Compiling SpecC for Simulation

Jianwen Zhu

Electrical and Computer Engineering
University of Toronto
Toronto, ON M5S 3G4
Tel: 416-946-5971
Fax: 416-71-2326
e-mail: jzhu@eecg.toronto.edu

Daniel D. Gajski

Informatin and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
Tel: 949-824-4155
Fax: 949-824-4155
e-mail gajski@ics.uci.edu

Abstract—**Systems-on-chip (SOC) design calls for the use of executable system design language (SLDL). SpecC is a C-based SLDL designed to embrace the IP-centric design methodology. In this paper, we present a SpecC-language-based approach for system level simulation of SOC. Pros and cons of this approach is compared against the existing library-based approaches. Furthermore, we discuss in detail the various design considerations for the SpecC simulation API as well as our reference implementation.**

I. Introduction

In the last few years, it has been a routine for papers and public speeches in VLSI-related conferences to start with the prediction of systems-on-chip as the inevitable outcome of Moore's law. Today, many semiconductor and telecommunication vendors are already designing and delivering systems-on-chip with, however, painstaking efforts. These efforts taken are a natural result of the new challenges posed by system-on-chip design.

One of the design challenge is the lack of proven top-down refining methodology, which brings the product concept at very high level all the way down to the register transfer level, at which current EDA tools are mature enough to produce first-time-right silicon. Such methodology starts with a specification at the so-called system level, and then makes judicious design decisions in different aspects of the design space. The decisions are mostly system architecture selection, as well the mapping from specification to the architecture. These design decisions are either taken or discarded based on performance evaluation of the intermediate designs.

Due to the pressure of time-to-market, it is not realistic any more to design system-on-chip completely from scratch. Hence the reuse of the so-called intellectual property (IP) components, becomes an absolute necessary. The IP components can be general purpose, for example, processor cores, processor peripherals, standard on chip bus interfaces, DSP cores, on-chip memories; or specialized, for example, Ethernet media access control block, viterbi decoder etc. The incorporation of the modeling and reuse of IP into the system-level design methodology hence becomes an attractive option. In fact, an IP-centric design methodology [1] has recently been proposed. In addition, a system level design language (SLDL) called SpecC [2] is proposed to facilitate this methodology.

In this paper, we address the verification challenge at the system level by describing the implementation of the SpecC simulator. The importance of verification to the IP-centric design methodology is three-fold: First, by efficient simulation, it functionally verifies if the system-level design specification is exactly what the designer needs in the first place. Second, again by simulation, it functionally verifies if an intermediate design is consistent with the specification. Third, by profiling typical design scenarios, it helps to analyze the quality metrics of an intermediate design such as speed and power. In short, verification serves as the touchstone of decision points in the refinement decision tree.

In the sequel, we first compare our approach of system-level simulation with other other proposals in Section II. We then discuss the design of SpecC simulation API (for Application Programmable Interface) in Section IV and show how the SpecC programs are compiled into C++ code with calls to SpecC API in Section V. We then describe in detail a reference design of the API in Section VI. The experimental results are discussed in Section VII.

II. Library-based vs Language-based

While the current industry standard for IP development and deployment is at the register transfer level (soft IPs) or physical level (hard IPs), many vendors realize that these abstraction levels are too low for system specifiers and implementors to communicate. The predominant languages used by system specifiers are C or its derivatives. This can be exemplified by the reference designs of telecommunication standards released by the ITU. While the English specification of components and systems helps to understand the principles, in many cases the "C specification" is the best reference for implementers who need to know every detail. For all practical reasons, it is desirable to use C/C++ to serve as the golden model between system specifiers and implementors.

Unfortunately, while C can serve as a good language for algorithmic specification, it cannot be readily used for modeling IPs at the register transfer level. To solve this problem, a number of efforts emerged, which happen to follow a similar technical and business strategy. CynApps recently announced its Cynlib [3], a C++ class library which provides features so that C++ can be used to model hardware. The Open SystemC Initiative, announced a similar library called SystemC [4] [5]. Another similar implementation with arguably superior simulation performance is the OCAPI library [6] developed by IMEC. If either product becomes a standard, the adopters can then freely exchange their IPs in C++—a similar goal accomplished by VSIA for soft IPs except that now the IP models can be simulated quickly using inexpensive or even free C/C++ compilers.

While these library-based approaches effectively turn C/C++ into a hardware description language and hence the specifiers and implementors now seem to speak the same language, the semantic gaps remain: the mapping from system functionality in the specification, which is often captured by various computational models, to the system architecture, which the implementors are supposed to explore, is not trivial. The mapping from algorithmic representation of IP components from providers to the RTL representations of IP integrators relies on behavioral synthesis tools which are not yet mature. It can be envisioned that CynApps and SystemC can be extended to describe system specifications much the same way as the Ptolemy project but the question is how easy these specifications can be understood by system-architecture exploration and behavioral synthesis tools, in contrast to humans or simulation engines only.

The SpecC Technology Open Consortium (STOC) [7], backed by Japan's top-tier electronics and semiconductor companies, was founded in 1999 to promote the adoption of SpecC as the specification language for system level design. While motivated by the same desire to move from RTL to C/C++ for system level design, SpecC takes a different approach than the library-based approaches taken by Cynlib, SystemC and OCAPI. The SpecC language was developed by first identifying the requirements of SOC design and then carefully devising a set of constructs with well-defined semantics. These constructs are an extension of the existing ANSI-C language. When a SpecC specification is simulated, the extended constructs are expanded into a set of simulation API calls, which are roughly equivalent to the API defined by SystemC and Cynlib open source libraries. The difference here is that the user interacts with the constructs and the library is hidden. In addition to the added abstraction and expressive power from the specifier's side, the SpecC approach also has the advantage that the synthesis tools have a much easier task of analyzing and understanding the specification than the library-based approach, where it is hard to differentiate the code used for specification from the code used for simulation.

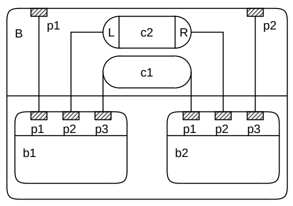
III. A Short SpecC Tutorial

Semantically, the functionality of a system is captured as a a hierarchical network of behaviors interconnected by hierarchical channels. Syntactically, a SpecC program consists of a set of behavior, channel and interface declarations.

A behavior is a class consisting of a set of ports, a set of component instantiations, a set of private variables and functions, and a public main function. Through its ports, a behavior can be connected to other behaviors or channels in order to communicate. A behavior is called a composite behavior if it contains instantiations of child behaviors. Otherwise it is called a leaf behavior. The functionality of a behavior is specified by its functions starting with the main function.

A channel is a class that encapsulates communication. It consists of a set of variables and functions, called methods, which define a communication protocol. A channel can be hierarchical, for example subchannels can be used to specify lower level communication. An interface represents a flexible link between behaviors and channels. It consists of declarations of communication methods which will be defined in a channel.

For example, Figure 1 shows a SpecC program. The example system species a behavior `B` consisting of two subbehaviors `b1` and `b2` and communicate via integer `c1` and channel `c2`. Note that the `par` statement specifies that `b1` and `b2` execute in parallel. There are other statements in SpecC which define different rules of composition. For example, the `pipe` statement specifies that the subbehaviors execute in pipelined fashion. The `try-trap-interrupt` specifies exception and interrupt handling. The `fsm` statements specifies a finite-state-machine like behavior. For a detailed treatment, please refer to the language reference manual contained in [1].



```
interface L { void write( int a ); };
interface R{ int read( void ); };

channel C implements L, R {
   int data; bool valid;

   void write( int x ) {
      data = x; valid = true;
      }
   int read( void ) {
      while( !valid ) waitfor( 10 );
      return data;
      }
   };
```

```
behavior B1( in int p1, L p2, in int p3 ) {
   void main( void ) { ....p2.write(p1); }
   };
behavior B2( out int p1, R p2, out int p3 ) {
   void main( void ) { .... p3 = p2.read(); }
   };
behavior B( in int p1, out int p2 ) {
   int c1;
   C    c2;
   B1   b1( p1, c2, c1 );
   B2   b2( c1, c2, p2 );
   void main( void ) {
      par { b1.main(); b2.main(); }
      }
   };
```

Fig. 1. SpecC example.

IV. SpecC Simulation API

SpecC defines an application-programming interface (API) for simulation. The goal of the simulation API is to separate simulator implementation from compiler implementation. In this way, C++ code produced by any SpecC compiler can be linked with any SpecC API compliant simulation library to produce an executable. In addition, a wrapper library can be developed to wrap around existing simulation library, such as SystemC, or Cynlib library, to make them work with SpecC.

The design of the simulation API is a matter of dividing the work of implementing SpecC semantics between SpecC compiler and SpecC simulator.

On the one extreme, one can choose to let the SpecC compiler to generate all the low level code to implement the SpecC semantics, thus effectively eliminating the need for the simulation library. This approach not only leaves a heavy burden to the compiler implementation, but also makes the system fairly inflexible. For example, whenever a new algorithm is developed to improve simulation efficiency, one has to rewrite a large chunk of the compiler code. Given the complexity involved in the compiler development, this option is obviously impractical.

On the other extreme, one can embed as much functionality as possible in the simulation library, thus effectively leaving the job of compiler simply translating the SpecC constructs into corresponding API calls. Note that due to the limitation on the expressive power of the host language (C++), it is not always possible to find a concise correspondent of SpecC construct. Note that the library-based approaches face a similar limitation, although the problem is alleviated by leveraging the operator overloading construct of C++, to make the use of API as intuitive as possible.

Our criterion on where to draw the line is to make the simulation API as simple as possible, while leaving the compiler to perform the trivial yet tedious work of translation.

This decision results in an API that is surprisingly small. In fact, it consumes only 6 names in the C++ name space, as well as 10 functions. The names are reserved for a set of C++ classes. Note that the API does not prescribe how the classes are implemented, that is, it does not assume the availability of any data members or member functions for each of the class, except the availability of a constructor of predefined form. In fact, the definition of these classes is not needed until the SpecC generated C++ code is to be compiled into object code using a host compiler, such as gcc.

A. API Classes

Figure 2 lists the six C++ classes reserved by the API. `behavior` is the super class of all translated SpecC behaviors. `channel` is the super class of all translated SpecC channels. `event` correspond the the SpecC event data type. `fork` is the class used to capture the information of a concurrent behavior. The `fork` constructor takes a behavior object as its argument. `try_block` and `exception_block` are the classes used to capture information of exception and interrupt handling. The `try_block` constructor takes a behavior object as its argument. The `exception_block` constructor takes a boolean flag indicating whether it is an exception or interrupt, a behavior object, and a list of events as its arguments.

```
class behavior;
class channel;
class event;
class fork;
class try_block;
class exception_block;
```

Fig. 2. API classes.

B. API Functions

Figure 3 lists the set of C++ functions defined in the API. Functions `start` and `end` give the simulation library a chance to initialize and finalize its private data structures. Function `abort` is used to abort the simulation in case of failure. Function `par` accepts a list of `fork` objects and will execute the corresponding subbehaviors in parallel. Function `pipe` accepts a list of `fork` objects and will execute the corresponding subbehaviors in pipelined fashion. Function `tryTrapInterrupt` accepts a `try_block` object as well as a list of `exception_block` object as its arguments. It will execute the behavior captured by the `try_block` object, while monitoring events captured by the `exception_block` objects. If a relevant event is notified during the execution, the corresponding exception handling behavior or interrupt handling behaviors will be executed. Function `waitfor` takes a number `delay` as an argument and will suspend the execution of the calling behavior until `delay` time units later. Function `wait` takes a set of events as arguments. It will suspend the execution of the calling behavior until any of the events are notified. Functions `notify` and `notifyone` take a list of events as their arguments. When called, `notify` will awake all the behaviors that are "waiting" on the corresponding events. On the other hand, `notifyone` will awake only one of the behavior that are "waiting" on the corresponding events.

```
void start( void );
void end( void );
void abort( const char* formatString, ... );
void par( fork* first, ... );
void pipe( fork* first, ... );
void tryTrapInterrupt(
    try_block *t, exception_block *f, ...
    );
void waitfor( long long delay );
void wait( event* first, ... );
void notify( event *first, ... );
void notifyone( event *first, ... );
```

Fig. 3. API functions.

V. Compiling SpecC for Simulation

SpecC compiler translates a SpecC program into a C++ program which contains the use of SpecC APIs.

A. Compiling SpecC classes

SpecC extends the C language with a set of SpecC classes, namely, behaviors, channels and interfaces. The SpecC compiler translates SpecC classes into corresponding C++ classes. The translated behavior inherits from class `behavior` and the translated channel inherits from class `channel`. The members of the SpecC class becomes the members of C++ class without modification.

The ports of SpecC classes become the C++ class data members. Furthermore, the types of the ports are converted into corresponding reference types.

The constructor of the translated C++ classes will be synthesized so that it takes a list of arguments with the same signatures as the port list. The constructor will automatically make sure all the data members inferred from the ports reference the corresponding arguments of the constructor.

For example, in Figure 6, behavior `X` has two ports `e1` and `e2` with type `event`. The translated C++ code makes `e1` and `e2` as data members with type `event&`. Also note the creation of the constructor `X`.

B. Compiling Basic Data Types

SpecC extends C language also with a new data type `event`. The API defines a C++ class `event`, which correspond directly the the SpecC data type. SpecC directly translates reference to data type `event` into the C++ `event` type.

C. Compiling Statements

SpecC extends C language with a set of new statements. Some statements are translated into calls to the corresponding API functions. For example, the SpecC `waitfor`, `wait`, `notify` and `notifyone` statements are directly translated into corresponding API calls.

Some statements are synthesized into low level C++ code without the use of API calls.

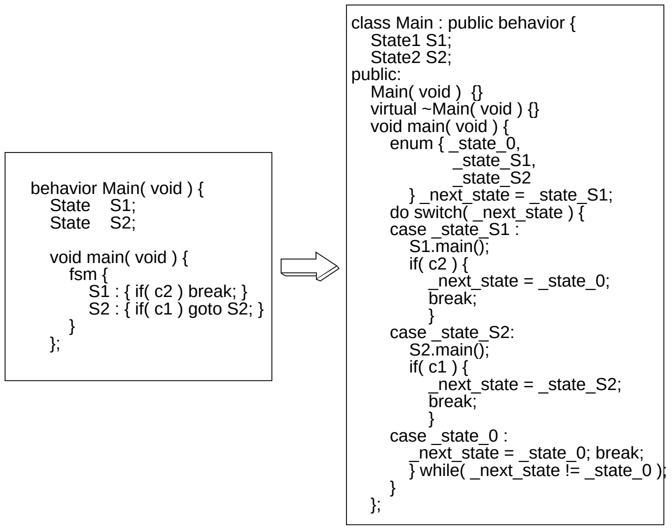
Figure 4 shows how a `fsm` statement is translated into low level C++ code. Note that variable `_next_state` is first synthesized with a synthesized enumerate type. A `switch` statement is then synthesized to branch upon the runtime value of `_next_state`. Each state in the `fsm` correspond to a `case` statement. The code for executing the state is automatically synthesized. The state transitions are synthesized into assignments to `_next_state`.

Another example is the SpecC timing statements, which are synthesized into low level C++ code using a real-time scheduling algorithm.

Most others involve both code synthesis and API call.

Figure 5 shows how a `par` statement is translated into C++ code. Note that for each concurrent behavior, a `fork` object is automatically synthesized and passed as argument to the `par` function. Since the `fork` object is a local variable, the creation and finalization of associated data structure is automatically managed by the C++ compiler using the constructor/destructor mechanism. The `pipe` statements are compiled exactly as the `par` statements except the name of the API call.

Figure 6 shows how a `try-trap-interrupt` statement is translated into C++ code. Note that a `try_block` object as well as two `exception_block` objects are synthesized for the monitored, exception handling, and interrupt handling behaviors respectively. The synthesized objects are passed as arguments to the `tryTrapInterrupt` API call.

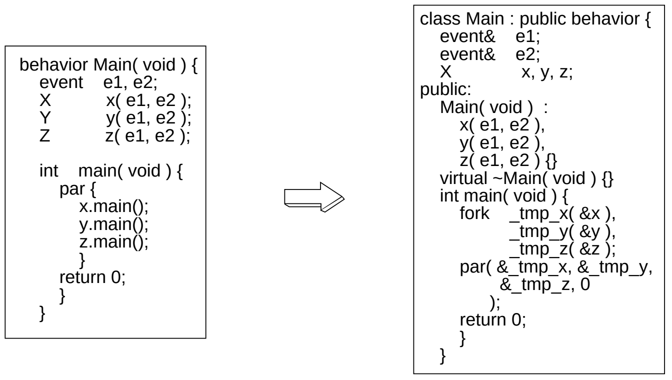


```
behavior Main( void ) {
    State    S1;
    State    S2;

    void main( void ) {
        fsm {
            S1 : { if( c2 ) break; }
            S2 : { if( c1 ) goto S2; }
        }
    };
```

```
class Main : public behavior {
    State1 S1;
    State2 S2;
public:
    Main( void )  {}
    virtual ~Main( void ) {}
    void main( void ) {
        enum { _state_0,
                    _state_S1,
                    _state_S2
            } _next_state = _state_S1;
        do switch( _next_state ) {
        case _state_S1 :
            S1.main();
            if( c2 ) {
                _next_state = _state_0;
                break;
                }
        case _state_S2:
            S2.main();
            if( c1 ) {
                _next_state = _state_S2;
                break;
                }
        case _state_0 :
            _next_state = _state_0; break;
            } while( _next_state != _state_0 );
        }
    };
```

Fig. 4. Compiling `fsm` statement.



```
behavior Main( void ) {
    event    e1, e2;
    X          x( e1, e2 );
    Y          y( e1, e2 );
    Z          z( e1, e2 );

    int    main( void ) {
        par {
            x.main();
            y.main();
            z.main();
            }
        return 0;
        }
    }
```

```
class Main : public behavior {
    event&    e1;
    event&    e2;
    X              x, y, z;
public:
    Main( void )  :
        x( e1, e2 ),
        y( e1, e2 ),
        z( e1, e2 ) {}
    virtual ~Main( void ) {}
    int main( void ) {
        fork    _tmp_x( &x ),
                _tmp_y( &y ),
                _tmp_z( &z );
        par( &_tmp_x, &_tmp_y,
                &_tmp_z, 0
              );
        return 0;
        }
    }
```

Fig. 5. Compiling `par` statement.

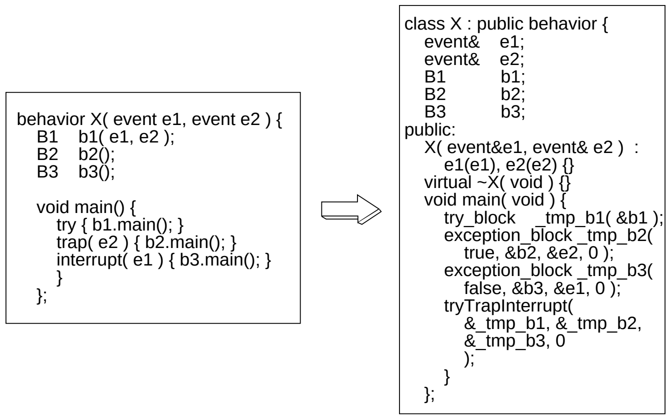
VI. SIMULATION LIBRARY REFERENCE DESIGN

We have developed a reference design, called SpecSim, for the simulation library. A detailed user's and programmer's manual can be found at [8]. As its name suggests, the purpose of the reference design is to demonstrate the implementation of the SpecC system. Performance optimization, on the other hand, is secondary. Nevertheless, design decisions are carefully made so that the simulator is reasonably simple, fast and flexible enough for for future improvement.

To achieve the goal, we have divided the implementation into several abstraction layers —- a strategy that has been successfully applied in many other areas of computer science, for example, operating system design and protocol design. While the third layer, namely the SpecC simulation API as described in Section IV, is realized by a set of C++ classes, the first and second layers are implemented by a set of COM interfaces [9].

A. Thread Layer

At the lowest level is the thread layer, implemented by three COM interface: `INative, IThread, IScheduler`. The purpose of this layer is to abstract away the implemen-



```
behavior X( event e1, event e2 ) {
    B1    b1( e1, e2 );
    B2    b2();
    B3    b3();

    void main() {
        try { b1.main(); }
        trap( e2 ) { b2.main(); }
        interrupt( e1 ) { b3.main(); }
        }
    };
```

```
class X : public behavior {
    event&    e1;
    event&    e2;
    B1        b1;
    B2        b2;
    B3        b3;
public:
    X( event&e1, event& e2 )  :
        e1(e1), e2(e2) {}
    virtual ~X( void ) {}
    void main( void ) {
        try_block    _tmp_b1( &b1 );
        exception_block _tmp_b2(
            true, &b2, &e2, 0 );
        exception_block _tmp_b3(
            false, &b3, &e1, 0 );
        tryTrapInterrupt(
            &_tmp_b1, &_tmp_b2,
            &_tmp_b3, 0
            );
        }
    };
```

Fig. 6. Compiling `try-trap-interrupt` statement.

tation detail of different multi-threading packages we might use, for example, the POSIX `pthread` package [10], the QuickThread package implemented by University of Washington [11]. While sharing a similar functionality, these packages differ in both performance and portability. Since choices of the multi-threading package to use on different platforms may vary, it is best that we establish an multi-threading API that can wrap around different packages. Furthermore, since the functionality we need and only need from the thread layer is a simple abstraction of execution context as well as the mechanism to perform non-preemptive context switch, we can use this API to mask away the other functionalities the package may provide, or to create the abstraction using the package's native primitives.

B. Discrete Event Layer

At the second layer is the discrete event layer, implemented by the COM interface `IDeEngine`. This layer implements the discrete event semantics of SpecC defined in [2].

At the heart of the discrete event layer is a scheduler which maintains a priority queue, called the timing wheel. The timing wheel contains a set of timed `Events` sorted by their time stamps. Each event contains a set of `Threads` and `Closures` (atomic functions) suspended for execution for a delay equal to the time stamp of the event. The interface functions such as `waitfor`, `wait`, `notify` adds threads and closures to existing events, or newly generated timed events, and inserting timed events to the timing wheel.

The scheduler is consulted whenever the native thread finds no thread to execute (deadlocked), in which case the scheduler will remove the event with the smallest time stamp, thereby advancing the simulation time. Execution will be resumed immediately for the `Closures` and the `Threads` contained in the removed events, thereby generating more events. This process is repeated until the timing wheel is empty or the simulation is aborted.

C. API Layer

The third layer directly implements the SpecC simulation API.

This involves the implementation of the classes defined in the API except class `event`, as it is already implemented in the second layer. The implementation of these classes involves only trivial bookkeeping tasks since their only function is to record the information necessary for the API functions.

Since the second layer provides the bulk of the simulation semantics, the implementation of the API functions are not hard either and usually involves only thread management. For example, the `par` function basically suspends the current thread, called the parent thread, and resume the set of threads, called the child threads, which are maintained by the `fork` objects it receives. It then resumes the parent thread when all the child threads finish execution. The `pipe` function implements the same fork-join behavior as the `par` function except it puts a loop around it. Furthermore, the pipeline flushing behavior is implemented. Similarly, the `tryTrapInterrupt` function starts the handler threads and let them wait on the corresponding events. It then starts the monitored thread. When awaked, the handler thread either aborts the monitored thread in case of exception handling, or suspend the monitored thread in case of interrupt handling. The functions `wait` and `waitfor` delegate their implementation to the scheduler defined in the second layer. The functions `notify` and `notifyone` move all or one threads in the events that they receive as arguments from the `Suspended` list to the `Ready` list. The functions `start` and `end` create and destroy the `DEScheduler` object.

VII. Implementation and Experiments

We have implemented the the reference simulation library as described and made an open source release at [8]. It has been ported to most commonly used platforms such as Sun Solaris and Linux x86 based on different thread packages such as pthread and quick thread. With the open source license, we expect this implementation can be evaluated, used, ported and enhanced by the community in an efficient way.

To test the robustness of the implementation, we have developed a comprehensive set of test programs to exercise hopefully every corner of the SpecC functionality. Furthermore, we have developed and simulated a number of real-life applications, some of which are listed in Table I. The types of applications range from embedded controller to high performance DSPs. The complexities of these applications, as shown in the second column of Table I, also give us a fair amount of confidence in the expressive power as well as the implementation robustness of the SpecC system.

The total simulation time of each application is not shown since many of the them are reactive systems, which by definition should interact with its environment in a continuous fashion. However, many of the examples are included in the SpecC release for those who are interested in improving or re-implementing the simulation library and would like to benchmark the simulation performance.

Application	Complexity (#lines)
Traffic light controller	1485
Elevator controller	2035
Forward Error Control Code	1227
FIFO	679
JPEG	1464
JBIG	3106
GSM vocoder	13000

TABLE I
SIMULATED SPECC SPECIFICATIONS.

VIII. CONCLUSION

The complexity involved in System-On-Chip design calls for the use of IPs and an IP-centric design methodology. It is critical for such a methodology to use a language to help the design specification and design exploration of an end application. It has becoming a consensus that such language should be C/C++ based and the SpecC language is one of the candidate solution. One critical issue in the SLDL based design environment is the verification methodology. This paper presents the implementation of the SpecC simulator, which helps both the design specification verification and design implementation verification by fast simulation. Our implementation strikes a balance between compiler complexity and simulator complexity, and is considerably smaller than alternative solutions. Our future work extends to the following directions: First, porting of the simulation library to more architectural platforms and more multi-threading packages need to be performed. Second, interfacing with other open-source C-based simulation libraries, such as SystemC and CynLib, should be performed. Third, simulation performance needs to be performed by the use of more efficient data structures.

REFERENCES

[1] D. Gajski, J. Zhu, D. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, March 2000.

[2] J. Zhu, R. Doemer, and D. Gajski, "Syntax and semantics of SpecC+ language," in *Proceedings of the Ninth Workshop on Synthesis and System Integration of Mixed Technologies*, Japan, December 1997.

[3] *CynLib Web Site*, `http://www.cynapps.com/CynApps/products/cynlib/opensource.html`.

[4] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design environment," in *Proceeding of the 34th Design Automation Conference*, 1997.

[5] *SystemC Web Site*, `http://www.systemc.org`.

[6] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens, "Hardware reuse at the behavioral level," in *Proceeding of the 36th Design Automation Conference*, New Orleans, June 1999.

[7] *SpecC Technology Open Consortium Web Site*, `http://www.specc.org`.

[8] *SpecSim Web Site*, `http://www.eecg.toronto.edu/˜jzhu/releases/specsim`.

[9] *COM Web Site*, `http://www.microsoft.com/com`.

[10] B. Lewis and D. J. Berg, *Multithread Programming With Pthreads*, Prentice Hall Computer Books, December 1997.

[11] D. Keppel, "Tools and techniques for writing fast portable threads packages," Tech. Rep. UW-CS E-93-05-06, University of Washington, 1993.