Hardware/Software Co-Visualization on the Electronic System Level using SystemC

(Embedded Tutorial)

Rolf Drechsler[1,2] Jannis Stoppe[1]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany

***Abstract*—Hardware design is increasingly shifted towards higher abstraction levels to cope with the complexity of modern systems. For this purpose, the concept of Hardware/Software Co-Design has been established. Systems are implemented in high-level languages such as C++ or SystemC, providing an executable prototype without requiring designers to decide in an early stage on whether features should be implemented in hardware or software. For these, just like for pure hardware or software systems, design understanding remains crucial.**

Hardware/Software Co-Visualization (HSCV) is a concept that has recently been proposed to assist developers in communicating features of their design. Especially for visualizations, designers need to be able to quickly generate the underlying data and the resulting visual representation. This paper illustrates the current state of HSCV and proposes a methodology to harmonize the visualization data being generated from Electronic System Level designs written in SystemC.

I. Introduction

Both, hardware and software design, are complex tasks. Designers need to handle a vast amount of information when designing either of them and the increasing amount of transistors per area leads to the requirement of both, more complex hardware (utilizing the increasing numbers of transistors per area) *and* software (utilizing the available computation power).

This complexity needs to be communicated. One way to do this is to visualize the according structures. In fact, visualization is an established methodology, with visual representations for both, hardware and software, being available.

However, visualization alone is not sufficient to handle larger designs. Instead, to handle the increasing complexity, higher abstraction description languages (such as SystemC) have been developed. This design paradigm, the development of systems on the *Electronic System Level* (*ESL*), includes the notion of Hardware/Software Co-Design. This approach encompasses the idea of designing systems that consist of both, hardware and software, without specifying further what part should be implemented in which domain. Instead, the design's features are described in a high-level programming language (such as C++) and merely describe the desired behaviour that should be refined into either hardware or software later on.

This way of describing systems not only enables designers to quickly prototype designs, it also represents a mixture of several existing means of description. Enabling designers to still use visualization as a tool to communicate properties while at the same time reflecting the dynamic, multi-dimensional properties of the design is a new field of research introduced in [9] called *Hardware/Software Co-Visualization* (*HSCV*).

This paper first illustrates the current state of hardware and software visualization means, then gives an overview of the issues when developing HSCV techniques and about the current state of HSCV and finally proposes another paradigm for the realization of HSCV technologies for ESL designs that are implemented using SystemC.

II. Hardware/Software Co-Design

Hardware design using dedicated hardware design languages is a complex and tedious task, especially when considering the size modern designs may have.

Modern high-level software programming languages on the other hand allow developers to quickly write software by providing a higher level of abstraction over the underlying architecture.

In order to still be able to quickly prototype a design to e.g. locate errors earlier in the design process or provide customers with a running simulation of a design, this concept of higher abstraction design is transferred to the hardware domain. The most prominent example of this high-level design approach is SystemC [19]. This C++ library comes with several constructs that enable a designer to describe hardware features in C++. It features classes for e.g. modules and signals and comes with a simulation kernel. The major idea is to build a virtual system using classes that inherit the provided module class and is interconnected using the given ports and signals. The logic within the modules is implemented in arbitrary C++ code. It may either consist of a synthesizeable subset, allowing the system to be translated into hardware, or any other C++ code, allowing the designer to quickly implement any desired functionality, e.g. using any libraries that are needed to quickly sketch out the functionality.

The resulting system consists of parts that represent hardware (such as the given modules and signals) and parts that may represent either hardware or software (such as the logic within the modules that is written in C++ and may be later translated to hardware or software that is running on the given module). The system hence combines all kinds of design paradigms.

III. Hardware and Software Visualization

As HSCV is closely related to both, hardware and software visualization, this chapter gives a short overview over existing techniques for these.

[2]See http://www.concept.de/

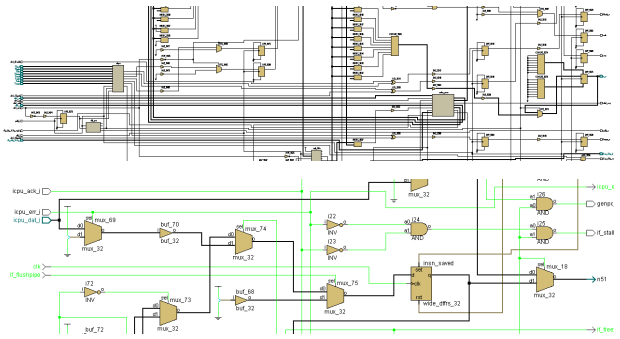

Fig. 1. Gate level (top) and register transfer level (bottom) visualizations generated using the commercial tools by Concept Engineering[2]

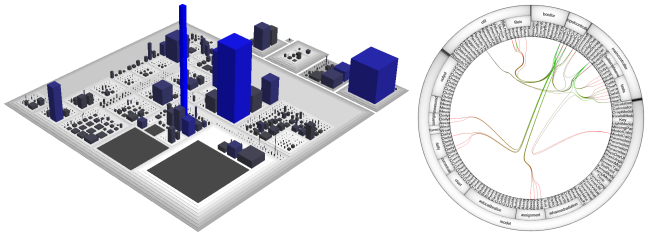

Fig. 2. Software visualization: The CodeCity visualization [26] illustrates the class structure (left), Extravis [7] uses e.g. a circular bundle view (right)

a) Hardware: Classic hardware visualization is an established tool to facilitate design understanding in the hardware domain.

As hardware is by definition a static structure, available visualizations behave accordingly. Figure 1 shows established hardware visualization views for gate level and register transfer level designs, illustrating logical blocks and the according connections.

The structure is usually represented in a graph: nodes and transitions are a natural data structure for the given elements, allowing the according algorithms to be used for further processing [8]. The issue in this case usually therefore is not the representation as such but the complexity: As hardware designs consist of up to several billion parts, displaying them is not a straightforward task.

The system behaviour is usually displayed in a different tool: waveform viewers show changing signal assignments over time, allowing designers to inspect how parts of their design interact. This means that current hardware visualization tools are essentially split into different tools for different tasks, thus requiring the designer to switch back and forth if information from both are needed.

b) Software: While software is at first rather similar to hardware as it consists of structural elements (such as classes) and their connections (such as relations between them) and behaviour (execution of functions etc.), there are conceptual differences between them upon closer inspection. The major difference being that structural features may be dynamic when e.g. an object is created in memory for a limited time. This way, the separation between behavioural and structural features is weakened, requiring either more dynamical structure descriptions or more structural behaviour descriptions.

Additionally, the structural information is more diverse in software: while hardware descriptions usually rely on signals to connect the different parts, the connections in software are more diverse: relations such as inheritance, composition etc. should not simply be represented in a single “is related” information.

The diverse information present in software has resulted in an equally diverse set of visualizations. Figure 2 illustrates two different approaches to visualize the source code of a given program, with lots of other approaches being available as well.

IV. Hardware / Software Co-Visualization

When a programming approach that encompasses both, hardware and software concepts, is applied, any visualization approach must do the same. In order to do that, the available language features are analysed in Section IV-A, existing approaches are outlined in Section IV-B and the suggested course of action is described in Section IV-C.

A. ESL Features: Taking Stock

Hardware/Software Co-Design merges the concepts of both, hardware and software, into a single, abstract description. Any visualization therefore needs to consider the features of both, hardware *and* software, and come up with a single, consistent visualization approach to communicate the design features.

The source code can be interpreted as a static, structural feature which does not change for the duration of a simulation.

The (simulated) hardware is more interesting in this case: while objects that represent hardware are created dynamically by arbitrary C++ statements, starting the simulation means that from that point onwards, the resulting architecture may no longer be altered. This means that the given hardware consists of *rather* static elements that first behave like arbitrary objects but are fixed after a certain moment during execution has passed.

The mixture of hardware/non-hardware types complicates this further. A SystemC module (which represents some kind of hardware part) may reference or contain arbitrary C++ objects. These may be either structural features (i.e. supposed to be part of the given hardware) or not (e.g. they merely required to quickly implement the module’s behaviour).

The same is true for communication: SystemC connections such as signals are usually used for inter-module communication, but as modules are ordinary C++ objects, they may as well simply call another object’s methods.

There are several other in-between constructs. E.g. variables that are shared between simulated hardware elements (such as global or static variables) have no real representation in hardware but can still be modelled as such. Generally, the differentiation between hardware elements and behavioural descriptions is often purely semantic and cannot be derived from the description itself.

The same is true for software: There may be “plain” C++ code that provides some kind of functionality that is not directly connected to the simulated hardware design, e.g. a system monitor that traces what is happening or a logger that writes certain events to the hard drive. The ESL source code that describes the system itself (e.g. methods that interpret

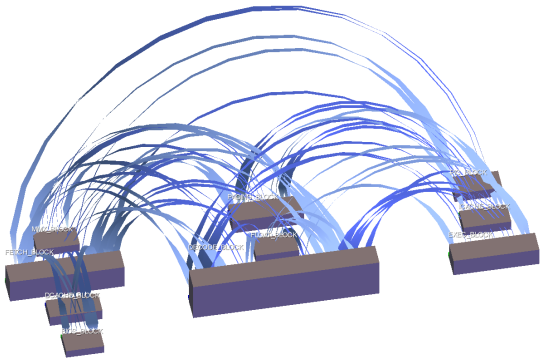

Fig. 3. Visualization of the SystemC RISC CPU example

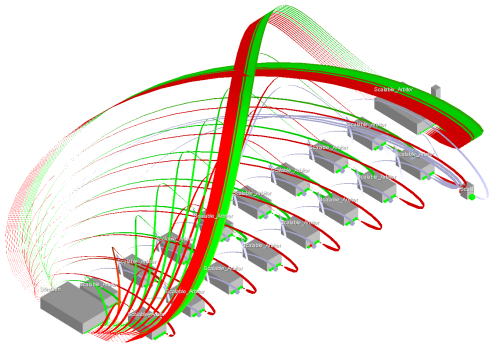

Fig. 4. Visualization of a SystemC arbiter design

the inputs of a module and calculate the output values) are similar, as they are merely methods that are being called during simulation. While this is still C++, it is semantically different; this is part of the simulated hardware system, the former is not. A third kind of software is the software that is running on the simulated system: a CPU that is built in SystemC will not run if it is not provided with software.

Generally, SystemC merges several different concepts from software and hardware development. While this allows for a more rapid development, it also means that finding consistent visual representations of the system's parts is not a trivial issue.

Even software and hardware visualization are not "solved" yet: if even these parts of the challenge have no satisfying beats-all solutions yet, it is even less the case for a design paradigm that combines both these approaches.

B. Existing Approaches

The closest tool to ESL visualization is SysML, which is a UML dialect to illustrate system designs. It re-uses some UML diagrams (such as the sequence diagram), modifies others (such as the block definition diagram, which is based on the UML class diagram) and adds new ones (such as the requirement diagram).

However, SysML, just like UML, is primarily a tool to plan a design. The reverse way, generating visualizations from an existing design, is an issue that, so far, has not been covered. One issue that impedes the generation of visualizations is the difficulty of analysing SystemC designs and extract their data in the first place: with SystemC being a C++ library, any analysis tool for SystemC has to support C++ in its entirety.

C++, however, is notoriously difficult to properly analyse when no further restrictions are applied. There is a wide variety of dialects for different compilers, each coming with its own libraries and additions. Additionally, as the system structure is defined by the modules that are created during the elaboration phase before the simulation starts, retrieving the system layout ultimately requires the extraction of the program state during the execution of the program. As C++ compilers tend to remove any information that is not needed for the execution of the program (including e.g. data about an object's types, fields, methods etc.) or, if it is required for execution, stores it in compiler-specific structures (such as the virtual function tables), it is inherently difficult to develop a generically applicable solution to extract information about a design from a SystemC implementation.

A variety of approaches has been implemented to extract the data, each of which differs concerning applicability, prerequisites, completeness and other criteria.

- Parsers [10], [16], [6], [4], [2] are based on the information that can be extracted from the source code and are thus limited concerning the supported language constructs and the ability to extract parametric design variations.
- Approaches that rely on the modification of compilers (and maybe the consecutive execution of a modified program) [18], [11], [17] are inherently limited to the compiler architecture they are based on, excluding any designs that rely on compiler-specific constructs from a different platform. The same is true for approaches that use a debugger to observe the running program [5], [20].
- As all compilers are generating and storing information for the debugger to be able to retrieve meta data about a running program, this data can also be used as a foundation for data retrieval methods. Relying on the debug data and reading it during execution allows for a detailed structural analysis but does not capture the behaviour of a design [22].
- *Aspect Oriented Programming* modifies the source code before compilation, injecting new features that can be used to observe the design [24]. It is hard to implement and debug though. Ultimately, it is limited in its expressiveness, too, and needs to be consistently updated to remain compatible with the developments of the different compilers and the C++ langauge itself.

While there are some approaches to then visualize the given data [12], [13], even these dedicated works are focusing on the data extraction and remain on a proof-of-concept stage concerning the visualization itself.

We consider the difficulty in generating the data to be visualized in the first place a major issue when it comes to the implementation of visualization engines for the ESL or SystemC in particular: if it requires a major effort to just get the data, then any visualization reduces itself to a proof-of-concept idea.

Without a common interface to pass generated data onwards to a visualization tool, engineers are more unlikely to take this step and build a visualization. An exchange format for ESL

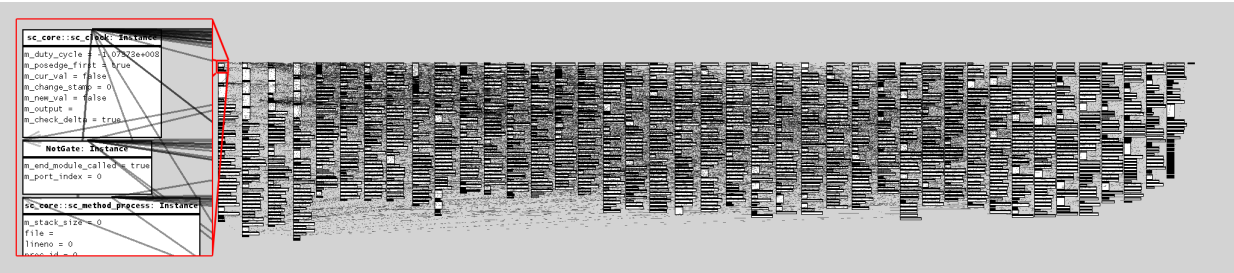

Fig. 5. Objects during runtime on the ESL [23]

data may be able to remedy this issue: If user interface experts can easily retrieve exemplary data and build tools for these, the entry threshold for building an according tool should be reduced significantly.

C. Visualizing SystemC

When extracting all available data, simply visualizing the given results using established means such as UML/SysML is not a valid solution as the data is too complex to handle. Figure 5 illustrates this issue for an object diagram: when all reachable objects are extracted from a running SystemC simulation, displaying them all does not make the system any easier to understand. When the additional information that is present in the ESL is added to the given representations, the complexity is going to increase even more, contradicting the idea of providing an easy-to-grasp interface to a design's distinct features.

This issue works both ways: adding hardware elements to software visualization approaches produces models that are about as hard to grasp as those that are based on adding software elements to established hardware visualization approaches (which already suffer from the given systems' complexity, as well).

We implemented a proof-of-concept visualization that uses data we can extract from a given SystemC design to illustrate a model of the system (as illustrated in Figures 3 and 4). The core structures are SystemC modules, linked via their ports using the signals present in the system. The visualization primarily focuses on the additional illustration of meta-data, using e.g. the height of the modules being shown to measure their complexity or the amount of memory they occupy. The tool is able to link back to the source code to provide designers with an easy way to edit a given module. It also displays behavioural information that can be read from VCD files stored during the simulation of the system.

However, we consider the interface between the visualization and the data retrieval method to be the most crucial element: in order to create a visualization, complex data retrieval methods had to be implemented first, with the visualization being closely tied to the given extraction methods. If the interface between visualization and data retrieval method was standardized, developing alternative approaches to either of the two could be done more quickly.

This interface would have to be able to properly reflect the structures present on both sides of ESL design, hardware and software. The data that can be extracted is basically always a set of nodes (representing entities) and connections, possibly with a collection of changes over time to reflect behaviour in some way.

With the extracted models usually representing a graph, using this as a smallest common denominator to directly benefit from developments in the visualization community is a natural step.

Using graphs as a common data structure for e.g. visualization is a step that has been proposed for software [14] and is a natural representation for hardware (which is a set of interconnected parts anyway). This step, coming up with a common base for further processing, is needed for Hardware/Software Co-Design as well.

This step clearly separates data retrieval (i.e. SystemC analysis) and application (e.g. visualization). Thus, the given extraction methods can be refined or tweaked regardless of the usage of the retrieved data: any third party visualization tool that supports the given data set may be used to display the given data. On the other hand, these tools that use the retrieved data, can be developed separately as well, allowing innovative, diverse solutions to be built on top of existing retrieval methods.

We have implemented a bridge that translates the information about a given ESL design from various sources into a neo4j [25] graph, in fact allowing not only the visualization of a given model but also enabling designers to e.g. quickly compare graph structures to e.g. a UML specification (which, of course, has to have an according representation as well). Figure 6 shows different representations of a neo4j graph for the class hierarchy of a SystemC design: the syntactic representation of the code (which basically just a hierarchical view of the given code elements) and the semantic representation (which more accurately shows e.g. the hierarchical structures of objects having methods, fields etc.).

There are several existing formats available to exchange graph data [21]. However, SystemC employs features such as changing data (as values on signals are altered over time) or even structural features that are changed during simulation (as new objects are instantiated), which usually is not part of a graph format. As a proof-of-concept, we implemented a method to translate the structure of SystemC designs into the GEXF format which supports such more complex features such as a hierarchy of nodes or changes in structure and content over time. It thus serves as a bridge to existing visualization tools such as sigma.js or gephi [1] (see Figure 7).

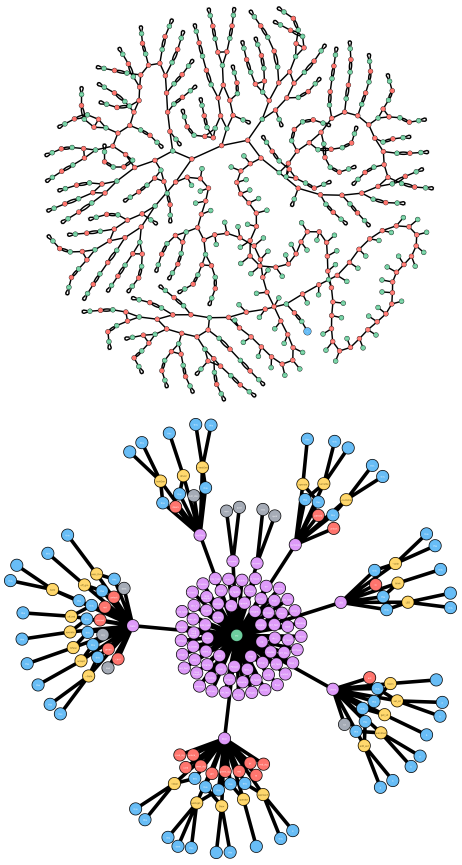

Fig. 6. Syntactic (top) and semantic (bottom) representations of the static code structures of an ESL implementation in the Neo4J graph database

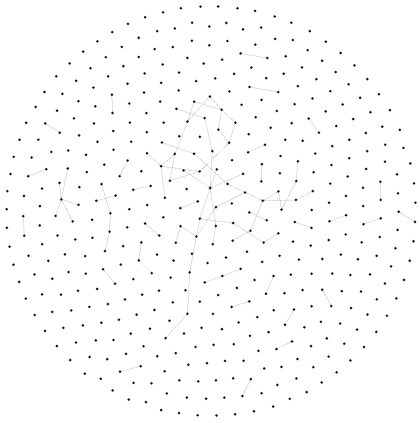

Fig. 7. Class inheritance of a SystemC design data in gephi [1]

As a first result, existing visualization toolkits can be used to handle the given data structures. These may provide different approaches to the display of the underlying data, actually giving users the ability to pick the tool that suits them best while analysing their design. This is not necessarily limited to the display, it may also be a question of the architecture as some may prefer web-based solutions such as *d3.js* [3] over e.g. local clients such as gephi or a question of the focus as some may only want to view the structures while others would like a database such as neo4j [25] to be able to use database queries to extract the desired information.

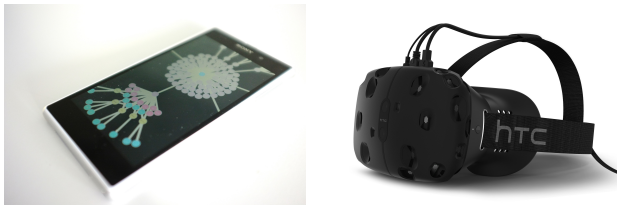

Fig. 8. Visualizations may be needed on handheld devices (left) or even VR headsets (right) [15], depending on the use case.

This format may thus serve as a reasonable foundation for HSCV: it provides the required features and is already supported by several visualization tools.

Concerning the visualization itself, performance is a straightforward necessity: with SystemC designs consisting of thousands of modules, classes and connections, the ability to handle the according amount of data smoothly is an obvious criterion.

However, the criterion that should benefit more from a given exchange format is accessibility. In order for such techniques to be used, they need to be readily available. The standard SystemC data extraction technique is the registration of signals and/or fields ot be included in a VCD in order to later view them as waveforms in a dedicated application. This should be regarded as an upper limit for interaction prior to the usage of data in order to make people indeed use the given techniques instead of e.g. just drawing diagrams manually.

Accessibility, however, concerns both sides of the visualization approach: it is an important criterion for the retrieval of the data *and* its visualization.

The former encompasses all kinds of extraction methods that retrieve a set of data concerning the structures and the behaviour of a SystemC design. As outlined before, with C++ being focused on performance and compilers usually removing everything that is not required for execution, this is already a tough problem.

The latter means that dedicated visualization tools need to be available where designers need them. That means that a visualization should not only be easy to load and use on a designer's workstation but that a common system design data format should be as ubiquitous, with visualizations available on both, low level (such as websites or handhelds) or state-of-the-art hardware (such as e.g. powerwalls or virtual reality environments) as illustrated in Figure 8.

This ubiquity of tools should be a key focus for HSCV tools. The previously mentioned common data exchange format (or lack thereof), however, is a key issue for this case. While there are several data retrieval methods available and still being worked on, there is – so far – no common target that this data is written to. This of course is a showstopper for any potential visualization tool to be developed: if a team needs to implement its own extraction methods in order to come up with a visualization solution, this binds resources to the degree of making the whole undertaking infeasible.

For GEXF, there is support by existing graph visualization engines, it supports the timed changes present in a dynamic system such as a SystemC design and our own data extraction methods have shown promising results.

We regard the usage of a common exchange format as a necessary and overdue first step towards more ubiquitous HSCV technologies.

This detachment of visualization and data retrieval should enable UI specialists to develop dedicated solutions for HSCV and incorporate techniques from other fields that have recently made an impact on user interaction. On handhelds, users may e.g. want to filter the given information using spoken, natural language. Using VR sets, the additional dimensions and immersive techniques could be used to browse the data. Websites, on the other hand, can be used to easily and quickly distribute information about a design or generate printable overviews for a documentation. All these use cases and also those that are not listed here should be quickly implementable: UI design often evolves around building several prototypical implementations and trying what works best, which is impossible without a common foundation.

The goal of the community around the analysis of SystemC designs should be to provide data to enable other engineers to create smart solutions. Common interfaces are a necessity for this and have not been established yet. The given approach is a suggestion to remedy this issue and provide easy-to-use, established intermediate data formats.

V. Summary and Conclusion

Visualization techniques for Hardware/Software Co-Design are still very much in its infancy.

There are lots of open questions concerning how to visualize these systems and and just as much room for improvements concerning the retrieval of the required data. This paper illustrated the current state of affairs for SystemC visualization and suggested that these two core issues (data retrieval and visualization) should be properly separated, using a common interchange format that is easy to pass on and work with.

GEXF, an XML-based graph format, was suggested as a solution for this issue that is already available, easy to handle and providing the needed features (such as behavioural and hierarchical structures).

VI. Acknowledgements

This work was supported in part the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E, within the project SPECifIC under the contract no. 01IW13001 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

References

[1] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 2009.

[2] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Sandeep Shukla. SystemCXML: An extensible SystemC front end using XML. In *Proceedings of the Forum on Specification and Design Languages*, pages 405–409, 2005.

[3] Michael Bostock. D3.js – Data Driven Documents. http://d3js.org/, 2012. Accessed: 2015-10-02.

[4] Carlo Brandolese, Paolini Di Felice, Luigi Pomante, and Daniele Scarpazza. Parsing SystemC: An Open-Source, Easy-to-Extend Parser. In *IADIS International Conference on Applied Computing*, pages 706–709, 2006.

[5] H. Broeders and R. van Leuken. Extracting behavior and dynamically generated hierarchy from SystemC models. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 357 –362, june 2011.

[6] Javier Castillo, Pablo Huerta, and Jose Ignacio Martinez. An open-source tool for SystemC to Verilog automatic translation. *Latin American Applied Research*, 37(1):53–58, 2007.

[7] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, and Bart Van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 100–109. IEEE, 2009.

[8] Rolf Drechsler, Wolfgang Günther, Thomas Eschbach, Lothar Linhard, and Gerhard Angst. Recursive bi-partitioning of netlists for large number of partitions. *Journal of systems architecture*, 49(12):521–528, 2003.

[9] Rolf Drechsler and Mathias Soeken. Hardware-Software Co-Visualization: Developing Systems in the Holodeck. In *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 1–4. IEEE, 2013.

[10] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: an efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 148–154, 2004.

[11] Christian Genz and Rolf Drechsler. Overcoming limitations of the SystemC data introspection. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 590–593, 2009.

[12] Christian Genz, Rolf Drechsler, Gerhard Angst, and Lothar Linhard. Visualization of SystemC designs. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 413–416. IEEE, 2007.

[13] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *FDL*, pages 646–658. Citeseer, 2003.

[14] Richard C Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 162–171. IEEE, 2000.

[15] HTC. HTC Vive. http://dl3.htc.com/us/press-kit/Vive Images.zip, 2015. Accessed: 2015-10-02.

[16] FZI Karlsruhe. KaSCPar - Karlsruhe SystemC Parser Suite, 2012. http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-kascpar-karlsruhe-systemc-parser-suite.

[17] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 79–88. ACM, 2010.

[18] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip. In *International Conference on Embedded Software (EMSOFT)*, pages 317–324, 2005.

[19] O.S.C. Initiative. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2006.

[20] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. An Integrated SystemC Debugging Environment. In *Embedded Systems Specification and Design Languages*, pages 59–71. Springer, 2008.

[21] Matthew Roughan and Jonathan Tuke. Unravelling Graph-Exchange File Formats. *arXiv preprint arXiv:1503.02781*, 2015.

[22] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from SystemC designs using debug symbols and the SystemC API. In *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*, pages 26–31. IEEE, 2013.

[23] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Validating SystemC Implementations Against Their Formal Specifications. In *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*, page 13. ACM, 2014.

[24] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Automated feature localization for dynamically generated SystemC designs. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 277–280. EDA Consortium, 2015.

[25] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity*, pages 217–218. ACM, 2012.

[26] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.