Summary of the Work of GAELS Project

Wei Song, September 2014

Contents

1. Introduction

1.1. *EDA Development for Asynchronous Designs*

Current EDA tools concentrate on synchronous designs. The assumption of a global clock allows EDA tools to analyse and compile synchronous designs using static timing analysis. Asynchronous designs have no such benefit of clocks. To analyse the behaviour of an asynchronous circuits, the tool needs to expand the state space of the whole design, which increases exponentially.

Rather than adopting the synchronous hardware description language, the available asynchronous EDA tools require users to provide designs in transition based languages, such as pertify and Balsa. These languages describe the circuit behaviour in transitions and handshakes.

The current asynchronous EDA tools are not user friendly for synchronous designers, since the tools require the circuit designers to understand the underlying theories of asynchronous circuits before writing an efficient asynchronous design.

1.2. *Asynchronous Verilog Synthesizer*

The asynchronous Verilog synthesizer (AVS) is a tool aiming to automatically convert a large-scale asynchronous RTL design into a globally asynchronous and locally synchronous (GALS) design. In this way, the original hardware designers do not need to understand asynchronous circuit while the tool can provide a GALS design which explores the benefits of low power and variance tolerance.

To achieving this target, the tool needs to understand the behaviour level data flow from the RTL level design, which in fact is a reverse engineering process. Utilizing the extract data flow, the tool is expected to automatically partition the design and generating proper handshake channels.

The current tool is still far from this target. The current AVS tool can compile large scale RTL designs, convert them into internal abstract formats, extract controllers, extract data paths, and do preliminary partitions. The key component which is still missing is how to estimate or back-annotate data rate to the data paths extracted.

2. Verilog Compiler

To analyse and potentially re-synthesize an existing RTL design, the design itself have to be parsed and converted into abstract syntax trees which can be understand and operated by software. Verilog and VHDL are the two most utilized languages in RTL designs. Choosing Verilog as the target language is based on two observations: Verilog shows more potential in behavioural level designs (considering System-Verilog) and the Grammar itself is easier to parse than VHDL due to the limited number of types. However, this decision also leads to the current dilemma that the OFDM transmitter (written in VHDL) provided by IHP cannot be analysed.

2.1. *Verilog Parser*

Although parsing Verilog is already easier than VHDL, it is still far from a straight forward task. The current AVS tool [1] understands most of the synthesizable syntax but not all of them can be elaborated and converted into the internal abstract graphic format, signal-level data flow graph.

Verilog parsers are not new and many tools have them, including open-sourced ones. The reason to re-invent the wheels is due to the difficulties in reusing existing parsers. The best open-source parser available right now is the one in the Icarus Verilog tool [2]. However the internal format of “vvp” is not necessary (even introducing new problems) and the parser/elaborator for simulation is quite different with the parser for synthesis. Learning the skeleton of the grammar files, a decision to rewrite a new parser was made.

Converting a multi-file and hierarchical Verilog RTL design into abstract syntax trees (ASTs) needs several steps. The order of these steps is important for the correct AST generation and more steps will be required to parsing future Verilog standards, such as the System-Verilog. The order used in AVS is described as follows:

1. Preprocess all Verilog files separately to handle Verilog macros.
2. Parse each Verilog file into modules with a list of literal statements.
3. Pass each module again to analyse each statement: store signal declarations in multiple signal databases and link each declared signals with all its fan-in and fan-out in the block.
4. Starting from the top modules, dynamically assigning the parameters of each sub-modules and rename it with the parameter suffixed.
5. For each renamed module, reducing all signals (mostly parameters) with constant values.
6. Find out all sub-modules in the renamed module and iterate again from step 4 until no sub-module found.

2.1.1. Verilog Preprocessor

A pre-processor is responsible to understand and unfold all the marcos used in a Verilog files. The Verilog pre-processor in the Verilog-Perl tool [3] has been adopted. Located in `[AVS]/preproc/`, it reads every Verilog file and generate a new file with all macros unfolded expect `` `line `` which is used to trace the original line count when reporting an error.

2.1.2. Verilog Parser

A Verilog parser reads a preprocessed Verilog file and converts it into preliminary ASTs, where all statements are stored in lists without any analyses (assignments, declaration, etc). The parser used in AVS is built from the Flex lexical analyser [4] and the Bison parser generator [5]. Located in `[AVS]/averilog/`, the parser generates ASTs directly in C++ objects. Nearly all synthesizable features of Verilog can pass the parser but not all of them will be converted to ASTs. Some features are silently neglected (but most of the parts being neglected are the non-synthesizable statements).

One problem of the parser is the unclear error report system. Although usual coding errors would be reported with the exact line count, the error report for unsupported features is difficult to understand. One of the reasons traces back to the LALR(1) parsing behaviour of the Bison parser. A more user friendly parser may be generated by ANTLR [6].

2.1.3. Verilog Abstract Syntax Tree (C++ STL)

The ASTs are implemented using C++ classes with the help of STL components in `[AVS]/netlist/`. The derivation of classes is listed as follows:

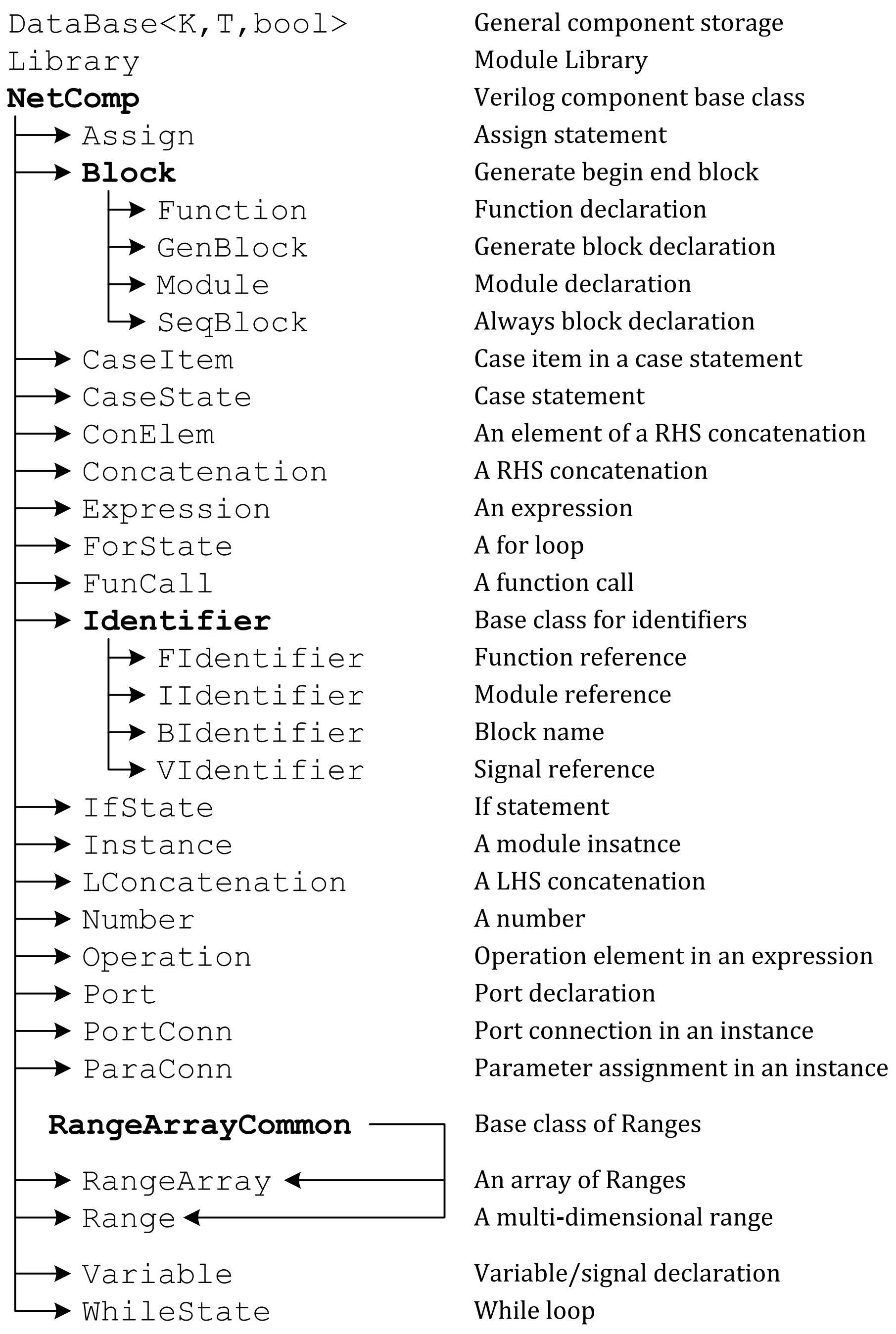
DataBase<K,T,bool> General component storage
Library Module Library
NetComp Verilog component base class
Assign Assign statement
Block Generate begin end block
Function Function declaration
GenBlock Generate block declaration
Module Module declaration
SeqBlock Always block declaration
CaseItem Case item in a case statement
CaseState Case statement
ConElem An element of a RHS concatenation
Concatenation A RHS concatenation
Expression An expression
ForState A for loop
FunCall A function call
Identifier Base class for identifiers
FIdentifier Function reference
IIdentifier Module reference
BIdentifier Block name
VIdentifier Signal reference
IfState If statement
Instance A module insatnce
LConcatenation A LHS concatenation
Number A number
Operation Operation element in an expression
Port Port declaration
PortConn Port connection in an instance
ParaConn Parameter assignment in an instance
RangeArrayCommon Base class of Ranges
RangeArray An array of Ranges
Range A multi-dimensional range
Variable Variable/signal declaration
WhileState While loop

Fig. 2.1. The class heritage tree of netlist components.

A container class, such as a Module, always block or even a right-hand side (RHS) concatenation, contains other AST component, or netlist components. The containing relationship between the netlist components is depicted as:

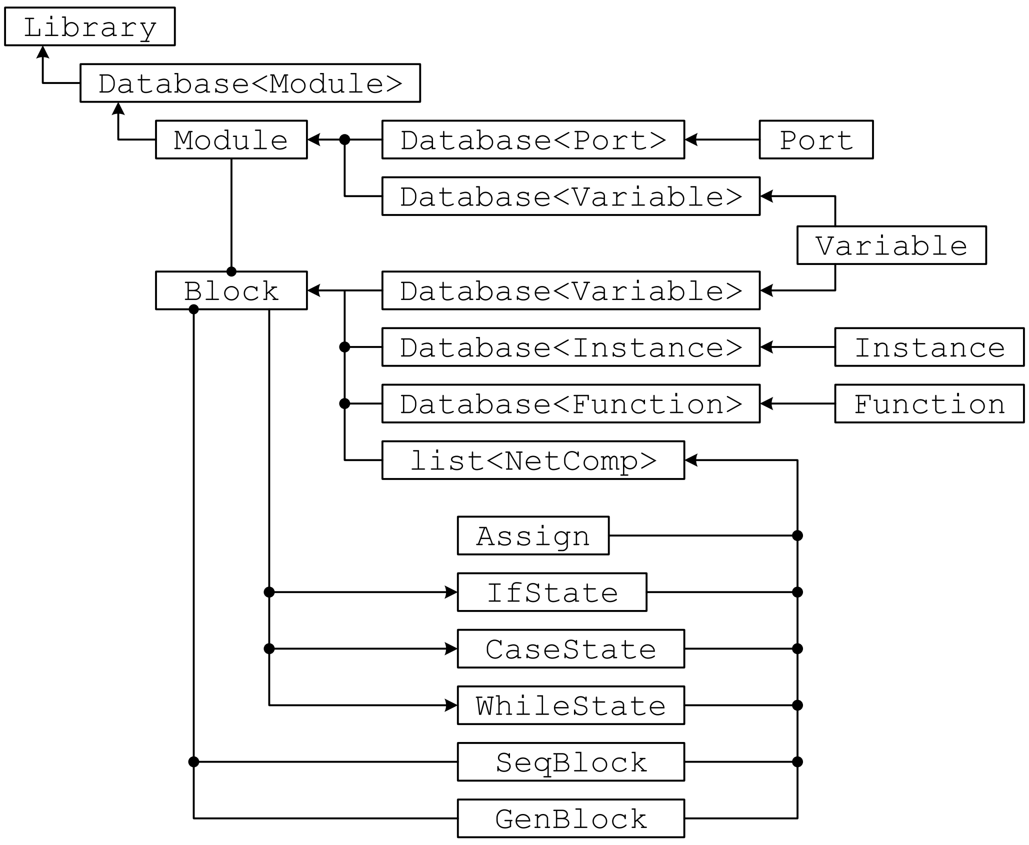

Fig. 2.2. Relationships among netlist components (statement level)

This figure reveals only the statement level relationship between netlist components. The class relationship inside statements are rather complex. Below shows the classes related to the assign statement. For other statements, please resort to the source codes in `[AVS]/netlist/`.

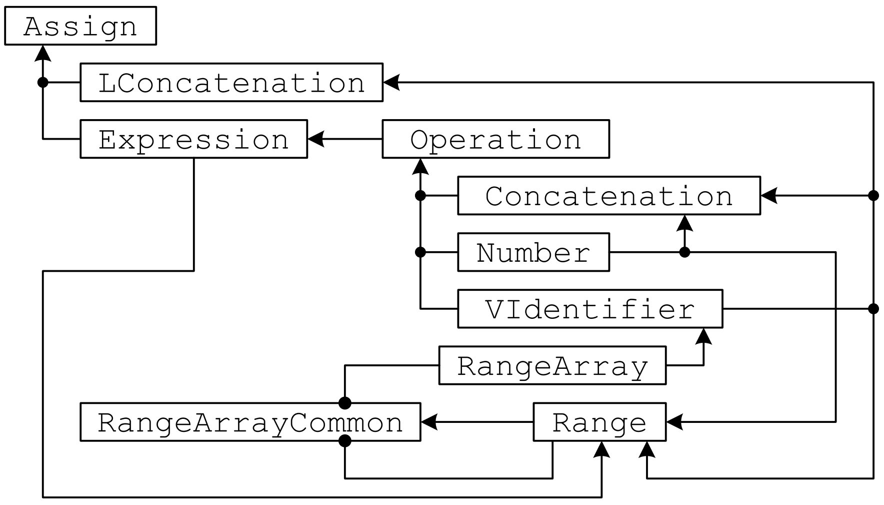

Fig. 2.3. Sub-components of an assignment

2.1.4. Design Elaboration

Elaborating or connecting a hierarchical design from the preliminary ASTs is not straight forward. Verilog supports initiating module instances before declaration and compile-time configuration using parameters. All these user-friendly features require the elaborator to collect enough information through traversing the hierarchical design multiple times, or multi-pass elaboration. In each pass, the tool must try its best to reduce the constant expressions and unfold structures reconfigured in previous pass.

2.1.4.1. Elaboration in parsing

Taking `Module` as an example, the elaboration in paring a module classifies all statements and stores them in separated databases. The AST `Module` class has 5 internal databases and 1 list of statements. In the 5 databases, the databases for parameters and ports are defined for `Module`, and the databases for variables, instances and functions are derived from `Block`. Initially, all statements, no matter a declaration or an instance initialization, are stored in the list of statements. After all statements are parsed, the elaborator in parser revisits the list to classify different types of statements:

1. For variable (`reg`, `wire`, `integer`) declarations, remove the declaration from the list and add a new record in the variable database.
2. For instance initialization, remove the initialization from the list and add a new record in the instance database.
3. For function declarations, remove the declaration from the list and add a new record in the function database.
4. For parameter declarations, remove the declaration from the list and add a new record in the parameter database.
5. For port declarations, remove the declaration, double check the port list and add a new record in the port database.
6. For all unnamed blocks, promote all internal statements to the current level and classify the new statements.

After the classification process, all statements remaining in the list are `always` blocks, `assign` statements or generation blocks which cannot be unfolded yet.

The last two operations of the elaborator in parsing are (1) linking all variable identifiers used in expressions to the declaration stored in the variable and parameter databases and (2) linking all internal blocks and instances with the module block containing them using a father pointer.

2.1.4.2. Dynamic parameter calculation and loop unfolding

Dynamic elaboration in linking the hierarchical design aims to eliminate all parameters and unfold loops parameterized by these parameters.

The elaboration process happens in several steps:

1. For each instance defined in an elaborated module, calculate the local values of the parameters and copy the sub-module with a new suffixed name.
2. In the newly copied module, calculate and replace the parameters with local values.
3. Traverse the new module. For each generation block, unfold all `for`/`while` loops and promote the unfolded statements to the higher level.

Reusing this dynamic elaboration until no instance found, the whole design should have all parameter databases empty. The only item in the list of statements in each sub-module should contain only `assign` statements and `always` blocks.

2.2. Signal-level Data Flow Graph

Instead working on the C++ AST, a more abstracted graphic representation of the RTL designs is extracted and used to analyse the circuit. The new representation is named signal-level data flow graph (SDFG), which is a directional multi-graph high-lighting the relation between Verilog signals.

2.2.1. Definition

The formal definition of a signal-level data flow graph (SDFG) can be described as:

A signal level data flow graph is a directed multi-graph denoted by a six-tuple *DFG* = (*V*, *A*, T_A, F_A, T_V, F_V), where:

V is a finite set of nodes representing the parallel components in the AST;

$A \subseteq V \times V$ is a finite set of arcs denoting the connection between components;

$T_A \in \{control, data, clock, reset\}$ is a finite set of available arc types;

$F_A : A \to T_A$ is a function mapping types to arcs;

$T_V \in \{seq\_block, combi\_block, i\_port, o\_port, module\}$ is a finite set of available component types;

$F_V : V \to T_V$ is another function mapping the types of all components.

For the following Verilog module, which is a traffic light controller:

```
module traffic(clk, rstn, red, green, yellow);
   parameter R = 0;  // red state
   parameter YR = 1; // yellow state after red
   parameter G = 2;  // green state
   parameter YG = 3; // yellow state after green
   input  clk, rstn;
   output red, green, yellow; // light control
   reg [1:0] state;     // state machine
   reg [1:0] state_nxt; // next state
   reg [5:0] cnt;       // second counter

   always @(posedge clk or negedge rstn)
     if(~rstn)
       state <= R;
     else
       state <= state_nxt;

   always @(state or cnt) // next state
     if(cnt == 0)
       case(state)
         R:  state_nxt = YR;
         YR: state_nxt = G;
         G:  state_nxt = YG;
         default:
           state_nxt = R;
       endcase // case (state)
     else
       state_nxt = state;

   always @(posedge clk or negedge rstn)
     if(~rstn)
       cnt <= 0;
     else if(cnt == 0)
       case(state)
         R:  cnt <= 2;
         YR: cnt <= 49;
         G:  cnt <= 4;
         default:
           cnt <= 49;
       endcase // case (state)
     else
       cnt <= cnt - 1;

   assign red = state == R ? 1 : 0;
   assign green = state == G ? 1 : 0;
   assign yellow =
     (state == YR || state == YG) ? 1 : 0;
endmodule
```

The generated SDFG can be depicted as:

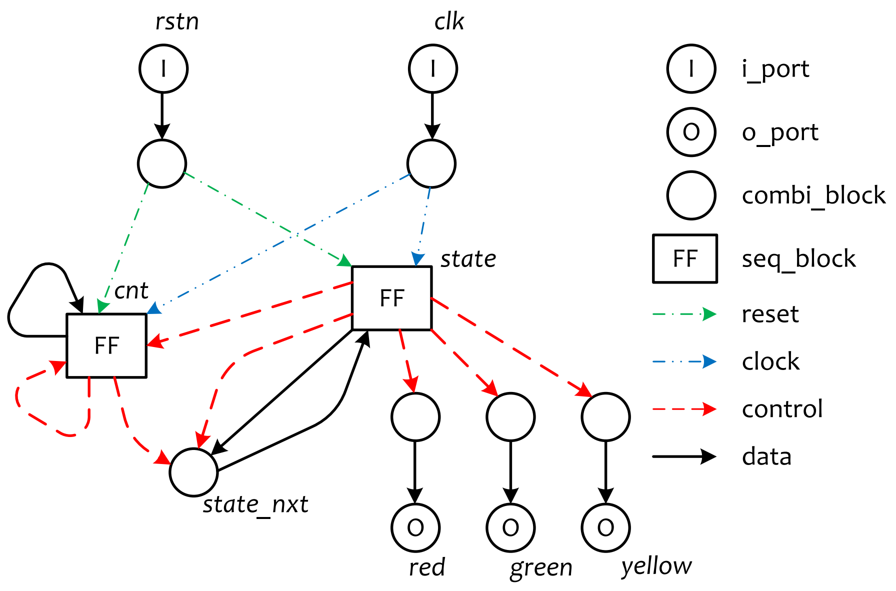

Fig. 2.4. Sub-components of an assignment

All sequential `always` blocks (*seq_block*) are depicted as rectangles labelled “FF”. Continuous assignments and combinational `always` blocks are denoted by blank circles. Input and output ports are represented by circles labelled with “I” and “O” respectively. If there is a module entity, it would be drawn as a rectangle labelled “Module”. The corresponding signal names are noted besides the nodes. Arcs are used to demonstrate the relations between nodes. If the value of a signal is determined by another signal through an assignment, such as `state_nxt` assigns the value of `state` in the first `always` block, a *data* arc is used to link them together, just as the bold black arrow from `state_nxt` to `state`. All the signals appeared in the condition statement of `if`, `case`, `for`, `while` and the condition operator “ `? :` ” are considered as control signals affecting the assignments of the node where they are used. In the traffic light controller, since `state` is used in both `case` statements as the case condition, two corresponding *control* arcs (a red dash arrow) is drawn from `state` to `cnt` and `state_nxt` respectively. The arcs for *reset* and *clock* are depicted in different arrows as well.

2.2.2. Relation Tree

To generate an SDFG, each `always` block is analysed to reveal the relation between the signals being assigned in the block and the signals read in the block. The AST of the block is first converted into a forest of relation trees.

The relation tree is a temporary data structure. In the forest generated for each `always` block, a relation tree is produced for each of the assigned signal. In the tree, a map is used to keep tracking the signals affecting the value of the assigned signal and the relation of this relation.

The forest is generated gradually through traverse the AST. When the whole AST is visited, all signals stored in the individual maps are translated into an arc in the SDFG.

The detailed implementation of the relation tree structure locates in `[AVS]/sdfg/rtree.hpp` and `[AVS]/sdfg/rtree.cpp`

2.2.3. Range (Interval) Calculation

Range calculation is commonly named interval calculus in math. It is important in Verilog as ranges, especially multi-dimensional ranges are used to describe multi-bit signals. It is found that range calculation is crucial in verifying multi-driver, no-driver, no-load, and parallel signal partition issues.

A range calculation library is provided in two formats: one is the AST range classes which support ranges with variables (`[AVS]/netlist/range.hpp; [AVS]/netlist/range_array.hpp; [AVS]/netlist/range_array_common.hpp`); and the other one is a complex calculation library for constant ranges (`[AVS]/cpprange`).

The cpprange library provides 3 types of ranges:
RangeElement: single dimension range such as [5:0].
Range: multi-dimensional range expression, such as [6:-3][7][1:0].
RangeMap: arbitrary calculation of multiple range expressions, such as [6:3][1:0] | [3:-2][5:4].

The supported operation types include:
Subset, overlap, connected, disjoint, equal, less, combine, hull, intersection, complement, etc.

For the detail information and examples, see https://github.com/wsong83/cppRange

2.2.4. Automatic Partition of Multi-bit Signals

One powerful implementation technique in Verilog is to using multi-dimensional array of signals in `generate` block and implement parallel processes or gates using parameters. For the following example:

```
reg [DW-1:0] i0, i1;
reg [1:0][DW-1:0] buf;
reg [DW-1:0] max, min;

always @(posedge clk) begin    // compare and swap
      buf[0] <= i0;
      buf[1] <= i1;
      max <= buf[0] >= buf[1] ? buf[0] : buf[1];
      min <= buf[0] >= buf[1] ? buf[1] : buf[0];
end
```

Although register array `buf` is declared as a single signal, it is used as two signals to buffer `i0` and `i1`. If `buf` is recognized as one signal, the generated SDFG would look as:

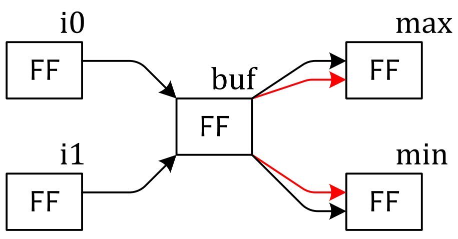

Fig. 2.5 SDFG without range partition

It seems OK, but the SDFG fails to reveal the fact that `buf` is used as two separated buffers. In more serious occasions, this ignorance of range leads to false combinational loops. By tracking the ranges of all registers on the left hand side of assign statements, the multi-dimensional or even multi-bit signals can be automatically partitioned. The SDFG generated with automatic range partition looks as:

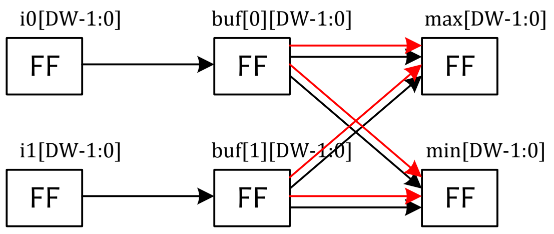

Fig. 2.6 SDFG with range partition

The detailed automatic range partition process involves the automatic range calculation is generating the relation trees and automatic connection of LHS registers with RHS partial registers. The related codes are located in:

```
[AVS]/sdfg/rtree.hpp
[AVS]/sdfg/rtree.cpp
[AVS]/sdfg/dfg_graph.hpp
[AVS]/sdfg/dfg_graph_avs.cpp
```

2.3. User Interface

Like many synthesis tools, AVS provides a basic command line based user interface. Using this interface, the user is able to run different commands, forming complex task by individual steps, store and load multiple tasks using scripts. All the user interface related codes are located in `[AVS]/shell`.

2.3.1. Tcl Command Line Environment

The user interface is built on the Tcl/Tk package adopting an opensource Tcl/C++ package named cppTcl (https://github.com/wsong83/cpptcl, original http://cpptcl.sourceforge.net).

The original cppTcl library has been modified to support add user defined functions with ability to change to underlying environment. In the case of AVS, AVS is able to add new commands which are able to change the data belonging to AVS rather than the name spaces in Tcl.

For the detailed modification and the use of cppTcl library, refer to http://wsong83.github.io/cpptcl/

2.3.2. Command Definition

Adding a new command in the Tcl command user interface is fairly easy. All the commands added by AVS are stored in `[AVS]/shell/cmd`. Each command has a cpp file to implement it.

Each command is a structure defined in the namespace of `shell/CMD`. The struct has 4 static members: `help()`, `exec()`, `name` and `description`. The `help()` method is called when showing the help information of this command. The `exec()` method is called automatically when this command is used in the Tcl command line environment. The `name` field is used to identify this command, and is the command name used in the command line. After all, the `description` field is used to show the summary of this command when listing all the command available in the command line.

To add the a new command in the Tcl command line environment, a new struct declaration needs to be added in the `[AVS]/shell/cmd/cmd_define.h` using macro `NEW_TCL_CMD`. Then in `[AVS]/shell/env.cpp`, a new Tcl command is defined using macro `FUNC_WRAPPER` and AVS_ENV_ADD_TCL_CMD. Once the three macros are set, the command line would be able to recognize this new command, call the `help()` method when showing the help information, and call the `exec()` method when the command is executed.

In the cpp file that defines the actual behaviour of the command, `boost::spirit` library is used to define and parse arguments. Therefore, the command is able to define and parse arbitrary argument formats. Many basic argument elements are provided in `[AVS]/shell/cmd/cmd_parse_base.h` to reduce workload.

The `exec()` method has two arguments, `str` and `pEnv`. `str` is the string of the command line parsed from the Tcl parser. The detail arguments are collected by parsing the `str` using the Boost Spirit parser. The final usable arguments are stored in the `arg` object. `pEnv` is a pointer pointing to the working environment where all the passed modules, libraries and graphs are stored.

For the detail of these commands, refer to the codes in `[AVS]/shell/cmd`. The simplest command could be echo (`[AVS]/shell/cmd/echo.cpp`), which can be a good start.

2.3.3. Error Reporting Mechanism

It is always possible to report errors in an ad hoc way, such as print a message to the shell. Many places in the AVS still printing information in this way due to the convenience. However this instructed way is not good for large systems. An error reporting system is defined and utilized in the parsing and elaboration part of the AVS system.

The error report system is initiated in the environment as an object named error (`[AVS]/shell/env.h`). The class definition isdescribed in `[AVS]/shell/err_report.h` and `[AVS]/shell/err_report.cpp`.

The `err_report` class is a function class. Inside the class, a map is used to store all pre-defined error types. Using error identifier (`errID`), an err_report object is able to identify the error type and print the error accordingly.

Using this system, defining a new error type is easy. Every error is defined as a macro in `[AVS]/shell/err_def.h` file. An error type comprises 4 fields: an error identifier, a severity level (fatal, error, warning or info), number of parameters and the formatted error message using the `boost::format` package.

To report using the error reporting system, the code needs to only call the err_report function by accessing it through the global AVS environment object. One such example is in `[AVS]/netlist/instance.cpp`. When elaborating an instance but the instance cannot find a matching module definition, the instance would report an error to say it cannot find the module:

```
    G_ENV->error(loc, "ELAB-INST-0", mname.get_name());
```

Where `G_ENV` pointing to the global environment, `error` is the error reporting system, `loc` is the line count in the original Verilog code, `ELAB-INST-0` is the predefined error type, and `mname.get_name()` get the name of the unfound module. The actual error message shown in the shell may look like:

```
    nova/src/top.v:104.12-60:   [ELAB-INST-0] Error: fail to find the
module named "fifo_seq_DW32".
```

2.3.4. Script

To show how script can be used to speed up testing, follows is a simple script I have used to test the OpenRISC processor. To load this script in the AVS shell, simple type "source test_or1200.tcl".

File: test_or1200.tcl

```
        # asynchronous Verilog test script
```

```
# set up the PATH for finding source codes
set search_path "../or1200-rel1/rtl/verilog"

# suppress pre-known warnings
suppress_message "SYN-PORT-2"
suppress_message "ELAB-VAR-2"
suppress_message "ELAB-VAR-3"

# read in the whole hierarchical design
analyze -format verilog or1200_alu.v
analyze -format verilog or1200_amultp2_32x32.v
analyze -format verilog or1200_cfgr.v
analyze -format verilog or1200_cpu.v
analyze -format verilog or1200_ctrl.v
analyze -format verilog or1200_dc_fsm.v
analyze -format verilog or1200_dc_ram.v
analyze -format verilog or1200_dc_tag.v
analyze -format verilog or1200_dc_top.v
...
analyze -format verilog or1200_spram_64x24.v
analyze -format verilog or1200_sprs.v
analyze -format verilog or1200_top.v
analyze -format verilog or1200_tt.v
analyze -format verilog or1200_wb_biu.v
analyze -format verilog or1200_wbmux.v

# elaborate the design with identifying the top level
elaborate or1200_top

# extract SDFG
extract_sdfg

# automatically detect controllers
report_fsm

# automatically extract data paths
extract_datapath

# using the data path to find potential partitions
report_hierarchy
```

3. Controller Detection

3.1. Motivation

It is assumed the automatic identification of controllers is important in the synthesis. To understand the behaviour of a RTL design in order to resynthesize it into a GALS design, a synthesizer may require the state space information of certain or even all controllers. Without the direct input from designers, automatically detect controllers is no easy task.

Current commercial synthesis tools detect finite state machines (FSMs) using coding styles. According to the Verilog user guide provided by the Synopsys HDL Compiler, an FSM must be written as a register assigned only with predefined values, used only in `case`, `if`, `==` and `!=` statements (expressions), and never used as ports. These strict restrictions match only the FSMs written in the standard form of one or two always blocks. Hierarchical FSMs and the counters used as controllers are rejected by the matching algorithm.

Using coding styles to recognize controllers is not sufficient enough for the purpose of understanding the behaviour of a RTL design. The synthesizer needs to recognize most, if not all, controllers, including FSMs, counters used as state machines, and state flags. To achieve this goal, a pattern matching algorithm has been proposed to recognise controllers in the extract SDFGs.

3.2. Detection Criteria

Once an SDFG is generated for a hierarchical design, 3 criteria are used to match a controller:

Criterion 1: Let a register be a controller, at least one of its output paths is a self-loop path which does not go through higher hierarchies.

Since the transition of a controller is restricted by its predefined state space, each transition always and must transit from a known previous state to a new state depending on the status of its environment. As a result, a controller must have a self-loop to identify the previous state. The extra hierarchical constraint is set for practical concerns. In normal RTL designs, the calculation of the next state is located in the same module containing the controller or sub-modules inside the controller module, which complies with the constraint. However, some RTL designs have manual test chains or combinational loops which may confuse the controller detection algorithm. The hierarchical constraint is added to prevent any global combinational loop from making a false self-loop to a non-control register.

Criterion 2: Let a register be a controller, at least one of its output paths is a control path towards another register.

The value of control register must has been used to control the modification of other registers; otherwise, this register is a non-control one on data paths.

Criterion 3: Let a register be a controller, all its input data comes from self-loop paths or constant numbers.

This criterion is inferred from the finite state space limitation of controllers. Although the state transitions depend on their environment, the state space of each controller is predefined. Therefore, the data inputs of a controller come from only itself or a predefined constant value.

The simple type system defined in 2.2.1 has been expanded to record the detailed features that inferred the type. The expanded type definition is defined in `[AVS]/sdfg/dfg_edge.hpp`:

```
    SDFG_DDP = 0x00001, // default data loop
    SDFG_CAL = 0x00002, // mathmatical calculation datapath
    SDFG_ASS = 0x00004, // direct assignment
    SDFG_DAT = 0x00008, // other type of data
    SDFG_CMP = 0x00010, // compare
    SDFG_EQU = 0x00020, // equal
    SDFG_LOG = 0x00040, // logic
    SDFG_ADR = 0x00080, // address
    SDFG_CTL = 0x00100, // normal control
    SDFG_CLK = 0x02000, // clk
    SDFG_RST = 0x04000, // reset
    SDFG_DAT_MASK = 0x0000F,
    SDFG_CTL_MASK = 0x00FF0,
    SDFG_CR_MASK = 0x0F000
```

Using the expanded types, the controller detection algorithm can tell the types of the controller, such as finite state machine, counter, address control and flag.

3.3. Path Iteration

The term path mentioned in previous sections denotes the logic connection from a register to another register. The arcs in the SDFG represent the relations between signals rather than registers. To figure out the type of a path, an algorithm is needed to find out all the possible paths and compile a type from all the arcs on the paths.

Several methods have been provided in `[AVS]/sdfg/dfg_graph_avs.cpp` to return the paths and types from a register and towards a register.

The type reduction of a path is based on two rules: (a) control arcs have higher priority than data arcs; therefore, a path is a control path if there is any control arcs on it. (b) in the sub-type of data or control, the type is decided by the first non-assignment type.

For example, for the following expression:

```
    Assign dout = (select && count -1 == 5) ? 45 : 0;
```

The path from count to `dout` is defined as `SDFG_EQU`. The path is a control path since the value of count is compared with 5, the data type derived from the minus operation is overridden by control. The final type is defined as `SDFG_EQU` because it is the first non-assignment type. The following `? :` operation will not change the type.

The underlying reasoning for these rules is to preserve the most significant relation between the source and the destination although it may not be right for all coding features.

The type reduction rules also allow the use of software caches. The path search and reduction methods do not need to find all paths and reduce the types individually. They calculate the final reduced type by calculating the types of partial paths using recursive functions.

3.4. Detection Results

Three projects from the OpenCore hardware repository has been chosen to test the performance of the controller extraction method. The original results are published in [7] and the updated results with type estimation are listed as below.

Fig 3.1 The results of controller extraction with type estimation

Design	SDFG Nodes	Registers	Controllers	FSMs	Counters	Addresses	Flags
OR1200	2284	147	30	6	4	1	20
RSD	1225	450	77	28	31	6	33
H264	8454	1060	62	41	31	7	4

Some features are added into the control indication. Comparisons (==, >, <) and Boolean operations (||, &&, !) are recognized as control expressions. As a result, more registers are recognized as controllers compared with the results shown in [7]. This does not necessarily indicate that fake controllers are recognized or some real controllers are neglected. The definition of a controller is not clear in practical code as the division between data paths and control paths. As for analysing the behaviour of a RTL design, it is better to recognize a fake controller rather than overlook a real one.

The type estimation of controllers is not accurate as well. Some controllers are recognized as multiple types. This can happen in real designs, such as a finite state machine has only two states (both FSM and flag). Generally speak the type estimation provides only an indication of the real types.

3.5. Future Work and Conclusion

The accuracy of controller extraction relies heavily on the accurate estimation of arc types. Due to the complexity of coding styles, the current type estimation and path type reduction algorithm are not perfect. They make mistakes sometimes and these estimation errors may affect the final controller extraction. Better type estimation method can improve the extraction accuracy; therefore, providing more usable information.

The current controller extraction method has already achieved a high extraction rate which is much higher than commercial tools. It recognizes controllers in the forms of finite state machines, counters and flags. The processing time is short. For the largest test case, the H264 decode, it takes only 10 seconds to extract controllers.

4. Data Path Detection

4.1. Motivation

The motivation of automatically detecting data paths in RTL designs lies in the need to analyse the underlying data flow. Most hardware designs can be recognized as several data paths controlled by multiple controllers. The controllers are either running locally using data-dependent or fixed control logic, or synchronized with other controllers. The communication and the behavioural of controllers are difficult to analyse. Data paths, on the other hand, are simple. They can be considered as linear or non-linear pipelines with fixed or variable data rates. Once the data paths of a hardware design are extracted, it is clear to see the data dependence between modules. This provides a clear direction of where to partition a large design into multiple small ones.

4.2. Data path extraction

The extracting method used in AVS is actually very brutal. Data paths are extracted from SDFG in two steps: removing all control arcs and trimming. The first step removes all control related arcs, including control, clock and reset. Then in the second step, believing that control related nodes are separated from the data paths due to the removal of control arcs, a recursive algorithm traverse the SDFG and remove all control related sub-graphs. When the recursive trimming is finished, only data paths remain in the final SDFG.

4.2.1. Control removal

Control removal is easy. The extraction algorithm traverses the whole SDFG and removes all arcs with a type other than data (any of the sub-type of data arcs). The aim of this step is to separate the control related sub-graphs from the data path sub-graphs. Considering a control related signal must controls the behavioural of other signals, there must be control arc on the paths from the control signal and the controlled signals. By removing the control arc, the control signal is cut off from the controlled signals.

4.2.2. Graph Trimming

As described in the ISCAS 2014 paper [8], the graph trimming step tries to remove all the control related sub-graph by trimming dangling nodes and arcs. The algorithm traverse the SDFG generated from the first step and removing all nodes (and arcs connected with it) qualified the following criteria:

- A combinational node with either no egress arc or no ingress arc.
- A register node without a non-self egress arc.
- An input port without an egress arc.
- An output port without an ingress arc.
- A module without an egress arc.

However, it is later found out that not all control related nodes are removed. Some control nodes form loops with other control nodes. These loops can escape from the above trimming criteria and remain in the final data path SDFG as disconnected sub-graphs. To further remove these loops, two extra traversal of the data path SDFG is required. One starts from input ports to output ports and removes any unvisited nodes and arcs. The other one travels backwards and also remove any unvisited nodes and arcs. This two extra step also automatically remove non-data ports as an extra benefit.

It is found that travel through the whole SDFG, basically flatten all hierarchies, requires a significant amount of memory. The running time also increases exponentially with the level of hierarchies. The alternative is to perform the trimming in a recursively way in each module. The modules on the higher hierarchies are processed before the lower hierarchies; therefore, some ports of a lower hierarchical module could be removed before the module is processed. Utilizing this information, the lower hierarchical module can safely remove all nodes that related to the removed ports.

The recursively trimmed data path graph is not equivalent to the flattened results due to some cross boundary loops. The side-effect of these loops is small.

4.3. Detection Results

The extraction results are very good at least for small-scale designs. For large-scale designs, the problem is more about how to judge the extracted data paths considering the lack of knowledge to the design itself.

For more analysis of the extracted data path, refer to [8]. Here shows the automatically extracted filter in the H264 design, which seems great.

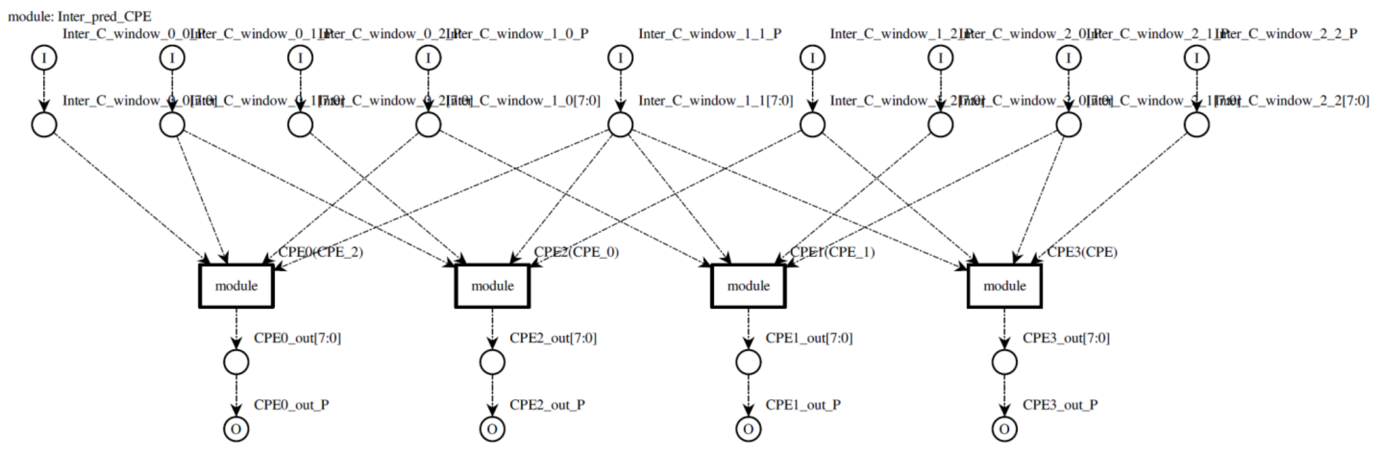

Fig 3.1 An automatic data path extraction from H264.

4.4. Future Work and Conclusion

Currently it is believe that the data path extraction successfully remove control related nodes and preserve data related nodes in the extracted SDFG. It is more likely that some nodes left in the extracted SDFG are still control related due to the lack of global information. For example, without user input, all input and output ports are assumed data related until found otherwise. The AVS tool allow user to define the type of each input/output port which can help the extraction.

One of the potential user asked whether the tool is able to extract Verilog code using the data path SDFG extracted. The SDFG has internal match with the original Verilog ASTs. Therefore, it is able to remove control related nodes in the Verilog using the removed nodes in the SDFG. However, there will be problems as when control signals are removed, what is the correct behaviour to remove the signal in the original Verilog. This can be a very interesting and maybe useful future research.

5. Interface Recognition and Design Partition

5.1. Motivation

It is very useful to partition a design into loosely connected synchronous sub-designs. However, it is unlikely a synthesis tool can intelligently break the original module definitions and regenerate sub-modules. The first step is to partition a design based on the original module hierarchy to see which sets of modules can be recognized as a sub-design.

To achieve this goal, the partition tools need to understand the module interfaces, such as data/control type, behaviour and throughput. The automatic data path extraction is able to tell the data related ports. For partitioning a design, the behaviour of data ports is more important than control ports. Therefore, this Section tries to automatically estimate the type of different types of data ports.

5.2. Detection Method

Static pattern matching is used to estimate the type of data ports. Currently the algorithm tries to tell if a data port is a wire/pipeline (no variable data rate control), memory interface or handshake interface.

The details of this pattern matching process can be found at
`[AVS]/shell/cmd/report_hierarchy.cpp`

5.3. Wire and Pipeline

The type of a port is identified by the estimated type of the driving node in the SDFG. This driving node is defined as the first non-assign node found by back-tracing from the port.

A port is recognized as a WIRE port if it is combinationally connected to input or output ports. That is to say, there is no registers between this node and the global inputs or global outputs.

A port is recognized as a PIPE port (pipeline) if there is no self-loop for any of its previous two levels of registers. The algorithm back-traces and iterate all the previous two levels of registers to see if there is any register has a self-loop. If no loop is found, it is believed that the port is driving by a pure mathematical calculation where the variable data rate is irrelevant (meaning no possible partition on this port).

5.4. Memory

A port is recognized as a MEM port (memory interface) if it or any of its driving registers is controlled by another register behaving as an address. This covers all normal features of RAM, ROM and registered RAM.

The test results show that this estimation is very accurate that all memory interfaces have been found and correctly identified.

5.5. Handshake

Handshake (HAND) interfaces are difficult to recognize. One of the problems is that there are so many ways to implement handshake communication between two modules. The pattern used to recognize a handshake interface needs to be general enough to match with them.

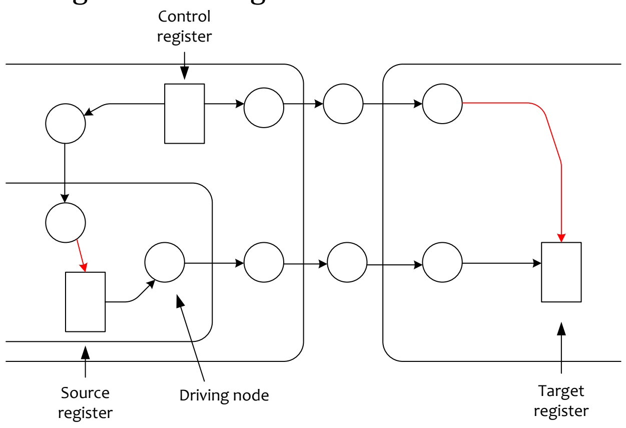

Fig 5.1 The pattern used for handshake recognition.

The pattern used to recognize handshake interfaces is depicted in Fig. 5.1. For a port, all registers that drive the port are collected as the source register. All registers driven by this port are collected as target register. Any pair of source and target registers belongs to different modules. For each possible pair, the controlling registers of both registers are found. If a controlling register is found controlling both the source and the target register, and it share a hierarchical module with one register but not both, the controlling register is recognized as the handshake controlling register and a handshake interface is found.

This pattern is believed to be general. For a handshake interface, there would be a enable, request, ready or ack signal which is driven by the local controller driving the data source and used as an indication by the receiving register. This indicates a controller to control the value of both the source and the target register. For a handshake interface, this shared register should either on the sender or on the receiver end.

However, this pattern does not recognize the global FSM which can also be used as handshake control. In this case, the controlling register shares the hierarchical module with both registers. It is found difficult to differentiate this pattern with the global enable signal.

5.6. Potential Partition

For each module inside a design, the tool tries to recognize the types of all its data input and data output ports. When the ratio of the ports estimated as MEM or HAND passes certain threshold, the module is recognized as a potential sub-design.

The tool will travel through the top to all sub-levels to find potential sub-designs and report them. User is allowed to change the threshold value; therefore, to control the number of potential sub-designs found.

5.7. Partition Results

Here is the summary result of the OR1200 design.

Summary:
```
or1200_qmem_top/or1200_qmem_ram(0.75)
or1200_cpu/or1200_rf(0.8)
           or1200_lsu/or1200_mem2reg(0.75)
                      [Other]or1200_reg2mem;
           [Other]or1200_freeze;or1200_except;or1200_operandmuxes;
           or1200_cfgr;or1200_sprs;or1200_alu;or1200_genpc;
           or1200_mult_mac;or1200_ctrl;or1200_if;or1200_wbmux;
or1200_ic_top/or1200_ic_fsm(0.833333)
or1200_dc_top/or1200_dc_fsm(0.75)
[Other]dwb_biu;or1200_immu_top;or1200_dmmu_top;iwb_biu;or1200_pic;or1200
_sb;or1200_pm;or1200_tt;or1200_du;
```

It is shown that the partition algorithm believes or1200_qmem_ram, or1200_rf, or1200_mem2reg, or1200_ic_fsm and or1200_dc_fsm modules can be used as sub-designs.

5.8. Future Work and Conclusion

The current partition algorithm is far from perfect. It can recognize the types of interfaces; however, this recognition is not very accurate, especially for the type of handshakes. Also it is unable to recognize bus interfaces yet. Even an interface is recognized, the algorithm feels difficult to understand the control underlying the interface. For complicated design, dozens of controlling registers have been recognized as the controlling registers for handshake interface, which is obviously useless for further behaviour analysis. It is required to find out the couple of key controlling signals (which may be a combinational signal rather than a register) that affecting a handshake interface, for better behaviour analysis. Right now, the controller extraction has very little help in the process of analysing the behaviour of interfaces. The tool cannot estimate the data rate of interfaces; therefore, a data flow analysis is unavailable yet.

The urgent job right now is to reduce the number of controlling registers recognized for handshake interfaces. Then it is possible to analyse the state spaces of these controlling signals and estimate the variable data rate, which is crucial for data flow generation.

6. Future Work and Conclusion

6.1. Current Issues

The current controller extraction algorithm is effective. It recognizes most, if not all, controllers in a RTL design. However, some non-control signals are mistakenly recognized as controllers. The type estimation of controllers is not perfect. Analysing the state space of controllers on the signal level (rather than binary level) is fundamentally difficult.

The data path extraction is believed to be brutal and effective as well. The algorithm works better for large RTL designs because more control relations are exposed for data path extraction. However, annotating data rates to the data path is a difficult problem. The tool can back-annotate saif (switching activities from simulation) files to the SDFG, but the back-annotation is very inaccurate for multi-bit buses.

Interface recognition provides only preliminary partition results. It cannot understand bus interfaces yet and the recognition for handshake interfaces is not effective. Too many controlling registers are recognized for each handshake interface, which makes further behaviour analysis difficult.

6.2. Conclusion

The asynchronous Verilog synthesizer (AVS) tool is able to compile large-scale Verilog RTL designs into a new abstract format named signal level data flow graph (SDFG). It provides an expandable command line environment for future development. The current commands provide the functionalities including extracting controllers, extracting data paths and preliminary partition analysis.

References

[1] W. Song, "Asynchronous Verilog Synthesiser," [Online]. Available: https://github.com/wsong83/Asynchronous-Verilog-Synthesiser. [Accessed September 2014].

[2] S. Williams, "Icarus Verilog," [Online]. Available: http://iverilog.icarus.com/. [Accessed September 2014].

[3] W. Snyder, "Verilog-Perl," [Online]. Available: http://search.cpan.org/dist/Verilog-Perl/. [Accessed September 2014].

[4] Flex Project, "flex: The Fast Lexical Analyzer," [Online]. Available: http://flex.sourceforge.net/. [Accessed September 2014].

[5] GNU, "Bison," [Online]. Available: http://www.gnu.org/software/bison/. [Accessed September 2014].

[6] T. Parr, "ANTLR (ANother Tool for Language Recognition)," [Online]. Available: http://www.antlr.org/. [Accessed September 2014].

[7] W. Song and J. Garside, "Automatic controller detection for large scale RTL designs," in *EUROMICRO Conference on Digital System Design*, Santander, Spain, 2013.

[8] W. Song, J. Garside and D. Edwards, "Automatic data path extraction in large-scale register-transfer level designs," in *International Symposium on Circuits and Systems*, Melbourne, Australia, 2014.