Journal of Student Research Volume 10 Issue 4 (2021)

A Higher-Level Synthesis Tool for Rapid-Prototyping of Verilog RTL Designs from FSMD Specifications

Stelios Papoutsakis[1] and Nazanin Mansouri[1#]

[1]Shiley School of Engineering, University of Portland, Portland, OR, USA
[#]Advisor

ABSTRACT

This work presents a light-weight synthesis tool, F2VGen (Finite State Machine to Verilog Generator), that generates *Register Transfer Level* (RTL) implementations modeled using Verilog *hardware description language* (HDL) from abstract specifications provided as *extended finite state machines* (FSMD). This approach takes synthesis yet to a higher level. In contrast to conventional high-level synthesis (HLS) where the design specification is given in behavioral Verilog, in this work specification begins at an even higher level of abstraction and is provided as a finite state machine that captures the data-flow. Designers can input the design specification using a graphical interface (GUI). The resulting synthesized design is implemented at the register-transfer level and is distinctly divided into a *data-path* (that performs the computations) and a *controller* (that controls the operation of the data-path). The process does not guarantee an optimized implementation in terms of area, power consumption or speed. However, it realizes the required functionality in hardware, and can be used in many settings where access to a fast prototype is required. The prototypes are implemented in reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) and present an effective solution when a fast proof of concept design is needed, and also as a stand-in model in situations when testing the interactive systems in actual hardware is required.

Introduction

Synthesis processes have been used to automate the design exercise and to overcome the design challenges arising from the ever-increasing size and complexities of modern digital systems. Synthesis evolution has been a continuous move upward, transcending the layers of hierarchy. Abstraction has played the key role in this evolution and allowed the designers to work at higher levels as we saw an upward shift from layout and transistor levels to logic level, register-transfer level, and finally behavioral level. Behavioral modeling abstracts away many lower-level design concerns and details and allows directing the focus on creative design. However, behavioral modeling still requires fluency in hardware description languages on the part of designer and a great deal of expertise tied to implementation issues. In this work, we bring the modeling yet to a higher level where the focus is purely on design and the implementation concerns are entirely abstracted away. The idea is to maintain a strict separation between the design and implementation. The design work can be done theoretically, for example, by a design architect, who can now create proof of concept implementations. F2VGen can generate Register Transfer Level implementations from the higher-level specifications that can be further mapped to actual hardware.

This paper uses Extended Finite State Machines (FSMD) for modeling the behavior and specification of the design. The behavior is described in an algorithmic style. While capturing the desired behavior of the design, such specifications are significantly closer to human language and thought process. Finite State Machines (FSM) are ubiquitous in the world of digital design. Many control-dominated designs e.g. traffic light, garage or elevator controllers are generally modeled as FSMs. The control logic of all types of designs are also generally modeled as FSMs. An FSM is an abstract mathematical model of a design, accurately capturing its detailed interface, and a computational machine that can enter a limited and defined number of possible states based on the sequence of its input values. The

machine does a predetermined calculation at each state and taking the input values into account, computes the output values and transitions into new states. Even though FSMs are powerful tools for modeling the design behavior, due to state explosion problem cannot be used for modeling the entire behavior of an arbitrary design. Every bit of a state variable doubles the state space of the design. Hence, while they can efficiently model the control flow of a design, FSMs are very limited when modeling the computations and remembering the intermediate results. Extended finite state machines as their name suggests, extend the definition of an FSM. They overcome the aforementioned limitations of FSMs, and can model the behavior of the design in its entirety.

This paper proposes an approach that relies on modeling the design specification at this "higher-level" of abstraction, and a "higher-level synthesis" tool that automatically creates RTL implementations from FSMD behavioral specifications. The input specification is a Moore machine, so the output values are determined by the state of the FSMD independent of the inputs.

The tool has two main components, the graphical user interface (GUI) and the synthesis engine. The design specification FSMD is input using the GUI and is captured as an annotated state transition diagram. The annotations define computations, variable assignments and other data-transfers in an algorithmic specification of the design. Once design entry is complete, the front end interface generates a custom high level textual description of the machine. A compiler then converts this description into an intermediate format that is fed as input into the backend synthesis tool. F2VGen can then create an RTL implementation of this design in Verilog. While the RTL synthesized designs generated by this tool may not be optimized, they are functionally correct and can be used as a golden model for verifying more optimal solutions. In addition, this implementation can be downloaded to an FPGA and used immediately. A main benefit of this tool is that it can generate a prototype rapidly. It eliminates the human variable of writing code and strictly narrows down potential errors to purely logical ones.

What follows is a discussion of the synthesis process and the details of the proposed synthesis tool, its comprising modules and its development. Section II presents the motivation for this work. A survey of related research is presented in Section III. Section IV is focused on in depth presentation of the technical approach. A summary of experiments is presented in Section V. Finally, Section VI is dedicated to concluding remarks and future work.

Motivation

Development of rapid prototypes have many different applications in digital design domain. F2VGen can be effectively used in these applications, particularly in test and verification domains. One such application is in fast development of "Bus Functional Models" (BFMs). When implementing a BFM, its accuracy is of utmost concern as opposed to its optimization. A different application in verification domain is development of reference (golden) models of the design that can be used in equivalence checking. This approach offers a great solution for directly translating design specification into an RTL implementation with focus on functionality. Another application is in hardware emulation. A forth use is in physical test, testing of reactive systems or any other application where the complete or partial behavior of the environment of a "Design Under Test" (DUT) need to be modeled and present in hardware. In all these applications, F2VGen synthesis tool can be used to rapidly prototype a solution.

Related Work

This work presents a novel idea implemented in an electronic design automation (EDA) tool with very unique features that can create a prototype design in minimal time from a very abstract specification model that is yet familiar to any digital design engineer or architect.

In [1] Deeptimahanti and Sanyal present UMGAR, a semi-automated tool that takes requirements written in a natural language format and generates UML models. Like F2VGen, UMGAR relies on parsing engines. However, it is very different in its practical application as it is not fully automated and requires human interaction and intelligence to remove irrelevant classes and to verify relationships between objects. In addition, the input requirements are

very restrictive. Pearce et al present Dave and their exploratory work investigating potential use of ML for translating natural language specifications into their corresponding Verilog HDL [2]. While the work is promising, the authors point to limitations and consider it exploratory. The input specifications are often simple, and while the tool correctly translate into Verilog most of the time, the success is not guaranteed (~94% rate). In [3], the authors present the tool GLAsT that is used to create System Verilog Assertions from natural languages. Their work parallels with this work in functional verification domain, but is targeted to verification and not design synthesis. All the above researches share a similar goal of shifting the focus in specification (in design or verification) upward and closer to human language, however, they have different premises, application and limitations

Technical Approach

The main objective of this work and the synthesis tool developed as a result is to discuss how a descriptive specification of a design as an FSMD (an annotated FSM) can be automatically translated into RTL Verilog code (implementation). To keep to traditional design of FSMs, the specification is provided to the tool graphically. The frontend of the tool is a graphical interface that allows design entry and converts the design input as an annotated graph into textual format. The tool backend is a compiler that translates the textual representation of FSMD into an RTL design modeled using the constructs of Verilog hardware description language. What follows is an in-depth discussion of this tool.

The Graphical User-Interface

The designer fully interacts with F2VGen synthesis tool through a graphical user interface (GUI). The input specification is an FSMD that is an abstract behavioral model of the design. FSMDs are annotated state transition diagrams (FSMs). While FSMs are defined by their inputs, outputs, states and state transitions, FSMDs extend this definition, and though can also be graphically specified using directed graphs, they must be defined by their inputs, outputs, internal variables, states, state transitions and state annotations. Like FSMs, nodes are used to denote states, and directed edges between the nodes to denote state transitions. Each directed edge is labeled with the input conditions for the corresponding state transition. Each node is labeled with annotations that define computations involving the variables, variable assignments and other data-transfers occurring in that state in an algorithmic fashion (Figure 3).

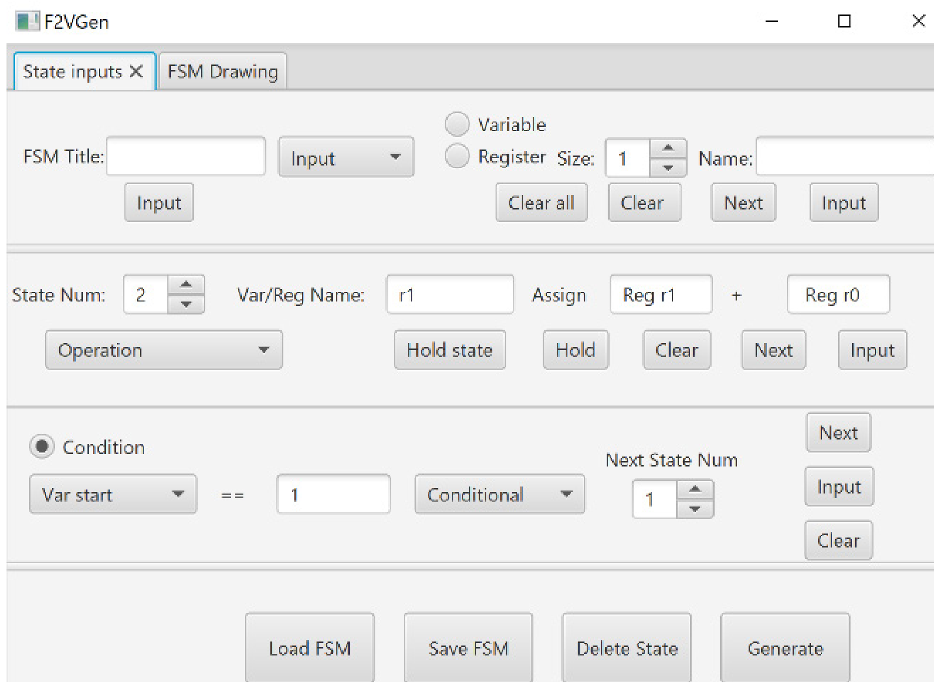


Figure 1. The Graphical User Interface

The tool's GUI provides the designer with graphical widgets to describe the FSMD, which include defining the FSM's inputs, outputs, variables, state assignments, state operations and branching (Figure 1). Additional design features, such as a visual representation of the FSM, are generated by the tool to further aid with the design process (Figure 2). By clicking on each state, the annotations corresponding to the state are displayed. Once the design entry is complete, from the information entered in the GUI a textual description of the FSMD is created, which is fed into the Compiler in the next stage.

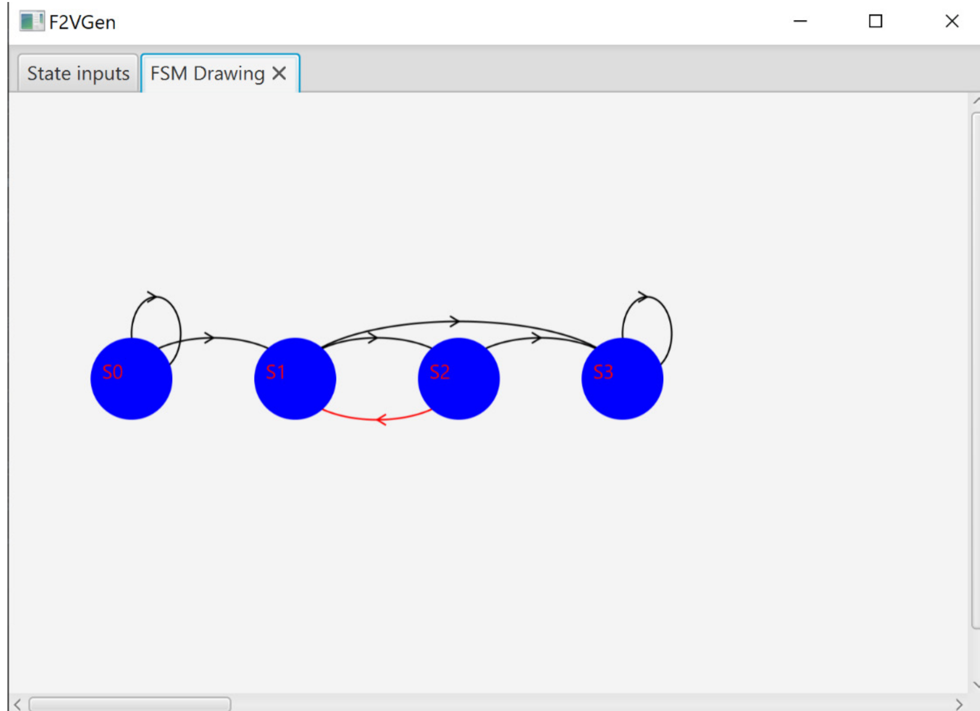
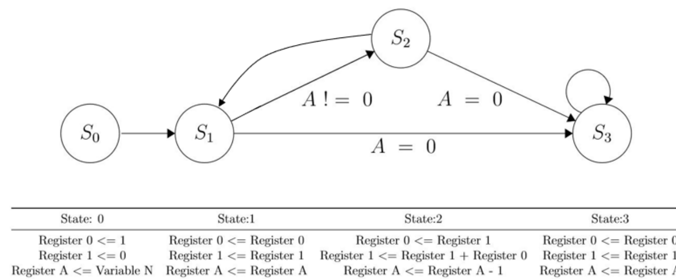


Figure 1. GUI's Auto-Generated Graphical Representation of FSM

Figure 3 shows an algorithmic specification of Fibonacci function also given in (1) as an FSMD.

$$F(n) \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$



State: 0	State:1	State:2	State:3
Register 0 <= 1 Register 1 <= 0 Register A <= Variable N	Register 0 <= Register 0 Register 1 <= Register 1 Register A <= Register A	Register 0 <= Register 1 Register 1 <= Register 1 + Register 0 Register A <= Register A - 1	Register 0 <= Register 0 Register 1 <= Register 1 Register A <= Register A

Figure 2. FSMD Specification of Fibonacci Function

This FSMD has one input, one output, 3 variables, 4 states and 7 state transitions. State annotations show the computations and variable assignments corresponding to each state. The textual representation of this FSMD, generated by the tool's front-end is shown in Figure 4. It is this textual specification of FSMD that is input to the compiler.

```
Start FSM
input Size 1 Var start Size 8 Var n
output Size 1 Var busy Size 8 Reg r1
State:0 Define
Size 8 Reg A = Var n
Reg r1 = 0
Size 8 Reg r0 = 1
Var busy = 0
Next State if Var start==1 State:1
Next State State:0
End
State:1 Define
Reg A = Reg A
Reg r1 = Reg r1
Reg r0 = Reg r0
Var busy = 1
Next State if Reg A==0 State:3
Next State if Reg A~=0 State:2
End
State:2 Define
Reg A = Reg A -  1
Reg r1 = Reg r1 + Reg r0
Reg r0 = Reg r1
Var busy = 1
Next State if Reg A==0 State:3
Next State State:1
End
State:3 Define
Reg A = Reg A
Reg r1 = Reg r1
Reg r0 = Reg r0
Var busy = 0
Next State State:3
End
End FSM
```

Figure 3. Textual Specification of Fibonacci FSMD

The Compiler

In this work, a compiler was developed that takes the FSMD modeled using a defined grammar and translates (synthesizes) it into an RTL implementation modeled using the constructs of the Verilog. Then, the objective of this compiler is to translate from a new textual FSMD language to Verilog language. The compiler heavily relies on the tool ANTLR [4] that will be introduced in the following sections. The development of the compiler can be summarized as follows:

1. Developing the FSMD Grammar
2. Developing the Lexical and Syntax analyzers to create a parse tree from the source language
3. Developing the Code Generator
4. These topics will be discussed in detail in the following sections.

Grammar Definition

A formal language is syntactically defined by its *formal grammar*. A formal grammar consists of the set of all rules that describe how to form valid strings of the language from its alphabet. Hence, the grammar is exclusively concerned with the syntax and not the semantics of the language constructs (their meaning). There are two types of grammar rules: *lexical rules* and *syntactic (parser) rules*. Lexical rules define a series of symbols that are used to create *tokens*. Tokens are used to partition the input stream into logical units. Syntax rules are more complex rules that contain lexical rules as well as other rules. They are used to group the tokens of the language into *phrases*. Every grammar has a top level rule which aims to match the entire text. If an input stream does not match this rule, the string is rejected

by the grammar. This hierarchy is what forms a *parser tree*, which is a tree structure where the branch nodes are parser rules and the leaves are the tokens that form the lexical rules.

```
register: ('Size' INT)? 'Reg' NAME;
INT: ('0'..'9')+;
NAME: (UPPERCASE | LOWERCASE)+ (INT)*;
fragment UPPERCASE: ('A'..'Z');
fragment LOWERCASE: ('a'..'z');
```

Figure 4. FSMD Grammar Rules

As the first step in designing the compiler of F2VGen, the *FSMD Grammar was defined.* Figure 5 shows some of FSMD Grammar rules. When developing this grammar, the goal was to create an English-like language that could behaviorally describe FSMDs. At a high level, an FSMD has some main elements: input signals, output signals, internal variables and states. Thus, the highest level parser rule needed to define these main elements. First, inputs are signals that are driven by external systems. They can be of any size.

In FSMD Grammar, "Var" is used to determine a signal which is driven by a source. In other words, the variable has no memory and cannot retain its value from previous states. Registers are variables which retain their value from state to state. In FSMD Grammar, "Reg" is used for these variables. Lastly, outputs are signals that are used to drive other circuitry. Output variables differ as they can be either defined as "Var" or "Reg". One level below the root rule is the state parser rule. This rule is used to define operations and branching conditions that occur. This rule allows for arithmetic operations (e.g. addition, subtraction, multiplication, division, …), logic operations (such as bitwise *AND*, bitwise *OR, …*) or other operations. "Reg" variables store the results of computations. If a register variable is not an output, it can be defined in the first state with its relative bit size. Designers can also drive output "Var" variables to certain values at each state. Finally, branching conditions are defined. Conditional branching is allowed through evaluating logical expressions formed using relational operations and involving either register or input variables. Figure 6 shows a simplistic 2 state FSMD defined based on FSMD Grammar rules.

Once FSMD Grammar was defined, any FSMD could be described using the grammar constructs of the defined language. The input FSMD stream can then be syntactically analyzed and a corresponding parse tree for it can be automatically generated. This will be further discussed in the next section. Parts of the parse tree of the FSMD description in Figure 6 is shown in Figure 7.

```
Start FSM
input Size 1 Var start
output Size 5 Reg sum
State:0 Define
Reg sum = 0
Next State if Var start==1 State:1
Next State State:0
End
State:1 Define
Reg sum = Reg sum + 1
Next State State:1
End
End FSM
```

Figure 5. A Simplistic FSMD

Lexical and Syntax Analyzers

When developing compilers, grammatical phrases of the source language are often represented by a parse tree. Hence, the first stages in developing a compiler, are developing the *Lexical Analyzer* and *Syntax Analyzer* modules that match

the stream written in the source language to the grammar rules and create a parse tree. In this work, for developing these modules the tool ANTLR was used.

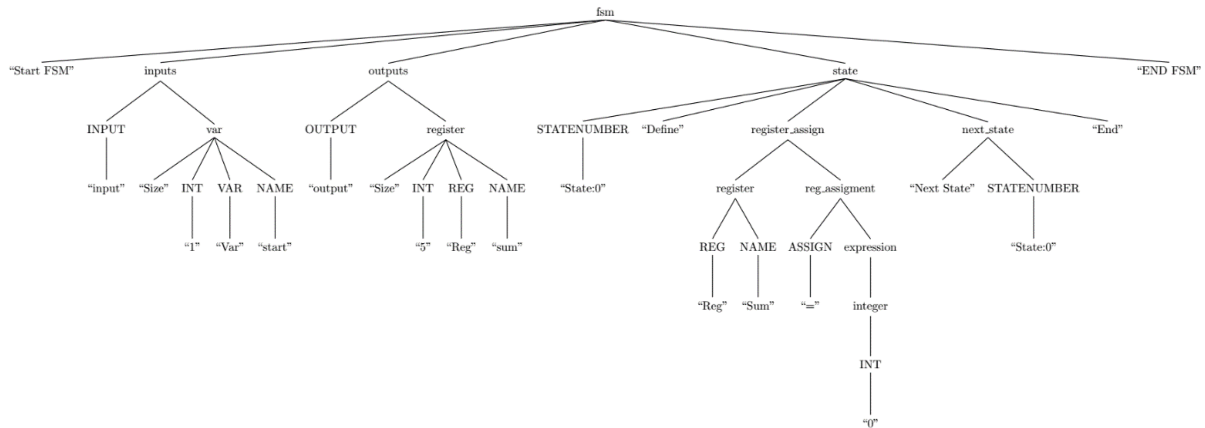


Figure 6. An Automatically Generated Parse Tree

ANTLR:

“ANTLR (Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees” [4]. ANTLR creates programming code that provides an API used to parse input data.

ANTLR provided the perfect support for developing the compiler’s front-end (Lexical Analyzer or lexer and Syntactic Analyzer or parser).

Once the complete FSMD Grammar was precisely defined, ANTLR was used to generate the lexer and parser modules. Given the grammar and an input FSMD specification, the compiler can now accept or deny it, and when syntactically correct, generate its parse tree.

Code Generation

In the final stage, the compiler needs to generate the code in the target language. The input of the *Code Generator* module is the parser instance, and the output is the RTL design in the target language - in this case Verilog.

ANTLR’s Parse Tree Traversal:

ANTLR also provides the necessary API used to traverse the parse tree for an accepted input. ANTLR provides a *walker*, which is used to traverse the parse tree that is generated by the parser instance. The walker first enters the root node and then traverses it depth first. ANTLR provides the ability to override *listener* methods. There are two types of methods for each node, *enter methods* and *exit methods*. These methods allow for arbitrary code execution when a walker reaches a certain node.

This infrastructure allowed us to develop a walker module that traverses the parse tree to extract and store all the necessary information about the input FSMD. Once the walker finishes its traversal, it uses this information for synthesizing the design.

Parse tree's operation nodes are analyzed and used for resource allocation and binding to functional units, its variable nodes are analyzed for register allocation and binding and the design's complete data-path is constructed using this information. The steering logic is added by analyzing the data-flow to complete the generation of the data-path. Parse tree's state nodes are analyzed and the information is used for controller generation.

The data path and controller of the design are created and connected by the tree walker and the Verilog code is created using data-path and controller templates. The complete RTL design is generated using the information collected during the tree traversal. Figure 8 shows the block diagram of F2VGen.

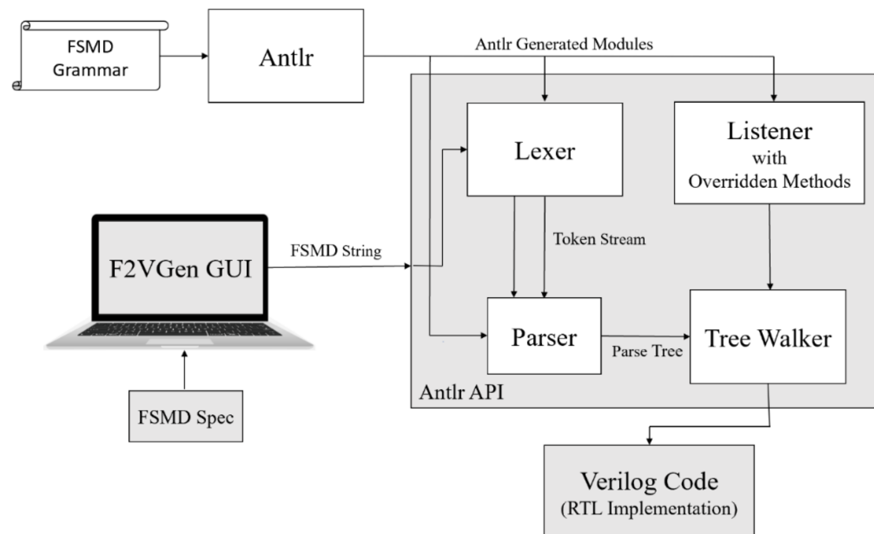


Figure 7. F2VGen Synthesis Tool

Experimental Results

In this work, several Verilog models of designs of small to medium scale have been successfully generated from their high-level FSM specification as shown in Table 1. The Verilog design implementations were successfully synthesized and mapped for download into a ZedBoard FPGA.

Table 1. F2VGen Generated Designs

Design	Inputs	Outputs	Variables	States	Synthesized	LUTs	Registers
Fibonacci	2	2	6	4	Yes	34	24
Factorial	2	2	5	4	Yes	52	16
Hannoi	2	2	5	5	Yes	21	16
Exponent	3	2	7	5	Yes	56	24
Polynomail1*	2	2	7	7	Yes	63	26
Polynomial2**	2	2	7	11	Yes	53	51
Polynomial3***	2	2	7	13	Yes	55	51
Is Even	2	2	5	5	Yes	10	8
Is Prime	2	2	8	6	Yes	142	32
Square	2	2	5	4	Yes	9	11

*$4x^2 + 6x + 7$
**$8x^4 + 2x^3 + 9x^2 + 5x + 2$
***$7x^5 + 4x^4 + 23x^3 + 8x^2 + 9x + 11$

Conclusion

This paper has presented F2VGen, a lightweight higher-level synthesis solution for auto generation of RTL designs (in Verilog HDL) from FSMD behavioral specifications. This tool successfully abstracts the behavioral modeling. This allows for designs to be rapidly prototyped and quickly synthesized to hardware devices such as FPGAs. The tool's GUI provides creative widgets that allow for incremental construction of the FSMD specification. Antlr provided the necessary modules to accept textual representations of FSDM and create a code generator based on that representation. While F2VGen's GUI is an academic proof of concept tool that is currently used for entering small to medium size FSMD examples, future work will aim to expand testing larger and more complex design specifications.

References

[1] D. K. D. a. R. Sanyal, ""Semi-automatic generation of UML models from natural language requirements"," in *ISEC '11: Proceedings of the 4th India Software Engineering Conference*, February 2011. https://core.ac.uk/download/pdf/59347307.pdf

[2] B. T. a. R. K. Hammond Pearce, ""DAVE: Deriving Automatically Verilog from English"," in *MLCAD '20: Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, November 2020. https://arxiv.org/pdf/2009.01026.pdf

[3] C. B. H. a. I. G. Harris, ""GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions"," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016. https://past.date-conference.com/proceedings-archive/2016/pdf/0334.pdf

[4] T. Parr, "The Definitive ANTLR 4 Reference", Pragmatic Bookshelf, 2013.