

Legend Of Bluespec

kr469

October 2021

Contents

1	Introduction	3
1.1	Motivation	4
1.2	Already existing tools	4
1.3	Proposed solution	5
2	Preparation	6
2.1	Understanding Bluespec	6
2.1.1	Rules	6
2.1.2	Modules and interfaces	7
2.1.3	Typeclasses	7
2.1.4	Provisos	7
2.2	Choosing tools	8
2.2.1	Haskell	8
2.2.2	Tcl (pronounced "tickle")	8
2.2.3	Python	9
2.2.4	Postmortem	9
2.3	Creating a grammar	9
2.3.1	What grammar is needed ?	9
2.3.2	Where to find this grammar?	10
2.3.3	Technical aspects of the grammar	10
3	Implementation	12
3.1	Crawling the packages	12
3.2	Loading Bluespec packages' information (TODO check grammar of s')	13
3.2.1	Lexical and syntax analysis	13
3.2.2	Understanding abstract syntax tree (AST)	13
3.2.3	Organizing data and functionality	13
3.3	Resolving types	15
3.4	Building the top level module	19
3.4.1	Instantiating new modules	19
3.4.2	Connections	20
3.4.3	Busses	20
3.5	Synthesizing the module	20
3.5.1	JSON interface	21
3.6	GUI	24
3.6.1	Backend	25
3.6.2	Frontend (React)	25

3.6.3	Frontend (React Flow)	25
3.6.4	Frontend (MUI)	25
3.6.5	Functionality	26
4	Evaluation	32
4.1	Overview	32
4.1.1	Polymorphism disclaimer	32
4.2	Example 1 - Connecting FIFOs	33
4.2.1	Platform Designer	33
4.2.2	My tool	34
4.2.3	Using text format to do same things and Qualitative metrics	36
4.3	Example 2 - Simple System-on-Chip	36
4.3.1	Platform Designer	36
4.3.2	My tool	36
4.3.3	Other differences	39
4.4	Example 3 - Working AXI4 masters and slaves	39

Chapter 1

Introduction

Making hardware is hard, and similarly to other problems in computer science, we make it easier by adding layers of abstraction. There are many layers involved in making hardware, from those abstract transistors to gates to logical operations to functions to modules to chips to multi-chip modules to whole devices. The layer this project focuses on is going from modules to chips. At this layer, we already have code for modules like memory, CPU cores, interconnects etc. Now what we need is an ability to compose them in such a way that they create a whole device.

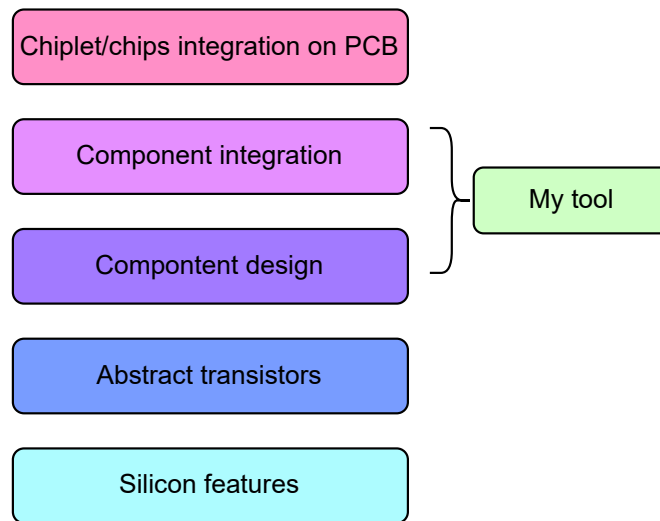


Figure 1.1: Stage one of node used to instantiate new modules

1.1 Motivation

Vocabulary

- Verilog - Hardware description language
- Bluespec - Hardware description language
- Intel Quartus Prime (IQP) - programmable logic device design software produced by Intel. (source [Wikipedia](#))
- Platform Designer - Tool inside IQP used to connect high level components.

Verilog is an old language with a minimal type system. It forces people to encode more complex interfaces using finicky naming conventions. To simplify the process of connecting components with many wires, people tend to use high-level tools like Platform Designer, which has built-in logic allowing for easy connection of components. This approach is not perfect and as we will see during the evaluation, a better option might be to move away from Verilog and use a more modern language like Bluespec. The problem is that tools like Intel Quartus Prime (IQP) do not have native support for Bluespec, and it forces Bluespec users to work with transpiled Verilog (from Bluespec) code. Unfortunately, because of typing richness of Bluespec (comparable to Haskell), arbitrary code cannot be transpiled, and before code can be transpiled it needs to be stripped of any arguments or other kinds of polymorphism. The process of transpilation also destroys high-level interfaces, and they must be recreated manually after importing Verilog code into IQP.

1.2 Already existing tools

Creating custom hardware became immensely expensive in the last few decades both in terms of money but also in terms of man-hours needed to create it.

This creates barriers to entry for new players, and also reinforces the strong positions of already adopted standards.

This also means that there is not as large an open-source community as around hardware and therefore one cannot simply use a plugin for Bluespec native high level integration tool because there are none.

My project will try to tackle a subset of functionality provided by a tool called Platform Designer that is a part of the Intel Quartus Prime package. Platform Designer is a GUI tool for high-level integration, designs generated by it are kept in verbose text file format. Unfortunately, those files tend to be megabytes long, making them hard to read and edit, for humans. There are other problems with using Platform Designer, like the cumbersomeness of importing new components.

Market share and justification for focusing entirely on comparisons with Intel Quartus Prime

It is a bit difficult to accurately judge the market share of IQP, but It is produced by one of the biggest designers and manufacturers in the world, and it is also used by Qualcomm according to [HG Insights](#). Using Google trends we can see that in recent years (since 2016) competitor Xilinx Vivado has overtaken IQP in interest at the global scale, in most developed countries like the US, Europe, and parts of Asia. There is roughly 50/50 split between IQP and Vivado. My personal observations suggest that Xilinx Vivado became highly popular in India which is a large country with a history of chip design, and this might be strongly skewing data in favour of Xilinx's product. Another argument to compare against IQP is that it was used last year as a part of the course, and it cannot be irrelevant if it was taught at Cambridge. Therefore, it is fair to not investigate Xilinx's software during evaluation (TODO: check if this has changed).

1.3 Proposed solution

During my project, I created an alternative JSON based file format, that is much simpler and human-readable, that can be interpreted using my tool to produce Bluespec code of a top-level module.

This file format is rather unimpressive on its own, and one might argue that it would be better to use Bluespec code instead of it. The interesting part of this project lies in the backend that can verify correctness of this file and produce set of typing information about produced Bluespec code.

But to truly appreciate the work I have done during this project, I also implemented GUI, which uses the same backend as the one used to interpret JSON files and is able to supply the user with typing information and possible choices when connecting components.

One can think of it as a Bluespec native high-level integration tool that uses a simple language server in the backend. This way Bluespec users can do high-level integration in the domain of not transpiled Bluespec code. Then after a satisfactory top-level module is created, it can be transpiled to Verilog and then imported into IQP.

Chapter 2

Preparation

2.1 Understanding Bluespec

Before I started this project, my only knowledge about Bluespec came from [Cambridge Bluespec Tutor](#) which covers basics, and it was completed by me a few weeks before the start of this project. To understand this project, I have learned about most of the type system of Bluespec, but for you, the reader, most of it is not relevant. Therefore, I prepared a short tour of things that will be important to understand in this project.

2.1.1 Rules

In Bluespec all computation is done in form of rules. Each cycle we will take a subset of all rules that we are going to execute in this cycle, rule is fired (executed) in cycle only if it's ready (or will be ready) and it's not conflicting with other rules (If this happens compiler must issue a warning, and picks arbitrary rule to fire from subset of conflicting rules). Each rule can fire at most one time per cycle. For rule to be ready to fire it needs it's implicit and explicit conditions to be true. Rule can be fired in a cycle even if it's not ready at the start of cycle, for example if you add item on an empty queue and then pop can happen in same cycle.

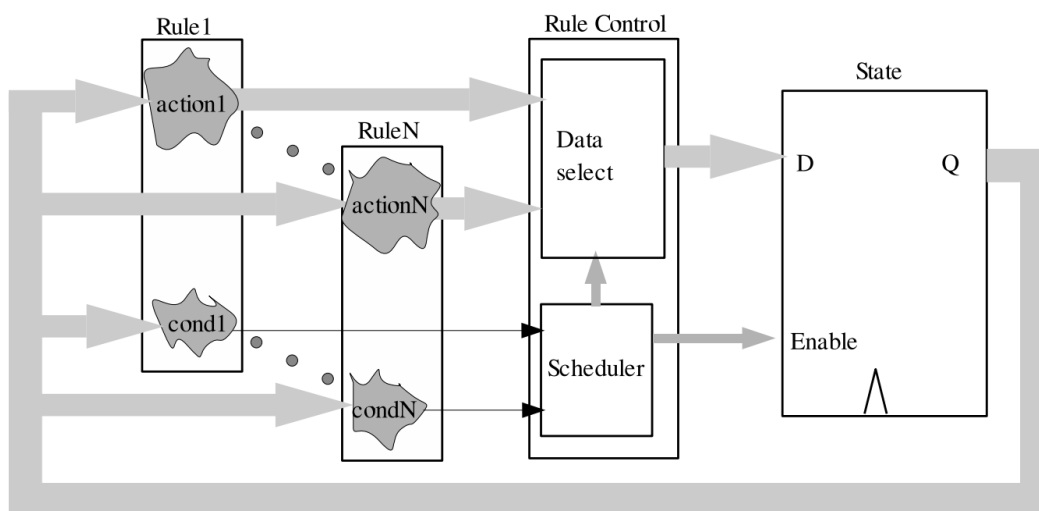


Figure 1: A general scheme for mapping an N-rule system into clocked synchronous hardware.

(TODO: add reference to the bsv reference document from which this image was taken)

TODO: piece of Bluespec with module using fifo and other rule explaining types of conditions.

2.1.2 Modules and interfaces

Module is unit of organization of logic. Modules don't have types, instead they implement some interface. This means that you can have multiple modules with different behaviour that implement the same interface. It might be helpful to think about similarly to different physical connectors (like USB-C, PCI-e). Interfaces are made up out of two types of things:

- Methods - that allow for interaction with the module, they come with few variants, but this is not important for now.
- Subinterfaces - That allow for more generalization, for example if you want to implement module that contains two copies of the same interface, you can just create a new interface that contains two instances of that subinterface and assign them accordingly.

2.1.3 Typeclasses

Typeclasses in Bluespec are used to group types for which specific functions or modules are implemented. Typeclass instance is (TODO describe what is it) and it can have provisos attached to it.

2.1.4 Provisos

One big feature of Bluespec typing is support for provisos. They create constraints on types that can be used with a given function and give information about the expected type of the output of the function during compilation. For example, we can have a polymorphic function `extend` of a type $\text{Bit}\#(a) \rightarrow \text{Bit}\#(b)$ where $\text{Bit}\#(x)$ is a vector of x bits, such function accepts vector any length a and always return vector of length $b = a + 1$. To make sure these relations stay like this, we would attach a proviso `Add\#(a,1,b)` which means that $a + 1$ must be equal to b .

Mentioned above `Add` is one of the Size Relationship provisos, other such provisos are `Mul` `Div` `Max` `Min` `Log`. Those provisos cannot be nested, but we still can express complex arithmetic constraints, thanks to size relationship type functions. They essentially represent the result of the operation on values that were supplied. There are only 8 of them, and they are `TAdd` `TSub` `TDiv` `TMul` `TMax` `TMin` `TLog` `TExp`. To better explain this, let us look at this simple example:

`Add\#(TExp\#(a),TDiv\#(a,8),c)` is a proviso that says that the following equation $2^a + \lceil \frac{a}{8} \rceil = c$ on parameters `a,b,c` must be satisfied. Otherwise, it will mean that supplied parameters are not valid and compilation won't succeed. Those equations can be also solved to deduce unknown parameters.

There is also another type of provisos that I will call *typeclass based* ones. Typeclass based provisos allow us to either ensure certain functions are defined for a given type or to deduce something. For example, `Bits\#(a,b)` means that for type `a`, there must be functions `pack` and `unpack` allowing for conversion to and from a vector of b bits. Moreover, this can be used to get the length in bits of given objects, on which we can later put constraints using Size Relationship

provisos. In my project, I will often check if there exists an instance of the `Connectable#(a,b)` typeclass for a given pair of types `a` and `b` to check if function `mkConnection` (used to connect two interfaces together) is defined. All of this can be a bit confusing, but if one looks closely, system of provisos and typeclasses is remarkably similar to Prolog the programming language. We have variables that are effectively the same as in Prolog, we have values that are integers, and we have parametric types and functions that are like Prolog's compound terms. Typeclass are like groups of rules with the same name, and provisos attached to Typeclass instances are like predicates attached to rules.

One significant difference is that provisos can be solved in arbitrary order, which means that unlike in the Prolog, we need the ability to solve sets of equations on integers to resolve them.

2.2 Choosing tools

I had effectively 3 choices for a language to write this project in. Here is some justification for why I have chosen Python.

2.2.1 Haskell

The compiler of Bluespec is written in Haskell. Therefore, there are several reasons why Haskell would be the right choice. Here are some of them:

- I can reuse already written logic.
- I will not diverge in terms of logic from Bluespec compiler.
- I will not need to write grammar to load packages.

But there are also a few particularly good reasons why I did not use Haskell.

- I never used Haskell before, and my experience with functional languages is equal to what was shown in the foundations of the computer science course.
- I never used Bluespec before, and my experience is limited to completing the Bluespec tutor provided by the computer science department.
- Even if I knew Haskell and Bluespec, industrial compilers are rather large and complicated. I also received advice to avoid modifying/extending them as a part of the project.

While each of those reasons on its own is something that can be managed. All of them combined would create an unacceptable amount of risk of not delivering the project. There was also a chance that the use of Haskell would not create a good story for the project. (This is quite subtle).

2.2.2 Tcl (pronounced "tickle")

This language is used as a scripting language in both Intel and Xilinx tools, While I will be reading packages using Tcl scripts provided by the creators of the Bluespec compiler (BSC), my understanding is that those scripts are just handy wrappers for some Haskell functions. This is also a foreign language to me, with a minimal presence online and negligible learning resources, making it difficult to learn. It is also quite obsolete, and there is not much tooling for it making it undesirable as a time investment (As an example of the desirable to learn technology. I learned for this project JavaScript just to create a GUI).

2.2.3 Python

Firstly, this is a language I have experience working with. Secondly, it is widely supported, and there is extensive tooling for it. Its flexible typing system allows for rapid experimenting, which was especially handy when I was not sure what the structure of the data would look like. It is also worth mentioning that Python has support for typing, and at some point, I decided to start adding typing hints around. My rule of thumb that I used was that if Pylance (Python language server) cannot deduce the type of something, I need more hints. Other reasons for using Python are:

- I will have many interesting problems to solve. Like creating and interpreting grammar.
- Code written in it can be used by a large community of python developers.
- I can use many tools like Lark (for grammar), Django (as a backend server), and SymPy (to solve size relationship provisos) that will speed up the process.

The two arguments against using Python are:

- Haskell would be better suited for this task. I explained above why it does not make sense for me to use Haskell.
- Python is usually slow. Fortunately, this is not a big issue thanks to caching and the use of PyPy (implementation of Python using JIT compiler).

2.2.4 Postmortem

Looking at those choices now (after finishing the project and knowing all encountered difficulties), I could have also considered C#, due to its speed and similar typing flexibility to Python. I also think Haskell would be a better choice if this were more like my pet project on which I would work for a few years to make it a commercially viable product.

2.3 Creating a grammar

This section could be moved into implementation.

Bluetcl

Bluetcl is a tool written in Tcl language that allows for inspection of Bluespec packages. It is a first-party tool that is included with the Bluespec compiler. There is not much documentation about it, but from my understanding, it is a wrapper around Haskell's code. From the user perspective, it is just a bash script that starts the Tcl terminal compiled with some libraries. From my perspective, I use it as an arbitrary command-line program, as this method of use is intended by the authors and contains some documentation.

2.3.1 What grammar is needed ?

Because I am using Bluetcl as a command-line tool, communication with it is done via text. While synthesizing commands is easy, parsing outputs is a bit more difficult, and I decided to do it

systematically. To do this, I need the grammar of all possible outputs that I am interested in. Bluetcl produces a range of outputs, but two of them that are particularly useful are descriptions of functions and descriptions of types.

2.3.2 Where to find this grammar?

Unfortunately, this grammar is not documented anywhere, so I had to reverse engineer it. This approach might not be perfect and might not cover every input, but it is good enough to parse every standard package and all packages used to compile Flute (RISC-V CPU with 5 stage pipeline). Here are other reasons to justify this approach:

- Heaps' law suggests that the number of unique words in a given body of the text is proportional to approximately the square root of the number of words in the text. Therefore, it is fair to assume that something similar will be true if we consider the number of unique grammar rules.
- This grammar, while different from the grammar of Bluespec the language maps a subset of Bluespec grammar, so we can supplement our deductions with a reference guide for Bluespec language. Therefore, accounting for cases we have not seen, but are possible.
- There are also some limitations on how exotic things can get in top modules. (TODO rephrase this)

2.3.3 Technical aspects of the grammar

This is EBNF grammar, I parse it using Lark library for Python, and I am using Earley parser, as it is capable of arbitrary length lookahead. The grammar I created contains roughly 90 rules, and I will not include all of them here, but I will show an example to give a feel of what is happening. (TODO: Think about dumping them into an appendix)

A useful feature supported by Lark is the ability to have regular expressions in the grammar. I am mentioning this as it is effectively like having a parser inside a parser. It is also worth mentioning that there exists a handy tool for debugging and creating grammar. It is located at this website: <https://www.lark-parser.org/ide/> and it can run the parser online and show created AST tree.

Parsing position TODO maybe find a better example with shorter line

Here is an example of parsing a small section of output that describes the location of a piece of code that is being described. I decided to show it as it's simple and shows different features that I needed to use.

----- Rules used -----

identifier_u: /[A-Z][\w\$_']* /

tcl_position: "{" "position" "{" tcl_path NUMBER NUMBER
["{" "Library" identifier_u "]" "]" }

tcl_path: ["%/" /((\.{1,2})|(\w|-|\s)+) /
["/" /((\.\.|\.)|((\w|-|\s)+)) /]* "." /\w+ /

----- Text to parse -----

{position {%/Libraries/Connectable.bs 25 1 {Library Connectable}}}

▼ tcl_position

▼ tcl_path

Libraries

Connectable

bs

25

1

▼ identifier_u

Connectable

Chapter 3

Implementation

3.1 Crawling the packages

Figure 3.1: Overview of process of crawling packages



To interact with Bluetcl, I have written a script using the Pexpect library. This script works by creating a subprocess of Bluetcl and exposes functions that allow for the performing of queries. The core of this script is a function called `call` that takes as input a string that is a command and returns the output of stripped of warnings or raises an exception if an error occurred (for example: in the case where a package was not found). To remove warnings, I make some assumptions.

- I only use a fixed set of commands (like `list packages`, `describe a type`).
- For those commands important output is always on the last line. (This was checked empirically)
- Output of command is always followed by `%` a character that never occurs in the rest of the output and marks the end of the output.

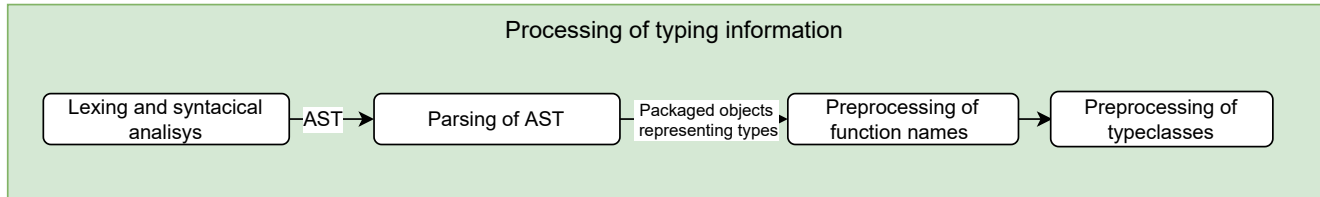
Thanks to them, I do not need a sophisticated system for parsing errors and or warnings. I can just make a quick search for the error, and if needed forward this to the user.

In total, this script allows me to:

- Initialize subprocess
- Add folder to search path of Bluetcl
- Load package (Bluetcl takes care of finding and loading dependencies)
- Get the list of loaded packages
- List functions in a package
- List types in a package
- Get information about types and functions in the package

3.2 Loading Bluespec packages' information (TODO check grammar of s')

Figure 3.2: Overview of parsing bluetcl data



As mentioned earlier output of bluetcl is text, and I need it in some better-organized form. In this section, I describe the process of doing this.

3.2.1 Lexical and syntax analysis

I decided to use the Lark library for lexical and syntax analysis. This library takes grammar and strings of text as input and returns an abstract syntax tree. The only difficult part here is creating grammar, but we have taken care of it during preparation. (TODO improve this last sentence)

3.2.2 Understanding abstract syntax tree (AST)

Given AST I needed to turn it into a more useful form. To do this I used `Transformer` the class provided by Lark. The way it worked is that I created a subclass of `Transformer` that implemented a function for each non-terminal / rule (Each non-terminal can have multiple rules and Lark allows for renaming of individual rules, for example, this allows parsing different non-terminals using the same function or parsing single non-terminal using different functions depending on rule that was used). `Transformer` will then perform a bottom-up transformation of AST, and for each rule, it will call the respective member function, with parsed children (non-terminals used by rule) as arguments. At the end of the transforming process, I have a list of objects that represent data from Blutecl.

One of the bigger inconveniences I encounter is the fact that Lark passes parsed children, as a simple list. This does not provide information about what exactly each child is, this is not a problem if the rule contains a fixed number of non-terminals, but otherwise, I employ a mix of length-based inference, and wrapping of data from children into tuples, to provide remove ambiguity.

What was quite useful at this stage is Python's type system flexibility, I was able to gradually build up `Transformer` class and add typing information only after I had a firm understanding of what type something needs to be. For some things, it was not immediately obvious what is the best way to represent them, and what often happens is I would first leave them as lists of unparsed things. Then after I got to the point where I needed to use them, I would decide what is an optimal way to represent them.

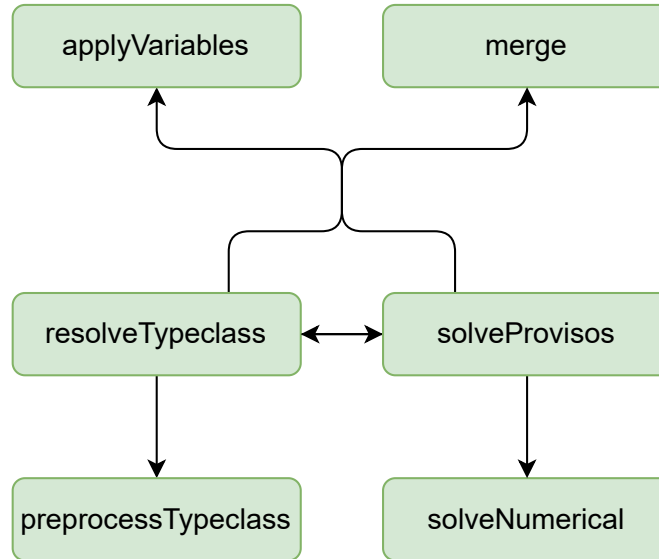
3.2.3 Organizing data and functionality

I created a class called `TypeDatabase` that provides simple functions that manage loading and parsing packages information. It is also capable of loading and storing it's state to pickle file. This

saves a lot of time as all this parsing and communication with bluetcl isn't particularly fast. This class is also a home to set of functionality used to resolve types. From my testing it takes about 370s to load roughly 250 packages that I found included with Bluespec compiler and created to compile Flute CPU. Loading those packages from cached files takes only few seconds.

3.3 Resolving types

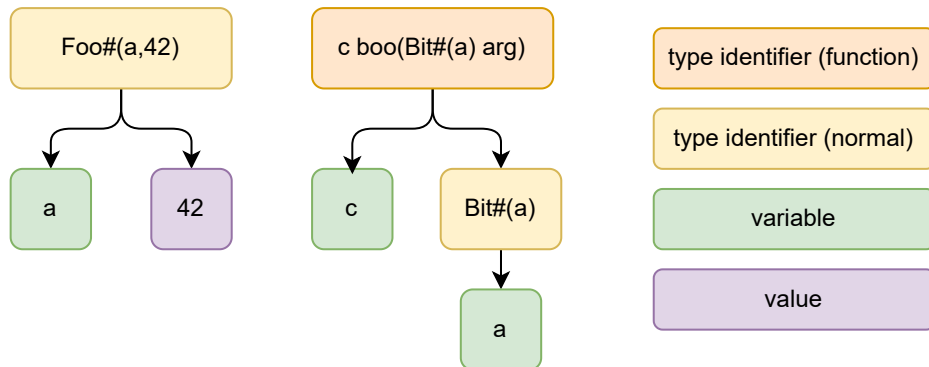
Figure 3.3: Dependencies between functions



This section assumes that one have read about typeclasses and provisos from [previous chapter](#). Before we can talk about the process of building top level module we need to talk about resolving types, as without it we could not do much beyond formatting strings.

Types of most things can be represented by a type identifier. Type identifier can be thought as a tree data structure where every non leaf node is another type identifier, and leaf nodes are values (integers) or variables. Example: type identifier `Foo#(a,42)` has a name `Foo` and two arguments: variable `a` and value `42`. Type identifiers can also describe functions. For example: `Bit#(b) boo(Bit#(a) arg)` where, `boo` is a function with one argument named `arg` (it is type identified by type identifier `Bit#(a)`), and it returns object of some type `c`. In the tree representation it looks like this:

Figure 3.4: Break down of a type identifiers as a tree



In my code there are 4 main functions that combined allow for type resolution:

- **merge** function takes two type identifiers and returns dictionary (map) from names of variables to their values. It's similar to Prolog's unification operator.
- **applyVariables** function takes a dictionary and a type identifier and returns a type identifier with all variables replaced by their values. This is more of a utility function.
- **resolveTypeclass** function takes a type identifier that is an proposed instance of a typeclass and returns dictionary of learned variable values if proposed instance is a valid one otherwise it raises an exception. In Prolog terms this would be like looking which rule can be applied and returning resolved variables from applying that rule.
- **solveProvisos** function takes a dictionary of variables and a list of provisos and returns a dictionary with variables learned by solving provisos or raises an exception if it's impossible to satisfy all the provisos. This is like checking predicates in Prolog.
- **preprocessTypeclass** function that takes a typeclass and produces a look-up dictionary of typeclass instances for given group of names present in the instance of the typeclass.
- **solveNumerical** function that takes list of size relationship provisos and returns dictionary of learned variables.

Merge function

This function works like two synchronized dfs's, that go through the trees (type identifiers) and apply logic based on nodes that are beginning currently visited. This function needs to account for quite a lot of special cases as each node can be one of the following: variable, value (integer), type identifier (normal), and type identifier (function). Fortunately, half of the cases are symmetrical to the other half. In broad strokes, here is the logic behind those cases:

- If one of the nodes is a variable, and the other is a value or another type identifier, then we set the value of the variable to a value of the other.
- If both nodes are values, we check if they are equal.
- If both nodes are the same kind of type identifier, we recursively compare their children. (If there are not of a function kind, we also compare their names)
- If both nodes are variables, then we need to check if it's possible to unify the values of those variables if so, we assign the value of those variables to the result of unification.
- In other cases, like when we need to unify type identifier with value, we raise an exception.

One minor problem that I encountered was that I had to keep track of which variables are equal, but I was able to avoid this problem altogether using shared pointers.

Apply variables function

This function is quite simple. Simply recursively traverses a type identifier and replaces all variables with their values.

Preprocess typeclass function

Earlier, I compared typeclass to groups of Prolog rules with the same name, one thing I did not mention is that typeclasses can also contain something called dependencies, they are something like marking inputs and outputs in Prolog. Those dependencies are used in typeclasses like `Bits#(a,b)` which are used to determine something, and they make sure that the resolution of a typeclass is always unambiguous.

For example, typeclass `Bits#(a,b)` has a dependency `a determines b` which means that you can use it to determine `b` from `a`, but if you know only `b` then you cannot use it to determine `a` because there may be multiple types that can be packed to the bit vector of the same length `b`. An example of determinable pair of `a` and `b` would be `a = Bit#(32)` from which we can determine that `b = 32`. It is also worth mentioning that we can have multiple tuples of dependencies. For example, we can have typeclass `Boo#(a,b,c,d)` with dependencies `(a,b,d) determines c` and `(a,d) determines (b,c)`. They mean that we need to know either variables `(a,b,c)` or `(a,d)`. From an implementation perspective, this is just extra complexity.

Now, I observed that most typeclass instances use non-function type identifiers, so I can create a look-up dictionary for every tuple of names of type identifiers in the instance. This way I find a set of typeclass instances that can be used to resolve a typeclass. I also need to account for instances that have functioned as one of their parameters as this introduces nasty polymorphism and I decided simply to put those instances in a special group called universal instances that are always checked in the resolve typeclass function.

Resolve typeclass function

This function in Prolog terms is like taking a predicate of a rule and then trying to check if it can be satisfied. This can be quite slow if there are many rules, but thanks to the lookup dictionary created during preprocessing. We can narrow down the set of rules to check. Then we can iterate over all typeclass instances (rules) and apply the following algorithm:

1. Merge candidate instance with one
2. If merging succeeded, attempt to solve provisos.
3. If provisos were successfully solved, return the dictionary of resolved variables, otherwise continue iteration.
4. If all iterations have failed, fail.

This procedure is a bit slow, so I speed it up by creating lookup dictionaries that take as a key a dictionary and as a value a typeclass instance. This method can cut down search time significantly, but there are still rules that use polymorphic functions, and those cannot be preprocessed away so easily. Therefore, I call them universal instances and search through them every time. While, I could invent many heuristics to speed up this process. In general, this is still as difficult as the halting problem.

Solve numerical function

This function takes a set of size relationship provisos and a dictionary of variables. Then it converts those provisos into a set of equations on natural numbers. If the set is unsolvable this function raises an exception. Solving is done using the SymPy library. From an implementation standpoint,

I just need to traverse the tree of type identifier describing proviso and for each [name of proviso](#) TODO (check this) I convert it into a mathematical expression. Exact expressions were taken from the Bluespec reference manual.

Solve provisos function

As established earlier, provisos can be of two types, one is based on typeclasses and those can be resolved using `resolveTypeclass`, and the other ones are size-based and require solving sets of equations on integers and this can be done using `solveNumerical` function. One big problem is the lack of order in which provisos must be solved. Therefore, I handle this by creating a set of provisos to be solved. Then I repeatedly iterate over this set and remove solved provisos. If the set does not change after iteration, and it is not empty this means no further progress can be made, and the function raises an exception.

How those functions can are used

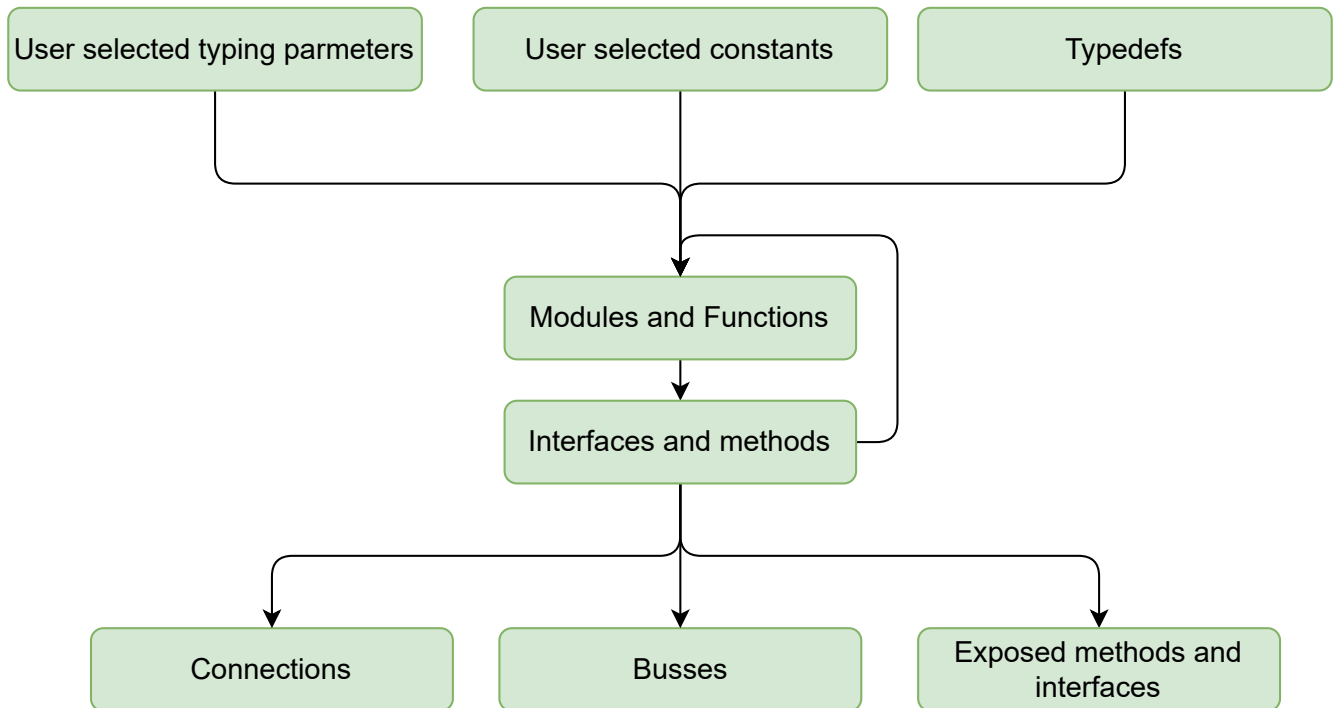
Using all of those functions, I can answer many questions like:

- What known thing can be used as *nth* argument of a given function ?
- Is it possible to call given function with given tuple of arguments ?
- What is the resulting type of function given certain arguments?

There are also more subtle uses of functions like `merge`, for example, when I need to deduce exact types of members of certain interfaces.

3.4 Building the top level module

Figure 3.5: Overview of the flow of data in the top level module



Above diagram shows flow of data during creation of top level module. User can specify all nodes of this diagram with exception of *Interfaces and methods* produced by *Modules and functions* as this is specified by typing information. Building the top level module can be broken down into few operations.

- Instantiating new modules.
- Connecting modules using `mkConnection` function.
- Connecting modules using busses, using functions of certain characteristics.
- Specifying exported interfaces and methods.

Operations of Instantiating new modules, Connecting and Creation of Busses are quite similar, so I will start by describing Instantiating new modules, and then how it fits with other operations.

3.4.1 Instantiating new modules

To instantiate new module, we need to do following things:

1. Convert inputted by user strings to type identifiers. This includes converting names of interfaces and it's members to theirs type identifiers.
2. Checking if arguments provided are of valid types.
3. Checking if typing information satisfies provisos.

4. Adding newly created interfaces and method to the namespace of known names.

From this list items 1 and 4 are resolved using few dictionaries that keep track of known names and their types. I also have a system that keeps track of which instantiated things depend on which other instantiated things. As for points 2,3 they are done using earlier described functions for type resolution.

3.4.2 Connections

Adding new connection is easy, it's basically addition of a new module. What's more interesting is that I keep track of all possible connections that can be made. This is later used to supply autocompletion in GUI(If user uses JSON based interface I can just print this data). Tracking of these connections is done in a naive way by just checking all possible $O(n^2)$ connections. A question that arises is "Can I do better?". While I don't have a concrete proof of this, because typing resolution maps quite nicely to Prolog, and Prolog is touring complete language, problem of finding all possible connection can be reduced to following problem.

Given n input values (number of interfaces and methods in instantiated modules) and k input programs (number of instances of typeclass `Connectable`) find all pairs of two input values for which at least one program halts with resulting value being true. This sounds like a lot of work, but thanks to those look-up dictionaries usually I need to check only few programs. I also can spread out the work over subsequent additions of new modules, and not recompute already known results. This way user will perceive lag roughly proportional to number of modules already instantiated. I also noticed that if with larger number of modules, PyPy's JIT compiler will step in, and it will produce better optimized code.

3.4.3 Busses

When it comes to busses most logic goes to synthesizing code for them, and not actually figuring out whether given data is valid because, for this I can fall back on logic designed to instantiate modules. To synthesize a bus I need to synthesize a routing function. Routing function is a function that takes a route and produces one hot vector, that specifies which slave is used for a given address. To create a routing function two things must be created. First I need to create a new function to be added to the database of known functions. To do this I just reuse my parsing tooling. As for creating code for it, I generate it by simply adding `if` statement for each address region(I don't need to do fancier logic as in hardware it will be optimized anyway). I also generate comments giving information about which slave occupy which address region.

3.5 Synthesizing the module

This is probably the simplest part of the project. I just interface over different kinds of things defined by the used and run some string formatting code. Even for busses, thanks to polymorphism in Bluespec I can quite easily write a routing function.

The main goal of the project is to be able to take a JSON with some data and synthesize a top module in the Bluespec language from it. This goal is more like a justification to create a language server like API. With this approach I have some simple core goal, but in the later sections we will see how thanks to all the work I have done, this API can be used to support GUI that aids the

user in creating a top module. Therefore, I will focus in this section on how the API works, and only later how is it used when reading JSON or as a backend to the GUI based app.

3.5.1 JSON interface

I decided that my “human-readable format” will be JSON. This is because wide support for JSON is already available in many languages. The only problem of JSON is that it’s more verbose than in some aspects Bluespec, and in theory it would make more sense to use Bluespec. However, this could introduce user confusion, as I’m not able to support arbitrary Bluespec features in my backend. My understanding is that supporting something only partially could lead users to think that my tool is broken, because it doesn’t match their assumptions about what it can do. While it can be argued that if I had written this project as extension of Bluespec compiler I could have supported all features of Bluespec (thanks to already written logic in the compiler). I explained at the beginning of this document, why doing this was infeasible for me as a part II project. At the same time using JSON allows me to support simplified semantics for things like bus creation, and features not found in Bluespec like specification of folders with additional libraries.

Structure of JSON file

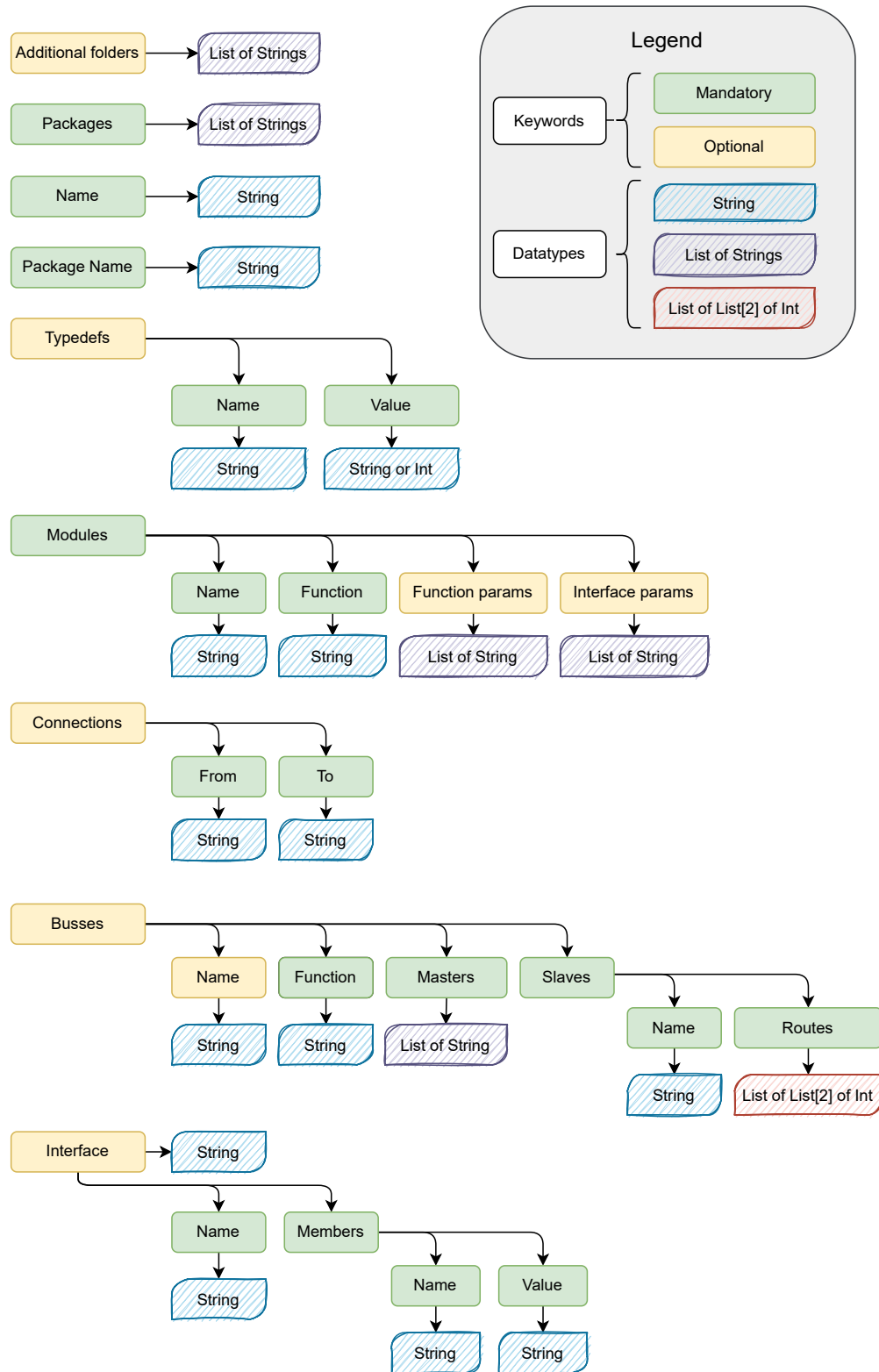


Figure 3.6: Disclaimer: In this diagram I'm using uppercase names but JSON file uses lowercase ones also all spaces are replaced with underscores. I'm also using s in name to signify a list of dictionaries with those keywords

More detailed description of keywords:

- `additional_folders` - list of folders that will be added to Bluetcl search path.
- `packages` - list of packages that will be used
- `name` - name of the module (defaults to “top”)
- `package_name` - name of created package
- `typedefs` - list of typedefs defined by a user
- `typedefs[i].name` - string starting with uppercase
- `typedefs[i].value` - integer or string (TODO check if string works)
- `modules` - list of modules or function to be instantiated
- `modules[i].name` - string being name of a module (used later to access such module)
- `modules[i].function` - lowercase name of a function used to create a module
- `modules[i].func_params` - list of strings that will be parsed, this can include numbers, names of interfaces of previously declared modules, `tagged <Keyword>` where `<Keyword>` can be replaced with struct creator.(TODO check if it works) This keyword is optional if function does not take any arguments.
- `modules[i].interface_params` - parameters passed to the interface of the result of the function, this keyword is optional if full type of the interface can be inferred from the type of function used to create a module, arguments, and provisos.
- `connections` - list of dictionaries containing definitions of connections
- `connections[i].from` - lowercase string with access path of a left side of the connection
- `connections[i].to` - lowercase string with access path of a right side of the connection
- `busses` - list of dictionaries describing busses
- `busses[i].name` - TODO remove this
- `busses[i].function` - lowercase name of a function used to create a bus
- `busses[i].masters` - list of access paths of interfaces to use as masters
- `busses[i].slaves` - list of dictionaries describing slaves
- `busses[i].slaves[j].name` - access path of a interface used as slave
- `busses[i].slaves[j].routes` - list of lists length two of integers describing starts and ends of address ranges on which slaves will be accessible. Starts of ranges are inclusive end are exclusive. (This data is used to generate routing function)

- `interface` - either a string describing an access path of interface to be exposed by module or a dictionary describing more complex interface made out of multiple member interfaces and methods
- `interface.name` - name of a type of the interface, if one is not present in loaded packages a new interface will be synthesized
- `interface.members` - list of members methods and subinterfaces of this interface
- `interface.members[i].name` - name at member will be accessible
- `interface.members[i].value` - access path of a method or a interface to be exposed at given name

Using JSON interface

If all I did was simply synthesizing a Bluespec file describing package from JSON, it would be mostly (one would need additional data to synthesize new interface) possible to do that just by doing simple string manipulation. I know this because my initial iterations were doing this. However, my tool can do much more than that. I to show those abilities I print additional data and verify correctness of given JSON. Bellow is rundown of this additional data, but to truly appreciate it I would recommend looking at how GUI is using this data.

- Dictionary of all possible connections better interface and subinterface and methods of instantiated modules.
- Dictionary of possible valid arguments for each function used to instantiate a module or a function.
- Dictionary of possible other masters and slave for each instantiated bus.
- Inferred types of every instantiated module and types of it's subinterfaces and member methods.

All this data can be useful when using complex modules from foreign packages, like a CPU core that doesn't have any direct connections, but It has many subinterfaces that can be connected to things like memory, external devices, etc.

3.6 GUI

Adding GUI was an optional goal mentioned in the project proposal. I originally intended to use a game engine, but Bluespec compiler is a Linux tool and game engines like Unreal Engine 4 or Unity3d are optimized to be used on Windows, so I decided against that idea. Instead, I found a library called React Flow, that provided simple API for creating graph based user interfaces. Therefore, I decided to create a GUI as a website. This was quite an adventure as It was effectively my first contact with web development. (I have done backend in .NET and C# for group project in last term, but it was mostly focused on writing queries to database and I had no contact with frontend development)

3.6.1 Backend

For backend, I had a choice between Django and Flask (as those are main python libraries for backend development). After reading about them in theory Flask was supposed to be better for small projects, like this one. However, I decided to use Django because I wanted to learn technology that will be more useful in the future. I followed official tutorial on Django and after initial pain with security adding new functionality was easy.

3.6.2 Frontend (React)

Frontend was written in React.js as you might have guessed from the name of the library I wanted to use. According to [Stack Overflow 2021 survey](#) React.js is the most popular frontend technology in 2021, so again I considered time spent on learning it to be good investment. One thing that took me quite a bit to get used to is everything being a function(React.js supports classes, but it's an old paradigm and I wanted to learn doing thing the “correct” way). This makes working with variables a bit tricky, as changes to state variable cause re-rendering of the whole component, and if variable is not a state variable, then it's value is going to be lost after re-rendering. If not careful is easy to cause feedback loop causing infinite re-rendering and subsequent crash of the application.

3.6.3 Frontend (React Flow)

Vocabulary

- Handle - Is a component of node that is used to start or end a connection. TODO include a image of a handle.

React Flow from developer perspective requires defining nodes (they can be basically and arbitrary React components), and some metadata about displaying a graph that is things like how edges look, how user can add new nodes, etc. The exposed API is quite simplistic. For example position of a handle is constrained to the middle of side of a node, and this library will generate CSS to put it there. Therefore, I needed to write some finicky CSS to align handle with text displaying its name. [TODO improve this sentence.]

Another thing that is problematic about React Flow is that it will cause node component to update every time it is moved. What I found is that if you wrap larger subcomponents of a node in `memo` from React, then performance stays quite good even if multiple nodes are moving as React will cache rendered subcomponents.

On the bright side after I figured out workarounds for those issues, I was left with many other things being handled by library. This included automatic graph formatting, and pretty Bézier curves for edges.

3.6.4 Frontend (MUI)

Default HTML buttons, and text boxes are ugly. So I decided to use React UI component library called MUI (previously Material UI). I don't have much to say about it except that use of it was frictionless, at least in my humble opinion it made my application look modern. I'm mentioning this as one of my complaints about Intel's platform designer was that it's UI is decades old.

3.6.5 Functionality

After teasing who I did things let's talk about features of my GUI. Using this GUI user can do following things.

- Instantiate new module.
- Connect two interfaces.
- Create a bus.
- Select methods and interfaces to be exported as a part of created module.
- Look at created Bluespec file.
- Run compilation followed simulation and look at output of both.

To do all of those things, user will need to interact with graph building tool. Bellow is overview of components user can choose from that will describe the top level module.

Instance node

This node is most important as using it user can instantiate new modules, convert between interfaces, and use functions in general. It can be in 3 main stages. During fist one it's an empty vessel using which one can specify it's name and function/module that will be used. In a second stage user specifies inputs and typing information. Finally after clicking *confirm update* node transitions into stage three and interfaces and methods of created object become usable.

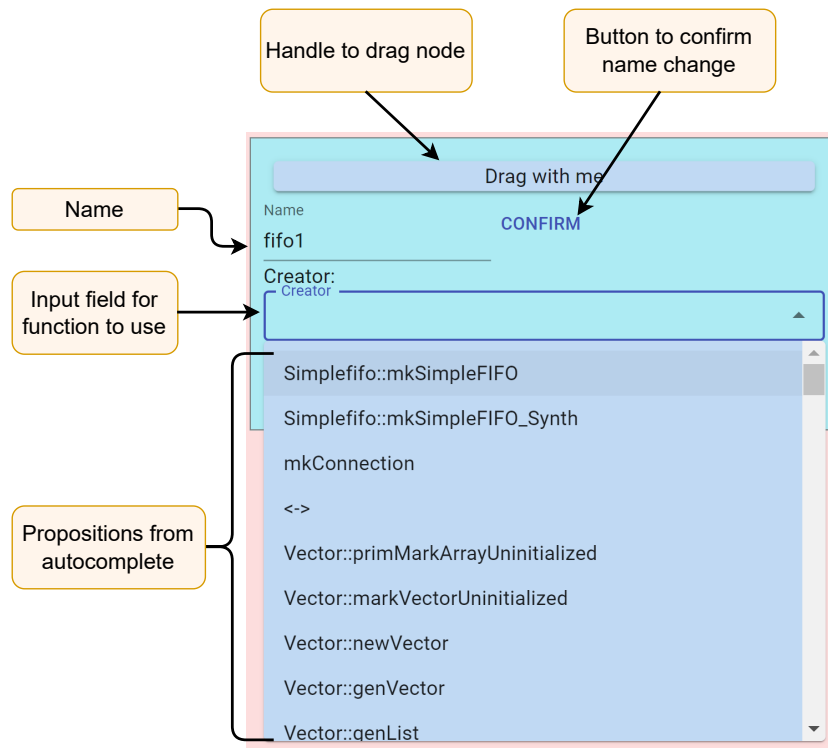


Figure 3.7: Stage one of node used to instantiate new modules

At stage one we will see the first field with autocompletion, it's the simplest one and simply lists all functions/modules user can use. It contains stand-alone functions as well as functions found as part of typeclasses. Node will transition to stage two when one of those its picked. At stage two user can, specify typing information. To do this they are presented with a tree like input structure. Values inputted there can be of either uppercase string describing type identifier, integers or lowercase strings that will be used as a variable that are meant to be resolved by backend. There is also a list of inputs that go directly to the function. Those inputs contain fields with autocomplete that will filter all typedefs found in loaded packages, interfaces and functions found in previously initialized nodes. There are also handles for that allow for drag and drop style of creating connections. If user choses value via text field, that is a member of previously initialized node then connection will be generated automatically. After user is satisfied with their choices they press *confirm update* button. At stage three user is presented *information about error of lack of them* this can take bar can be in one of three states, its functionality is shared between nodes.

- Waiting for confirmation - When backend is processing data.
- Instantiation successful - If everything went smoothly.
- Error with text that tries to describe and error that occurred. This usually occures if after user supplied data that is not able to satisfy provisos of function used.

In case of lack of errors user is given an overview of the interface of the object created with it's subinterfaces and methods and their types. Attached to them are handles for starting connecting using DnD.

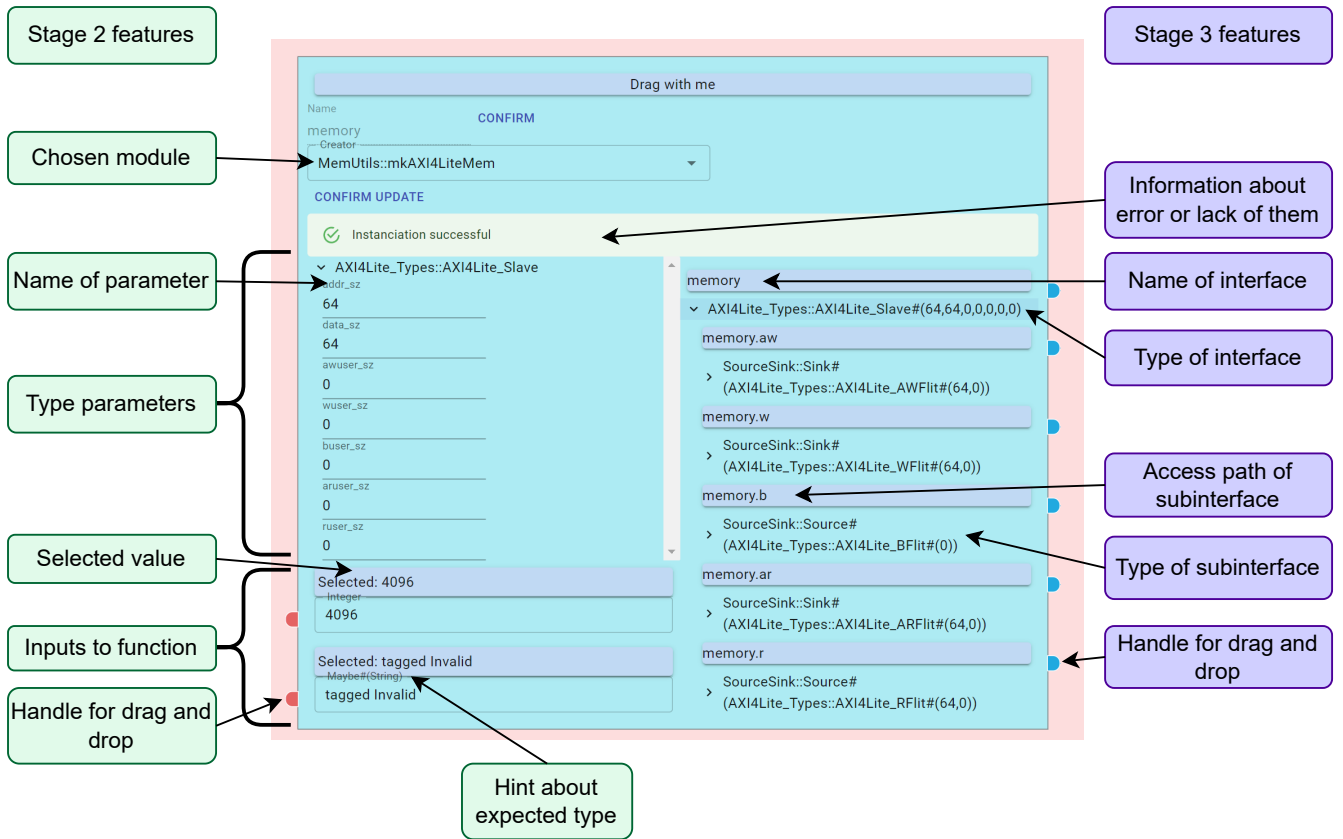


Figure 3.8: Fully instantiated node with descriptions of features, divided into stages two and three

Connection node

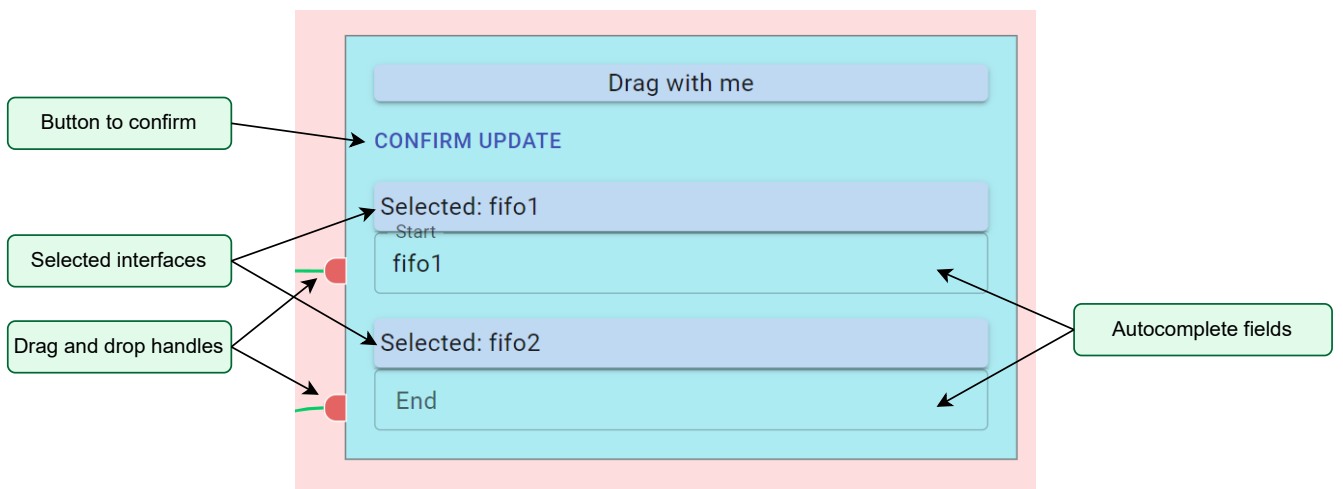


Figure 3.9: Connection node, connecting two fifo modules

This is the simplest node it contains two inputs similar to ones found in the instance node. The key change hides in autocomplete, *start* field will show all modules that can be used to start connection, whereas *end* will only autocomplete to things that can be connected to the thing specified in the start field. Power of this node lies in the fact that it uses typing information in the

backend, and it will work with new any types as long as there is applicable instance of typeclass `Connectable` which should be common as it's part of the standard libraries.

Bus node

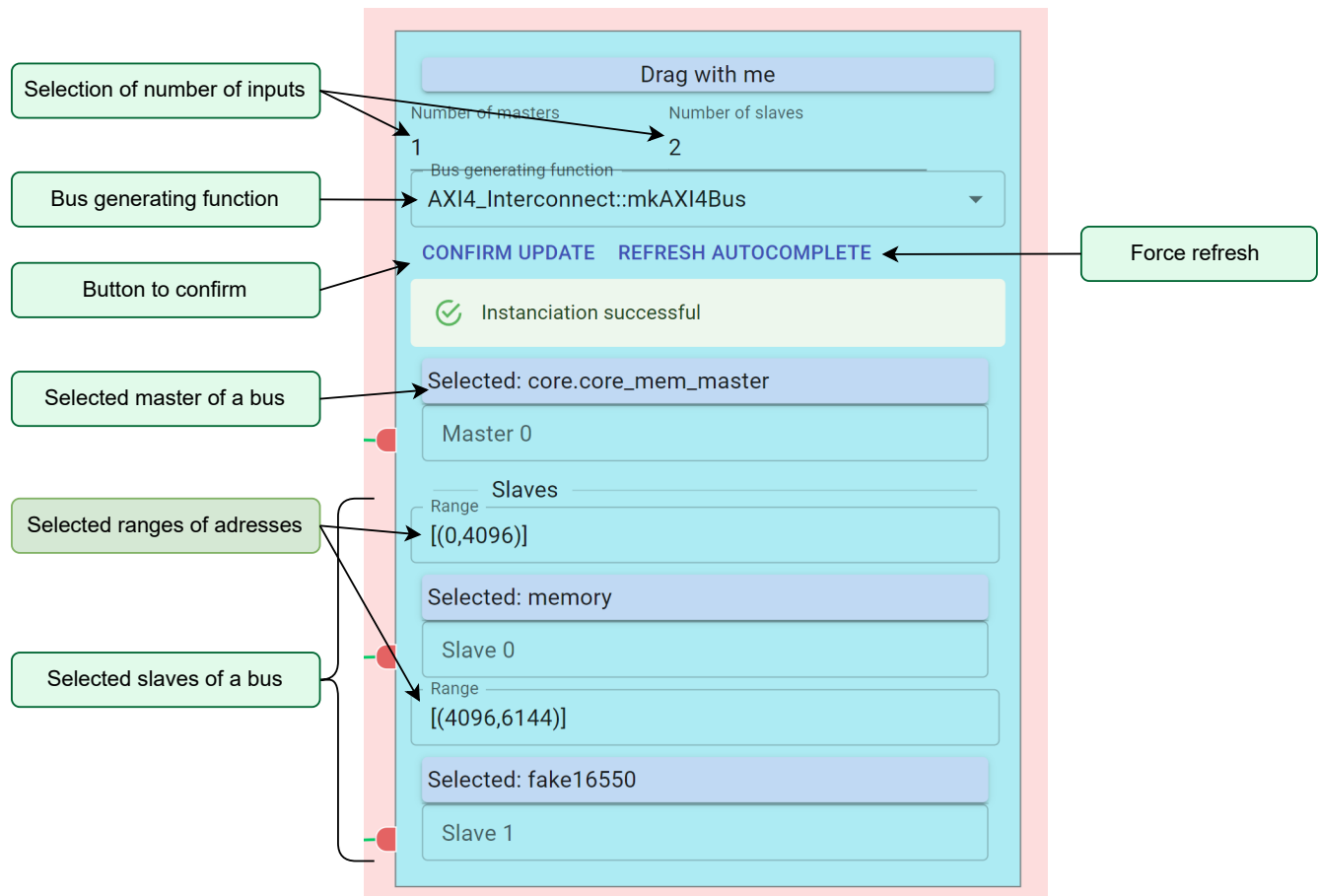


Figure 3.10: Bus node connecting one master and two slaves using AXI4 interface

Similarly to connection node most of the components were reused, and interesting are details. This node works only with a subset of functions, that have specific something (TODO find a word to use). Fortunately used don't need to look for those functions autocomplete field will only show you valid ones. (It might be possible that some non bus function will slip through as I don't want to be overly aggressive and require things like having "bus" in the name of the function, but from my testing my simple check narrows down options to less than ten from few hundred found in all packages, I worked with.).

Autocompletion for masters and slaves is a bit more nuanced as it has two levels of precision. At first level it will show all masters and slaves that can be used with chosen function, but not necessarily together. However, after first successful initialization of a node (using *confirm update*), options for masters and slaves will be narrowed down to only ones that can work with masters and slaves selected during first initialization. This node also contains fields to select ranges of addresses on which slaves will be visible.

Exported interface node

Drag with me

Name of top-level module: top

Name of new interface: Subset

Number of exported things: 2

If only one thing is exported it's name and name of interface is irrelevant and can be left empty. Top level will assume it's type

CONFIRM UPDATE

Instantiation successful

Export name: cpu_imem

Selected: core.cpu_imem_master

Exported Interface/Function

Export name: interrupt_sources

Selected: core.core_external_interrupt_sources

Exported Interface/Function

Figure 3.11: Overview of features of a node allowing to export things from a created module

This node is allows user to specify name of a function creating a module they designed. Name of the interface of which will be used by this module if it will need to export more than one thing, and most importantly all the exported interfaces. If a name of an interface is already known,

system will try to populate members of that interface with members provided, if the name is not known it will simply create a new interface using all the typing data from the members. It also contains a simple feature which will generate name based on access path if user won't provide one.

The intention behind this node is that one might want to create their system partially in Bluespec and partially in Verilog as there are vast libraries of IP written in Verilog. Using this node one can create a new interface on a fly that does not have much use in Bluespec world, but if transpiled to Verilog it will only contain things user specified, making process of importing it into tools like Platform Designer much easier.

Futher work (TODO this can be removed)

It's worth keeping in mind that for GUI like this there is multitude of things that can be added. Here is list of things that could be added:

- Ability to add comments that partition the graph (like in UE4).
- Inputs node to be able to select inputs to a module. Therefore allowing for parametrization.
-

Chapter 4

Evaluation

Throughout working on this project I have learned a lot about QSys and Bluespec, and how they interact. This means that I will need to adjust slightly it from the project proposal to better represent problems that are solved. Firstly my understanding is that Platform Designer tool is shared between QSys and QSys Pro.

4.1 Overview

My evaluation will be divided into two sections each with three subsections. First section will compare text file formats of QSys and mine and second section will compare GUI of QSys and mine. Each of the subsection will go through three examples:

- First one will show simple example of two connected fifos, and difficulty of connecting a third one.
- Second will show an example of simple system-on-chip (SOC). With one Flute core, memory module and fake interface with outside world, all connected by a bus.
- Third example will two AXI4 masters and two AXI4 slaves connected by a bus.

First two examples will produce valid devices, but they won't do anything interesting. Last example will produce a working top level module that will demonstrate practical use of the tool.

4.1.1 Polymorphism disclaimer

Before we even create text files that we are going to compare we talk about an issue when working with QSys and Bluespec. That is the fact that Bluespec can't generate Verilog code of functions that are polymorphic. Nor there is support for Bluespec in QSys nor Xilinx Vivado. My understanding is Verilog type system is not rich enough to express things that would be needed to support complex polymorphism found in Bluespec. Therefore, any module used with QSys will need to be converted into it's synthesizable version stripped of any arguments and transpiled into Verilog. One might argue that this is an issue with Bluespec and not QSys, but I would argue that it is still a problem that needs fixing I have solution for It, that is working with Bluespec natively rather than Verilog files.

4.2 Example 1 - Connecting FIFOs

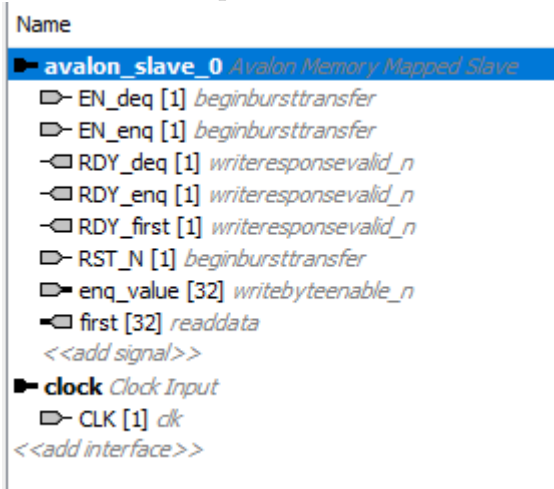
In this example our goal is to create two connected FIFOs and then modify the file to connect a third one. FIFOs that I will be using have a copy of FIFO interface found in default Bluespec libraries.

- Enq - If queue is not full then enqueue the data.
- Deq - If queue is not empty then remove first element.
- first - If queue is not empty then return first element.

In Bluespec there are provisos attached to those functions, and if in given cycle user will want to perform illegal operation (like dequeue when queue is empty) then rule in which this operation would be performed simply won't fire in that cycle. In verilog there is no notion of such provisos nor method so everything needs to be synthesized to wires to aid this process Bluespec uses Ready-Enable micro protocol. Every method has Ready and Enable wires. When Ready is high, then method is allowed to be executed. When Enable is high, it signifies that method must be executed.

4.2.1 Platform Designer

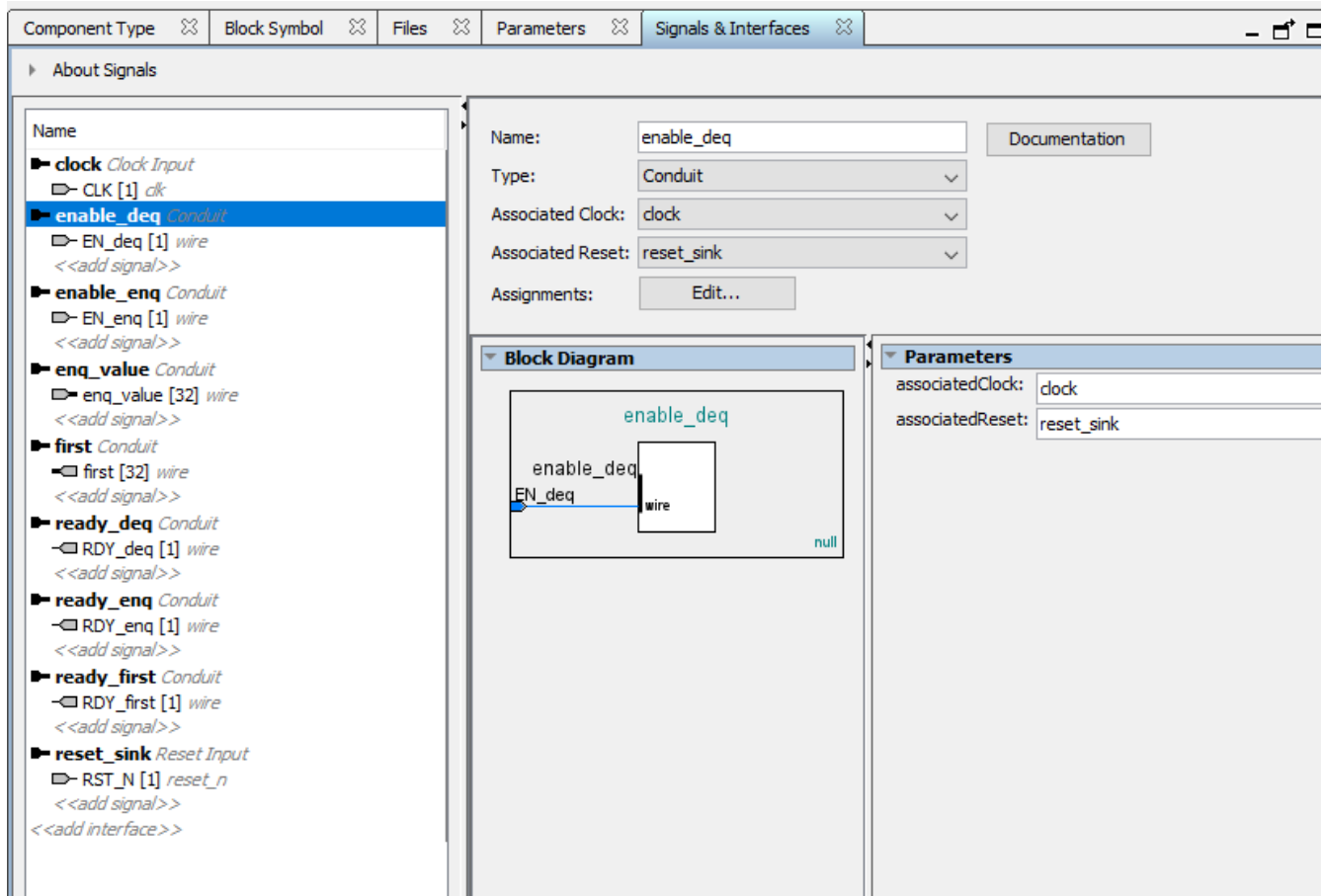
To create a system with two connected FIFOs, we first need to import them into platform designer. Here we encounter first problems. As mentioned before, Bluespec can't generate Verilog code of functions that are polymorphic. So we must decide on the width of the data stored in such FIFO before transpiling to Verilog. After analysis of Verilog file by platform designer it decides that all wires found are parts of Avalon slave interface. This is an obvious error, and we need to correct it.



This is unfortunately easier said than done. There are few options of assigning those wires, but neither of them perfect. Firstly we will need to choose a type of interface those will represent. We have over 30 options to choose from, but most of them are variants of larger protocols like AXI4. The only option that will make sense for us is Conduit, which basically means bank canvas. We could create a conduit for each method but after consultation with my supervisor he recommended creating a conduit for each wire. So I did as he recommended, and this produces following set of interfaces. This requires quite a bit of manual work, listed below.

- Create 8 new interfaces (QSys figured out what to do with clock wire on its own, so we have one for free) this includes giving them a name and assigning a reset signal.

Figure 4.1: After assigning wires and Interfaces



- More each wire to its respective interface and change signal type of each wire to something generic like wire from type given to it as part of Avalon slave interface.

With all of those wires we can connect obvious things together like `first` to `enq_value` and clock signals, but then we run into a problem that with all wires that represent Ready-Enable micro protocol because we would like to enable enqueue in second queue only if both wires `ready_deq` and `ready_first` are high in the first queue, otherwise because we don't know how our FIFO is implemented we might run into a problem where we might get bugs(for example we lose data because we `ready_deq` became high cycle faster than `ready_first`, or in opposite scenario we would read stale/corrupted data). Unfortunately we can't create such logic using platform designer and one way of solving this would be to create special connector module that would do this logic(I'm going to come back later to explain why this approach might be infeasible in general case).

4.2.2 My tool

Using my tool whole phase of importing and setting up wires can be skipped. Connecting is as simple as creating a new node connection node and selecting chosen FIFO's interface. Extending two connected FIFOs to three is as simple as adding one more FIFO and one more connection node.

Figure 4.2: Three connected FIFOs using my tool



4.2.3 Using text format to do same things and Qualitative metrics

To count number of tokens I'm using following regexp `(\w|\.)+`. This regexp will match sequences of alphanumerical characters with dots and underscores. (TODO draw graphs) Raw numbers: 431 tokens to import FIFO, 347 for two connected FIFOs, and 415 for three connected FIFOs. My solution: 31 for two connected FIFOs and 43 for three connected FIFOs.

4.3 Example 2 - Simple System-on-Chip

In this example we will create a simple system-on-chip (SOC) with one Flute core, memory module and fake interface with outside world, all connected by a bus. This example is designed to even out playing field by making all connections use AXI4 interface which is understood by platform designer.

4.3.1 Platform Designer

Again we start with importing our modules. This time we need to import two of them, one for fake 16550 interface and one for flute core. As for memory module we will use one from QSys library to save time and effort.

Unfortunately platform designer can't figure out how to organize wires into interfaces like AXI4 master or AXI4 slave. So we need to do it manually. This wouldn't be so painful if not for the fact that we need to set type of each wire according to it's name(for example `cpu_imem_master_awready` has type `awready`). Each of those AXI4 interfaces has 37 wires, and if we account for all other interfaces of Flute core we get 150 wires that individually need to be assigned to interfaces. Similarly, our fake 16550 interfaces also uses 40 wires in total. When assigning this through GUI it took me around 16 minutes to click through all the assignments. (For context tools like Xilinx Vivado have heuristics that are able to do a lot of this automatically, but my understanding is that still this works only for certain known interfaces)

After initial pain of importing modules we arrive at the place where Platform designer shines. It is able to show possible connections between AXI4 masters and slaves, on a two 2 grid-like layout. Then connecting two together is as simple as clicking cross-section of two wires on the screen.

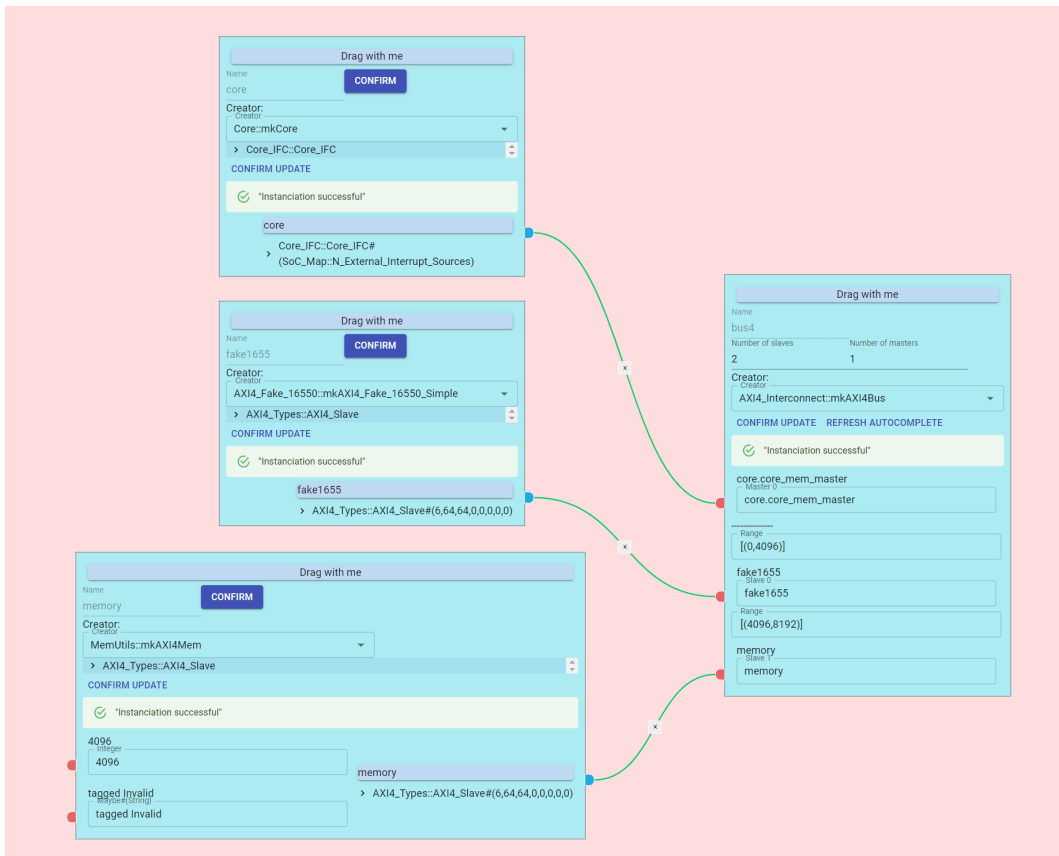
4.3.2 My tool

Again using my tool we don't need to do anything to import modules, as all the typing information is provided by the package. To connect those together we create a special bus node, select function for creating a bus, select number of masters and slaves, then assign each accordingly. While I'm unable to show nice 2d grid of possible connections, I have two-stage autocompletion. Stage 1 autocompletion becomes available after user selects function to create a bus. Such functions can be polymorphic, so I show all interfaces that might be valid individually, but not necessarily together. Stage 2 autocompletion becomes available after user selects at least one master and at least one slave, and clicks confirm update. After that bus is locked to specific type of masters and slaves and autocompletion narrows to show only masters and slaves that will work together.

Figure 4.3: Connected Sys-on-Chip using Platform designer

Use	Connections	Name	Description
<input checked="" type="checkbox"/>		clk_0	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
<input checked="" type="checkbox"/>		fake16550_0	fake16550
		clock	Clock Input
		reset_sink	Reset Input
		altera_axi4_slave	AXI4 Slave
<input checked="" type="checkbox"/>		fluteCore_0	fluteCore
		clock	Clock Input
		avalon_slave_0	Avalon Memory Mapped Slave
		en_dm_dmi	Avalon Memory Mapped Slave
		core_mem_master	AXI4 Master
		reset_sink	Reset Input
		cpu_imem_master	AXI4 Master
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...
		clk1	Clock Input
		s1	Avalon Memory Mapped Slave
		reset1	Reset Input

Figure 4.4: Connected Sys-on-Chip using my tool



4.3.3 Other differences

There is a number of subtle differences that are worth talking about, but were not pronounced by this example.

Platform designer has ability to automatically convert between AXI4 interfaces and Avalon ones, and also within those families it allows for some degree of flexibility when parameters don't match exactly. When I create busses I'm limited by typing system, and way functions for creating buses were defined. Therefore, if user wants to do similar conversions they would need to use some functions and perform those conversions by themselves.

Another slight difference is the fact that in Platform designer, each master has its own bus, and while slaves can be connected to any number of masters they need to be connected individually.

4.4 Example 3 - Working AXI4 masters and slaves

This example