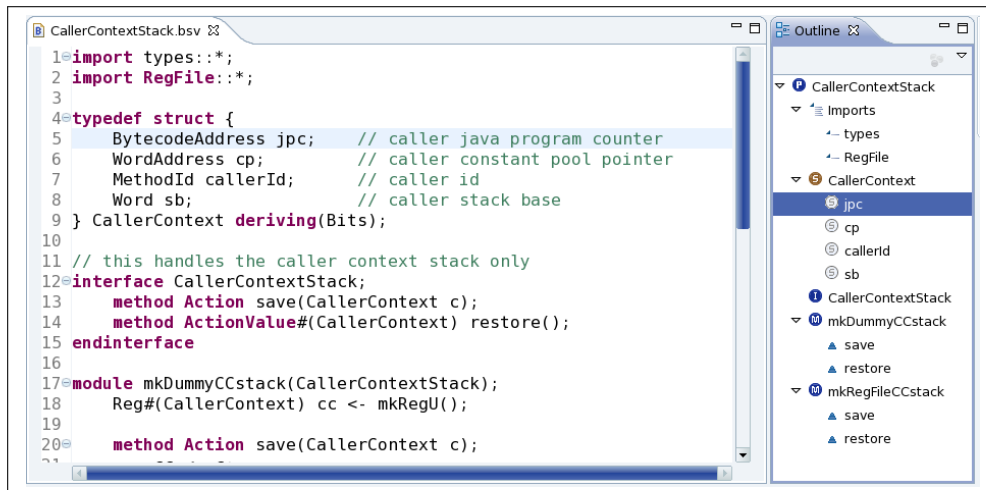


Figure 5.11: Content outline view.

Table 5.1: Outline types

Type	Icon	Type	Icon
Package		Module definition	
Exports		Export declaration	
Imports		Import declaration	
Interface declaration		Sub-interface declaration	
Method definition		Function definition	
Rule definition		Type synonym	
Enumeration		Enumeration element	
Struct		Struct member	
Tagged union		Tagged union member	

Explorer and navigator view

These two views provide a convenient access to BSV projects. The explorer view (fig. 5.12) tags error markers to source files and displays the main BSV file, while the navigator view presents the CST for each file (fig. 5.13). Both views have preset filters to hide the compiler output files.

Perspective

The BlueSVEP perspective places the explorer and other views in a predefined arrangement. The navigator view is not displayed by default and has to be enabled manually.

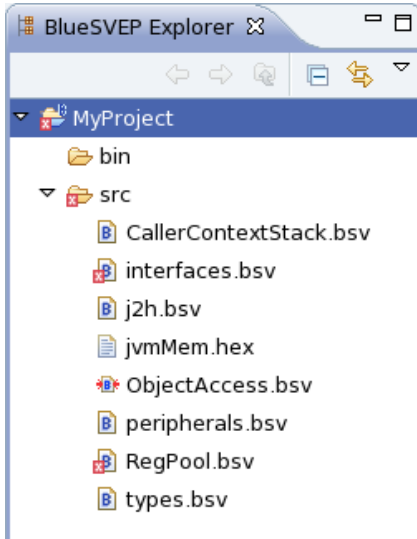


Figure 5.12: Explorer view.

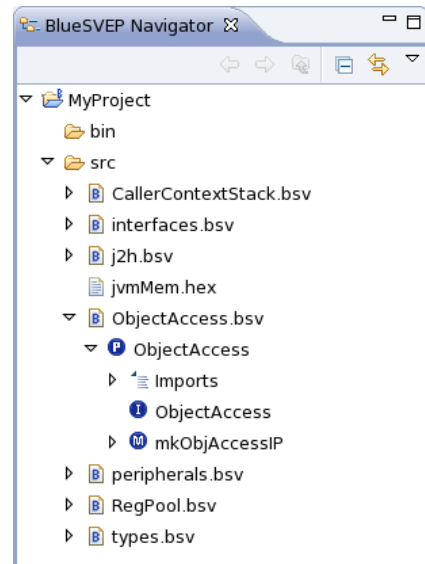


Figure 5.13: Navigator view.

5.3 Installation instructions

To install BlueSVEP more easily, a remote update site is available. In this way, it is possible to install and download the plugin directly in Eclipse. This requires to create a feature project and an update site project, which gather information about the comprising plugins and build them as `jar` files. To install BlueSVEP follow these steps:

1. Open in Eclipse the menu Help->Software Updates->Find and Install.
2. Select “Search for new features to install” and click on next.
3. Click on “New Remote Site”, choose a name (e.g. BlueSVEP update site) and set the URL to “<http://esd.cs.lth.se/sw/BlueSVEP/update/>”.
4. Include the created remote site and click on next.
5. Follow the installation routine further.

6 Conclusion and outlook

In this project, a Bluespec SystemVerilog Eclipse plugin was implemented. The plugin is capable of project management, syntax highlighting and linking of compiler errors with source code. Furthermore it provides a source file parser for error feedback while typing. The created BSV grammar for this parser is well changeable, in case of future language developments or further improvements of the parser.

One problem while building up the grammar was the slight differences between the grammar description and the BSV compiler acceptance. The grammar described in the reference manual is not always complete and correct as well. The language by itself also offers sometimes more than one way to write certain parts (e.g. short forms), which increases the required number of production rules.

For now, the parser is not capable of parsing the whole BSV grammar. For example, the state machine sub-language is not supported. Based on these missing parts, the parser cannot parse certain syntax constructs. The parser stops also in the case that too many errors occur in one line and if there is no suitable error production rule.

Using the JastAdd core plugin was an convenient solution, compared to handwritten code. The current lack of documentation was compensated by using the JastAdd Java plugin as a reference project. Rewrites and attribute equations were also a gain for the project. It requires indeed more effort while designing the AST, especially the inherited attribute equations, but the advantages prevail.

There are many possible ways to improve the usability and extend the plugin's feature list. The most important task is definitely to be able to parse the whole BSV grammar and to be as conform as possible to the BSV compiler. Workbench wide type analysis and further error checks should spare further compiler time. The error tolerance and the parser stability has to be improved as well to provide a good working plugin.

Another important feature is the integration of simulation data generation and presentation. The BSC compiler is able to generate Verilog output and simulation data for Blueview. This is already done partially through the project preference pages and

through setting a project's top module. To display the data it would be convenient to be able to start the tools directly from Eclipse.

The present project management could be extended by an additional BSV file wizard. This wizard could already contain a code template for a basic package, which is completed by the wizard.

The explorer view can be extended in several ways. E.g. it could display the model elements so that an error is directly tagged to an element. The output folder could be hidden, as it is not important for developing and the presentation of the source folder could use own icon.

The JastAdd core plugin provides further useful features. Source code auto-completion, for example, can ease programming and speed up development. Visualizing package and module dependencies or searching for their uses provide a better overview. Changing existing source code with automated refactoring is much more convenient and less error-prone.

List of Figures

1.1	Project build-up	7
2.1	The AST for the BSV sub-language.	9
2.2	concrete syntax tree for the example program in listing 2.2	10
2.3	CST produced by a RAG.	10
3.1	Output without options.	17
3.2	Output with -oI.	19
3.3	Output with -oC.	20
3.4	Output with -iC.	20
3.5	Output with -iCI.	21
4.1	Parser goal: <code>Program</code>	22
4.2	Common nodes, 1st part	23
4.3	Common nodes, 2nd part	24
4.4	Package definition, 1st part	25
4.5	Package definition, 2nd part	26
4.6	Conditional statements	27
4.7	Types	29
4.8	Type synonym definitions	30
5.1	Syntax highlighting for syntax words, strings, comments and normal text.	39
5.2	Preferences for syntax highlighting	39
5.3	Folding markers	40
5.4	Hover comment for an <code>InterfaceDecl</code> object	41
5.5	Syntactic error from parsing.	41
5.6	Semantic error from unbound type variable analysis.	42
5.7	New Project wizard.	42
5.8	Project specific settings.	43
5.9	Global compiler settings.	44

5.10 BSC compiler feedback.	44
5.11 Content outline view.	45
5.12 Explorer view.	46
5.13 Navigator view.	46

Listings

2.1	Grammar for a subset of the BSV language	8
2.2	Example program based on the sub-language	9
2.3	Rewrite rule with condition	11
3.1	Section from the JFlex input file	12
3.2	Section from the Beaver input file	13
3.3	Section from the AST definition file	14
3.4	Example AST definition file	17
4.1	Grammar for <code>AttributeLine</code>	25
4.2	Grammar for package statements	25
4.3	Grammar for <code>IfStmt</code>	26
4.4	Scanner rules for system tasks.	28
4.5	Type grammar	28
4.6	Type synonym grammar	29
4.7	Collection of imports	31
4.8	Rewrite <code>PackageDef</code>	32
4.9	Attributes for unbound type var check	32
4.10	Rewrite <code>TypeVar</code>	33
4.11	Collect errors method from <code>ErrorCheck.jadd</code>	34
4.12	Type analysis method from <code>TypeAnalysis.jadd</code>	34
5.1	Grammar for <code>ModuleDef</code>	37
5.2	Equations for content outline view elements	38

Bibliography

- [1] Inc. Bluespec. URL: <http://www.bluespec.com>, 2008. 5
- [2] Bluespec, Inc. *Bluespec SystemVerilog Reference Guide*, revision: 31 august 2007 edition. URL: http://www.bluespec.com/wiki/index.php?title=BSV_Documentation. 6, 7, 20, 28, 41
- [3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1986. 8
- [4] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. 9
- [5] G. Hedin. Reference attribute grammars. *Informatica (Slovenia)*, 24, 2000. URL: http://www.cs.lth.se/home/Gorel_Hedin/publications/2000-RAGsInformatica-PreliminaryVersion.pdf. 10
- [6] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*. Springer-Verlag, 2004. URL: http://www.cs.lth.se/home/Gorel_Hedin/publications/2004-ReRAGs-LNCS.pdf. 11
- [7] T. Ekman and G. Hedin. URL: <http://jastadd.org>, 2008. 12, 14
- [8] JFlex. URL: <http://www.jflex.de>, 2008. 12
- [9] Beaver. URL: <http://beaver.sourceforge.net/index.html>, 2008. 13
- [10] Graphviz. URL: <http://www.graphviz.org>, 2008. 16, 18
- [11] Eclipse. URL: <http://www.eclipse.org>, 2008. 19, 35
- [12] log4j. URL: <http://logging.apache.org/log4j>, 2008. 21
- [13] D. Rubel E. Clayberg. *Eclipse - Building Commercial-Quality Plug-ins*. Addison Wesley, second edition, July 2006. 35
- [14] Eclipse Wiki. URL: <http://wiki.eclipse.org/>, 2008. 35