What C++11 means to SystemC?

By David C Black of Doulos Inc

Abstract

In September of 2011, ISO approved an update to the C++ standard, which is known variously as ISO/IEC 14882:2011, C++0x and C++11. This paper takes a quick look at some of the features and illustrates how they can change the way we write SystemC code for the better. Discussion considers potential impacts to performance and code quality. Code examples are accompanied with comments on experiences using the new features and what limitations were encountered.

Introduction

SystemC means a lot of things to the electronic design community. For many it describes a way of modeling Electronic System Level designs (ESL), and plays a major role in architectural exploration and creating virtual platforms that allow early software development before hardware is available. Technically, SystemC is simply a standardized C++ library with an event driven simulation kernel and facilities to simplify modeling of hardware components using C++.

With the recent releases of standards for both C++ and SystemC, it seems like a good time to see how the standards interact. Some perspective on the two standards is a good place to start, so the next section explores the history of the two standards.

Brief history of C++ and SystemC

C++ has a long history dating back to the 1970's when Object Oriented Programming (OOP) techniques were becoming established and Bjarne Stroustrup was investigating how C might be extended to support OOP. Over the years C++ evolved to be much more than just an OOP language, and is now termed a multi-paradigm language. Be that as it may, C++ was quite popular and was eventually standardized in 1999 by the International Standards Organization (ISO). Since that time it has undergone several updates.

The following table summarizes the history of C++.

Year	C++ Activities
1970's	Bjarne Stroustrup begins investigation
1990	ISO Standardization begins
1998	C++98 ISO/IEC 14882-1998
2003	C++03 ISO/IEC 14882-2003 (minor)
2007	C++TR1 ISO/IEC 19768-2007
2011	C++11 ISO/IEC 14882-2011 (aka C++0x) – includes TR1

Table 1 - C++ History

It should be noted that due to the long time before standardization was formalized, many C++ compilers took an even longer time before becoming compliant to the 1998 standard. This meant that compatibility across platforms was an issue for several years until well into 200x. This played a role in SystemC as we shall see.

More recently, and the subject of this paper, C++ received a major update in the form of ISO/IEC 14882-2011, which was ratified in September 2011. This update has been in the works for some time and was variously referred to as C++0x due to the expectation it would be finalized sometime in the later half of the first decade (2004-2009). It is now humorously referred to by some as C++0xB as in 2000 + 0xB = 2011.

Meanwhile, SystemC was introduced to the electronic design community in 1999 and underwent several revisions culminating most recently in IEEE-1666-2011. Early 1.x versions of the standard were notoriously cumbersome and the lack of conformance amoung C++ compilers didn't help. By the time version 2.1 was released, it was common practice to use the GNU C++ compiler version 3.1; although, version 2.93 was still supported for some time. The 2011 standard was finally published in January 2012, and is available for download at no charge via www.systemc.org.

The following table summarizes a history of SystemC.

Year	SystemC Activities
1999	SystemC v0.9 established
2000	SystemC v1.0 approved
2001	Open SystemC Initiative formed
2002	SystemC v1.02
2005	TLM 1.0 release; SystemC v2.1 released; IEEE-1666-2005 approved
2007	SystemC v2.2 released (IEEE compliance)
2008	TLM 2.0 approved
2010	OSCI merges with Accellera
2011	IEEE-1666-2011 approved (incorporates TLM)
2012	SystemC v2.3 expected

Table 2 - SystemC History

It should be noted that at the time of this writing, the open source proof-of-concept implementation version 2.3 has not been released to the public, but is expected to be available in the summer of 2012.

Complete List of changes to C++11

This paper is focused on how the changes in C++11 will affect coding practices in SystemC.

The following table lists all of the C++11 changes to the standard along with an indication of which C++ compiler versions support it. Hyperlinks in the table bring up specific text related to each change.

Language Feature	Proposal	Clang	GCC
Rvalue references	N2118	2.9	4.3
Rvalue references for `*this`	N2439	2.9	[Note]
Initialization of class objects by rvalues	N1610	2.9	Yes
Non-static data member initializers	N2756	3.0	4.7
Variadic templates	N2242	2.9	4.3
Extending variadic template template parameters	N2555	2.9	4.4
Initializer lists	N2672	No	4.4
Static assertions	N1720	2.9	4.3
`auto`-typed variables	N1984	2.9	4.4
Multi-declarator `auto`	N1737	2.9	4.4
Removal of auto as a storage-class specifier	N2546	2.9	4.4
New function declarator syntax	N2541	2.9	4.4
New wording for C++0x lambdas	N2927	No	4.5
Declared type of an expression	N2343	2.9	4.3
Right angle brackets	N1757	2.9	4.3
Default template args for function templates	DR226	2.9	4.3
Solving the SFINAE problem for expressions	DR339	2.9	4.4
Alias templates	N2258	3.0	4.7
Extern templates	N1987	2.9	Yes
Null pointer constant	N2431	3.0	4.6
Strongly-typed enums	N2347	2.9	4.4
Forward declare enums	N2764	No	4.6
Generalized attributes	N2761	No	No
Generalized constant expressions	N2235	No	4.6
Alignment support	N2341	3.0	No
Delegating constructors	N1986	3.0	4.7
Inheriting constructors	N2540	No	No
Explicit conversion operators	N2437	3.0	4.5
New character types	N2249	2.9	4.4
Unicode string literals	N2442	3.0	4.5

Language Feature	Proposal	Clang	GCC
Raw string literals	N2442	3.0	4.5
Universal character name literals	N2170	No	4.5
User-defined literals	N2765	No	4.7
Standard Layout Types	N2342	3.0	4.5
Defaulted and delete functions	N2346	3.0	4.4
Extended friend declarations	N1791	2.9	4.7
Extending sizeof	N2253	No	4.4
Inline namespaces	N2535	2.9	4.4
Unrestricted unions	N2544	No	4.6
Local and unnamed types as template arguments	N2657	2.9	4.5
Range-based for	N2930	3.0	4.6
Explicit virtual overrides	N2928	3.0	4.7
	N3206		
	N3272		
Minimal support for garbage collection and reachability-based leak	N2670	No	No
Allowing move constructors to throw [noexcept]	N3050	3.0	4.6
Defining move special member functions	N3053	3.0	(core)
Concepts [not part of C++11]	-		4.6
Concurrency	-		Partial
Sequence points	N2239	No	
Atomic operations	N2427	No	No
Strong Compare and Exchange	N2748	No	4.4
Bidirectional Fences	N2752	No	No
Memory model	N2429	No	No
Data-dependency ordering: atomics and memory model	N2664	No	No
Propagating exceptions	N2179	2.9	No
Abandoning a process and at_quick_exit	N2440	No	4.4
Allow atomics use in signal handlers	N2547	No	No
Thread-local storage	N2659	No	No
Dynamic initialization and destruction with concurrency	N2660	No	No
C99 Features	-		No
__func__ predefined identifier	N2340	2.9	
C99 preprocessor	N1653	2.9	4.3
long long	N1811	2.9	4.3
Extended integral types	N1988	No	4.3

Usability & convenience improvements

First, several of the new C++ features will greatly simplify certain coding tasks. Take for example the following C++07 code:

```
list<int> my_list; // assume list<> required
my_list.push_back(1);
my_list.push_back(8);
my_list.push_back(42);
typedef list<int>::const_iterator list_t;
for (list_t it(my_list.begin());
     it != my_list.end(); ++it) {
  fifo->write(*it)
}
```

Figure 1 – C++07 iteration

Many will instantly recognize the STL (Standard Template Library) container idiom for visiting every element of a container (std::list); however, the code is somewhat tedious. The following is the equivalent written using new C++11 constructs. New syntactical elements are highlighted.

```
list<int> my_list { 1, 8, 42 }; // initialize

for (int& val: my_list) { // foreach

  fifo->write(val);

}
```

Figure 2 – C++11 iteration

Notice first the new initialization technique using curly brackets {} as an initializer. This feature has a lot more power than shown here, but should be compelling enough with this simple example. Second, the use of a *foreach* syntax is quite compelling. No more do we need the typedefs and indirection of iterators.

Next consider a method to write a value after a time delay. In C++07, this requires a method to be defined and several explicit constructions for SystemC datatypes:

```
sc_fifo<sc_bigint<96> > fifo;
void delayed_write(sc_time& t, sc_bigint<96> v)
{
  wait(t);
  fifo->write(v);
}
sc_process_handle h = sc_spawn(
  sc_bind(&delayed_write
         , sc_time(11.5,SC_NS)
         , sc_bigint<96>("0x9E5C1234")
));
```

Figure 3 – C++07 spawning

The constructors can be simplified using C++11's new user-defined literals, which could be incorporated into a future version of SystemC:

```
#if __cplusplus >= 201103L /* check for standard version */
  sc_bigint<128> operator"" _big(const char* p) // user defined
    { return sc_bigint<128>(p); }
  sc_time operator"" _ns(int t) // user defined
    { return sc_time(t,SC_NS); }
#endif
```

Figure 4 – C++11 User defined literals (potential additions to SystemC)

Finally, using lambda's, the original code can now be rewritten as:

```
sc_signal<sc_bigint<96>> s; // no space!
auto ph = sc_spawn([&](){wait(11.5_ns);s.write(0x9E5C1234_big);});
```

Figure 5 – C++11 spawning

This greatly reduces the coding overhead and makes it much more readable.

Using lambda's has some other potentials. For many it has been a complaint that Verilog has a simple *continuous assignment* that makes simple logic equations easy such as:

```
assign      y = a + b ;
always_reg @(posedge clk) q <= a + b ;
```

Figure 6 – SystemVerilog simplicity

With the addition of some simple macros, SystemC can have this ability too.

```
/*global*/ list<sc_process_handle> assigns, always_ffs;
#define ASSIGN(sensitive,equation)\
assigns.push_back(sc_spawn([&](){ for(;;){\
  wait(sensitive);\
  equation;\
}})

#define ALWAYS_FF(clock,equation)\
always_ffs.push_back(sc_spawn([&](){ for(;;){\
  wait(clock);\
  equation;\
}})
```

Figure 7 – SystemC possible new macros

Now it is possible to code the following:

```
ASSIGN(   a_sig|b_sig,   y_sig = a_sig + b_sig );
ALWAYS_FF( clk.pos(),   q_sig = a_sig + b_sig );
```

Figure 8 – SystemC with C++11 potential simplicity

Safety improvements

Convenience is not the only type of improvement. Safety is a big concern when coding. I am often asked how to avoid certain coding problems. With C++11, several features will help ensure your intent is understood.

The following example contains several of the many safety enhanced features of C++11:

```
class Parent {
  int m_x{5}, m_A[3]{1,2,3}; // Initialize
public:
  Parent(float f) : m_x(int(f)) {} // Constructor
  Parent () : Parent(3.14) {} // Reuse constructor
  Parent (const Parent&) = default; // Document intent
  explicit operator string(); // Explicit conversions
  bool stop_here(void) const final; //Disallow override
  void some_func(int x);
};
class Derived : public Parent {
  void some_func(int x) override; // Intended override
};
static_assert( sizeof(long long)>= 4 // Compile-time check
              , "long long too short");
```

Figure 8 – SystemC with C++11 potential simplicity

In the preceding, reuse of constructors means less likelihood of cut-n-paste errors and obviates the need to create helper functions for this purpose. The "default" declaration allows a functional declaration that you intend to use the default behavior, which in this case is a default copy constructor. The "final" keyword disallows for a derived class to try overriding a method in a parent. The "override" keyword avoids mistakes due to mistyping a method's signature when attempting to override a parent's behavior. Finally, static_assert allows for clearer error checking at compile time.

Of course your mileage may vary and certainly there is still plenty of room for better compilers and lint checkers.

Performance improvements

Finally, it should be noted there are some improvements that simply enhance code efficiency and improve performance. The most notable of these features is the "move constructor", which was designed for situations where a copy can be replaced with a move since the original is about to be destroyed. This has resulted in big improvements in STL performance, and presumably may be applicable to SystemC and TLM implementations.

How to get/use

The concepts explored in this paper were created using a pre-release version of SystemC 2.3 proof-of-concept simulator and GCC version 4.7 on both Mac OS X Lion (64-bit) and Linux Fedora Core 15 (64-bit) using the –std=c++11 compiler switch. Presumably other compilers will have similar abilities soon (see table 2 earlier in this paper).

For more information on feature descriptions, please see:

- http://en.wikipedia.org/wiki/C%2B%2B11
- http://www.cprogramming.com/c++11/what-is-c++0x.html

For information on compiler support, the following web resources should be helpful:

- http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport
- GNU http://gcc.gnu.org
- Clang http://clang.llvm.org/

The real challenge may be in properly installing new versions of the toolchain to support C++11 (e.g. GCC 4.7). We highly suggest installing tools in a location separate from the default location on Linux systems due to the way in which Linux systems rely on the compilers for other uses. The popular SourceForge 'modules' package (see http://modules.sourceforge.net/) has helped immensely in managing some aspects of this type of non-standard installation.

If you are interested in examining some of the code that was used during our exploration, you may download a small tarball containing an example of the code from the KnowHow section of the Doulos website.

Example code tarball availale: http://www.doulos.com/knowhow/systemc/cxx11/

Experience

For the most part, we found the C++11 features just work. The only real challenges were understanding exactly how the features work. Fortunately, there are plenty of descriptions to read on the web with examples. There is certainly a lot of room for more experimentation, but what little was done has given us a real appreciation for the potential benefits.

Recommendations

What do we recommend? First, design teams should start learning now. Unless you take the time to try the features, your code will continue to be somewhat bloated and less readable than it could. This is particularly useful for modeling only teams who are less likely to be dependent on tools outside the C++ tool chain.

We do note that most EDA tools are not using the latest version of C++ tools, and in many cases they are several years behind the state of the art. Ask your vendors for support, and explain how the features will help get designs to market faster and with better quality. Of course this is not required if you are only using SystemC in a stand-alone environment, but most folks are importing 3rd party IP. This means IP providers need to start looking at C++11 as well.

If you happen to be using High Level Synthesis tools (HLS) for your C++ design, it may be some time before vendors start supporting C++11; however, it's not too early to start asking.