The Hardware Design Toolchain
Approaches and State of the Art

Fredo Erxleben

August 27, 2014

We will hate the tools (FCCM 1996 prediction for 2001)
We will still hate the tools (FCCM 1998 prediction for 2003)
We will merely dislike the tools (FCCM 2000 prediction for 2005)
We [will] hate the tools more (FCCM 2007 prediction for 2012)

1 Motivation

Since the introduction of integrated circuits, hardware complexity has increased rapidly and constantly. This complexity naturally is a hard thing for humans to handle once it reaches a certain threshold. As a consequence, the need for tools arises to enable the people involved in the hardware design process to continue working on, advancing and improving the matter. While this is a fact for any evolving branch of science and production, the speed, by which the tools adapt varies greatly. Taking software development as a comparison, we find that there are often a lot of tools available for one task, each one of them filling a niche or being tailored with a special use-case in mind. In addition, the integration between these tools becomes increasingly tight through the extensive use of plugins and common file types\(^1\). This allows the creation of well interlocking tool-chains with a high level of user comfort and productivity for virtually every requirement that might arise. In the following, a selection of tools used for hardware design will be presented in an attempt to outline where weaknesses in the currently available tool-chains for hardware design are found. Due to the sheer amount of different approaches made over the years and tools that were developed with the intention of helping to improve the design process, it is not possible to look at them all or in more detail. Instead, in the following, an overview over approaches made to create tool-chains for hardware design or single tools to be used in them, shall be given. It will also be outlined, what their current state in productive use is.

2 Criteria

When reviewing tools or tool-chains there are several aspects to look at. Some of the most essential ones are listed here, which influence how easily and productively humans can use the given software.

The field of application describes which and how many different use-cases a tool can

\(^1\)Famous examples for the first are *eclipse*, *emacs* or *vim*; For the second, one might think of *JSON*, *.tex* or *XML*
cover. This is often represented as points on the extended Gajski-Kuhn-chart\(^2\) [9].

**The design of human interaction** influences the way the user works with the software. It positively strongly affects the learning curve and productivity when using a tool, supporting multiple techniques for solving a given task, being intuitive and communicating effectively with the user.

**Availability and openness** are relevant for obtaining a broad user base and acceptance. People have very diverse premises with respect to hardware and operating systems. Being available for a lot of different setups makes it easier for more users to apply the tools, while available source code gives interested developers the opportunity to adapt and improve them.

**Tool-chain integration capabilities** refer to the ability of a tool to be extended by plugins or to operate with the file formats used by other software.

### 3 Languages Used in Hardware Design

Before researching tools themselves, a short overview over the hardware description languages these tools are based on is mandatory. Since virtually no hardware down to the mask level is publicly available, these languages serve as an interface typically at the register transfer level. Everything below that is normally specific to the hardware vendor who applies his own parser and compiler. While this practice enables a lot of optimization for the given hardware by the vendor, it also binds a considerable amount of his development workforce.

#### 3.1 VHDL

VHDL serves as standard in the world of hardware design and most tools support processing or creating hardware descriptions in this language. This is mostly due to its wide application range that allows to use VHDL for descriptions on different abstraction levels: For hardware interfaces, implementations as well as test benches as shown in figure 1. As a trade-off this makes the language very diverse\(^3\), and structurally complex. Its origin is found in languages like *Ada*, which makes learning it more difficult for people more used to languages like *C/C++* or *Java*.

![Figure 1: Green dots show the application range of VHDL](image)

#### 3.2 Verilog

An also often used HDL is *Verilog*. While the basic *Verilog* language can not cover the same range of application as *VHDL*, extensions like *SystemVerilog* exist, that increase the amount of use-cases. In combination with compilers that support its features, *SystemVerilog* is also used for the verification of hardware designs. Since *Verilog* and its variations are designed to resemble *C*-style languages, it tends to be used

---

\(^2\)Since testing is such an important task in the hardware design process, it is only reasonable to include it when discussing use-case coverage of tools.

\(^3\)VHDL has 99 reserved keywords [16].

\(^4\)An extensive discussion about the pros and cons of teaching either *VHDL* or *Verilog* can be found at [http://ask.slashdot.org/story/09/05/31/187208/vhdl-or-verilog-for-learning-fpgas](http://ask.slashdot.org/story/09/05/31/187208/vhdl-or-verilog-for-learning-fpgas).
like a procedural language by new users, which contradicts the way an HDL works\(^4\).

3.3 The Relation between Languages and Editing Tools

It could be argued that any tool, capable of handling a given HDL also inherits the languages descriptive potential. But this ignores the fact that hardware designs tend to get very complex very fast and it is in the applied editing tools responsibility to assist the user by providing

- Different views on the design, e.g. by offering schematic representations as well as textual ones or allowing certain things to be blended out.
- Means of navigating the code easily
- Aids for reading and interpreting the code\(^5\)
- Possible code completions
- Shortcuts for often repeated tasks\(^6\)
- Support for code refactoring
- Enforcement of conventions\(^7\)

Currently, editing tools deployed in IDEs like ISE, Quartus, Sigasi Pro or Vivado only cover a small subset of these requirements. This lack of user support has a notable negative impact upon the hardware designers productivity [20].

4 Tools for Specifications

Whenever a hardware design is to be made the first thing to be done is a specification describing the intended structure, behavior and limits. In the most cases these specifications are done in textual form, aided by the use of tables or graphical sketches. Apart from this very general similarity there seems to be no consensus at all about how a specification is to be written. There have been case studies to show that UML might be deployed to improve specifications and reduce ambiguity compared to nearly pure textual variants [1]. In addition it has been tried to embed a formal specification into VHDL and to show that this could be used for automated verification [24].

5 High-level synthesis

High-level synthesis usually describes the transformation of a hardware description given in an abstract representation\(^8\) into a form which can be synthesized directly into hardware.

5.1 HLS from higher level languages

A current trend are attempts to generate an HDL description from other established programming languages. These approaches initially explored functional languages like Haskell [6] because of the relative ease of translating them into HDL [25]. Other efforts have been made to re-purpose existing software programming languages like C/C++ [27], Java (especially referring to JHDL [4]) or using Matlab [2].

5.2 HLS from visual representations

Starting the design process with sketching a visual representation has proven a good practice in the software world, where UML has become an important tool for development. Up to now, there is no widely accepted equivalent for the hardware world, even though there have been made efforts to either use UML itself for hardware description [7] or create a visual representation for VHDL [21]. Still, no holistic agreed upon visual representation for hardware designs exist so far\(^9\) even though suggestions were made [11]. As a result there are nearly no tools

---

\(^5\)Mostly associated with syntax highlighting, this also covers context based help

\(^6\)Think of instantiating a component

\(^7\)For example naming conventions, code formatting or required documentation

\(^8\)For example, as a program written in a high-level language or a schematic representation of the structure.

\(^9\)Even for the most basic elements like logic gates there exist multiple wide-spread circuit symbols
that employ a possibility of designing hardware by creating visual representations of either the hardware structure or behavior\textsuperscript{10}. Further research and development regarding the topic seems to have nearly ceased after the basics have been patented \cite{26}. Promising approaches have been made by HDLDesigner, Qucs and fritzing IDE\textsuperscript{11}. The latter offers the design to be made from a view representing either the setup on a breadboard or a schematic representation to be refined later as a PCB layout. Examples are shown in figure 2 and, regarding Qucs, figure 3a in the appendix.

\section*{6 HDL generation}

\subsection*{6.1 HDL generation by evolutionary algorithms}

There have been approaches to use evolutionary algorithms to generate hardware descriptions. While this might have potential for optimizing existing designs, it was shown that designs made from scratch by this method tends to create solutions that are not practically usable. If the specified limiting conditions for the generator algorithm are not exhaustive, the resulting design might violate any implicitly made assumptions. A good example for this phenomenon can be found at \cite{18}.

\subsection*{6.2 HDL generation by dedicated tools}

A further possibility of generating HDL is the use of dedicated programs that are developed for creating efficient solutions for a given design problem. Using this approach, the user has a method on hand to quickly achieve a custom tailored design without having to spend much time on either adapting existing ones or recreating them from scratch.

\subsection*{6.2.1 Example: FloPoCo}

FloPoCo \cite{8} is a generator for arithmetic cores, primarily aimed at FPGAs. It allows for the fast generation of arithmetic operators implementations as well as the user-provided definition of completely new operators and test benches. Designs created with FloPoCo tend to be very efficient with respect to hardware usage and execution speed due to the optimization done by the generator. While its command-line based nature does not offer the comfort, one might be used to have in graphical environments\textsuperscript{12}, it allows the easy integration into other software.

\section*{7 Test and Verification}

Testing a design and verifying that it meets the specification it is based on, are often repeated tasks during development. Because verification is tightly linked with the specification of the design, it obviously benefits from a formal representation that can be evaluated automatically.

\subsection*{7.1 Challenges Faced when Testing a Design}

Since most hardware can not be altered after production, any error in the implementation may result in the hardware becoming unusable for its intended purpose. As a result, a lot of development time is invested into testing and verification\textsuperscript{13}. Both tasks have shown to be difficult for several reasons:

- There are no tools available, \emph{explicitly destined} for testing hardware designs or the \emph{comfortable} creation of testbenches.
- Tool-assisted verification of a design requires the specification to be formulated in a way available proving algorithms can understand. It was mentioned in section 4 that this has not yet been established apart from research approaches.

\textsuperscript{10}For example as flow charts or in the form of state machines
\textsuperscript{11}fritzing does not offer an export into an intermediate hardware language but can generate SPICE-netlists
\textsuperscript{12}Provided they were done properly.
\textsuperscript{13}Common estimations are around half of the total development time.
• Testing is pretty much limited to either checking assertions at runtime, if the used HDL supports it, or manually inspecting waveforms as the result of a simulation. Using hardware interfaces like JTAG produces a huge amount of data and requires specialized tools to find the information relevant for the user.

Also, testing effort is impacted heavily by translation and, in the case of testing a synthesized design, place-and-route times, as well as the frequency of test runs. Research proposals and estimations about their benefits already have been made [20].

7.2 Verification with PROMELA and SPIN

As shown in [22], it is possible to formulate a hardware design as a PROMELA\footnote{Process Meta Language}-model. This model can be verified by the open source model checker SPIN. Apart from this proof of concept there seems to have been no further work on automated translation of hardware descriptions into PROMELA models, integration of the shown approach into tools used in production or further research on the possibilities and limits of the presented technique.

8 Documentation

Any hardware design is most likely complex enough to require an extensive documentation for users and developers alike. This demand is further increased by the habit of developers to use abbreviations, macros or other shortcuts the applied design environment might have to offer. As a result, any design which is not sufficiently documented, will be hard to understand for anybody not familiar with it. This in turn increases the time users or other developers have to spend on trying to figure out the behavior of a given design instead of working productively with it. While a lot of documentation tools exist in the software world, for hardware designs automated generation of documentation files is neither as advanced nor as common. A reason might be, that HDLs tend to be difficult to parse, making it hard for automated tools to extract relevant information. Attempts to an integrated documentation focus on VHDL and are present in the form of VHDL-Doc, the doxygen VHDL-module and VHDocl. An example of the use of included documentation is given by Sigasi Pro as shown in figure 3b.

9 Tool-Chains

In most cases, having only one tool will not suffice to cover all use-cases that the design process brings with it. Therefor the need arises to combine tools into tool-chains. In practice this has proven to be a difficult task.

9.1 Exchange Formats

For any tool to communicate with another, it is necessary to have a way to represent the transferred information which is understood by both. It is the purpose of an exchange format to provide such means of common ground. While in some cases a wide-spread HDL can serve this purpose, there are often enough cases where either the sets of HDLs, supported by the tools in question, do not intersect or are not suited to represent the layer of abstraction on which the exchange is supposed to happen\footnote{VHDL, for example is not suited to represent a netlist}.

9.1.1 EDIF

The Electronic Design Interchange Format was first released in 1985 with the intention to create a unified exchange format to be used as a standard by tool vendors. There exists a multitude of vendor-specific implementations, varying in features that are supported. This in turn defies the intended purpose of EDIF to be a universal exchange format which eventually led to the discontinuation of its development. Many tools still support the format to offer some kind of backwards-compatibility.
9.1.2 BLIF

Another approach to the problem of communication between tools was made by the Berkeley Logic Interchange Format, developed by the University of California in 1992. While tool support for this format seems to be very sparse, the format or its subset BLIF-MV [17] is used for academic research regarding automated reasoning about circuit structures [3]. An EDIF to BLIF converter exists [19]. There are also implementations for conversion between VHDL and BLIF. For that task, tools like blif2vst or vl2mv can be found, although only as source code.

9.2 Tool Interoperability

Attempting to create tool-chains out of the existing tools turns out to be difficult. Proprietary software, like ISE, Vivado or Quartus and others, normally is not working well with other products but rather supplies all functions needed for the design process by itself and uses also proprietary formats for intermediate files. When, for example, comparing the supported file formats for ISE [17] and Quartus [18] the common ground reduces to VHDL, Verilog and EDIF.

Specialized tools, that only cover parts of the design process are also rarely build to operate with wide varieties of formats. In table 1 in the appendix, a set of these kind of software is given, together with the use case they focus on and possible ways of information in- and output, where applicable. Given the number of different options some tools support, it is not helpful to list all of them with all possible variations and capabilities. Here, the selection is limited to one intended use-case and the usage of VHDL to give an idea of which components a tool-chain might be constructed. Another, more elaborate, overview over a variety of tools is provided by [9]. If and how a given tool can be fit into an intended tool-chain always has to be decided for the individual case. Vice versa, finding a tool that fulfills given requirements can make extensive research necessary.

Even with the right choice of tools, it is still mostly up to the user to create a tool-chain by applying the software in question manually or using scripts. In an attempt to ease this task, Sigasi Pro uses the features provided by Eclipse for the creation of a tool chain.

10 Open-Source Tools

10.1 Considerations on Using Open-Source Tools for Hardware Design

Being open source and, as a consequence freely available, gives a tool certain advantages:

- It is easier for the tool to become widespread, since no company policy, copyright or licensing hinders the application of the tool by anybody. This aspect also plays an important role when considering portability of tools between different platform configurations.
- Users can give feedback more directly and observe how this feedback is handled. They might even be able to provide requested changes by themselves.
- Adaptation or extension to a given use-case can be done with less effort then in a closed-source solution.
- Open source software has a very high number of potential developers.

All of these points do not guarantee for a tool to be able to compete with commercial solutions, as long as they are not exploited. Furthermore, most of the tools needed for hardware

---

16 A plausible strategy behind this behavior is to tie the customer to a companies products in the attempt to secure market shares. Following this reasoning, tools of different vendors are supposed to not operate well with others.


19 It is not common for company-developed tools to even provide an API for extensions. Notable exceptions are Xilinx’ JBits (previously XBI) [14] [13]
design tend to be very specific and complex software. They require a quite large amount of initial work before they become interesting enough for other users to join the development. Since hardware specifications tend to be treated as company secrets, even open source tools may have to rely on the hardware providers implementation at some point. To circumvent these and other problems, there are an increasing amount of openly available hardware platforms [28].

10.2 Currently Available Open-Source Tools

When looking into existing open source tools for hardware design and synthesis, it is notable that there are a number of programs available for many different use cases\(^\text{20}\). Some of these projects are no longer actively developed as of 2014, but still in a usable state. Already mentioned where Qucs and fritzing, which both are currently active and promising projects. The research in JHDL spawned the JHDL-CAD tool [15] while for VHDL one might want to employ the GHDL or FreeHDL environment. Both are command-line based, but can generate simulation environments for a given design\(^\text{21}\). For low-level synthesis and verification, the toolset ABC [12][10] is available and actively developed.

Most companies, producing software for hardware design, rarely offer full support for operating systems like Linux or BSD. In most cases, some packages for wide-spread distributions are available, albeit mostly neither in official repositories nor managed by a package maintainer.

11 Improvements

Modern tools used in hardware design have been developed in great numbers and are an active research topic. Still they are often perceived as inadequate when compared to tools used in software engineering\(^\text{22}\).

11.1 Suggested Improvements

A lot of improvements and research thrusts have already been made [20]. They cover the need for better algorithms and specialized tools for specific parts of the tool-chain as well as the need to integrate these tools better into each other. It was also observed that design decisions very often tend to be made ad-hoc while implementing and that often overlooked points in the overall process still are specification and formulation of design properties and restrictions. Improvements in this area will most likely impact the verification process positively. Due to the complexity of the hardware-design process, no tool will be able to fulfill all the arising needs properly. Also, being not able to always employ the most fitting tool for a certain part of the design process, notably decreases productivity. As a consequence, chaining tools together becomes more important, as more specialized tools get involved. This requires them to support for multitudes of exchange formats, which has been rarely achieved as of the time of writing.

11.2 Further Challenges

Additionally to the challenges already outlined by others, there are other tool-related topics that need improvement to further enhance the ease, quality and speed with which hardware can be designed. Most tools still need a lot of experience and in-depth knowledge about hardware design to be used effectively. As a consequence it is necessary that tool producers look a lot more into user interaction and usability issues. If a future tool would be easier to use, it will also become easier to learn and adapt and thus improve the teaching of hardware de-

---

20 Most of them are small and not very much advertised, so it is challenging to be aware these tools exist and find reliable sources.

21 The FreeHDL simulator is generated as C source code which may then be compiled. This allows for modification of the simulator with relative ease.

22 Software engineering is most likely referred to by hardware designers, since this is a field of application they often experience first hand.
sign since the new user can focus more on the task itself then struggling with the software to achieve the intended result. Further support for teaching as well as production would come from support offered by the tool in the form of improved content awareness. This would for example enable the software to offer support for the used HDL as discussed in section 3.3, point out possible design flaws or suggest appropriate design patterns.

Since the currently available tools mostly fall either in the category of relatively short-lived purely academic approaches or, on the other hand, are developed and maintained by companies that are keen on guarding their intellectual property, lots of development potential for improving the design tools, residing in their user base, is wasted\textsuperscript{23}. While the impacts and benefits of open source collaborative development are a never ending hot topic ([23][5] et al.), it is agreed upon, that a tight and direct connection between users and developers can greatly benefit a product.

\textsuperscript{23}Again, it is most likely that hardware designers are also somewhat knowledgeable regarding software engineering.
List of referenced tools

In the following section, an overview over the previously referenced tools will be given. For each of them the license, the developer or developing company and a link for further information will be given. Also it will be shortly explained what their field of application is.

**ABC** is a set of tools to synthesize and verify low-level hardware descriptions. Since it originated at Berkeley University, it consequently supports the **BLIF** format. The source code can be found at [https://bitbucket.org/alanmi/abc/overview](https://bitbucket.org/alanmi/abc/overview).

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley Logic Synthesis and Verification Group</td>
<td>Unspecified</td>
<td><a href="http://www.eecs.berkeley.edu/~alanmi/abc">http://www.eecs.berkeley.edu/~alanmi/abc</a></td>
</tr>
</tbody>
</table>

**doxygen** is a documentation tool that supports inline documentation for multiple languages, including VHDL.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimitri van Heesch</td>
<td>GNU GPL</td>
<td><a href="http://www.doxygen.org">www.doxygen.org</a></td>
</tr>
</tbody>
</table>

**FloPoCo** is a generator for floating point cores. Based on given parameters it can do high level synthesis for custom pipelines and operators. (See also section 6.2.1)

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community-based (Project lead: Florent de Dinechin)</td>
<td>under discussion as of 2014</td>
<td><a href="http://flopoco.gforge.inria.fr">flopoco.gforge.inria.fr</a></td>
</tr>
</tbody>
</table>

**FreeHDL** is an open-source generator for hardware-design simulators. While it can be used stand-alone it is more often found as a backend of tools like Qucs.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
</table>

**Fritzing** is an open source hardware project that provides a whole development chain from an IDE down to custom manufactured PCBs. Although being more oriented towards hobbyists, this tool offers potential towards educational use, especially due to being freely available.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community-based</td>
<td>GNU GPL, version 3</td>
<td><a href="http://fritzing.org">fritzing.org</a></td>
</tr>
</tbody>
</table>

**GHDL** is an **gcc**-based **VHDL**-simulation generator. It uses **VHDL** files as input to generate and compile executable simulators for the given design.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community-based (Project lead: Tristan Gingold)</td>
<td>GNU GPL</td>
<td><a href="http://ghdl.free.fr/">ghdl.free.fr/</a></td>
</tr>
</tbody>
</table>

**HDLDesigner** is an IDE that supposedly features a complete tool-chain and visual editing capabilities. Even acquiring the trial version requires a lengthy registration with the producer,
so it was no further looked into the capabilities of this software.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mentor Graphics Corporation</td>
<td>Proprietary</td>
<td><a href="http://www.mentor.com">www.mentor.com</a></td>
</tr>
</tbody>
</table>

**ISE** is an IDE specialized at development for Xilinx' FPGAs and CPLDs and the predecessor to **Vivado**. It offers basic project management, some editing capabilities, hardware design simulation and -synthesis. While being freely available in the basic version, buying a license is required if one wants to use all features. In any case a registration with Xilinx is mandatory.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Inc.</td>
<td>Proprietary</td>
<td><a href="http://www.xilinx.com">www.xilinx.com</a></td>
</tr>
</tbody>
</table>

**JHDL-CAD** is a set of tools compiled into a design created with **JHDL**. As of 2014 it appears to remain a purely academic approach and further development seems to be discontinued. Support for **EDIF** can be found separately at [http://reliability.ee.byu.edu/edif/](http://reliability.ee.byu.edu/edif/).

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brigham Young University</td>
<td>Custom open source license</td>
<td><a href="http://www.jhdl.org/">www.jhdl.org/</a></td>
</tr>
</tbody>
</table>

**Quartus** is an IDE specialized at development for Altera FPGAs and CPLDs. It offers basic project management, editing capabilities, design flow planning, hardware simulation and synthesis. While being freely available in the basic version, buying a license is required if one wants to use all features. In any case a registration with Altera is mandatory.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera Corporation</td>
<td>Proprietary</td>
<td><a href="http://www.altera.com">www.altera.com</a></td>
</tr>
</tbody>
</table>

**Qucs** is a tool for digital and analog circuit design with focus on visual representation. A notable capability is the integration of components specified by **VHDL**- or **Verilog**- files. It also features a simulator based on **FreeHDL**.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community-based</td>
<td>GNU GPL</td>
<td><a href="http://qucs.sourceforge.net/">http://qucs.sourceforge.net/</a></td>
</tr>
</tbody>
</table>

**Sigasi Pro** is a HDL-editing IDE based on the **Eclipse**-platform. It supports **VHDL** as well as **Verilog** and provides a lot of the features outlined in section 3.3. The software is prepared to be set up as part of a tool-chain with other tools like **GHDL**. A trial version is freely available from the **Eclipse marketplace**.

<table>
<thead>
<tr>
<th>Developed by</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sigasi nv</td>
<td>Proprietary</td>
<td><a href="http://www.sigasi.com">www.sigasi.com</a></td>
</tr>
</tbody>
</table>

**SPIN** is a freely available model checker based on **PROMELA**-models. Due to the descriptive capabilities of the modeling language, **SPIN** is often used in a many different scientific
VHDLDoc is a basic VHDL documentation tool that originated from kdoc. The used markup language attempts to be close to JavaDoc.

VHDLDoc

Developed by Christoph Schwick
License GNU license
Link schwick.home.cern.ch/schwick/vhdl.doc

VHDocl is a VHDL documentation tool that attempts to improve where doxygen and VHDLDoc have lacking. It is capable of working with doxygen-style documentation as well as its own notation.

VHDocl

Developed by Volker Schatz
License GNU GPL, version 3
Link www.volkerschatz.com/hardware/vhdocl.html

Vivado is an IDE specialized at development for Xilinx’ FPGAs and CPLDs and the successor of ISE. It offers basic project management, some editing capabilities, hardware design simulation and -synthesis. While being freely available in the basic version, buying a license is required if one wants to use all features. In any case a registration with Xilinx is mandatory.

Vivado

Developed by Xilinx Inc.
License Proprietary
Link www.xilinx.com

References


Figure 2: Different visual representations offered by fritzing for creating a circuit.

![The platine view](image1.png)
![The schematic view](image2.png)
![The PCB layout view](image3.png)

Figure 3: Qucs example of pop-up documentation.

![A simple design represented by Qucs](image4.png)

Table 1: Possible building blocks of a tool-chain for hardware design

<table>
<thead>
<tr>
<th>Tool</th>
<th>Focus</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>doxygen</td>
<td>Documentation</td>
<td>Annotated VHDL</td>
<td>HTML</td>
</tr>
<tr>
<td>FloPoCo</td>
<td>High-level Synthesis</td>
<td>Command-line parameters</td>
<td>VHDL Executable simulator</td>
</tr>
<tr>
<td>FreeHDL</td>
<td>Simulation</td>
<td>VHDL</td>
<td>Executable simulator</td>
</tr>
<tr>
<td>GHDL</td>
<td>Simulation</td>
<td>Java source code (VHDL)</td>
<td>VHDL Executable simulator</td>
</tr>
<tr>
<td>JHDL-CAD</td>
<td>High-level synthesis</td>
<td>VHDL</td>
<td>VHDL Executable simulator</td>
</tr>
<tr>
<td>Sigasi Pro</td>
<td>HDL Editing</td>
<td>Annotated VHDL</td>
<td>HTML</td>
</tr>
<tr>
<td>SPIN</td>
<td>Verification</td>
<td>VHDL</td>
<td>Model checker as C</td>
</tr>
<tr>
<td>VHDLDoc</td>
<td>Documentation</td>
<td>Annotated VHDL</td>
<td>HTML</td>
</tr>
<tr>
<td>VHDocl</td>
<td>Documentation</td>
<td>Annotated VHDL</td>
<td>HTML</td>
</tr>
</tbody>
</table>