Extending gNOSIS for System Verilog HDL Static Analysis

Scott A. Carr, Richard Neil Pittman

Microsoft Research

September 2015

Technical Report

MSR-TR-2015-68

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Extending gNOSIS for System Verilog HDL Static Analysis

Scott A. Carr, Richard Neil Pittman

Microsoft Research

Abstract

Software engineering tools for Hardware Design Languages (HDL) lag behind traditional software development tools by decades. However as heterogeneous computing becomes more pervasive, productive programming in HDLs will become vital. To this end, we have developed gNOSIS a static analysis platform for Verilog HDL. In this project we have extended gNOSIS to support System Verilog. A good analogy is C is to C++ as Verilog is to System Verilog, that is System Verilog is a superset of Verilog with more sophisticated features. This report details the challenges, approach, and progress we've made towards supporting System Verilog in gNOSIS.

1 Introduction

Using existing tools debugging a hardware design is much more time consuming than debugging software. Partly this is due to the long compilation time inherent in hardware design synthesis. Changing the set of traced signals requires recompilation which may take hours or minutes. To bring HDL programming productivity up to par with traditional software programming productivity, we need new tools to identify bugs without compiling all the way to hardware.

HDL tools will become increasing valuable as heterogeneous computing becomes more pervasive. Computing on devices other than traditional CPU can accelerate performance, increase security, and reduce power consumption. There is a trend now towards putting GPUs and FPGAs in data centers to tap into these benefits. The challenge is to allow HDL developers to quickly design, test, and debug on heterogeneous platforms.

2 Background

In both academia and industry the creation of HDL development tools is far behind software development tools.

2.1 State of the Art in Industry

Much research and development in industry has gone into improving programming productivity in traditional software development. Advances include automated testing, automatic test generation, test coverage analysis, symbolic execution, stronger type systems, time traveling debuggers, and many others. However, HDL programmers do not have these tools at their disposal. In industry tools are tied to a particular vendor and are very expensive compared to software development tools. One approach industry has taken is developing new languages to replace existing HDLs. Usually these language are higher level and synthesize to an HDL. Examples include Bluespec [5] and OpenCL [6].

2.2 State of the Art in Academia

Compared to software languages, the research community has largely ignore HDLs.

Bounded model checking has been applied to System Verilog assertions [3]. However, the technique only covers a subset of the language. Reducing the size of debug traces is another area of past work [4].

In contrast, academia has been intensely focused on improving software developer productivity. Much research has gone into high productivity languages, software engineering tools, and even empirically studying software teams. However, benefits of this research has not crossed the gap from software development to hardware development.

2.2 gNOSIS Project

The gNOSIS project was created to provide state-of-the-art development tools to HDL programmers. Previous gNOSIS projects have improved automatic board-level debugging of FPGA designs [1] and assertion mining for verification [2]. The overall goal is to shorten the development and testing time of hardware designs.

2.3 Glossary

Preprocessor: a program that expands *define*, i*nclude*, and other preprocessor directives. The gNOSIS preprocessor is not part of this Technical Report.

Lexer: the part of the parser that turns strings of characters in to a discrete set of token identifiers.

Grammar: a set of rules that define the language accepted by the parser. These rules are comprised of combinations of tokens.

Actions: C# code that executes when a grammar rule is matched. In the case of gNOSIS, the actions build the AST

Abstract Syntax Tree (AST): a structured representation of the parsed System Verilog code.

Analysis: code that traverses the AST and extracts information from it. For example, type checking is an analysis.

2.3 ANTLR

The gNOSIS project uses the ANTLR [7] parser generator for its parser. The input to ANTLR is one or more grammar files and it generates several files. A simplified example of the ANTLR process is shown in Figure 1.

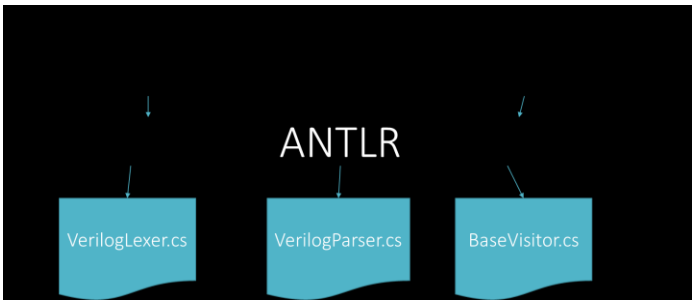


Figure 1 - ANTLR Inputs and Outputs

To build the gNOSIS System Verilog parser, we invoke ANTLR on our main grammar file (Verilog.g4) and it produces seven files:

VerilogLexer.cs and **VerilogLexer.tokens**

VerilogLexer is a class that transforms the string of input characters into tokens. The .tokens file is a text file with all the token mappings.

VerilogParser.cs

VerilogParser is the class where the parsing logic is implemented. It matches the tokens produced by the lexer to grammar rules.

VerilogBaseListener.cs and **VerilogListerner.cs**

These files are not used.

VerilogVisitor.cs

This file declares the interface *VerilogBaseVisitor* implements.

VerilogBaseVisitor.cs

Our actions (ParserActions.cs) inherit the base class defined in this file, *VerilogBaseVisitor*. For each grammar rule, ANTLR generates a *VisitRule_name* method. The base visitor will visit all the child nodes of a given rule automatically but if we want to take some action when we visit a particular rule (ex: build an AST node) we override that visit method.

Once ANTLR has generated the parser files, an execution of our tool looks like Figure 2. The source file is given to the lexer as a string. The string is divided up into tokens and given to the parser. The parser matches the rules in the grammar and when a rule is matched invokes the asssociated visitor method. The base class for these methods is generated by ANTLR but we much override these methods to add our own AST functionality. The we invoke our analysis on the AST and give the information we have discovered about the code to the user. In the diagram, blue boxes are autogenerated code and green boxes are code we wrote.

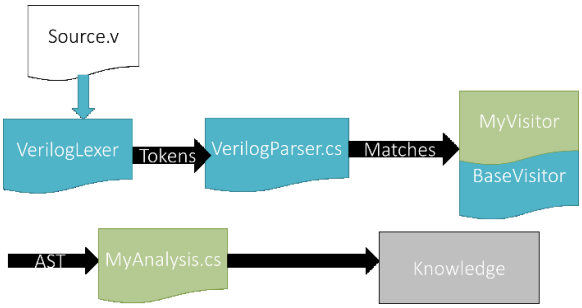
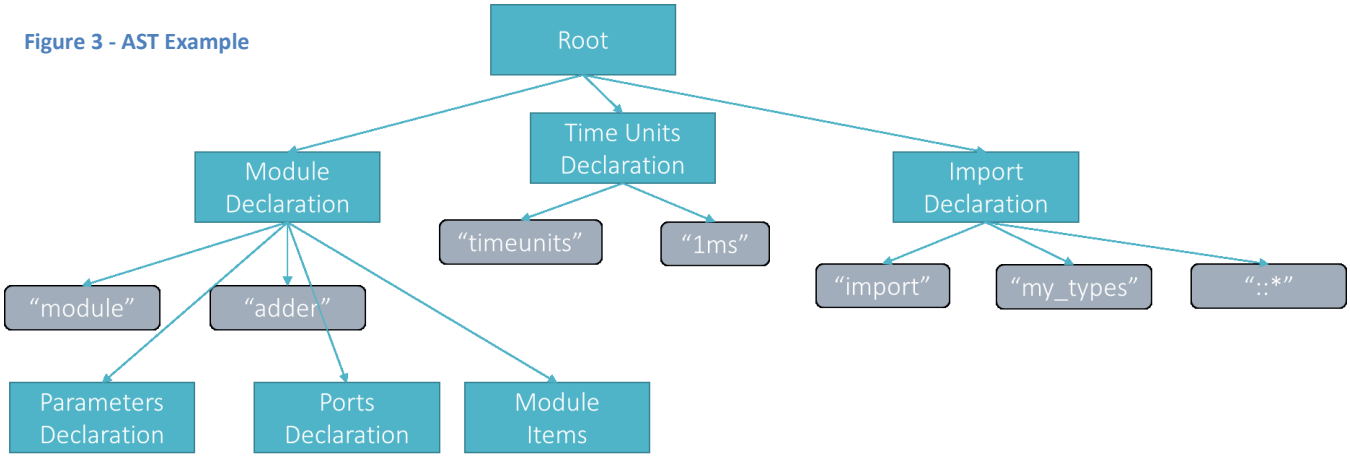


Figure 2 - Tool Flow Diagram

2.4 Project Structure

The gNOSIS project currently consists of three main pieces: the parser, the AST, and the analysis. The parser consists of the grammar files in *CombinedAntlr* project and the generated files described in **Section 2.3 ANTLR**. The AST consists of the astlib_*.cs files in the *oracle_avlib* project and the actions in ParserActions.cs in the same project. The analysis consists of several source files in the oracle_avlib including symbol_table.cs, atlib_clock.cs, and astlib_axiom.cs. An example AST is shown in Figure 3. The root of the AST is a list of nodes. There is a node class for most rules. Each node class has fields to hold the

Figure 3 - AST Example



parsed text matched by the grammar rules. In most cases the fields of the class nodes are strings containing in the input source text.

The *ParserTester* project is a basic test harness for testing the grammar and actions. ParserTester.cs is a batch test harness for running the parser/actions over many files. RulesTester.cs is for testing the parser/actions on a single file or even partial files.

The *FormatConverter* project converts a grammar in the format of the IEEE System Verilog specification into the format of an ANTLR grammar. We had to perform some manual steps to prepare the specification grammar for *FormatConverter*. The specification uses red font to denote literals so these had to be quoted manually. ANTLR uses the traditional regular expression notation of '+' for repeats one or more times and '*' for repeats zero or more times but the specification uses square and curly braces respectively to represent this. Running *Formatconvert* again is probably unnecessary as the complete converted grammar is in the *CombinedAntlr* project. The System Verilog grammar (from the specification) is implement in the .g4 files that begin with System_Verilog_*. Note that these are not currently used since we decided to build up on the existing Verilog grammar, but they can be referenced when expanding the current grammar to support more System Verilog features.

2.5 Build Structure

The project *CombinedAntlr* is the bottom of the project hierarchy. This project contains the grammar files. When any grammar file changes Visual Studio will start a new build of the *CombinedAntlr* project. However, Visual Studio does not know how to compile grammar files, so the *CombinedAntlr* project has a custom build step. The step simply invokes ParserBuilder.exe. *ParserBuilder* is command line C# project in oracle.sln. *ParserBuilder* is a standalone executable that invokes ANTLR on the grammar files in *CombineAntlr* and puts the generated files in *oracle_avlib*. *ParserBuilder* was originally designed to build multiple grammars but this behavior was deprecated. It checks to see if the grammar files have a newer "last write" timestamp than the generated files and if so, invokes ANTLR to generate new parser code.

The ANTLR generated code is part of the *oracle_avlib* project so as far as Visual Studio can tell, this project builds like a normal C# project.

3 Differences between Verilog and System Verilog

The most significant difference between Verilog and System Verilog is user defined types. To convert a Verilog grammar to System Verilog, a necessary step is: everywhere a type is used, the set of built-in types in Verilog needs to be replaced with any valid System Verilog type.

Other notable differences include: increment and decrement operators, return statements, arbitrarily dimensioned variables, packages, imports, exports, more assignment operators, and more built in types.

Because we took the approach of incrementally adding functionality to the existing Verilog grammar all the new System Verilog rules appear at the bottom of Verilog.g4 in the *CombinedAntlr* project.

4 Refactoring work

The existing Verilog grammar had lexer tokens, grammar rules, and actions interleaved in a single file. ANTLR supports this by enclosing the C# code for the actions in curly braces. We

decided to refactor the grammar into three separate parts (lexer, grammar, and actions).

The first reason why we did this was we found that including the string literals for the lexer tokens was very error prone. As an example, consider the following rule:

```
Case_statement : ‘case’ ‘(‘ expr ‘)’ case_body ;
```

This grammar rule implicitly defines three lexer tokens, namely the case keyword, open parenthesis, and close parenthesis. However if we make a small typo in the rule like:

```
Case_statement : ‘case’ ‘( ‘ expr ‘)’ case_body ;
```

Now we have introduced a new token (open parenthesis followed by a space). Combined with another lexer rule that ignores whitespace, we now have a parser where an open parenthesis followed by a space can only ever appear in a case statement. We made just this mistake in the course of the project and this bug was very difficult to hunt down.

To mitigate this problem we (by convention) forbid the definition of new tokens in grammar rules. This way ANTLR will return an error if we make a typo in a grammar rules (using the name of the token rather than the token literal).

Separating the grammar rules from the actions also has several benefits. First, having the actions in line with the rules adds visual/mental clutter. Mixing the two made comparing System Verilog rules to Verilog rules even more difficult. Determining if two rules are the same means comparing all elements of the rule and then recursively descending into the elements. Merging rules while also maintaining the rule actions is too many things to keep in mind at once. Second, separating rules and actions allows them to be tested and developed separately.

In fact, the biggest benefit to separating the lexer, rules, and actions is now they can be tested, debugged, and developed independently (or at least hierarchically).

5 AST and Extensions

The AST is a structured representation of the source code built by the parser actions. At a high level, there is an AST node type for most grammar rules and the job of the actions is to build up this tree. The fields of the nodes are typically strings containing the parsed source code.

It was one of our goals to remain as backwards compatible as possible, so whenever a construct in System Verilog easily mapped to a Verilog construct we reused the existing Verilog node. However there are some features of System Verilog that have no equivalent in Verilog. In particular new AST nodes were added for imports, packages, and user defined types. In System Verilog the program can define arbitrarily many types whereas there are a finite number of types in Verilog. The AST nodes that can be produced by System Verilog rules have a pointer to a type node. The original Verilog nodes had an enumeration for the type but it is not possible to enumerate all the types in System Verilog.

6 Challenges

Translating the IEEE System Verilog Specification’s grammar into the format ANTLR accepts posed some challenges. The IEEE grammar is written as a description of the language, not as a parser generator input. More specifically, the specification grammar is left recursive which is problematic for recursive

decent parser generators such as ANTLR. A left recursive grammar is essentially one where a grammar rule can match itself in an infinite loop, or it can match itself through a series of intermediate rules. The specification grammar is textually ambiguous. It is only made unambiguous by differentiating text in different color fonts as either being literals or symbols. For example, curly braces in red font denote literal curly braces (meaning the curly braces appear in the actual source code that is part of the rule) whereas curly braces in black font denote repetition of a rule zero or more times. We also observed an instance of a rule containing an apparent typographical error.

Similar to the left recursion problem, the specification grammar has rules that can match an empty string. This is a problem for a generated parser because a rule that can match an empty string can be matched infinitely many times. To address this issue, the rule must be expanded to make it match at least character. This problem pops up in situations like:

```
function_declaration :
function_type function_name
‘(‘ list_of_arguments ‘)’
‘endfunction’ ;

list_of_arugments: argument*
```

In the above example, *list_of_arguments* can match an empty string (it can match argument zero times). The fix is simple. We refactor the example to be:

```
function_declaration :
function_type function_name
list_of_arguments ‘endfunction’ ;

list_of_arugments : ‘(‘ argument*
‘)’ ;
```

The refactored version cannot match an empty string. The shortest string *list_of_arguments* can match is “()” or an empty list of arguments.

Different design decisions might be made when writing a specification grammar versus a grammar that is input to a parser generator. A person designing a grammar for parsing might try to keep the number of rules as small as possible. This is because the parser ends up needing code specific to each rule. We did not investigate reducing the number of grammar rules in-depth but some of the rules in the specification grammar are redundant, meaning multiple rules can match the same input source text. System Verilog suffers from the ambiguity problem particularly because of implicit data types. According to the specification grammar the source string:

```
x = 5;
```

is ambiguous. The statement could be an assignment to a previously declared variable named *x* or it could be a declaration of an implicitly type variable named *x*.

The second major challenge after the specification grammar was converted to ANTLR format, was merging the new System Verilog grammar with the existing framework. We did not want to have to have two separate implementations for System Verilog and Verilog. This was the most significant design decision of the project. Theoretically, a System Verilog grammar should be backwards compatible with almost all Verilog code. The exceptions are when a Verilog file uses System Verilog keywords to name variables or modules. For example, it is valid in Verilog to name a module *class* but *class* is a reserved keyword in System Verilog. One of the main benefits of a single grammar is it will be easier to maintain

and we only observed one instance of a reserved System Verilog keyword being used as a variable name in a Verilog file in our test set.

Merging the grammars into one meant determining what could be reused and what could not, and writing new AST nodes and analysis for the incompatible pieces. The new grammar is a single grammar that parses System Verilog and is a superset of the original grammar. This meant that the original AST and analysis would be preserved for input source files that were pure Verilog. However, this also meant adding to the original grammar rather than creating a new one from scratch. To avoid having duplicate rules we had to find the common subset between the Verilog and System Verilog grammars. Unfortunately, to compare two rules we must recursively compare all their children so this was time consuming. It also lead to a problem where a construct in Verilog is very similar to a concept in System Verilog but with a slight difference. This meant it would be easiest in the grammar to just add to the existing rule but this has cascading ramifications in the AST node and analysis. If for example, System Verilog's grammar leads to adding a new field to an original Verilog class all the code that handles this type of node should be updated to reflect the change. The strategy we employed was to extend the base Verilog class to a new class whenever possible. This way the base class methods would probably be sufficient for most System Verilog nodes even if some fields were ignored.

7 Progress and Unfinished Work

The current version can parse and build an AST for all the System Verilog and Verilog files in our test set. Analysis (symbol mining, clock mining, and type checking) is still under development.

The parser actions throw a NotImplementedException when it visits a node for which we have not implemented an action. Users can try/catch this exception if desired.

Symbol mining is a bit more complicated than building the AST because the base class symbol miner may or may not work for new nodes on a case by case basis. Manual testing is needed to verify symbol mining works on new nodes.

8 Future Work

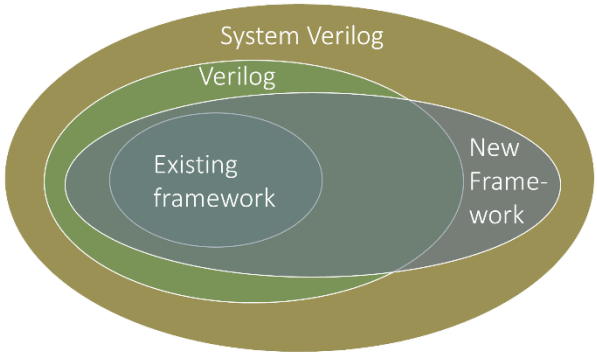


Figure 4 - Conceptual Diagram of Project Subsets

Extending existing analysis to all System Verilog node types and building an analysis on top of these types is future work.

Some of the new features of System Verilog do not have a clear analog in Verilog so how to handle them requires some thought and potentially requires new analysis functionality. These include:

- Enums
- Imports, exports, and packages
- Casts
- Forward Typedefs
- Implicit types

Take for example importing and exporting packages. In Verilog the way to import

something was with an include preprocessor directive so the preprocessor did the work of resolving includes before the static analyzer even saw the source code. In System Verilog we would need to build a mechanism for resolving imports that includes importing a whole package and only importing selected items.

Casting suggests a potentially interesting static analysis where we may be able to statically determine if a cast is valid or not. In some cases it is not decidable, but C# compilers will try to perform this check and will report an error to the user if the compiler is sure the cast will never succeed.

8.1 Edge cases and language coverage

As Figure 4 depicts, neither the original grammar nor the new grammar cover the entire language specification. We decided to prioritize parsing and analyzing the parts of the language that actually appear in our set of examples. Edge cases and complete language coverage could be addressed in future work. For example, UDP declarations and instantiations are not currently supported by our tool.

9 Discussion

Parsing is a difficult problem that will never be solved because it comes up in many different contexts and use cases. The parser for a compiler and the parser for a static analyzer might target the same language but be completely different. For example, maybe I want my static analyzer's parser to accept programs with syntax errors so I can help the user fix them. Or maybe my static analyzer should be stricter than the compiler and not accept programs with implicit casts to help the programmer find casts they might not be aware of.

The idea of looking for implicit casts and doing type checking in general is one interesting area of future work for this project. A program can be type checked much faster than the full synthesis process. Ideally the HDL programmer's IDE would be constantly performing lightweight static checking in the background as the type to catch errors as quickly as possible.

An interesting area of recent work in parsing is incremental parsing. Traditional compilers worked in batch mode. They read the source files, generated the code, wrote the code to disk and stopped. The compiler started from scratch every time even when recompiling code it had already seen. In IDEs and bug finding tools this is not the desired behavior. When a program is small recompiling/reanalyzing everything is fine, but when programs grow to thousands or millions of lines of code we do not want to have to reanalyze the whole program when one line changes. This is an interesting direction for static analyzers that are integrated into IDEs because they can quickly and incrementally analyze the code in the background to give the programmer real time feedback. Integrating gNOSIS into Visual Studio in this way could be one direction for future work.

10 Summary

In this project we extended the gNOSIS framework to support parsing and analyzing System Verilog source code. The main challenges of this project were maintaining backwards compatibility with existing code and supporting both Verilog and System Verilog with a single grammar through adding new rules to the existing grammar. We have begun work on

a type checker for the new types introduced by System Verilog that will add System Verilog programmers in quickly finding errors in their code.

11 References

1. Md. Ashfaquzzman Khan, Richard Neil Pittman, Alessandro Forin. “gNOSIS: A Board-Level Debugging and Verification Tool.” Proceedings of the IEEE Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 43-48. 2010.

2. Mehrdad Majzoobi, Richard Neil Pittman, and Alessandro Forin. gNOSIS: Mining FPGAs for Verification. MSR-TR-2011-141

3. Wille, Robert, et al. "Identifying a Subset of System Verilog Assertions for Efficient Bounded Model Checking." Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on. IEEE, 2008.

4. B.Keng and A.Veneris, “Path Directed Abstraction and Refinement in SAT-based Design Debugging,” in IEEE/ACM Design Automation Conference (DAC), 2012.

5. Nikhil, Rishiyur, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on* , vol., no., pp.69-70, 23-25 June 2004

6. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems Stone, John E. and Gohara, David and Shi, Guochun, Computing in Science & Engineering, 12, 66-73 (2010), DOI:http://dx.doi.org/10.1109/MCSE.2010.69

7. Parr, Terence. "The definitive ANTLR reference: building domain-specific languages." (2007).