Connecting a C++ based Structural Verification Tool to the Web

A new bridge between C++ and the Web

Carsten Schmitt, Christoph Jäschke, Claudia Wolkober, Ulla Herter
IBM Deutschland Research & Development GmbH
Schönaicherstr. 220
71032 Böblingen, Germany
{carsten.schmitt,jaeschke,ck,ulla.herter}@de.ibm.com

Abstract.

This paper presents a new approach to combine a web front-end with a C++ application. It is based on the integration of a web server and an interpreter into the C++ application using dynamic linking while the interface between the web scripts and the application is automatically generated using the open source Simplified Wrapper and Interface Generator (SWIG). This way accessing data and functionality of the C++ process from web scripts is possible. Recompiling is only required if either the C++ application or its interface to the front-end changes. All other changes to the web front-end can even be made while the C++ application is running. This kind of software architecture has proven to work well in context of a C++ based structural verification tool. The described concept is used to present the verification results and the tool's functionality to the user in a web browser and thus improves the overall usability. Especially visualizing parts of the circuit logic in the browser is helpful and improves the collaboration of the hardware development teams across multiple sites while keeping the development effort of the front-end under control.

1. Introduction

The development and verification of today's server processors is getting more and more complex. In the field of hardware verification many different methods and tools exist to perform this task. A global, distributed environment puts new demands on developers and toolchains. One aspect in this context is the usability of tools and their support for collaboration. While there are many applications which have already been moved to the cloud[1], there exist lots of applications which have been developed prior to the web area or without the intention of providing a web front-end to the user. Many of those applications have been developed using C++ for various reasons. In this work, a concept is presented which allows the elegant combination of a C++ application for structural hardware verification and a web front-end in order to improve the usability of the tool.

[1] Especially newly developed social platforms and a variety of business services.

The paper is arranged in eight sections. After this introduction, section 2 provides a quick overview on the concept of grammar based structural verification. Afterwards, a C++ structural verification tool is introduced, for which we have developed a web front-end based on the concept described in this paper. As a basis of understanding, section 3 then introduces some key terms and concepts from the field of web technology. Following, section 4 discusses the requirements and circumstances, which led to the conclusion that using web technology in combination with our C++ tool is advantageous. The next section provides an overview on related work which has been evaluated in context of this paper. The main section 6 then in detail describes our concept of bridging a C++ application and a web front-end. Subsequently, section 7 provides a small example which should help to better understand the previously described concept. The paper concludes with section 8 where the overall results are summarized and possible future items are discussed.

2. Grammar based Structural Verification with a C++ Tool

Grammar based structural verification is one method to verify that circuits are logically correct before they go into production. In contrast to performing a simulation by stimulating the inputs of a black box and checking if the outputs behave as expected, structural verification works differently. As the name suggests, in structural verification one performs a static analysis looking at the structure of a circuit. In this context structure roughly means the connectivity and combination of the different hardware gates and components. The analysis usually takes place on a netlist. For example the clocking of a chip makes use of some typical components like Clock Controllers (CCs), Local Clock Buffers (LCBs) and latches. Those components have to be interconnected in a certain manner to make the clocking work properly. In structural verification, one indirectly verifies the functionality of the chip by verifying that the components are hooked up correctly. There are different ways to perform this task - one is to use a kind of EBNF grammar to define the expected structures. One advantage of this approach is the compact description of the expected structures.
We have developed a C++ verification tool [6] which is able to verify the structure on netlists which reach from a small component to a huge full chip (millions of gates). The tool has been implemented in C++ to meet the resulting memory and performance requirements. The language offers many degrees of freedom and hence allows fine grained design decisions for example with respect to memory usage. From the user perspective, the C++ program is a console application without interaction. Once started with a certain configuration, it performs the analysis task and writes the results into various text files after completion. Beside other results one is a list with detected design problems. Over the time, a list of usability requirements arose - especially with respect to the readability of the output.

3. Web Technologies Overview

This section covers some basic web concepts which are used later in this paper. More or less the World Wide Web is based on three core ideas - the Hypertext Markup Language (HTML), the Hypertext Transfer Protocol (HTTP) and Uniform Resource Locators (URLs). Basically the web content itself is written in HTML. The content is hosted by a web server which can be accessed using HTTP while a certain content is available under a certain

URL. A client (usually a browser) now requests a certain URL from the web server[2]. The HTML content is transferred from the web server to the client and afterwards interpreted and rendered by the browser.
While the content on the web server has been static at first, it nowadays can be dynamic as well. Dynamic in this context, means that the HTML returned to the browser is generated on server-side right after the request comes in. Instead of reading in plain HTML content, the web server reads in a script which is then passed to an interpreter. One of those interpreters developed over time is the Hypertext Preprocessor (PHP) [15]. The interpreter executes the script and returns the generated HTML back to the web server. This allows the creation of dynamic contents - for example based on a database.
However, even a dynamically generated website would remain static within the browser without an additional scripting language executed directly in the browser. In this context JavaScript established itself as a standard. Initially it is stored on the web server like the HTML content and likewise transferred to the client where it is then executed. JavaScript allows manipulating the HTML tags and thus the appearance of the website. In addition it allows asynchronous HTTP calls (i.e. without reloading the complete website) to the web server in order to reload additional content[3]. This concept is also known as Asynchronous JavaScript and XML (AJAX).
The last item to address here is the Scalable Vector Graphics (SVG), which is a vector image format for two-dimensional graphics with support for interactivity and animation. SVGs can be stored or generated on server-side and then efficiently transferred to the client. Afterwards, the browser takes over the rendering of the SVG. Using JavaScript the developer is able to manipulate the appearance of the SVG in the browser and it is even possible to embed JavaScript code inside the SVG which can be hooked up to certain SVG events. In this work, SVGs are used to display an interactive circuit [13] to the user directly in the browser as shown in figure 1.

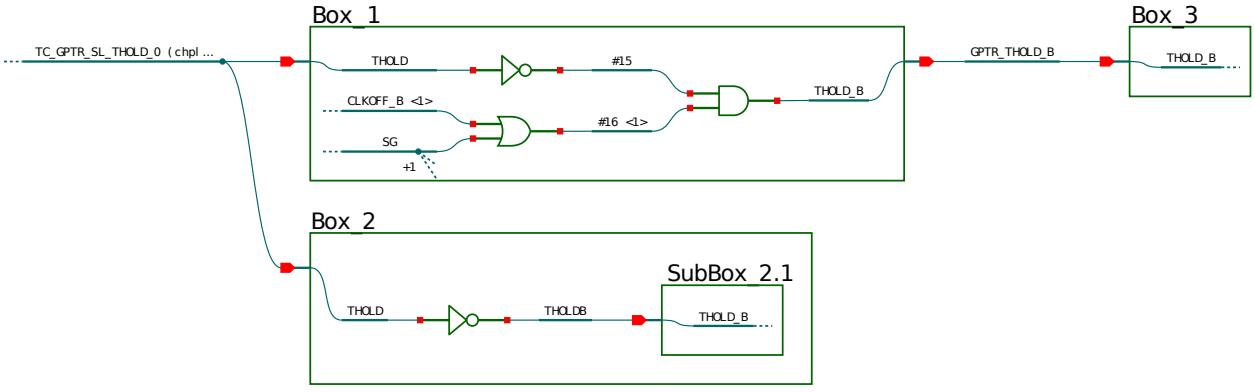

Figure 1: Interactive circuit SVG rendered by the C++ application and displayed in the browser.

[2]The URL addresses the web server and the desired content.

[3]Search engines use this mechanism to propose search results to the user while still typing.

4. New Requirements and Considerations for a Web Front-end

As mentioned in section 2, new usability requirements came up. In general, a more intuitive processing of the results and interactive usage of the available data were requested. Translated to features, this more or less boiled down to a graphical representation of the traversed circuit parts, a list of detected design problems, a search engine for components, and some other convenience functions[4]. From the perspective of front-end development the requirements so far could all be met by a variety of GUI toolkits.
However, there are further requirements which led us to the development of a web application instead of a C++ GUI. One is the ability to run on multiple host- and client environments. When building a web application, the support for all platforms comes for free[5]. Furthermore, accessing the application boils down to clicking a link instead of starting an executable. This also implies that no additional deployment of the software to the user is required.
Another important aspect is collaboration. Debugging a hardware problem often involves multiple hardware designers and verification engineers from various sites and components. This implies the need for sharing data across sites. In case of using a web front-end, a user is able to share data just by copy-pasting a link to colleagues. In this context the security aspect also plays an important role. Web applications nowadays provide some standard security concepts. For example, an encryption between client and web server via HTTPS is already built-in into most browsers.
Still, one constraint to keep in mind is implied by the web architecture itself. Conceptually, having a network between back end and front-end can increase the response time of the front-end. The delays can usually be kept low by only transferring a subset of the data to the client[6].
For the given reasons, we have decided to build a web front-end for our C++ verification tool. In this context, beside the development of the front-end itself, the aspect of bridging the C++ application and the web front-end became a central task. Our solution is presented in section 6.

5. Related Work

There already exist different ways to bring C++ and web development together. The C++ Web Toolkit (WT) project [4] is a library for developing web applications with C++. The API is widget-centric and uses patterns of desktop GUI development tailored to the web. The web design itself takes place in C++ using the offered components. CPPCMS [3] uses its own template language which then is converted into a C++ source file. TreeFrog [7] is a web application framework that generates MVC like C++ code patterns based on the input parameters. Afterwards, the generated code framework can be enriched with functionality and compiled. C++ Server Pages (CSP) [11] allow an integration of C++ code into a HTML template. A CSP compiler then takes the template file and translates it into a C++ file.

[4] For example, viewing the HDL, generating status charts or linking to a design documentation.
[5] The availability of a suitable browser is sufficient.
[6] In our case this is for example relevant when displaying entries of the netlist.

This file is compiled using a C++ compiler[7]. All those approaches have in common that they require a recompilation of the C++ source code after each front-end change.
Another way to connect a C++ application with a web front-end is the Common Gateway Interface (CGI). CGI is an extension of a common HTTP server which allows the execution of scripts and binaries[8] located in a configured directory. The script or binary is executed by the web server in another process. The communication between the web server and the executed program takes place via pipes. Hence, there is no way to directly access any data structures of the executable - especially not of an already running process.

6. Creating a Bridge between C++ and the Web

In order to build the web front-end described in this work, we use state-of-the-art web technologies like PHP, HTML, CSS and JavaScript. However, note that the presented concept is generic and not necessarily bound to the use of PHP. Thus, the more generic term scripting language will be used for the following concept description.
As described in section 3, the web is based on a client-server concept. Hence, in order to allow a client (browser) to access any kind of data, a web server is required. There are multiple implementations available which could be used to embed a web server into a C++ application - for example mongoose [9], libhttpserver [10] and libmicrohttpd [5]. Up to this point, the idea of having an embedded web server as shown in figure 2 is not new. Well known applications like the Open source VideoLAN Media Player (VLC) [12] provide web interfaces to the user using a built-in web server. The C++ web toolkits described in section 5 are supplied with an embedded web server as well. However, one major drawback of this approach is, that each modification of the web front-end requires a recompilation of either a template or the (mostly generated) C++ code. In addition, some approaches force the developer to use partly inappropriate languages to create the web front-end. The concept presented in this work avoids those drawbacks.

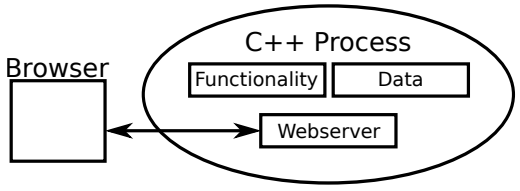

Figure 2: A C++ application with an integrated web server.

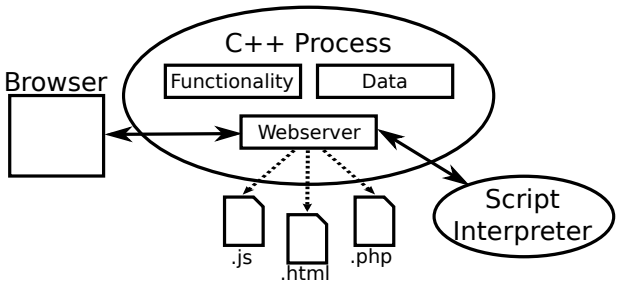

Figure 3: Front-end described by web files stored outside the C++ application.

In order to avoid the recompilation step, the web front-end source code and the C++ code have to be split. That means the front-end can no longer be part of the C++ application as well as the C++ code cannot be part of a scripting language any longer. In order to achieve that, the concept shown in figure 2 has to be extended to what is shown in figure 3.

[7] This works similar to the Active Server Pages (ASP) - `http://www.asp.net`

[8] This can be any kind of executable - for C++ see `http://www.gnu.org/software/cgicc`

The web front-end files (e.g. HTML, JavaScript, CSS, PHP, images and other web sources) are located in a folder and therefore are independently stored from the C++ application (just like in a common web server setup). Once a request comes in to the web server, the respective resource is read and processed. In this context, the processing of script files requires a suitable script interpreter. The concept as shown can already execute scripts using the interpreter. In case the user accesses a PHP file, the script is read and handed over to the interpreter. The interpreter executes the script and finally returns the result to the web server. Afterwards, the web server sends the result further back to the client which issued the initial request. Typically this concept is implemented using a fork-exec construct [8]. The web server forks itself and the new child process then does an exec call to the interpreter. The original web server process (the parent process) now can communicate via pipes with the interpreter (the child process). However, so far there is no mechanism which allows accessing any C++ data or functionality from a script. In order to make that possible, two additional changes have to be made to the concept.

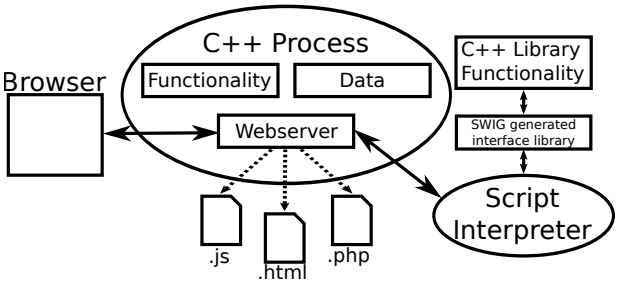

Figure 4: Interface to C++ library outside actual application generated by SWIG.

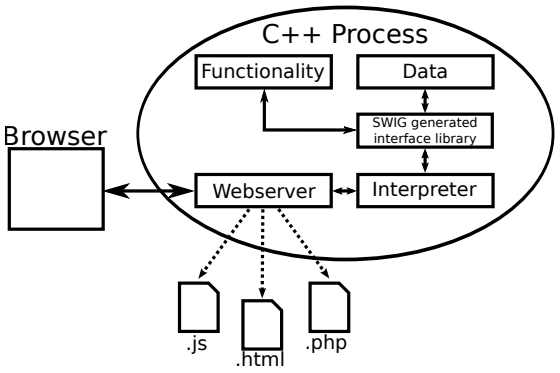

Figure 5: Interface to C++ application with integrated interpreter generated by SWIG.

Figure 4 shows the next concept stage. In order to allow access from a script to a C++ application, a software development tool called Simplified Wrapper and Interface Generator (SWIG) can be used [2]. This software package allows the generation of an interface between a C++ application and a scripting language. Currently about 25 different scripting languages are supported. The only required input to this tool is an interface description which lists the C++ classes and functions which should be made available to the scripting language. For example, when choosing PHP as scripting language, the code for a shared interface library and a `.php` file is generated. After compilation, the shared interface library can be loaded as a PHP extension by the interpreter and the `.php` file is just included. Finally, calling a C++ function from a PHP script just looks like a call to a PHP function (see listing 3). This way, recompiling the C++ source is only required if it has changed. Furthermore, executing the SWIG compiler is only required if the C++ interface has been modified. Changes to the web front-end, which do not involve interface changes, do not require any recompiling and can even be issued while the C++ application is running.
Finally, the only remaining problem is that the interpreter has been started as a new process in a completely new address space. That means, so far the only way to communicate between parent- and child-process is the pipe between them. This implies that even loading the generated shared interface library as an extension does not allow access to the parent-process.

Instead, an undefined reference problem will be issued when the shared interface library is loaded. This can be solved by doing a fourth concept modification as shown in figure 5.
One way to achieve that is moving the interpreter into the same process instead of calling it as a child-process from the web server. This turned out to be relatively simple since many interpreters are itself implemented in C and hence have a suitable C API. The web server then needs to be modified, so that instead of executing the interpreter as a child-process, it executes the interpreter directly via its C API. We managed to embed Python, Perl and PHP interpreters for testing purposes using their C APIs [9]. Finally, all components - the web server, the interpreter and some additional program logic - are packed into one dynamic library which then is loaded dynamically by the main C++ application.
Based on this final concept, the front-end can be modified without recompiling the C++ source code and even without restarting the C++ application - at least as long as the interface does not change. Furthermore, no export or import of data is required and C++ functionality can directly be executed from the browser. This turned out to be very helpful for the software project presented in this paper. It allows executing the C++ hardware traversal engine and rendering its result in the C++ process and just returning an SVG vector graphic to the browser. From the perspective of software development, this also has the advantage that only the SWIG interface file is to be maintained and that the web front-end developer can interact almost independently and in parallel to the C++ developer.

7. Sample Implementation

This section provides details of our implementation and demonstrates the previously described concept. For this implementation, the PHP interpreter has been selected since PHP is quite popular in the field to web development and hence supports lots of features. Figure 6 provides an overview on the whole compile and execution process. In addition, some code snippets are shown further below. At first, a C++ application is compiled (1). Listing 1 and 6 illustrate a small application holding an STL vector of integers which is initially filled with a few numbers.
Secondly, a SWIG interface specification as shown in listing 2 is necessary to allow PHP scripts accessing the required parts of the C++ application. This description language is very close to the C++ syntax. Its purpose is to specify the subset of classes and functions to be used by the scripting language[10].

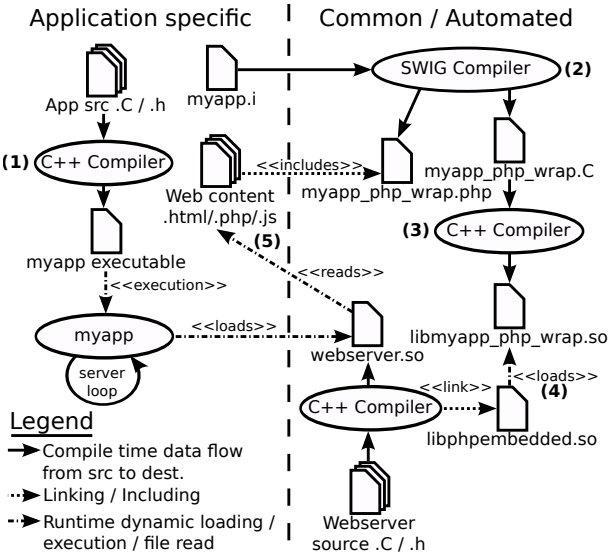

Figure 6: Compile, link and execution flow Left: Application specific part, Right: Common/automated part.

[9] For PHP there is a package called php-embed [14].
[10] A detailed description of the syntax can be found at `http://www.swig.org/doc.html`

In the next step the SWIG interface file is passed to the SWIG compiler (2) as shown in listing 4. This step generates a PHP interface file (here `myapp_php_wrap.php`) and a C file (here `myapp_php_wrap.C`).

```
#include <vector>
class MyAppT {
private:
    static std::vector<int> sMyVec;
public:
    static void init();
    static const std::vector<int> * getMyData();
};
```

Listing 1: myapp.h - Header file of the C++ application.

```
%include "std_vector.i"

%template(vector_int) std::vector<int>;
class MyAppT {
public:
    static const std::vector<int> * getMyData();
};
```

Listing 2: myapp.i SWIG interface file defines the C++ functionality available from the scripting language.

```
<?php
header('Content-Type: text/html; charset=utf-8');
include_once("myapp_php_wrap.php");

$vec = MyAppT::getMyData();

for ($i = 0; $i < $vec->size(); $i++) {
    echo "i=" . $i . " -> " .$vec->get($i) . "<BR>";
}
?>
```

Listing 3: myapp.php includes the generated myapp_php_wrap.php and represents the actual web front-end.

```
./swig -php -c++ -o myapp_php_wrap.C myapp.i
```

Listing 4: Executing the SWIG compiler myapp_php_wrap.so generates and myapp_php_wrap.php.

```
i=0 -> 123
i=1 -> 456
i=2 -> 789
```

Listing 5: Output of myapp.php in a browser.

```
#include "myapp.h"

std::vector<int> MyAppT::sMyVec;

void MyAppT::init() {
   sMyVec.push_back(123);
   sMyVec.push_back(456);
   sMyVec.push_back(789);
}

const std::vector<int> * MyAppT::getMyData() {
    return & sMyVec;
}

void main(void) {
  MyAppT::init();

  // Do something else ...

  // Finally, enter the web server loop

}
```

Listing 6: myapp.C - The source file of the C++ application. First, a static STL vector is initialized. Instead of exiting the program, the web server library is loaded and the server is started (code is not shown).

Afterwards, the generated `myapp_php_wrap.php` is included from the actual front-end PHP script as shown in listing 3. In this case, the STL vector is first requested using the static function `getMyData()`. Then the PHP code iterates through the vector and prints out all elements. The PHP interface to the STL vector (for example the `size()` and the `get()` functions) are generated by SWIG using the already predefined `std_vector.i` file[11]. It is included in the SWIG interface definition in listing 2.
In step (3) the generated `myapp_php_wrap.C` file is compiled to a shared library which can be dynamically loaded by the PHP interpreter as an extension (4). At this point, it is important to note that generally there are two different scenarios. The first one is that all required symbols are linked into the shared library (see figure 4). In this case no independent C++ application is running and only the C++ functionality provided by the shared library can be used by the PHP script. This for example makes sense if certain algorithms have to be implemented in C++ for some reason. The second scenario is the one used in this

[11] The list of STL containers supported by SWIG depends on the selected scripting language.

work. In this case most of the functionality is not part of the shared library itself - that means the symbols in the library are undefined at compile time (see figure 5). However, once this shared library is dynamically loaded by the main application the previously undefined symbols will become available since they are defined by the application itself[12]. Basically this is how access to the running C++ process is realized.
When running the `myapp` executable, the vector is filled with some numbers. Finally, instead of exiting the application, the web server library is loaded and the execution loop is entered (this code is not shown). This also includes loading the SWIG interface library `libmyapp_php_wrap.so` as a PHP extension. The process is now listening on a configurable HTTP port.
Visiting a certain URL from the browser[13] results in a request to the web server's web content folder (5). In case of a PHP script, the file is read and passed to the linked PHP interpreter via the C API. The PHP script is now executed. As described in section 6, the calls to C++ functions have been wrapped by PHP functions. The generated PHP code calls functionality of the loaded PHP extension which is the link to the actual C++ application. The result is returned by the interpreter and the web server finally sends it back to the browser where it is displayed as shown in listing 5.
As already mentioned in section 6, the implementation of the described concept requires a few modifications of the web server. Basically, the web server's processing of PHP scripts has to be changed. Instead of reading the PHP script and passing the content to the PHP interpreter via a pipe, the PHP interpreter's C API has to be used. This requires some additional setup code to initialize the interpreter. Once the described implementation is in place, only the application specific part (see left side of figure 6) is subject to changes. The only interface specification to maintain between C++ application and front-end is the SWIG interface file.

8. Results and Future Work

In this paper, we present a new approach to build a front-end for a C++ application based on standard web technologies. The concept allows the powerful combination of a C++ application with the advantages of a web front-end while the front-end can be developed using the standard web development techniques. This kind of software architecture has proven to work well and might serve as a role model for other C++ based projects not necessarily bound to hardware verification. Figure 1 from section 3 shows a circuit generated by our C++ verification tool and rendered by the browser.
Our web front-end is developed with respect to the requirements described in section 4 and is based on the presented concept. It greatly improves the usability of our C++ structural verification tool while keeping the development effort of the front-end under control. In this context, especially developing the front-end independently from the C++ application and without the need of recompiling turns out to be helpful. Moreover, supplying a front-end fix for a running session turned out to be practical since running the C++ verification tool on a full chip model can take multiple hours.

[12] The flags RTLD_GLOBAL and RTLD_LAZY are required for `dlopen()` - cf. [1]
[13] For example `https://127.0.0.1:4321/myapp.php`.

As described in section 4, the deployment process does not involve any installation on the user-side. Hence, it is quite easy to extend the web front-end step-by-step. The ability to use existing JavaScript and PHP frameworks improves the development efficiency and thus quickly led to a working front-end. In general, using the standard web presentation mechanisms helps to keep the front-end clear while still keeping the development effort low. Furthermore, it greatly improves the collaboration in distributed teams[14].
The web front-end, and especially the SVG based graphical circuit representation in the browser helps users to get a much better understanding of the tool output. Here the SVG mechanism turned out to be well suited for the interactive presentation of circuits to a user. It allows users to interactively traverse forward or backward through the circuit and share the results with colleagues by just copy-pasting the link. Beside a rendering engine implemented in C++ and a few lines of JavaScript code to navigate through the SVG, this feature did not require any more coding. Related content and other views on the data are reachable by clicking links. This way, the user can decide on its own whether to open the new content in a new browser tab or window.
For the future, there are plans to further extend the front-end of the presented structural verification tool. All this work will be based on the introduced concept. Beside other features, the search engine and the generation of status reports will be improved. In addition, a visualization of parent/child relationships between components is planned.

References

[1] Standard for Information Technology Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013), pages 1–3906, April 2013.

[2] Beazley, David M.: SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[3] Beilis, Artyom: CppCMS - High Performance C++ Web Framework. `http://cppcms.com`. Accessed: 2015-10-13.

[4] EMWEB bvba: C++ Web Toolkit. `http://www.webtoolkit.eu/wt`. Accessed: 2015-10-13.

[5] Grothoff, Christian: libmicrohttpd - Small C based web server library. `https://www.gnu.org/software/libmicrohttpd`. Accessed: 2015-10-13.

[6] Jäschke, Schmitt C., Herter U., Wich T., and Rust J.: Strukturelle Verifikation mittels parser-gesteuerter Netzlisten-Traversierung. In Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Dresden, Germany, February 22-24, 2010, pages 227–236, 2010.

[14] For example, sharing information by just copy-pasting a link is very helpful.

[7] Kazuharu, Aoyama: High-speed C++ MVC Framework for Web Applications. http://www.treefrogframework.org. Accessed: 2015-10-13.

[8] Kerrisk, Michael: The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press, San Francisco, CA, USA, 1st edition, 2010, ISBN 1593272200, 9781593272203.

[9] Lyubka, Sergey: Mongoose - Embedded web server for C/C++. https://github.com/cesanta/mongoose. Accessed: 2015-10-13.

[10] Merlino, Sebastiano: C++ Library for creating an embedded Rest HTTP server. https://github.com/etr/libhttpserver. Accessed: 2015-10-13.

[11] Real Time Logic, LLC.: C++ Server Pages (CSP). https://realtimelogic.com/products/barracuda-web-server. Accessed: 2015-10-13.

[12] Video LAN Development Team: VideoLAN Media Player. http://www.videolan.org/vlc. Accessed: 2015-10-13.

[13] Wurm, Christoph: Visualizing hierarchical structures for hardware chip verification. B.S. Thesis, DHBW Duale Hochschule Baden-Würtemberg, August 2012.

[14] Zend Technologies Ltd.: Embedded PHP SAPI Library. http://www.php.net. Accessed: 2015-10-14.

[15] Zend Technologies Ltd.: Hypertext Preprocessor (PHP). http://www.php.net. Accessed: 2015-10-14.