A Breath of Fresh AIR: Programmatic Construction for SystemC

Wild-and-Crazy-Idea Paper

Rainer Dömer
Center for Embedded and Cyber-physical Systems,
University of California, Irvine, USA

***Abstract*—Chisel has achieved a tremendous impact in chip design. Termed a Hardware Construction Language (HCL), Chisel very effectively eases the creation of RTL models in a Hardware Description Language (HDL), i.e., Verilog. Raising the level of abstraction, system designers use a System-Level Description Language (SLDL), such as SystemC, for modeling and simulation, but there is no standard framework for constructing an application model and generating its corresponding description. In this position paper, we make the case for a new *System-Level Construction Language (SLCL) ProCSy* that enables the programmatic generation of SystemC models. Using an explicit Application Intermediate Representation (AIR), we outline how SystemC models can be efficiently constructed by automation and convenient programming in Python, rather than being written by hand or generated through ad-hoc scripts.**

***Index Terms*—System-Level Construction Language, system-level design, modeling and simulation, SystemC.**

I. Introduction

For the co-design of systems with hardware and software, chip designers have raised the level of abstraction from the pin and cycle-accurate Register Transfer Level (RTL) to the system level where modules exchange payload data via function calls, i.e., Transaction Level Modeling (TLM). Here, SystemC [1] is used as the standard language for modeling, simulation, and synthesis.

Unfortunately, writing SystemC models is difficult and very time consuming. When teaching SystemC in the classroom, system-level concepts of modules connected by channels are easy to grasp, but there is a steep learning curve of actually writing SystemC code due to the complex syntax and semantic requirements of SystemC and the underlying C++ language. The discrepancy between what you want to model (e.g., a diagram on the whiteboard) and what you have to write (i.e., SystemC source code) is so large that even system designers in industry often forgo the system-level modeling due to the presumed (and often real) lengthy coding with high costs.

To address this problem, we propose a programmatic approach to construct SystemC models and replace tedious coding tasks with automation.

A. System Level Construction Language

Since around 1990, a formally defined Hardware Description Language (HDL) is the standard starting point at RTL for the design of digital chips, namely VHDL [2] (mostly in Europe) and Verilog [3] (predominantly in the United States). A decade later around 2000, the need to include software in the specification model resulted in the definition of a System-Level Description Language (SLDL), such as SpecC [4] and then SystemC [1]. Table I lists these description languages at their respective abstraction levels.

TABLE I
Need for a System-Level Construction Language (SLCL)

	Description Language	**Construction Language**
System Level	SLDL: SpecC, SystemC	*SLCL: ProCSy*
Hardware (RTL)	HDL: Verilog, VHDL	HCL: Chisel, PyMTL

Around 2010, Chisel[1] [5] introduced a new approach to RTL synthesis. Rather than describing the intended design, Chisel constructs a RTL model and is, therefore, termed a Hardware Construction Language (HCL). The extraordinary impact on productivity of focusing on *construction* over mere *description* is best exemplified by the ease with which RISC-V processor systems can now be generated with Chipyard [6].

We aim to bring this idea of *construction* up to the system level, as shown in Table I. We propose a new System-Level Construction Language (SLCL) named *ProCSy*[2] for the programmatic construction of SystemC models.

B. Key Ideas

Our proposed ProCSy strategy is based on five objectives.

1) Programmatic construction: Choose convenient programming in Python
2) Separation of the elaboration and simulation phases: Dynamic construction produces a static structure for simulation
3) Application Intermediate Representation (AIR): Construction with labor-efficient functions and automation
4) Static verification of structural hierarchy: Analysis of fully constructed hierarchy and connectivity
5) Dynamic validation of functional behavior: Model simulation using standard SystemC or parallel C++

We describe each objective in the following Section II.

[1]Chisel: Constructing Hardware In a Scala Embedded Language
[2]ProCSy (pronounced ”proxy”): Programmatic Construction for SystemC

979-8-3195-3997-7/26/$31.00 ©2026 IEEE

2026 Forum on Specification & Design Languages (FDL) | 979-8-3195-3997-7/26/$31.00 ©2026 IEEE | DOI: 10.1109/FDL71572.2026.11724687

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on November 13,2026 at 19:26:38 UTC from IEEE Xplore. Restrictions apply.

II. Proposed Approach

The system design flow with ProCSy begins with construction in Python, separates the traditional SystemC phases of elaboration and simulation with an explicit intermediate representation, then verifies the static model structure, and finally validates the functionality by dynamic simulation.

A. Programmatic Construction

In essence, ProCSy replaces describing with programming. Rather than manually describing the intended design model in complex SystemC/C++ code with strict syntax and semantic rules, a ProCSy program generates the SystemC model. For its convenient syntax and powerful native data types, we choose Python as the programming language. Technically, any language could be used, but object orientation is essential (this is consistent with Chisel [5] embedded in Scala [7] and [8]–[10] hosted in C++).

With Python as the host language, there is no code duplication due to separate declarations and definitions and no need for the intricate syntax and semantics of SystemC [1], such as preprocessor macros, template classes, new operators, pointers, and explicit constructors[3]. Instead, "correct by construction" SystemC code will be automatically generated in compliance with TLM and TLM-2.0 guidelines, including connectivity with the proper ordering of port and socket bindings.

Programmatic construction with Python can also use unrestricted[4] control flow or list comprehension to generate objects and adjust or delete them. During code generation, many data types may be automatically deduced, such as socket connections or ports mapped to channels.

B. Separation of Elaboration and Simulation Phases

As depicted in Fig. 1, the standard SystemC flow consists of model compilation followed by its execution. During execution, SystemC distinguishes between the elaboration and simulation phases, but these are tightly coupled by the in-memory data structures of the model hierarchy and connectivity. Here, the lack of proper runtime reflection in C++ stands in the way of inspecting the elaborated model. This is a well-known problem for SystemC analysis and synthesis tools[5].



Fig. 1. Traditional SystemC flow

In contrast, the construction flow with ProCSy explicitly separates the elaboration and simulation. As shown in Fig. 2 at the top, an application model is first constructed, then compiled and executed. Here, the design model is fully elaborated during construction and then represented as a purely static tree in the generated SystemC model. In other words, dynamic construction produces a static structure for simulation.

Notably, construction and simulation use different host languages, i.e., Python and SystemC, respectively. This two-language approach is similar to Chisel [5] where Scala constructs Verilog, but in contrast to [10] where SystemC serves as host for both elaboration and simulation[6].

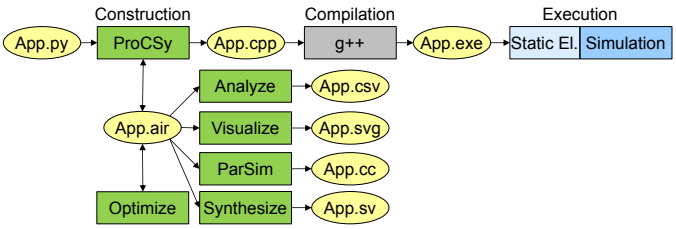

Fig. 2. Construction flow with ProCSy

More importantly, the ProCSy construction result is not hidden, but explicitly exposed. On the one hand, it is visible as a static instance tree in the generated SystemC source code. On the other hand, it can be saved as a text file `App.air` in Application Intermediate Representation (AIR) format, which then can be further processed, i.e., inspected, analyzed, visualized, annotated, optimized, simulated, or synthesized. Note that this intermediate representation can be seen as the system-level equivalent of the Flexible Intermediate Representation for RTL (FIRRTL) [11] in the Chisel construction flow.

C. Application Intermediate Representation (AIR)

Programmatic construction requires a flexible Application Programming Interface (API) to a library that offers a set of convenient and labor-efficient functions, as well as the automation of tasks that complement the system designer's expertise. This section outlines the facilities that our envisioned AIR library provides to the ProCSy programmer in the form of a Python module.

1) Flexible elaboration: Traditional elaboration in SystemC follows a strict order. First, objects including modules and channels are instantiated by use of top-down constructor calls, usually starting with a `Top` module at global level or in the `sc_main` function. Second, the constructor function bodies are executed in bottom-up fashion for each object's initialization. Third and fourth, two instance hierarchy traversals are performed calling the optional callbacks `before_end_of_elaboration()` and `end_of_elaboration`, respectively.

This strictly ordered elaboration may or may not be what the system designer has in mind when building the design model. ProCSy eliminates this artificial restriction and allows flexible construction and composition of components in the desired order.

[3]Non-trivial constructors suffer from duplication of `sc_module_name` and specific ordering of member initializer lists.

[4]"Verilog and VHDL cannot express recursive generate statements" [11]

[5]Pinapa [8] works around this obstacle by inspecting the GCC AST to extract the model's structure; PinaVM [9] inspects the LLVM IR and uses JIT compilation to extract the structural design; [10] uses the GDB debugger Python API to extract a Verilog netlist.

[6]Using the same host language, [10] struggles with typing, distinguishes `sc_elab_const<T>` and `sc_elab_vector<T>` types which are "mutable during elaboration" but constant during simulation.

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on November 13,2026 at 19:26:38 UTC from IEEE Xplore. Restrictions apply.

Fig. 3 shows an example of building a grid of RISC-V processors. A two-dimensional array of processor modules is instantiated by intuitive `for` loops that iterate over the desired number of `rows` and `cols`. The processors are then connected by queue channels (i.e., `sc_fifo<image>` in TLM), one vertical and one horizontal connection between neighbors. Finally, ports and corresponding bindings are added at the North and South sides of the grid. At the East and West sides, open ports (which are not allowed in SystemC models) are automatically handled (either removed or connected to constant inputs, as illustrated by the `PlugOpenPorts` function).

```
def Construct(name="ProcessorGrid", rows=8, cols=8):
    grid = air.Module(f"{name}_{rows}x{cols}")
    for r in range(rows):
        for c in range(cols):
            p[r][c] = grid.AddInstance(RVcore(32,"imac"))
    for r in range(rows-1):
        for c in range(cols-1):
            grid.ConnectByQueueChannel("image", p[r][c], p[r][c+1])
            grid.ConnectByQueueChannel("image", p[r][c], p[r+1][c])
    for c in range(cols):
        p[0][c].Connect(grid.AddPort(f"pin{c}", In))
        p[rows-1][c].Connect(grid.AddPort(f"pout{c}", Out))
    grid.PlugOpenPorts()
    grid.SaveAIR(f"{name}_{rows}x{cols}.air")
    grid.ExportSystemC(TLM1, f"{name}_{rows}x{cols}.cpp")
    return grid
```

Fig. 3. Constructing a grid of processors with AIR API.

We emphasize that ProCSy's flexible elaboration strategy also resolves another serious limitation of SystemC, namely the omission of object deletion. SystemC pays a lot of attention to constructors, but completely ignores destructors, leaving them alone until the end of simulation for the C++ compiler and operating system to cleanup. SystemC's *"only build-up, never take down"* approach severely hinders the model construction.

In contrast, ProCSy allows to freely add and remove objects during construction, since it completely builds the model in the AIR data structure before it generates the SystemC source code. A straightforward example is the modeling of a processor grid where all nodes are the same but with a few exceptions. Fig. 4 illustrates this with a `Customize` function that may be applied to the grid created in Fig. 3. Here, a processor at a given position (r, c) in the grid is being replaced by an application-specific 64-bit alternative and its connectivity is adjusted accordingly.

```
def Customize(grid, r, c, replacement=None):
    grid.RemoveInstance(grid.p[r][c])
    grid.RemoveDanglingChannels()
    if replacement:
        grid.p[r][c] = grid.AddInstance(RVcore(64,"i"))
        if r==0:
            p[r][c].ConnectToPort(grid, f"pin{c}", In)
        else:
            grid.ConnectByQueueChannel(p[r-1][c], p[r][c])
        if r==rows-1:
            p[r][c].ConnectToPort(grid, f"pout{c}", Out)
        else:
            grid.ConnectByQueueChannel(p[r][c], p[r+1][c])
```

Fig. 4. Customizing a grid of processors with AIR API.

2) Automation: The construction of a model hierarchy and connectivity in SystemC often involves a lot of repetition. Rather than tedious writing or duplication of code with minor adjustments, programmatic construction can assist with automation. The ProCSy motto is here to "write less code manually, complete it automatically".

The examples above have already demonstrated the automatic handling of unconnected ports. A similar approach can be taken for incomplete connections. Rather than mapping every single port of one component to another, a *bundle*[7] of ports can be connected as one unit. Here, data types, directions, and bit widths or vector sizes can often be deduced automatically, based on the parameters of the involved components or the flow of data. For example, the data type for the ports in Fig. 3 and Fig. 4 can be automatically inferred from the `image`-type queue channels. Note that such automatic type inference[8] significantly reduces the effort needed to specify matching connections and at the same time minimizes "type mismatch" errors that would otherwise be reported during SystemC compilation.

Automation also plays a fundamental role when ProCSy generates the source code for SystemC simulation. The code generators follow established guidelines and conventions and apply standard modeling styles automatically, i.e., for TLM or TLM-2.0 compliance. A simple example here is TLM-2.0 socket binding in the mandatory direction from initiators to targets.

D. Static Verification of Structural Hierarchy

As discussed in Section II-B above, the result of the flexible elaboration in ProCSy is intentionally exposed as a file in AIR format. We emphasize that this file represents a static model of the hierarchical composition and connectivity. It is complete, contains all instantiated modules in their actual configuration (with constant parameters), and only those. This model is *finite*. Its quantified size yields certainty, enabling static analysis and making design decisions free from doubt.

As illustrated in Fig. 2, the fully constructed hierarchy and connectivity can be inspected in a text editor, either as generated SystemC code or in AIR format. The AIR file can also be analyzed by dedicated tools, e.g., for statistics on the type and number of components present, or for synthesis. A visualization tool can graphically display the hierarchical connectivity as a block diagram and provide interactive zoom functionality, hierarchy browsing, or connection highlighting.

The intermediate representation can also be modified by dedicated tools which load the AIR file, apply their processing using the AIR API, and then save it again in AIR format. Examples include tools to add or modify annotations, or apply optimizations. As a specific use case, an estimation tool can flatten the hierarchy, resulting in a model that is linear proportional to the design size. Not only does this provide an

[7]We reuse the concept of IO bundles from Chisel [5].

[8]This automation is similar to the `auto` type in modern C++ which deduces a type, for example, from an initializer.

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on November 13,2026 at 19:26:38 UTC from IEEE Xplore. Restrictions apply.

accurate count of its components, it can also serve as an early indication of the chip area after synthesis.

E. Dynamic Validation of Functional Behavior

The functional correctness of the model after construction is validated dynamically through simulation. The constructed SystemC description can be compiled against the standard Accellera proof-of-concept (POC) library and then executed in the same fashion as with the regular SystemC design flow [1]. A stimulus module feeds test vectors into the design under test, timed discrete event simulation (DES) is performed (with progress logs and/or traced waveforms, as desired), and a monitor module validates the generated output vectors.

Since IEEE SystemC simulation [12] is sequential and often slow [13], alternative parallel approaches are desirable. One option is simulation with the Recoding Infrastructure for SystemC (RISC)[9] [14]. Here, the static hierarchy constructed by ProCSy can eliminate many limitations, and fast and safe parallel simulation becomes possible.

Another alternative to validate a model's functionality is the execution in a pure C/C++ context with POSIX multi-threading. Although this method is untimed and is only applicable to TLM-style data-flow models with directed task graphs [16], it can take advantage of maximum parallelism and produces the fastest functional execution. Note that these advanced simulation options become possible only through designated tools that analyze and export an AIR model for the desired validation method.

III. Next Steps and the Road Ahead

We have conducted some early experiments on a proof-of-concept prototype of the proposed AIR library. For a small set of simple applications, such as the processor grid in Fig. 3, we successfully generated and simulated the corresponding SystemC TLM and TLM-2.0 descriptions. Here, we have observed strong indications of productivity benefits, including fewer lines of code to write and correspondingly less designer time in specification. These parameterized models can also be easily configured and scaled, with zero compiler errors and warnings for the generated source code.

While these promising results raise confidence in the ProCSy design flow, they need more scrutiny and maturity. We will further refine and carefully evaluate the ideas outlined in Section II with coverage of leaf modules' functionality, additional benchmark applications, and real-world case studies.

As a stepping stone on the road ahead, we now provide this position paper on the wild-and-crazy-ideas track to the 29th Forum on specification and Design Languages (FDL) in the hope of bringing a fresh breath of AIR into the system design community.

[9]The RISC compiler `risc` [14] can perform parallel-access conflict analysis of certain SystemC models at TLM or TLM-2.0 abstraction and then instrument the executable with protection for multi-core parallel simulation. However, RISC requires a model with a completely static instance tree, or the front-end processing with a very limited dynamic elaborator `elab` [15].

IV. Related Work

At RTL, Domain-Specific Languages (DSLs) [17] and HCLs include Chisel [5], PyMTL [18], [19], PyRTL [20], and Bluespec [21], which use Scala, Python, and SystemVerilog host languages, respectively, to construct synthesizable Verilog models.

At the system level, where SystemC [12] is the target language, previous works focus on the elaboration phase that precedes the actual simulation. Since the host language C++ lacks proper runtime introspection and reflection, existing tools extract the structure and connectivity of an elaborated model from SystemC kernel data structures during execution. Such dynamic elaboration tools include Pinapa [8], which uses the GCC Abstract Syntax Tree (AST), and PinaVM [9], which inspects the LLVM Internal Representation (IR) and utilizes just-in-time (JIT) compilation.

Similarly, "Hardware construction with SystemC" [10] extracts an elaboration database using the GDB debugger Python API. This work also promotes "elaboration-time programming" and distinguishes compile-time parameters from "elaboration-time constants". While it starts from the same premise as the proposed ProCSy, it uses SystemC as the host language and generates synthesizable code at RTL, not at the system level.

A compiler-based instrumentation approach is implemented in the Recoding Infrastructure for SystemC (RISC) [14]. This open-source framework [22] performs static conflict analysis for out-of-order parallel discrete event simulation. To obtain a static instance tree, a dedicated front-end `elab` [15] instruments the elaboration phase and extracts hierarchy and connectivity information, but many restrictions apply. In contrast, our proposed ProCSy eliminates these limitations by providing a fully static model structure via AIR.

V. Conclusion

This position paper proposes programmatic construction for SystemC, named ProCSy, as a solution to fill the need for a system-level construction language. The transformation that Chisel and FIRRTL have brought with Verilog construction, the proposed ProCSy and AIR promise for SystemC.

We have outlined the ProCSy flow in five steps, namely (1) programmatic construction, (2) separation of elaboration and simulation, (3) construction with automation, (4) static verification of structure, and (5) dynamic validation of functionality. In short, the traditional SystemC flow of "describe, compile, elaborate and simulate" becomes "construct, generate, and simulate".

We hope that the ProCSy ideas stimulate healthy and fruitful discussions and become a valuable stepping stone in shaping the future of system design with SystemC.

Acknowledgments

The author gratefully acknowledges and thanks Yanda Li, Lars Luchterhandt, Wolfgang Mueller, and Yutong Wang, for fruitful discussions, valuable contributions, and constructive criticism that refined the ideas in this position paper.

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on November 13,2026 at 19:26:38 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[2] IEEE, "IEEE standard VHDL language reference manual," *IEEE Std 1076-1987*, pp. 1–218, 1988.

[3] IEEE, *IEEE Standard Description Language Based on the VERILOG Hardware Description Language, 1364-1995*. USA: IEEE Standards Office, 1996.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[7] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*. Sunnyvale, CA, USA: Artima Incorporation, 1st ed., 2008.

[8] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip," in *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, (New York, NY, USA), p. 317–324, Association for Computing Machinery, 2005.

[9] K. Marquet and M. Moy, "PinaVM: a SystemC front-end based on an executable intermediate representation," in *International Conference on Embedded Software*, (Scottsdale, USA), p. 79, 10 2010. SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic).

[10] R. Popov and M. J. Moiseev, "Hardware construction with SystemC," in *Design and Verification Conference (DVCon)*, 2018.

[11] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, Nov 2017.

[12] IEEE, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[13] R. Dömer, "Seven obstacles in the way of standard-compliant parallel SystemC simulation," *IEEE Embedded System Letters*, vol. 8, pp. 81–84, Dec. 2016.

[14] T. Schmidt, G. Liu, and R. Dömer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation," in *Proceedings of the Design Automation Conference (DAC)*, June 2017.

[15] T. Schmidt, G. Liu, and R. Dömer, "Hybrid analysis of SystemC models for fast and accurate parallel simulation," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 226–231, 2017.

[16] Y. Wang, Y. Li, and R. Dömer, "GPCC: Grid of processing cells compiler with pareto-optimal design space exploration," in *MBMV 2026; 29th Workshop*, 2026.

[17] Z. Han, K. Devarajegowda, M. Werner, and W. Ecker, "Towards a Python-based one language ecosystem for embedded systems automation," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pp. 1–7, 2019.

[18] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 280–292, 2014.

[19] C. Batten, N. Pinckney, M. Liu, H. Ren, and B. Khailany, "PyHDL-Eval: An LLM evaluation framework for hardware design using python-embedded DSLs," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, MLCAD '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[20] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, 2017.

[21] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pp. 69–70, 2004.

[22] G. Liu, T. Schmidt, Z. Cheng, D. Mendoza, and R. Dömer, "RISC Compiler and Simulator, Release V0.6.0: Out-of-Order Parallel Simulatable SystemC Subset," Tech. Rep. CECS-TR-19-04, Center for Embedded and Cyber-physical Systems, University of California, Irvine, Sept. 2019.

Authorized licensed use limited to: Access paid by The UC Irvine Libraries. Downloaded on November 13,2026 at 19:26:38 UTC from IEEE Xplore. Restrictions apply.