Automatically Connecting Hardware Blocks via Light-Weight Matching Techniques

Jan Malburg, Niklas Krafczyk, Görschwin Fey

*Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{malburg,thuriasz}@informatik.uni-bremen.de

†Institute of Space Systems
German Aerospace Center
28359 Bremen, Germany
Goerschwin.Fey@dlr.de

Abstract—In modern chip design, many different blocks are assembled in a single chip. Normally, these blocks have been written by different developers or even licensed from other companies. Correctly connecting all blocks is a tedious task. State of the art tools for automatically generating the connections either require identical port-names or additional user input describing the intended connections.

In this paper we present an automatic approach for connecting different blocks. In contrast to previous approaches, we neither need exact name matching of the port-names nor additional user input. An evaluation showed the advantages of our approach. For seven of eight designs our approach generated better connections than a previous approach, including a design which has been optimized for being used with the previous approach.

A second goal of this paper is to understand the limitations of the presented light-weight matching techniques.

I. INTRODUCTION

Modern chip designs are composed out of several different blocks. Blocks are typically described in a Hardware Description Language (HDL). Often these blocks are from different developers or even third party blocks licensed from other companies and have to be assembled into a single chip. Writing the corresponding connections is a tedious task for a developer, as he needs to connect several hundreds or even thousands of different ports. Hence, automation is desirable.

We consider a technique as automatic if it only requires the blocks and the number of instances of each block. In this sense tools are not considered automatic if they require further user input like descriptions of the intended connections as, for example, the tool MKTREE [1]. An example for an automatic tool based on this definition is the openly available Emacs Verilog-Mode [2], which allows automatic generation of the connections. Additionally, a developer can add hints to support the generation.

In this paper we propose a technique which automatically generates the connections between a set of blocks. Our technique uses heuristics to compute likely connections between the different blocks using the similarities of the port-names, the bit-width of the ports and the data-direction. In contrast to similar techniques, our technique does not require 100% matching of the port-names to find the connections. This is an advantage, as often there is no 100% match of the names, either because the blocks are from different companies with different naming conventions or the naming convention includes pre- and suffixes for the data-direction and -width. A secondary research goal of this work is to evaluate the limitations of light-weight approaches for automatically connecting modules. Therefore, no computation intensive functional techniques are used.

We evaluated our approach on eight different designs of different size, authors, functionality, and origin. As comparison we use Emacs Verilog-Mode. With respect to a quality metric, which approximates the amount of work such a tool saves a developer, our approach generates equal or better connections than Emacs Verilog-Mode for all of the eight designs considered. This even includes a design which has been optimized for being used with Emacs Verilog-Mode. For one design, for which Emacs Verilog-Mode was not able to create any connections, our approach generates a perfect set of connections.

Finally, we present cases which cause poor results for the approach and possible future improvements.

The remainder of this paper is organized as follows: Section II describes related work. Our approach is described in Section III including the different heuristic and connection strategies we are using. In Section IV we present the used quality metric and the evaluation of our approach. In Section V the limitation of the evaluated light-weight techniques and future improvements are discussed. Section VI concludes the paper.

II. RELATED WORK

In this section we discuss existing tools for generating connections between ports of different modules. However those techniques either require exact name matching of the port names or additional user input.

With SystemVerilog 1800-2005 [3] the implicit port instantiation operators \( . * \) and \( . \text{name} \) have been introduced. The operators allow shortening the instantiation list of modules. However, they require a complete match of the signal names with the port-names and that the signals are already defined in the module. In contrast, our approach neither requires that signals are already defined nor that there is a 100% matching between the names. The implicit port instantiation operators and our approach have in common that they expect correct data-width and data-direction.

The Emacs plugin Verilog-Mode [2] adds several options for automatic instantiation to the text editor Emacs. The option AUTOINST allows instantiation based on direct name matching similar to the \( . * \) operator for SystemVerilog. Further, a user can use regular expressions in order to lift the requirement of exact name matching. Additionally, Verilog-Mode provides AUTOWIRE and AUTOREG which automatically declares wires and registers based on the module’s input and output definitions. Still this approach either requires exact name matching or complex user input, in form of the corresponding regular expressions, to automatically create the port connections. Similarly, Emacs plugins for VHDL exist as well [4].

MTREE [1] is a tool to create connections between several Verilog modules. For this MTREE utilizes a special description language to reduce the coding effort for a developer. Therefore, MTREE is not an automatic tool in our sense and has the disadvantage that the developer has to learn the description language used.

ShapeUp [5] is a tool which uses a connection description written in Click, a language originally designed to describe network systems for generating the connection between modules. Further, ShapeUp utilizes module-interface descriptions based on the IP-XACT standard [6] in order to ensure correct wiring between the different modules. To some extend ShapeUp is even able to automatically generate converter blocks in cases where the interfaces do not completely match, for example in presence of different bit-widths. However, both the Click description of the intended connection and the IP-XACT description of the modules need user interaction. Thus ShapeUp is not considered an automatic tool in our sense.

This work was supported in part by the German Research Foundation (DFG, grant no. FE 797/6-1)
In [7] Avnit et. al. present an approach to automatically generate protocol converters for the communication protocols of two blocks. Their approach needs a description of source protocol and target protocol in form of a finite state machine and a mapping of data- and control-ports. Based on this they compute a hardware block implementing a finite state machine, which translates one protocol into the other. Their approach differs from ours as we assume matching protocols where they explicitly assume nonmatching protocols. Further, their approach needs user input in form of the finite state machines and the port mapping and uses computational costly formal analysis.

III. Technique

In this section we describe our technique and the heuristics we are using. The implementation of our technique is based on the parser of IcarusVerilog [8]. Consequently, our implementation only supports Verilog. However, the basic idea should work for other HDLs, like VHDL or SystemC as well.

The basic flow of our approach is shown in Figure 1. As input a list of Verilog modules, the source code of each module, and, optionally, the amount how often each module should be instantiated is used. First, a set of prohibiting heuristics is applied on the modules. The prohibiting heuristics mark connections as forbidden either because they are very unlikely or would result in nonsynthesizable code. Then supporting heuristics are used which compute the likelihood of a connection of being correct based on name comparison. The last step in the computation is the application of a connection strategy which uses the results of the heuristics. Some strategies additionally use a threshold or a safety approximation. The connection strategy returns the generated connections between the different modules.

A. Prohibiting heuristics

In this section we present the heuristics which mark connections between two ports as forbidden connections either because such a connection would result in invalid Verilog code or the connection is extreme unlikely.

• No output-to-output connections:
  Such a connection is not allowed by Verilog.
• No Output-to-InOut connections:
  A connection of an InOut port with an Output-port is valid in Verilog. But this type of connection is unlikely and might often result in multiple driver errors during synthesis.
• No connection between ports with different bit-width:
  Like the previous case, this results in correct Verilog code but is rather unlikely.
• No connection between two ports of the same module-instance:
  This is also very unlikely as the information could be forwarded inside the module.
• No connection of a port with itself:
  Such a connection cannot be expressed in Verilog.

B. Supporting Heuristics

Here we present several heuristics to compute a likelihood for two ports to be connected. These heuristics assign values to pairs of ports in the range from zero to one, where zero means unlikely and one means very likely.

1) Name matching ($N_m$): The first set of heuristics we present is based on the similarity of the port-names. For any sensible naming convention for port-names, those names should contain information about the data, which is expected to be send over that port. Consequently, if two ports should be connected, the same data is to be send over both ports. Over one of the ports as input and over the other port as output. Thus it can be assumed that the ports which should be connected have to some extend similar names.

We implement three different heuristics based on the string-similarity metrics Jaro-distance [9], Levenshtein-distance [10] and longest-common-substring.

• Jaro-distance ($N_m_{Ja}$):
  For computing the Jaro-distance, first the relation of matching characters gets computed, where characters are considered matching if they are identical and their position does not differ beyond a value defined by the length of the string. Then the amount of transpositions is counted. A transposition is a set of two matching characters, which are in different order in the two strings. After that, the distance is computed as the average of the relation of the matching characters to the length of the strings and the relation of matching non-transposed characters to matching characters.

• Levenshtein-distance ($N_m_{Le}$):
  The Levenshtein-distance, also known as edit-distance, is defined as the minimal number of edit operations which have to be applied in order to change one string into the other. An edit-operation is defined as either, deleting a character, adding a character, or the replacement of a character. The largest possible Levenshtein-distance of two strings is the length of the longer of those two strings.

• Longest-common-substring ($N_m_{LCS}$):
  The longest-common-substring is the length of the longest substring which both strings have in common. This value then is normalized based on the average size of both strings. The idea in using the longest-common-substring is that in many naming-conventions characteristics of the port, like direction and bit-width, are included in the name in form of a pre- or suffix.

For our heuristic computation the result of the distance computation is normalized to the range between zero and one, where zero means no similarities between the strings and one means identical strings. The normalized value is the resulting likelihood.

2) Extended name matching ($E_Nm$): Like the normal name matching, but additionally, the port name is also compared to the module name of the ports to which it should be connected, the submodules, if any, which use the port and the port names for those submodules. The maximum score of all these computations is used. As for name matching extended name matching can be combined with the different string metrics resulting in the three heuristics $E_Nm_{Ja}$, $E_Nm_{Le}$ and $E_Nm_{LCS}$.

3) Event checking ($Ev$): In synchronous designs, there exists always a clock signal and most likely a reset signal. If the event checking heuristic is used, a top level signal $clock$ is assumed to exist as well as a signal $reset$. The sensitivity lists of the modules and submodules are checked, and it is assumed that reset and clock signal are signals which appear in the sensitivity lists of the modules. For those signals which appear in the sensitivity list, their names get compared to the names $clock$ and $reset$, as normally the clock signal is named $clock$ or $clk$ and the reset signal $reset$ or $rst$. The result of this comparison is used as the assigned likelihood. Because, this heuristic only compares port-names with $clock$ and $reset$, the heuristic only assigns likelihoods to such connections.

C. Score and Threshold

Over all heuristics a score is computed for the connection between two ports. For this the likelihoods computed by the different heuristics get summed up. Also the prohibiting heuristics assign likelihoods
to the possible connections, hereby the following rules apply: The
heuristic which checks for port direction assigns 0.5 to connections
between input and InOut ports and to all other legal connection the
likelihood 1.0. Forbidden connections get the likelihood 0.0 assigned.
Correspondingly, the heuristic which checks for port width assign a
likelihood of 1.0 to connections with matching port widths and 0.0
otherwise.

A threshold on the computed score has been defined in order to
decide if a connection is likely or not. As the likelihood of all used
heuristics get summed up, the maximal score value depends on the
number of used heuristics. Therefore, the value of the threshold is
defined as a fraction of the maximum score. Preliminary experiments
showed that choosing a threshold equal to two third of the maximal
reachable score is effective; thus this is the value we are using for
the experiments in this paper.

### D. Safety Approximation

For a port, there are often several ports to which it can connect.
However, input ports are only allowed to connect to a single other
port. All those possible ports present alternatives to which the port
can be connected. Potentially, some of those connections might not
have the highest score, but do not have any other likely connection
and the connection with the highest score may have many different
other possible connections. In this section we present a safety approx-
imation which also considers the score of alternative connections.
The basic idea of the safety approximation is to compute, how much
more likely a connection is compared to the other connections the
associated port could be part of. Let $s(i,j)$ be the score for the ports
$i$ and $j$. Further, let $s_{\text{max}}$ be the highest possible score and $P$ the
set of all ports. Then we define the safety approximation $c(i,j)$ between
the port $i$ and $j$ as:

$$c(i,j) = b(i,j) \cdot \frac{s(i,j)}{s_{\text{avg}}(i,j)} \cdot \frac{s(i,j)}{s_{\text{max}}}$$

With $b(i,j)$ is a factor based on the prohibiting heuristics:

$$b(i,j) = \begin{cases} 0 & \text{connection marked as forbidden} \\ 1 & \text{otherwise} \end{cases}$$

and $s_{\text{avg}}(i,j)$ is the average score which $i$ and $j$ reach:

$$s_{\text{avg}}(i,j) = \frac{1}{2 \cdot |P|} \sum_k (s(i,k) + b(i,k) + s(j,k) + b(j,k))$$

### E. Connection Strategies

We implemented several different strategies for choosing the con-
nections between the ports.

1. **Greedy approach using the score ($S_{i,j}$):**
   - Always the connection with the highest score is created first.
   - This might make other connections invalid. Connections are
     created until no further connection with a score higher than or
equal to the threshold is possible.

2. **Greedy approach using the safety approximation ($S_{i,j,f}$):**
   - Like $S_{i,j}$ but creating the connection with the highest safety
     approximation first.

3. **Preferring already connected modules using the score ($S_{m,h,f}$):**
   - Like $S_{i,j}$, however when choosing the next connection to
     create, for connections between two modules which already are
     connected, the scores of these connections get multiplied by 1.1.
   - This is due of the fact, that often a package of logical information
     is sent over a combination of several ports.

4. **Preferring already connected modules using the safety approxi-
mation ($S_{m,cm}$):**
   - Like $S_{m,h,f}$ but using the safety approximation instead of the score.

### IV. Evaluation

In this section we evaluate our approach on several designs. In
order to reduce possible bias we use designs of different size, of
different purpose, and from different authors. For the evaluation we
removed the sub-module instantiations from the top-modules of the
designs and then applied our approach to recreate the instantiations.
We used Emacs Verilog-Mode as a baseline-comparison. We have
chosen Emacs Verilog-Mode because it is a freely available tool,
which uses similar inputs as our approach, however a developer may
add hints, in form of regular expressions, to improve the result. A
connection is considered correct if it is identical to a connection of
the original top-module; otherwise the connection is considered
incorrect. Thus, the resulting connection is optimal, if and only if it is
semantically equivalent to the original top-module.

For comparing the results a quality measurement has been defined.
Let $n_{\text{corr}}$ be the number of correctly generated connections, $n_{\text{all}}$
the number of all generated connections and $n_{\text{orig}}$ the number of
correctly expected connections. Our quality measurement $q$ is then
defined as:

$$q = \frac{n_{\text{corr}} - (n_{\text{all}} - n_{\text{corr}}) - (n_{\text{orig}} - n_{\text{corr}})}{n_{\text{orig}}} + 1$$

The intention of this measurement is to approximate the amount of
work a developer can save by using a tool\(^1\) for creating the
connections. $n_{\text{corr}}$ is the amount of correct connections, i.e., work
a developer has saved; however, a developer has to remove the
connections the tool has incorrectly created ($n_{\text{all}} - n_{\text{corr}}$) and, finally,
he has to add all missing connections ($n_{\text{orig}} - n_{\text{corr}}$). The rest of
the formula normalizes the value such that 0 means no work is saved,
corresponding to the case of the design without any connections, and
1 means that all connections are correctly generated, i.e., all work is
saved. The formula may return negative numbers; this is the case
when the correction of the returned solution is harder than connecting
the modules completely by hand.

Table I gives an overview of the designs used for this evaluation.
SHA3 is a design for computing the SHA3-hashcode of the input
data. The Wishbone-interface is an interface for register and memory
access, used in many designs on the OpenCores.org-website. The
interface allows a master and several slaves on a single bus. The
specification includes the naming convention for the correspond-
ning ports. The design Wishbone-specification only consists of sub-
modules with the correspondingly named ports but without any actual
logic. SD Mass Storage Controller is an SD-Card controller. The
design uart2spi is a protocol translation block between UART and
SPI which can additionally parse some simple commands directly.
MIPS789 and OpenRISC 1200 are two different RISC processor
designs. The OpenRISC 1200 supports floating point arithmetic and
includes data- and instruction caches, a memory management unit,
a timer, an interrupt controller, a debugging unit and two Wishbone-
interfaces. OpenSPARC T1 core is a core of the OpenSPARC T1
processor. The OpenSPARC seems to be developed using the Emacs

\(^1\)Approximation because it assumes that the work a developer needs to
remove a wrong connection and to create a correct connection is identically.
Under this assumption the value is an exact value.
Verilog-Mode as it includes the corresponding hints and the original module instantiation is in the form Emacs Verilog-Mode creates.

We have measured the runtime of our approach for the different designs. The measurement was conducted on an Intel Core2 Duo 2GHz, running Archlinux 3.9.3-1 and all heuristics have been used. Table II shows the processor time our technique required. The column parsing gives the time required by the IcarusVerilog parser to generate the abstract syntax trees for the design with all its sub-modules. The technique has been applied as presented in this paper, without any special optimization. As our approach requires that each port-name is compared to each other port-name the runtime of our approach is quadratic in the number of ports.

In the following paragraphs we present the results for our approach. Due to space limitations, we only present the average results over all designs (Figure 2) as well as the detailed results for the designs OpenSPARC T1 core (Figure 3) and tiny-AES (Figure 4). The OpenSPARC T1 core has been chosen because it includes hints for the Emacs Verilog-Mode and therefore is considered optimal for Emacs Verilog-Mode. The tiny-AES design was chosen because it shows the quality of the result on average.

As already mentioned the OpenSPARC design is especially interesting for a comparison between our approach and Emacs Verilog-Mode because the design is optimized for the usage with Emacs Verilog-Mode. This includes a corresponding naming convention and hints for the connection generation process. The results are shown in Figure 3. First, we notice that again $S_{hf}$ achieves the best results and is able to achieve larger $q$-values than Emacs Verilog-Mode for most combinations of heuristics. This design nicely shows that coding conventions which help other automatic tools, or are even required for the tools to be used, help our approach as well. But our approach does not require a developer to add supplementary information for the tool, like the regular expression for the Verilog-Mode or the connection rules for MKTREE. Even in cases where regular expressions are used to define the correspondence of signals, like for Emacs Verilog-Mode, the port-names should still have a strong similarity, which is sufficient for our approach.

If we have a closer look at the connection strategies, we can see that the qualities of $S_{cm}$ and $S_{cmm}$ are heavily fluctuating. Changing a single heuristic can change the resulting $q$-value by 0.5 or more. For example, the combination of name matching with the Jaro string metric and extended name matching with Levenshtein achieves a $q$-value above 0.8. However, by changing the string metric for name matching, the $q$-value drops below 0.1. Similar effects appear for the designs MIPS789, OpenRISC 1200 and to some extent for the designs SD Mass Storage Controller and Wishbone-specification. This shows that $S_{cm}$ and $S_{cmm}$ are rather unreliable and hence, not very useful.

Considering the results for tiny-AES design in Figure 4, we see that the result deviates from the other designs with respect to two attributes. First, the connection strategies $S_{cm}$ and $S_{cmm}$ achieve the best results for all combinations of heuristics. Second, using $E_v$
improves the quality of the results for the tiny-AES design in 26
of 48 of the cases and never reduces the quality of the result. In
contrast, for the other designs using Ev only improves the result
in very rare cases and in many cases even reduces the quality
of the result. Interestingly, in all cases at least 27 correct connec-
tions has been created. This is the same number of connections
Emacs Verilog-Mode creates. Also, the best results are those where no other
connections have been created. In the cases where more than those 27
connections have been created, the quality of the additionally created
connections is not better than randomly created connections.

Further, the Wishbone-specification is to mention. Our approach
using Shf creates the correct wiring (q-value equals 1) for any
combination of heuristics which contain name matching or extended
name matching. Further, \(S_{em,hf}\) is able to create an optimal result in
most (15 out of 28) cases. The connection strategies \(S_{em}\) and \(S_{em,em}\)
only achieve q-values between 0.4 and 0.9. Again those combinations
achieve the lowest q-values which include the Ev heuristic.

Finally, we compare our approach with the results of Emacs
Verilog-Mode. Table III gives the results when applying Emacs
Verilog-Mode to the designs and values achieved by our approach.
The column “highest” shows the highest achieved q-value over all
combinations of heuristics and connection strategies. The column
“best avg.” shows the result for \(Nm_{Lev}\) combined with \(ENm_{Lev}\)
and the connection strategy \(S_{hf}\), which was the combination achiev-
ing the highest average q-value. First, we notice that in case of the
Wishbone-specification Emacs Verilog-Mode is not able to create
any connection. This is due the fact that the naming convention
used for the specification includes a suffix for the port direction.
Hence, there is no exact name matching. In contrast our approach

generates a perfect result. Second, in case of the SD Mass Storage
Controller Emacs Verilog-Mode only creates incorrect connections
resulting in a negative q-value. Finally, in case of the OpenSPARC T1
tcore, although created to be used with the Emacs Verilog-Mode and
corresponding hints are provided, incorrect connections are generated
and again the technique presented in this paper achieved better result.
Altogether we see that the presented approach is able to generate
connections at least as good as Emacs Verilog-Mode and in many
cases even better. Even as only the combination with the highest
average q-value is considered, our technique achieves better results
in seven of eight cases.

Additionally, the results for the Wishbone-interface, which repre-
sents one interface to combine blocks from different sources, suggest
that our approach is effective for combining several blocks from
different origin to a single chip with connection strategy \(S_{hf}\).
V. LIMITATIONS AND FUTURE WORK

One goal of this work is to understand the limitations of the light-weight techniques. In this section we discuss the limitation helps the approach to achieve very good results, like in the case of the Wishbone-interface for which a very strict naming convention is applied and a perfect result is achieved. Also in case of the OpenSPARC core, where a naming convention is used to allow efficient use of the Emac Verilog-Mode, the results are very good. However, the presented approach suffers from poor naming conventions.

Further, the approach suffers from its inability to create glue logic. Especially, the SD Mass Storage Controller uses multiplexer schemes to connect outputs of several sub-modules to one single input. Similarly, several designs use the concatenation of several signals as inputs to a sub-module.

We propose two advanced approaches for future work: First, a functional analysis that matches trigger conditions of one module with output sequences of other components. Second, validating the result of the automatically connected design with respect to the specification. Finally, as seen for the tiny-AES design, further work is required with respect to preventing incorrect connections.

VI. CONCLUSION

In this paper we have presented an approach for automatically creating the connections between different logic blocks. The presented approach does not apply computationally expensive functional analysis. Instead our approach uses several heuristics which either mark a connection as invalid or assign a score to the connections based on a string metric to decide whether they are likely to be correct.

We compared our approach against Emacs Verilog-Mode with respect to eight different designs of various sizes. Our approach achieved better results for seven of those designs and an equally good result for the eighth design. Especially, our approach was able to achieve better results for the design which was optimized for being used with Emacs Verilog-Mode. Our approach generated a perfect connection in case of the Wishbone-interface a standardized interface to connect different blocks.

Further, the evaluation showed the limitations of the light-weight techniques and thus the cases for which more advanced approaches have to be considered.