Towards a Multi-dimensional and Dynamic Visualization for ESL Designs

Jannis Stoppe[1] Marc Michael[2] Mathias Soeken[1,2] Robert Wille[1,2,3] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany
[3]Faculty of Computer Science, Technical University Dresden, 01187 Dresden, Germany

***Abstract*—Current state-of-the-art approaches for the visualization of systems at the Electronic System Level are dominated by static and two-dimensional schemes which miss important features for a proper design understanding. We propose a system level visualization approach that overcomes these limits in two different ways. First, it aims for displaying several design properties using a multi-dimensional visualization space and, second, makes use of dynamic techniques to emphasize behavioral information. Overall, the visualization approach enables a better design understanding but also allows for an easier detection of irregularities in the considered design. We present preliminary results based on a prototypical implementation.**

I. Introduction

Visualization techniques play a central role in design understanding of complex circuits and systems. Existing state-of-the-art visualization techniques for descriptions at the *Electronic System Level* (ESL, [1]), e.g. provided in SystemC, are dominated by static and two-dimensional approaches that mainly focus on the representation of the structure in terms of blocks and connections between them. Annotations may additionally been used e.g. in order to highlight asserted connections in different colors or displaying simulation results at the respective ports. However, important other features which are crucial to a proper design understanding at the ESL are hardly covered by these solutions:

1) Previous approaches rely on a two-dimensional visualization focusing on the structure of the system. But ESL designs are more diverse and require the consideration of further properties and metrics such as code size and timing. Hence, a *multi-dimensional visualization* is needed allowing for the representation of several design properties at the same time, and therefore enabling the designer to understand the connection and correlation between them.
2) Previous approaches rely on a static representation, e.g. the representation of the structure or completed simulation runs. But ESL design heavily include behavioral descriptions. Hence, a *dynamic visualization* is needed that offers to emphasize on the behavioral aspects of the design such as simulation traces and their effect on the system state.

In order to implement a visualization approach that considers these features for a design language such as SystemC two questions need to be addressed:

1) How can that information be retrieved and
2) how can the information be visualized?

In this paper, we consider these questions. We discuss how the required information can be extracted from an ESL design and propose new schemes for their visualization. This leads to a dynamic and multi-dimensional visualization approach for SystemC designs which has been prototypically implemented and evaluated. The results of first case studies show promising application scenarios of the proposed solution.

II. ESL Design and the Need for Visualization

In order to keep the paper self-contained, this section briefly reviews the design at the ESL using the programming language SystemC as well as corresponding schemes for the visualization of the resulting designs.

A. ESL Design using SystemC

The complexity of hardware designs is still growing exponentially [2]. In order to be able to design such systems, apart from reusing large parts, more abstract system descriptions have been developed. One of these more abstract layers is the ESL. The goal of the corresponding descriptions is to provide a working simulation of a hardware system (less abstract than e.g. a SysML specification on the *Formal Specification Level* (FSL, [3])) that does not require all information concerning *how* the hardware is supposed to be implemented (more abstract than e.g. a VHDL description on the *Register Transfer Level* (RTL)).

The de-facto standard for ESL development is SystemC [4]–[6]. SystemC is a C++ library that offers structures to prototype a system and simulate the result. While the behavior is specified in the SystemC implementation, it is not needed to determine whether a certain part will later be implemented in hardware or software. This specific feature is usually referred to as *Hardware/Software Co-Design* and can be regarded a central feature of ESL design, as it relieves the designer from deciding how certain parts should be implemented and lets her focus on the desired behavior instead.

This feature has a wide range of implications, especially with regard to the decision of making SystemC a C++ library. A single hardware module is not limited in any way. The description may contain complex loops, large memory allocations and disallocations, network communication, cloud-based

computation, calls to external programs etc., all of which might be considered to take place instantaneously for the purpose of simulation.

The resulting design is composed of simulated hardware elements (such as modules and signals) and the description of their behavior (which can be any C++ program code).

Although the overall complexity is reduced (compared to traditional RTL designs), the diversity of elements results in more complex structures available for building the system itself.

In order to better grasp a design, a visualization is usually presented to the designer. However, the traditional hardware visualization techniques do not work well with ESL designs due to the mixture of hardware and software elements. While the hardware parts of a hardware/software co-designed system can of course be displayed as such, a proper visualization of the remaining parts has, to the best of our knowledge, yet to be developed.

B. Visualizing ESL Systems

Visualization is an important issue for several tasks during the design process. It can e.g. be used to more rapidly understand a system, to locate errors in running systems, or to illustrate the project's documentation.

Conventional visualization techniques that focus on hardware design do not cover the abstract layers and usually assume that the modeled hardware is the only part of the system that needs to be displayed. ESL designs, however, may have large software parts that cannot be translated to hardware elements and, hence, are not properly visualized thus far. This mixture of hardware and software yields two main questions that need to be addressed for visualizing an ESL design:

1) How can the information needed for this visualization be retrieved and
2) how should the retrieved information become visualized?

The former question is a problem that is especially apparent in SystemC: As C++ does not offer sophisticated reflection or introspection methods, the extraction of program information at runtime is difficult at best. While SystemC acknowledges this problem and provides an API that allows the extraction of `sc_object` instances, user defined types that are instantiated and/or referenced by the simulated system should also be part of a visualization of the system. Also, unlike a hardware system that has a distinct state for each clock cycle, ESL designs may execute certain functions without keeping trace of intermediate states, making the location of certain errors by means of a visualization of clock-accurate states impossible.

The latter question deals with the differences in paradigms between hardware and software design. A hardware system usually consists of a static architecture that changes its states to generate a certain result. The changing states are often timed using a clock, resulting in systems that have change states at fixed intervals. ESL systems do not follow such a pattern. Although they model hardware systems that do behave similarly, that behavior is often generated by much less homogeneous patterns. In case of SystemC, the non-restrictive permission of any C++ construct gives the designer the option to use all kinds of behaviors that have no resemblance in classic hardware systems.

Hardware/software co-visualization therefore is a topic that — although offering vast potential benefits — is highly complex and has only recently been brought up [7]. While there are visualization techniques for both, hardware and software systems, not all methods of both domains may work well together or show a consistent system. Both, the visualization itself and the back-end to retrieve the needed data are non-trivial problems to address, but a more accurate representation of ESL designs might help to grasp the features of a given system more easily.

III. Data Extraction

This section deals with the question how the information needed for the visualization can properly be retrieved. For this purpose, the existing approaches for information retrieval are briefly reviewed first. Then, the proposed solution is sketched.

A. Previous Work

In order to extract the information to be visualized the design needs to be inspected in some way. Several solutions have been proposed to achieve this.

One approach is to use a custom SystemC parser [8]–[11]. Although the idea might seem straightforward, the fact that the simulated SystemC design is created at runtime and might depend on inputs that are not available before the execution of the program seriously limits the applicability of this approach. A parser that could analyze any given SystemC design would have to inspect all possible paths through the program (at least until the end of the elaboration phase) which is equivalent to solving the halting problem. Also, the differences between C++ dialects and available libraries (which would all need to be supported by a parser) makes a full SystemC parsing support infeasible.

A different approach to this problem is to use existing compilers to inspect the structure of the SystemC program [12]–[15]. The usage of libraries does not pose a problem in this case. Furthermore, the resulting program can simply be executed until the end of the elaboration phase to be able to extract the program state at that point. That is, most of the problems stated above are solved. Nevertheless, approaches like this are highly compiler-dependent and need to be redeveloped once a new compiler (or just a new version of an existing one) is released.

Recently, another approach has been proposed [16] that uses (1) the compiler-generated debug symbols to extract static information about class structure, source code etc., and (2) the SystemC API to extract the instantiated SystemC objects at runtime. Consequently, all required information can be extracted without the need to modify either the SystemC library or the compiler. In fact, this approach is less intrusive than the compiler approach while still taking advantage of the existing compilation backend. For our visualization goal,

we make use of this approach in order to retrieve the needed information. For this purpose, several steps are carried out independently. These steps are described next.

B. Extracting the Module Hierarchy using the SystemC API

The SystemC API allows to retrieve all registered objects that inherit the `sc_object` type, including all modules, signals, and ports and their hierarchy. The simulated hardware parts of the ESL design can therefore be retrieved from the running program without interfering with the compiler or the library.

Although the call to extract the design could be hooked to the `end_of_elaboration` listener, offering the designer the function to call at the point at which she wants the design to be extracted (or even extracting it at several points) was the preferred solution for our case. This allows for a step-by-step extraction of the elaboration phase to see the system being instantiated.

C. Extracting Static Information from the Debug Symbols

Instead of parsing the source code or modifying the compiler to save the static source code information, the approach presented in [16] relies on the debug symbols to obtain the needed information.

The debug symbols are stored by the compiler to be able to give the programmer information about the program state during the debugging process. They usually contain all the structural information of the given program, including links to the source code files. Although the different compilers are using different formats, they are usually either human-readable (i.e. parseable) or accessible using an API.

This means that instead of modifying the compiler (and thus restricting the project setup to the one modified version), a common and pre-existing interface is used to retrieve the static information.

D. Snapshots and VCD for Behavior Retrieval

The repeated execution of the former methods results in a series of snapshots of the program state that can be used to depict the behavior of the system. If the debug-based method is available, the changing values of instances can be tracked without any further instrumentation, apart from calling the extraction method. This allows e.g. for the tracking of values of field variables.

In addition to the methods presented in [16], reading VCD files during the simulation of the system results in a (nearly) real-time display of values that are tracked by the designer during the simulation. Even this straightforward method can be used to display certain metrics (e.g. how often a signal has been switched etc.) to show which parts of the design are used frequently.

E. Additional Information

Supplementing the extraction methods from [16], additional metrics to support the visualization were extracted directly from the source code. Although this might at first challenge the notion of using an extraction method that is independent from parsers to be able to read the full C++ code base, these additional measures are mere supplements and need the sources to be extractable.

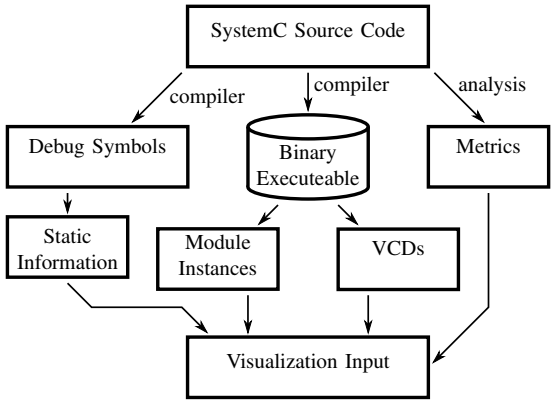


Fig. 1. System architecture for extracting the data to be visualized.

As the designer is writing source code to describe the system, measuring the source code is a logical step to illustrate different aspects of the program. Features such as code complexity cannot be extracted without access to the source code but might be interesting for the designer when viewing the system. This additional information is therefore parsed from the C++ sources if possible.

An overview of different extraction means is given in Fig. 1. Different sources at different steps during the compilation workflow are used to gather as much information about the SystemC program as possible before and during the execution.

In summary, the data extraction methods are based on the methods proposed in [16] but allow the exploitation of additional sources in order to provide a more detailed view of the system.

IV. Visualization

Having all desired information available, the next step is to properly visualize them. In this section, existing approaches are briefly reviewed before the concepts needed to visualize ESL designs are discussed.

A. Previous Work

Current visualization approaches focus on either software or hardware designs. For each, there are a variety of methods or standards available.

Software visualizations have to deal with systems that are constantly redesigned at runtime: Object instances, which resemble the concept of a “thing” that does something like a hardware part, are created and deleted at will. However, unlike hardware, the program logic itself mostly follows strictly linear patterns. Although parallel algorithms have started gaining traction with the widespread availability of multi-core systems and have always been a focus of super computing systems, they are still linear patterns that interact at certain points. UML as a standard to design and visualize software systems proposes several vastly different views to grasp all aspects

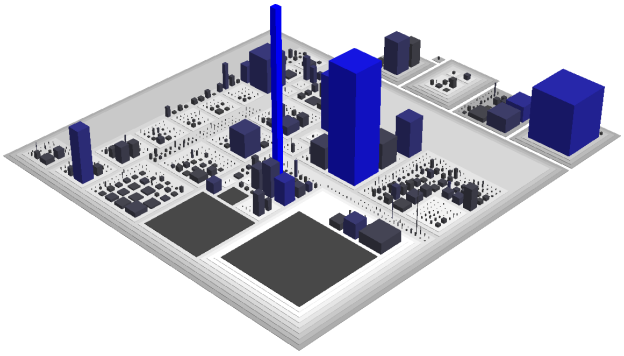
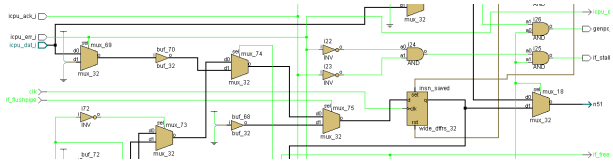
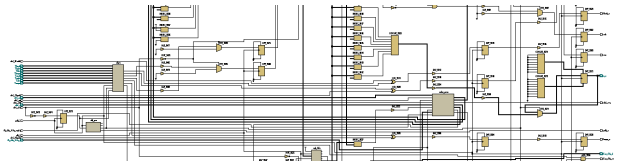


Fig. 2. Visualization as CodeCity



(a) RT level visualization



(b) Gate level visualization

Fig. 3. Classic hardware visualization focuses on static, printable images of a system.

of a software system, all of which statically represent either structure or behaviour. The main notion is that UML is a language that was designed to be printed.

With the advances in computation power that is available on even mediocre systems, more advanced solutions have been proposed: visualizations such as *gource* [17] or CodeCity [18] use 3D engines to display a software system. In order to visualize different properties at once, the CodeCity-metaphor has received attention in the domain of software visualization. Here, different design properties are displayed in different "dimensions" of the visualization, i.e. the number of attributes, methods, and lines of a Java class have been mapped to three-dimensional cubes that represent buildings in a city. Classes from the same package were placed in the same district to emphasize structural interrelation. As an example, Fig. 2 shows a picture of a CodeCity taken from [18]. Unproportional looking buildings immediately pinpoint the designer to problematic classes in the software project. The visualization reveals classes that are too complex in terms of code and may better be split into subclasses or classes that are not well-balanced in terms of their number of attributes to number of methods ration.

Despite the fact that the CodeCity-metaphor is easily comprehensible for designers, the metrics that have been applied for Java source code cannot applied directly to system level programs written in SystemC. Besides structural information such as lines of codes, number of signals, number of attributes, number of methods, and connectivity also quality metrics such as complexity, maintainability, as well as test and verification coverage are of high interest to the designer. In particular we are interested in integrating quality metrics into the visualization. In software, e.g. condition complexity [19] measures the number of linear independent paths in a program. For hardware an entropy-based concept has been presented in [20].

Overall, while the CodyCity-metaphor is a proper visualization technique allowing for a multi-dimensional visualization, its concepts need to be redeveloped in order to explicitly support the requirements of ESL designs.

Classic hardware visualization on the other hand usually evolves around established descriptions for the various levels of design. This starts on the transistor level, encompasses the gate level and ends on the register transfer level (Fig. 3 provides some examples). All these visualizations evolve around the core concept of hardware: That the system by itself is fixed and only the information being encoded in it is changing. These values are often visualized using waveforms that, while accurate, are not necessarily the best to see connections between and patterns of the signals. While these concepts do represent the hardware appropriately, they are not well suited to illustrate the dynamics of ESL designs.

However, to the best of our knowledge, the combined hardware/software co-designed systems at the ESL have not been visualized at all until now. Only recently, a single work envisioned such systems but did not offer a prototypical implementation [7].

B. Paradigms

The goal of our solution is to present a working system visualization that displays different visualization schemes in a coherent environment. Even if different concepts require different visualization techniques, they should be an integral part of each other. If viewing both in the same environment is for some reason not feasible, at least going from one to the next should be seamless in both directions. A single visualization for hardware, software, and behavior is anticipated to avoid repeated swapping of views and to illustrate that the system in question is indeed a single whole and not a collection of separated parts.

The proposed system uses CodeCity [18] as a baseline. The representation of elements using simple geometric shapes (mostly boxes) does not only keeps the system requirements low but also all shapes on a common ground level. This simplifies the orientation in the three-dimensional space. Generally, this semi-3D view (three-dimensional objects on a two-dimensional plane) also allows for a simplification of the navigation: the camera requires less degrees of movement freedom to view all objects (in our case only a two-axes pan motion, a one-axis rotation and a zoom), simplifying the controls to a degree where it is possible to use it on a touch screen without losing any navigation abilities.

Modules themselves are merely instantiated objects, albeit of a particular type. However, other objects that are referenced

from the described system should be displayed as well. Apart from the different semantics, there is no real difference between these two, so displaying the objects in a similar manner seems reasonable.

However, presenting all this information at once is usually too much to be displayed on a single screen. Hence, we apply a hierarchical view that uses several levels level of details. By "zooming in", more details of the respective components will blend in. The application of this technique results in an intuitive way to get more information about something in particular: just get closer to it to see more about it.

To visualize important correlations of system metrics, such as lines of code and complexity of each SystemC module, the designer should be able to choose which metrics are important to him and how they have to be visualized, for example as height or ground size of a box. Maintainability could be a suitable metric for this purpose. It reflects the adaptability and modifiability of a SystemC module which is required to correct errors or to improve the performance. The maintainability index as proposed in [21] already provides a proper definition for this. The goal of such an individually customizable system representation is to help the designer to obtain information she needs about certain parts of the system quickly.

Another important part is the behavior of the system. Using time as a dimension to display itself seems a more straightforward solution than the classic idea of timelines or flowcharts, especially when it comes to monitoring running ESL simulations. While displaying all of a simulation's states is out of the questions due to the discrepancy between monitor refresh rates and simulation speeds, several metrics can be used to analyze and quantize the system changes over a certain timeframe. Such metrics could visualize system changes e.g. by different colors over time. As an example, one can show how often signals are used in a simulation or the individual activity of each module. Furthermore, non-functional properties can be considered if they are available such as power consumption.

Showing the design's behavior in a dynamic visualization for a longer period of time allows the designer to detect correlations and irregulations among components and signals which helps for a better design understanding and to find bugs, respectively.

In conclusion, while there are visualization approaches for software and hardware systems, especially the hardware visualizations do not go far beyond classic paper drawings of circuits and therefore do not really make much use of the opportunities a computer-based visualization provides. The visualization of hardware/software co-designed systems which contain a mixture of both has not been done before beyond printable layouts (e.g. in SysML). This chapter presented the issues a visualization for such systems needs to deal with and presented possible solutions to be implemented.

V. Prototype

A prototype that shows several of the aspects outlined above was implemented using LibGDX [22]. LibGDX is a cross

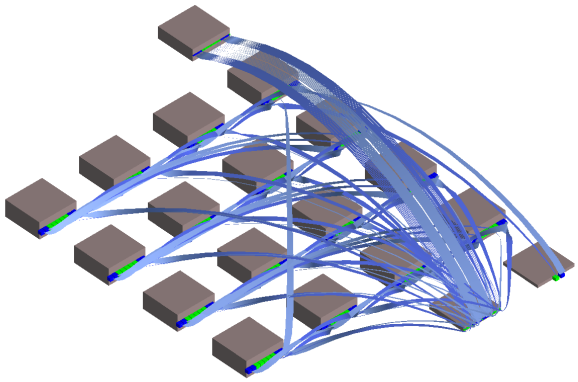


Fig. 4. SystemC modules and connections visualized: modules are represented by boxes, signals by connecting lines.

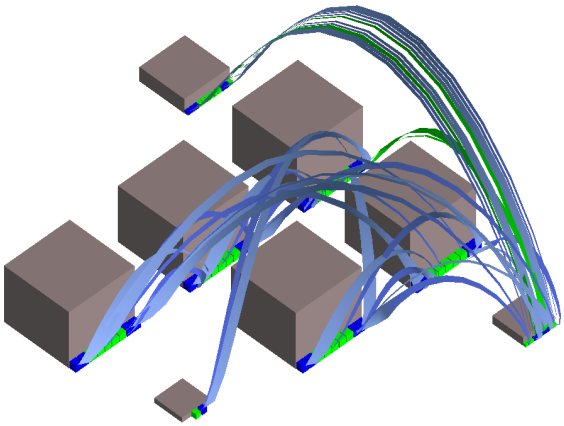


Fig. 5. SystemC Design metrics: Modules in this image are dispalyed as high as its class has lines of code and have a base area that resembles their respective code complexity

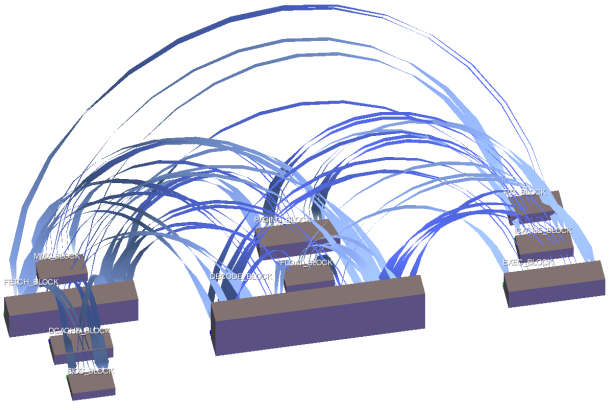


Fig. 6. Visualization of the SystemC RISC CPU example. Module height represents the size of a module in memory, module width illustrates its amount of ports.

platform framework which allows to run the prototype on various systems.

The data representation, as seen in Fig. 4, is fixed concerning its basic structure (e.g. a module is always a gray box), but different attributes can be mapped to the parameters of the given object (e.g. a module can be as high as its memory consumption or as shown in Fig. 5, as large as its class has lines of code and has a base area that resembles its code complexity). The connectors are smaller boxes attached to their belogging modules. The color of each connector indicates its connector type: Green represents input connectors and blue output connectors.

For the prototype, a simple layout solution that groups submodules in squares was used. While this is just a quick and simple solution, it still allows the concept to be illustrated. Also, there is currently no routing solution used: Connections between ports are illustrated using Bézier paths that evade other modules by describing a three-dimensional arc above the ground plane.

The behavior of the system can be displayed by using the standard log/dump-files created during a running simulation. The activity of the system can be displayed in real-time while the simulation is running or offline. While this limits the data to be displayed to that which can be extracted using standard dump-file extraction methods (and therefore e.g. excludes custom types), it is universally applicable for any SystemC design and already widely used. Immediate compatibility to existing project setups is in this case an important factor. Using other means to retrieve changes that are usually not part of the manually selected fields and signals would be an obvious next step for the visualization.

While the current state of the implementation does not cover all the aspects outlined in Section IV, it illustrates the first steps towards them. This working proof of concept offers an interactive view that enables the user to navigate through her SystemC design in an innovative and intuitive way. As a result, a new approach is provided which gives the designer a quick overview of a system and aids the design understanding on a different level than source code.

VI. Conclusion

An approach has been presented to system visualization that does not rely on printable outputs but instead focuses on a comprehensive, interactive representation. For this purpose, new visualization paradigms are exploited that have not been considered in previous ESL visualization techniques. These paradigms incorporate multidimensional and dynamic features in order to represent correlation between several metrics and to illustrate behavioral information, respectively. The approach has preliminarily been evaluated based on a prototype. While this prototype is not yet feature-complete, it shows how a different approach to hardware/software co-visualization can result in an intuitive interface to a given design that embraces all concepts that are part of it instead of separating them.

References

[1] B. Bailey and G. Martin, *ESL Models and their Application: Electronic System Level Design and Verification in Practice*. Dordrecht, Heidelberg, London, New York: Springer, 2009.

[2] P. A. Salvadeo, A. C. Veca, and R. C. Lopez, "Historic behavior of the electronic technology: The wave of makimoto and moore's law in the transistor's age," in *VIII Southern Conference on Programmable Logic (SPL)*, 2012, pp. 177–181.

[3] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards verification-driven design based on natural language processing," in *Forum on Specification and Design Languages*, 2012, pp. 53–58.

[4] Accellera Systems Initiative, "SystemC," 2012, available at http://www.systemc.org.

[5] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[6] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2005.

[7] R. Drechsler and M. Soeken, "Hardware-software co-visualization: Developing systems in the holodeck," in *Proceedings of the 16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS*, 2013, pp. 1–4.

[8] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: an efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies*, 2004, pp. 148–154. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.7049

[9] F. 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.

[10] J. Castillo, P. Huerta, and J. I. Martinez, "An open-source tool for SystemC to Verilog automatic translation," *Latin American Applied Research*, vol. 37, no. 1, pp. 53–58, 2007. [Online]. Available: http://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S0327-07932007000100011

[11] C. Brandolese, P. Di Felice, L. Pomante, and D. Scarpazza, "Parsing SystemC: an open-source, easy-to-extend parser," in *IADIS International Conference on Applied Computing*, 2006, pp. 706–709.

[12] K. Marquet, M. Moy, and B. Karkare, "A theoretical and experimental review of SystemC front-ends," in *Forum on Specification and Design Languages*, 2010, pp. 124–129.

[13] C. Genz and R. Drechsler, "Overcoming limitations of the SystemC data introspection," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 590–593. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874764

[14] 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*, 2011, pp. 357–362.

[15] K. Marquet and M. Moy, "PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2010, pp. 79–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=1879032

[16] J. Stoppe, R. Wille, and R. Drechsler, "Data Extraction from SystemC Designs using Debug Symbols and the SystemC API," in *IEEE Computer Society Annual Symposium on VLSI*, 2013.

[17] A. H. Caudwell, "Gource: Visualizing software version control history," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. New York, NY, USA: ACM, 2010, pp. 73–74.

[18] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *International Conference on Software Engineering*, 2011, pp. 551–560.

[19] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[20] B. Menhorn and F. Slomka, "Design entropy concept: a measurement for complexity," in *Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2011, pp. 285–294.

[21] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and application of an automated source code maintainability index," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997.

[22] M. Zechner and R. Green, "What's next?" in *Beginning Android 4 Games Development*. Springer, 2011, pp. 647–651.