2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Moving SystemC to a New C++ Standard

Ralph Görgen, OFFIS, Oldenburg, Germany (ralph.goergen@offis.de)

Philipp A. Hartmann, Intel, Duisburg, Germany (philipp.a.hartmann@intel.com)

***Abstract*—With C++11, a long expected update of the C++ programming language has been standardized, followed by another update to C++14, bringing in additions to close some remaining gaps. These two releases of the C++ standard have received much attention in the C++ community as they have introduced a number of features that can significantly improve productivity, safety and performance of C++ programs. In this paper, we briefly review the relation of SystemC to its C++ foundation and give recommendations how to move the IEEE 1666 SystemC standard and its implementations towards building upon these new C++ standards.**

Keywords—IEEE 1666; SystemC; C++; C++11; C++14; standardization;

I. INTRODUCTION

After the initial release as ISO/IEC 14882:1998, it took more than ten years to finalize the next revision of the C++ programming language standard. This long awaited major update of C++ has been ratified and released in 2011 and is informally known as "C++11". Along with this new standard, the development model for the C++ language standard has been switched to a decoupled one based on so-called *Technical Specifications* (TS) enabling a much faster update cadence. As a result, the next major standard update took place in 2014 already ([1], "C++14") and another one is expected for 2017 [2]. Unlike with earlier C++ standards, implementers have been very quick to support newly added language features together with or shortly after the standards' releases. At least three major implementations for C++14 are already available today.

With C++ being the technical foundation of SystemC, the evolution of the C++ language is very relevant for SystemC as well. In practice, many of the new C++ features can already be directly used in SystemC models today, provided that the underlying C++ implementation supports them. This includes features like automatic type deduction, lambda functions, or direct initialization, which are especially valuable in the context of SystemC modeling as demonstrated in [3], [4], and [5]. In order to leverage this evolution towards increased modelling productivity, safety, and performance in SystemC, an update of the C++ foundation of SystemC is needed.

In practice, the evolution of SystemC can be divided into two aspects: the evolution of the SystemC *standard* and language reference manual (IEEE Std. 1666, [6]), and the development of the SystemC *implementations* provided by the several vendors, including Accellera's open-source proof-of-concept implementation [8]. Regarding the underlying C++ foundation, these two aspects come with different steps and requirements, which are discussed in the following Sections II and III. In Sections IV and V, we discuss some semantics preserving language and implementation improvements, which can be introduced in SystemC on top of C++14. Extending the SystemC language itself with new features based on modern C++ primitives is out of the scope of this paper.

II. IEEE STD. 1666 EVOLUTION STRATEGY

As required by the IEEE Standards Association, the IEEE Std. 1666 must be updated at least every ten years. The current IEEE Std. 1666-2011 Standard SystemC Reference Manual [6] refers to C++ as its underlying technology in *Section 1.4 Relationship with C++* and *Section 2. Normative references*. More specifically, it currently references ISO/IEC 14882:2003, the most recent C++ standard when IEEE 1666-2011 was ratified. The standard requires that SystemC shall be used in conjunction with ISO/IEC 14882:2003. Strictly speaking, this means that using C++11 features together with the SystemC library is not conforming to IEEE 1666-2011. Hence, this reference needs to be updated.

The next revision of IEEE Std. 1666 is currently planned for the end of this decade. We propose to link the next revision to **C++14** (ISO/IEC 14882:2014, [1]) to officially allow using the new C++ features together with SystemC. Moreover, such features are allowed to be used in the standardized API as well. This enables several

major language improvements, and by the time of the release of the next revision of IEEE Std. 1666, most productively used compilers and tools are expected to support C++14.

Given the more frequent release cadence of C++ standards mentioned in the introduction, an additional clarification on the use of more recent C++ standards (or implementations) should be added to IEEE Std. 1666 as well. Here, we propose allowing the use of future versions of C++, provided that the execution semantics of the API defined in IEEE 1666 is explicitly defined by the implementation. This means, users and implementations agree on the chosen C++ standard.

III. SystemC Implementation Evolution Strategy

On the other side, SystemC implementers – including the Accellera SystemC Language Working Group [7], providing the proof-of-concept implementation – release updated versions more frequently to address errata, distribute bug fixes and to include new features beyond the definitions of IEEE Std. 1666. Usually, these new features address immediate modeling needs or are provided for field testing before being submitted for inclusion into the next IEEE revision[1].

In order to address the inclusion of new C++ features based on C++11/C++14, several practical aspects for SystemC implementers and users need to be considered. Today, not every EDA tool and compiler provides full support for C++1x in their bundled C++ implementations. Although the list of supported features is growing, SystemC users may run into problems if their SystemC implementation relies on not yet supported C++1x features. Hence, we recommend a step-by-step introduction of new features that is accompanied with an Opt-Out mechanism in the first phase. This allows users to benefit from bug fixes and other improvements even if they are bound to older compilers or build configurations.

A. *Opt-Out mechanism for C++1x features*

A straight forward setup to enable optional inclusion of certain features is the use of the C++ preprocessor. We recommend to introduce a "top-level" switch to limit the underlying C++ standard in a SystemC implementation via a symbol called `SC_STD_CPLUSCPLUS`, with values following the `__cplusplus` symbol as defined by ISO/IEC 14882:

ISO/IEC 14882	__cplusplus
14882:1998, 14882:2003	199711L
14882:2011	201103L
14882:2014	201402L

A similar `IEEE_STD_1666_CPLUSPLUS` symbol could also be added to IEEE Std. 1666-20xx, referencing the base version of C++ for the current SystemC standard. Users could then request a reduced C++ language subset by defining this value to a smaller value than provided by the compiler itself:

```
#ifndef SC_STD_CPLUSCPLUS // default to compiler's value, if not set by user
# define SC_STD_CPLUSPLUS __cplusplus
#endif
```

As a quality of implementation measure, implementations could then define detailed feature macros for the individual C++ language features, which are controlled by the top-level setting. This would then allow further control of the C++ subset, taking compiler or platform specific limitations into account.

B. *ABI compatibility checks*

Many (modern) C++ implementations provide manual selection of a specific C++ standard based on a configuration switch (e.g. `-std=c++11` on GCC-compatible compilers). In some C++ implementations, this leads to changes in the compiler's *Application Binary Interface* (ABI). Additional changes to the ABI can be introduced by some of the C++ annotations mentioned below. When combining user models with a SystemC implementation

[1] This is in accordance with IEEE 1666-2011 [6], Section 1.4: *Implementors and users are free to extend SystemC* [using the mechanisms provided by the C++ language], *provided that they do not violate this standard.*

library[2], the compatibility of the compiler and library configurations must be ensured in order to avoid runtime errors. We recommend extending the commonly used "compatibility check" in the SystemC implementations to include the C++1x-related configuration into the set of consistency checks between the library and the user models.

IV. SystemC Language Improvements

In the following, we propose improvements to the SystemC language and standardized API. Some are low-hanging fruits and require only the addition of annotations wherever they make sense, others are more complex and need to be discussed and defined in detail first.

A. Add `default` and `delete` annotations

The keywords `default` and `delete` can be used with special member functions such as constructors or operators. They express explicitly that the default implementation should be used here or usage of this function is not allowed. In SystemC, this can be used to disable copying and default construction of SystemC objects.

B. Add `constexpr` annotations

Generalized constant expressions – marked with `constexpr` qualifier – are functions/variables that contain simple statements only and that can be evaluated at compile time. Beyond improvement of execution performance, constant expressions are allowed to be used as template arguments, which opens a variety of applications in SystemC, for example in high-level synthesis.

C. Add `override` and `final` annotations

The keywords `override` and `final` are used to explicitly mark a method that overrides a base class method or to mark a base class method that shall not be overridden in any derived class. This feature improves code quality by avoiding accidental creation of virtual functions or overrides.

D. Add `noexcept` annotations

This annotation marks functions that do not throw exceptions and it is used for compiler optimizations.

E. Remove reference to Boost library

The current implementation of the SystemC library refers to the Boost C++ Library [9], which is not covered by an international standard. The new C++ standards provide features that are similar to the Boost features used in SystemC. Hence, they can be replaced by standard features and the dependency on the Boost library can be removed from IEEE Std. 1666, *Section 5.5 sc_spawn_options and sc_spawn*[3].

F. Replace internal enums with strongly typed enumerations

In C++11, type-safe enumerations have been introduced. In contrast to C++03 enumerations, they do not automatically convert to integer, e.g., comparison of two values of different enumeration types is not allowed. In SystemC, such strongly typed enums can be used to replace the internal enum types like `sc_logic_value_t` or TLM phases.

G. Move support for SystemC objects

Another feature introduced in C++11 are so-called *rvalue references* to implement *move semantics* for objects. Rvalue references are specific references to temporary objects and *move* means extending the temporaries lifetime by assigning it to another reference, effectively *moving* it to another scope. This avoids unnecessary copying of objects and is especially useful for *NonCopyable* objects. In the SystemC language, many classes forbid default construction and/or copying of its instances, for instance modules or signals. Enabling move semantics for these objects could facilitate their creation and handling a lot. However, the definition of the exact move semantics is not straightforward. It requires some more discussion, which makes this a more long-term goal.

---

[2] Binary compatibility across different SystemC implementations is not guaranteed by IEEE Std. 1666.

[3] This is an obvious example, where the SystemC implementation ABI changes depending on the selection/deselection of this feature in a user model.

V. SystemC Implementation Improvements

The following recommendations are only relevant for SystemC implementations and do not affect the standard, while still improving usability and safety:

A. *Add '[[deprecated]]' attribute to deprecated features*

Explicit attribute support has been added to C++11 as well, providing a standardized way for annotations without relying on implementation-specific extensions. The `[[deprecated]]` attribute allows a compiler to issue warnings when deprecated features are used. This is safer and more convenient than runtime checks and warnings. Especially when deprecated features are used in corner cases, runtime warnings might occur very rarely. In contrast, compiler warnings occur whenever a deprecated feature is used somewhere in the code. All classes and functions that are listed as deprecated in the SystemC standard ([6], Annex C) can be marked with this new attribute in the implementation.

B. *Use `nullptr` as null pointer constant*

C++11 introduces `nullptr` as an explicit null pointer constant. It provides better type-safety because it is distinguishable from integer constants, for instance when calling a function that has both integer and pointer overloads. Moreover, `nullptr` is not convertible or comparable to integral types, except for bool in explicit contexts. In the SystemC implementations and user models, this new keyword can be used to replace `NULL` or comparable constants for pointer initialization.

Leveraging new C++ language features inside a SystemC implementation can of course bring additional benefits. However, in order to provide support for more (and older) C++ implementations, SystemC implementers might need to stick to a smaller C++ subset for some time.

VI. Conclusion

In this paper, we discussed the required steps for moving SystemC to newer C++ standards, both related to the IEEE standardization as well as some practical considerations for SystemC implementers. While the SystemC standardization side is quite straight forward, the implementation side is more complex, especially when targeting multiple C++ implementations (i.e. multiple compiler versions with different C++ standards support). In order to enable an incremental move towards modern C++ usage in the SystemC ecosystem, we suggest a configurable selection of the C++ standard, combined with consistency checks to ensure binary compatibility. Additionally, we summarized a set of immediate improvements, both to the SystemC language as well as purely on the implementation side.

In summary, the modern C++ standards can provide significant modeling improvements in productivity, safety and simulation performance. In the LWG, we have started exploring practical solutions to bring the optional C++1x support into the proof-of-concept implementation.

References

[1] ISO/IEC 14882:2014, Programming Languages—C++. http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029

[2] Standard C++ Foundation. https://isocpp.org

[3] David Black. “What C++11 means to SystemC”, NASCUG 2012, http://nascug.org/events/17th/black_cpp11_2_27_2012.pdf

[4] Ralph Görgen, Philipp A. Hartmann and Wolfgang Nebel. “Automated SystemC Model Instantiation with modern C++ Features and sc_vector”. In Proceedings of DVCon Europe 2015. November 2015.

[5] Roman Popov, “Problems with SystemC syntax”. Accellera Forum, http://forums.accellera.org/topic/5472-/

[6] IEEE Computer Society. “IEEE Standard for Standard SystemC Language Reference Manual”, IEEE Std 1666-2011, Jan 2012. https://standards.ieee.org/findstds/standard/1666-2011.html

[7] Accellera Systems Initiative. “SystemC Language Working Group”. http://accellera.org/activities/working-groups/systemc-language

[8] Accellera Systems Initiative. “Core SystemC Language and Examples”. http://accellera.org/downloads/standards/systemc

[9] Boost C++ Library. http://www.boost.org/

[10] Bjarne Stroustrup, Herb Sutter, et.al. “C++ Core Guidelines”. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines