The Standard SpecC Language

Masahiro Fujita
Department of Electronic Engineering
The University of Tokyo
3-8-1, Hongo, Bunkyo, Tokyo, 113-8656, JAPAN
+81-3-5841-6673
fujita@ee.t.u-tokyo.ac.jp

Hiroshi Nakamura
Research Center for Advanced Science and Technology
The University of Tokyo
4-6-1, Komaba, Meguro, Tokyo 153-8904, JAPAN
+81-3-5452-5162
nakamura@rcast.u-tokyo.ac.jp

ABSTRACT

This paper introduces SpecC language, a system level description language based on C, and its consortium, SpecC Technology Open Consortium (STOC). Currently SpecC language version 1.0 is publicly available. SpecC technology covers SpecC-based design "methodology" as well as SpecC language itself. In this paper not only SpecC language but also SpecC-based design methodology are briefly discussed. The SpecC language specification working group (LSWG) under STOC is discussing on SpecC version 2.0. We also give a summary of the discussions being made by LSWG targeting version 2.0. We plan to formally release version 2.0 in the beginning of 2002. The main goal is to precisely and exactly define the formal semantics of SpecC language especially on the semantics relating to parallel and concurrent statements and event control mechanisms. These are the issues on which SpecC version 1.0 does not give clear and concise semantics. With these clarifications given by SpecC version 2.0, varieties of supporting tools for SpecC can consistently and easily be developed.

Keywords: Hardware Description Language, System Level Design, C-based Hardware Description, Formal Semantics, System Synthesis, High-Level Synthesis, Formal Verification

1. Introduction and the SpecC design methodology

As semiconductor technology advances, entire systems can be realized within single LSIs as System-on-a-Chip (SoC). Designing a SoC is a process of entire system design from specification to implementation design and also a process of both hardware and software development. Performance of the designed systems fully depends on both hardware (LSI) and software on top of LSI. Correct partitioning between software execution and hardware execution must be taken for high performance with low implementation cost, and integrated specification for both software and hardware is indispensable. Therefore, in order to design SoC, specification process and implementation design process must be smoothly and tightly coupled.

Figure 1 shows a design flow for SoC. Systems to be designed have both software and hardware, and IP in various design levels should be tried to be used as much as possible in order to shorten the design costs. In the right most part of the figure, the design levels that are covered by various languages are shown. SpecC is covering design levels from specification to behaviors. It can describe both software and hardware seamlessly and a good tool for rapid prototyping as well.

Two key elements to solve the integration of specification and design phases in the System-on-a-Chip (SoC) design process are: 1) a consistent and continuous design process from specification design to implementation design which covers both of software and hardware; and 2) design reuse, which means not only component IP but also specification IP and design IP. Over the years, many languages and data formats have been proposed. Recently, C/C++ based hardware (LSI) design methodologies are emerging. However, the current approaches do not efficiently cover these two key elements. Such approaches include UML, which is primarily used for specification design, and C/C++ and VHDL and/or Verilog, which are primarily used for implementation design.

SpecC-based design methodology is shown in Figure 2. SpecC covers specification model, architecture model, communication model, and implementation model with easy access and use of IP in corresponding design levels. Designers start their design processes with specification in SpecC. Then appropriate architectures that realize specifications are explored, and corresponding communications among components in architecture models are generated. Finally software models and hardware models as implementation models are the output of the SpecC design methodology.

In the following, we briefly review SpecC language version 1.0, which is currently available, and version 2.0 which is under discussion and will be available to the public by early next year. In section 2, we present key ideas of SpecC language version 1.0. Then in section 3, we give overview of SpecC Technology Open Consortium (STOC), which is an organization for promotion of SpecC language. In section 4, we give summary of the discussion points targeting SpecC version 2.0. Section 5 gives concluding remarks.

2. SpecC Language

SpecC methodology and language have been designed and implemented to integrate the specification and the design phases in the SOC design process. Originally developed at University of California, Irvine, with sponsorship from several companies,

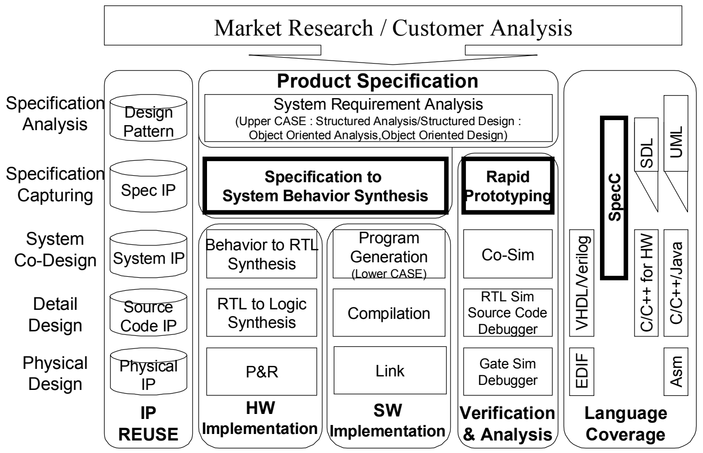


Figure 1. SOC Design Flow

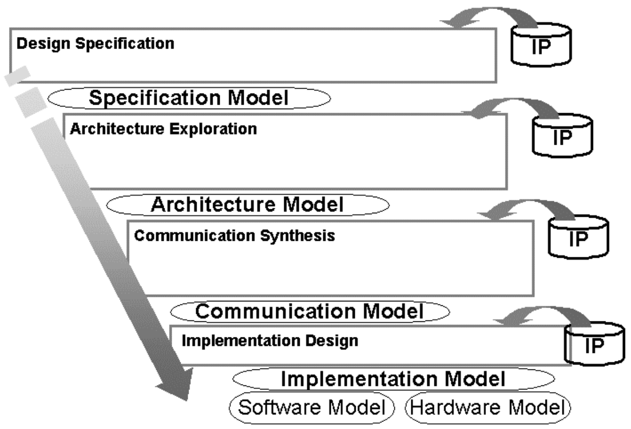


Figure 2. SpecC-based design methodology

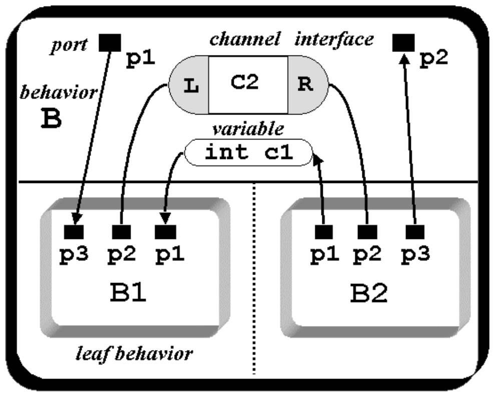


Figure 4. Communication between behaviors through channels

SpecC language is a system specification description language based on C. It allows the same semantics and syntax to be used to represent specifications for a system concept, hardware, software, and, most importantly, intermediate specification and information during hardware/software co-design stages.

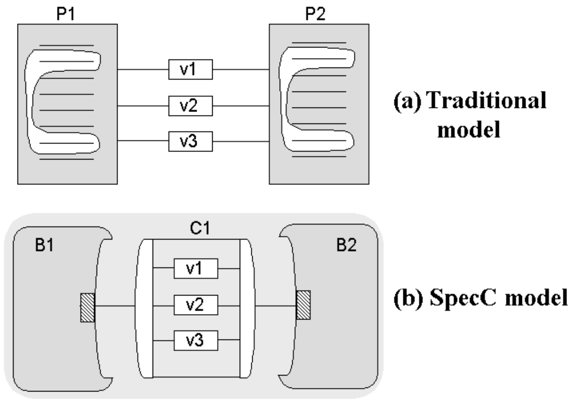


Figure 3. Communication through channels

Right now SpecC version 1.0 is publicly available. One key point in SpecC is the clear separation between communication and computation bodies in system level descriptions. With this clear separation, same descriptions can easily be used both for software and hardware (or both combined) development. As shown in Figure 3, in traditional approaches, communication among concurrent processes are just through shared variables, and control on the data transfer between the two processes are done by the statements which are in lined into the two process descriptions. Therefore, it is not easy or almost impossible to separate communication and the computation. In SpecC model, communications among processes are done through channels and control mechanisms for communication are described explicitly in the description of channels. This makes it very easy to explicit to separate the communication from the computation.

Structure hierarchy can also be described in SpecC as shown in Figure 4. In hierarchical designs, by using channels for communications, it can be easily seen how things are processed within a module in a hierarchical design as shown in the figure.

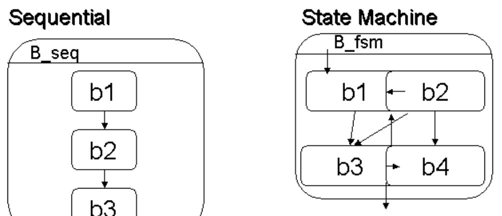
Also, SpecC has several ways to describe targeted control mechanisms:

(1) Sequential descriptions just like regular C

(2) Specialized syntax for finite state machine descriptions

(3) Explicit way to describe “parallel behaviors”

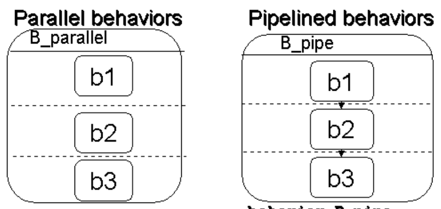
(4) Explicit way to describe “pipelined behaviors”



```
behavior B_seq
{
 B b1, b2, b3;
void main(void)
{
 b1.main();
 b2.main();
 b3.main();
 }
};
```

```
behavior B_fsm
{
 B b1, b2, b3, b4;
void main(void)
{
 fsm{ b1:{...}
      b2:{...}
      b3:{...}
      b4:{...}
    }
 }
};
```

Figure 5. Sequential and state machine description



```
behavior B_parallel
{
 B b1, b2, b3;
void main(void)
{
 par { b1.main();
       b2.main();
       b3.main();
     }
 }
};
```

```
behavior B_pipe
{
 B b1, b2, b3;
void main(void)
{
 pipe{ b1.main();
       b2.main();
       b3.main();
     }
 }
};
```

Figure 6. Parallel and pipelined behaviors

Figure 5 and 6 give ideas on sequential, FSM, parallel, and pipeline statements. Sequential statements are just like regular C description. By using FSM statements, explicit state transitions can be clearly described. Parallel statements explicitly describe parallel execution of multiple processes whereas pipeline statements give parallel execution of multiple processes in pipelined ways.

With these flexible descriptions mechanisms, system level description targeting both software/hardware combined systems can be smoothly described. Here due to space limit, we briefly review (2)-(4) with illustrations, since those are main difference from regular C languages in terms of language constructs.

3. SpecC Technology Open Consortium (STOC)

STOC was established on November 10, 1999 by 24 companies and organizations worldwide for securing wide acceptance of a SpecC language-based methodology. Activities of STOC will be publicized at the consortium's web site (http://www.specc.gr.jp/eng), where the introduction of SpecC technology, the consortium's activities, and research results are available. Activities of the STOC include two technical working groups (WG).

- The Case Study WG that examines the ability of SpecC language through descriptions of real applications, and collects design knowledge to establish design guidelines or methodologies. According to the current schedule, a period of the WG activities is one year, and 14 members will exchange evaluation results of SpecC language ability and design know-how based on the specification description experiences.
- The Language Specification WG that maintains and establishes the specification of SpecC language while evaluating and enhancing the current specification to improve its efficiency and application scope. Members are recruited from worldwide STOC members, and the activity scope also includes standardization of relevant tools like SpecC Compiler.

Objectives and current activities of both WGs also can be tracked from the STOC web site.

4. Discussions for version 2.0

After releasing version 1.0 in early February 2001, LSWG has identified several issues to be clarified in the language. SpecC is under revision mainly on the following points:

- Clarification of detailed semantics of SpecC language
- Enhancement of usability by incorporating several syntax notations
- Others, such as tool friendly issues

Here we briefly summarize the discussions on detailed language semantics targeting SpecC version 2.0. Since this is under discussion, anything presented here could change in the future.

4.1 Why Semantics?

SpecC is a system level description language and a wide variety of designers are expected to use SpecC, including hardware designers and software designers. Since ways of thinking of hardware designers are sometimes significantly different from those of software designers, the semantics of SpecC should be clearly defined from the viewpoints of both hardware designers and software designers. The importance of formal semantics is emphasized also by the fact that varieties of design assistance will be required for system level design. Specifications in SpecC will be the input of not only simulation but also synthesis, verification, and others. As for the synthesis, it would be a case that the description is partitioned into hardware and software parts, and the former is then synthesized into RTL hardware. It would be another case of synthesis that they are bound with IP cores with modifying communications between cores. Thus, a wide variety of synthesis tools will emerge for system level design assistance. The same situation will occur in simulation and verification. Therefore, the semantics of SpecC should be defined independently from their execution engines.

The reference manual of SpecC Version 1.0 (LRM v1.0) [1] and its reference compiler [2] were already published and announced. Since the formal semantics is yet one of the most important issues, SpecC Language Specification Working Group [3] is trying to formalize its semantics, which will be included in the next version 2.0

4.2 Discussions items

One of the characteristics in SpecC is the separation of computation and communication. The communication can be specified by using either explicit channel or shared variables. Whereas the semantics of explicit channel is quite clear, that of shared variables currently contains ambiguity. This is because the semantics of parallel behaviors is not well defined. Thus, our discussion started from the semantics of "par" statement, which specifies parallel behaviors. SpecC also provides "pipe" statement to specify pipelined behaviors, which is of course a part of parallel behaviors. Though the semantics of "pipe" is also important, we focus on the semantics of "par" at first because "pipe" can be defined by using "par". Once the semantics of "par" is defined clearly, that of "pipe" will be clear.

Concerning to "par", the following items are picked up for the discussion.

- What order is permitted in the scheduling? : Is the scheduling non-preemptive or preemptive? Is it deterministic or not? If it is non-deterministic, then what degree of non-determinism is permitted?
- How to assure mutual exclusive access? : In SpecC, "wait/notify" is provided to support synchronization. However, no primitives are provided to support mutual exclusion. It will be helpful to introduce a primitive supporting for mutual exclusion which is well suited to "wait/notify".
- Semantics of synchronization primitives? : Synchronization of concurrent behaviors is a critical issue. However, some of the rules in LRM v1.0 are vague and the semantics can only be fully understood with the help of the examples and the *note* sections in LRM v1.0. The vague point is the propagation scope of notified event and the relationship between the scope and the "simulation time" in SpecC.
- How to disable exceptions temporarily? : In order to handle asynchronous exceptions correctly, it is required to disable other exceptions during the execution. Thus, it will be helpful to introduce some primitives that disable exceptions temporarily.
- When is variable assignment reflected to other concurrent behaviors? : This issue is closely related to the first issue. Without clear definition on the timing of variable assignments, it is hard to specify concurrent behaviors that share variables. Thus, it is highly recommended to establish well-defined semantics on this issue.

Since the first item of scheduling is the most fundamental among these issues and affects the discussions on the other items, we first started the discussion on scheduling.

4.3 Semantics of "par"

Figure 7 is an example of parallel behavior. In this example, behavior `a` and `b` are executed in parallel. Behavior `a` contains two sequential statements `st1` and `st2`, whereas behavior `b` contains one statement `st3`. The first question is, in which order these three statements are executed? Given the LRM v1.0, the scheduling is non-preemptive. Then, not only non-preemptive scheduling of "`st1 -> st2 -> st3`" and "`st3 -> st1 -> st2`", but also preemptive scheduling of "`st1 -> st3 -> st2`" are permitted.

```
main(){
par{  a.main();
      b.main();} }

behavior a{
main(){ z=y;      /*st1*/
        x=z+20;  /*st2*/ }}

behavior b{
main(){ y=x+z+1; /*st3*/ }}
```

Figure 7. Example of "par" statement

4.3.1 Sequentiality

Before clarifying the concurrency between statements, we have to define the semantics of sequentiality within a behavior. The definition is as follows. A behavior is defined on a time interval. Sequential statements in a behavior are also defined on time intervals which do not overlap one another and are within the behavior's interval.

For example, semantics of behavior `a` in Figure 7 is defined on time axis as shown in Figure 8. Suppose the beginning time and the ending time of behavior `a` are *Tas* and *Tae* respectively, and those for `st1` and `st2` are *T1s*, *T1e*, *T2s*, and *T2e*. Then, the only constraint which must be satisfied is;

$$Tas <= T1s < T1e <= T2s < T2e <= Tae$$

Statements in a behavior are executed sequentially but not always in continuous ways. That is, a gap may exist between *Tas* and *T1s*, *T1e* and *T2s*, and *T2e* and *Tae*. The lengths of these gaps are decided in non-deterministic way. Moreover, the lengths of intervals, (*T1e – T1s*) and (*T2e – T2s*) in Figure 8, are also non-deterministic.

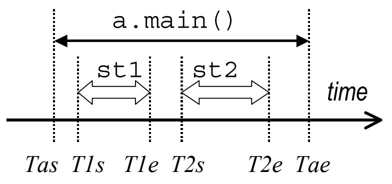


Figure 8. Semantics of Sequentiality

4.3.2 Concurrency

Behaviors invoked by "par" statement are executed concurrently. The definition of the concurrency is as follows. The beginning time of all the behaviors invoked by "par" statement are the same, and the ending time of all the behaviors invoked by "par" statement are also the same. For example, semantics of "par" statement in Figure 7 is defined on time axis as shown in Figure 9. Suppose the beginning time and the ending time of behavior a are *Tas* and *Tae* respectively, and those for behavior b are *Tbs*, *Tbe*. Then, the only constraint which must be satisfied is;

Tas = *Tbs*, *Tae* = *Tbe*

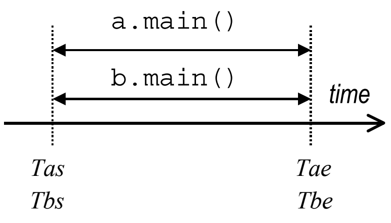


Figure 9. Semantics of Concurrency

Once the sequentiality and concurrency are defined, the semantics of the description in Figure 7 is clearly defined as illustrated in Figure 10. The followings are all the constraints to be satisfied.

- Tas <= T1s < T1e <= T2s < T2e <= Tae (sequentiality in a)
- *Tbs<= T3s < T3e <= Tbe* (sequentiality in b)
- *Tas = Tbs, Tae = Tbe* (concurrency between a and b)

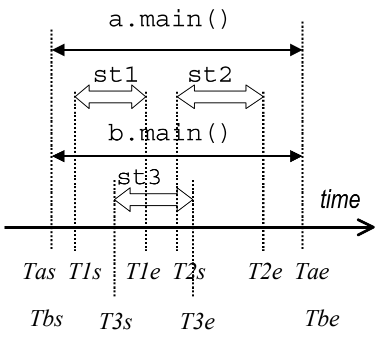


Figure 10. Scheduling for the example of Figure 7.

Note that there are no deterministic rules on the lengths of st3 st1, and st2, and on the lengths of the gap between statements, st3 may overlap with st1 and/or st2, or may not overlap with st1 or st2.

4.3.3 Atomicity

The next question is whether the overlaps between statements are permitted or not. The answer depends on the granularity of atomicity. In the LRM v1.0, atomicity is not described explicitly.

One policy is that each assignment statement is atomic. Under this policy, overlaps between statements are not permitted. Then, the statement of "y=x+x", for example, is always the same as "y=2*x" because x is not updated by any other statement during the execution of this statement. Thus, this policy will reduce the degree of non-determinism. On the other hand, this assumption is completely different from C language although SpecC is extended from C language, and thus it would be confusing. For instance, it is quite ambiguous what will happen if a function call is included in an assignment statement. Another disadvantage of this policy is that quality of synthesized hardware and/or software may be degraded because this atomicity should be guaranteed by hardware and/or software.

It would be another policy that no atomicity is assumed, which is opposite to the above policy.

We are also discussing this issue and going to reach the consensus that no atomicity is guaranteed in any operation, even in the memory access. Thus, statements may be preempted at any time in their execution. Due to this feature, overlaps between statements are permitted. Another result from this feature is "y=x+x" is not always the same as "y=2*x". This is because the first read of x may not be executed at the same time as the second read of x and because x may be updated by other parallel behaviors between the two reads. Therefore, if designers would like to use shared variables in a safe way, they should use those variables with explicit synchronization.

4.3.4 Relationship with Simulation Time

SpecC has two primitives to support the specification of timing called "simulation time": "waitfor" and "do-timing". Given the LRM v1.0, "waitfor" statement specifies execution time (or delay). Whenever the simulator reaches a waitfor statement, the execution of the current behavior is suspended for the specified amount of simulation time units. The do-timing construct is used to specify timing constraints in terms of minimum and maximum number of time units. The LRM v1.0 says the do-timing construct specifies synthesis constraints and the way that the simulator performs the constraint validation is implementation dependent.

In order to make the semantics of sequentiality and concurrency be sound with these primitives, the relationship between the length of each interval and the "simulation time" must be defined soundly. The definition is that the length of each interval on which a statement is defined is quite small and infinitely close to 0 in "simulation time"'. In other words, execution of each statement does not change the "simulation time". Going back to Figure 8, this definition is intuitively described as " (*T1e – T1s*) and (*T2e – T2s*), the lengths of statements' intervals, are infinitely close to 0". Note that this definition does allow that (*T1s– Tas*), (*T2s – T1e*), and/or (*Tae – T2e*), the lengths of gaps, have non-zero value.

```
main(){
  par{ a.main();
       b.main();} }

behavior a{
  main(){ z=y;         /*st1*/
          waitfor(2); /*NEW*/
          x=z+20;      /*st2*/ }}

behavior b{
  main(){ y=x+z+1; /*st3*/ }}
```

Figure 11. Example with "waitfor"

Figure 11 is an example where waitfor(2) statement is inserted between st1 and st2 of Figure 7. This waitfor(2) increments "simulation time" by 2. According to the above rule and the semantics of sequentiality and concurrency, there are three candidates on the timing when st3 is executed as shown Figure 12. Note that the length of the interval st3 is infinitely close to 0 whereas the interval of the behavior a and b has the length of 2.

Now, LRM v1.0 gives another rule that says that active threads are executed without changing the "simulation time". Thus, `st3` must be executed immediately without changing the "simulation time" before `waitfor(2)` as shown in Figure 13. Thus, `st3` must precede `st2` in this example.

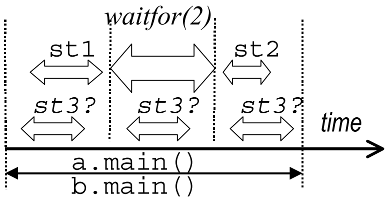


Figure 12. Candidates for Scheduling

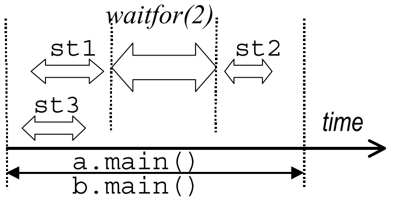


Figure 13. Scheduling for the example of Figure 6.

```
main(){
  par{ a.main();
       b.main();} }

behavior a{
  main(){ z=y;          /*st1*/
          waitfor(2);
          x=z+20;       /*st2*/
          notify e;     /*NEW*/}}

behavior b{
  main(){ wait e;       /*NEW*/}}
          y=x+z+1;      /*st3*/ }}
```

Figure 14. Example with "wait/notify"

Then, what will happen if `st3` is not an active thread?

In SpecC, "wait/notify" statements are used for synchronization. The semantics is that "wait" statement suspends the current thread from execution until one of the specified events is "notified". Since "wait" suspends a thread for a certain amount of "simulation time" unit, the next concern is how statements are scheduled if "wait" statements exist.

Suppose another example of Figure 14 where the synchronization statement of ``notify/wait" is inserted into Figure 11. In this example, "`wait e`" suspends `st3` until the specified event `e` is notified by "`notify e`". Here, "`notify e`" is scheduled only after the completion of st2 due to the sequentiality in behavior a. Thus, it is guaranteed that `st3` is scheduled after `st2`. Consequently, the example of Figure 14 is executed as shown in Figure 15. Note that the scheduling of Figure 15 is one of the candidates shown in Figure 12.

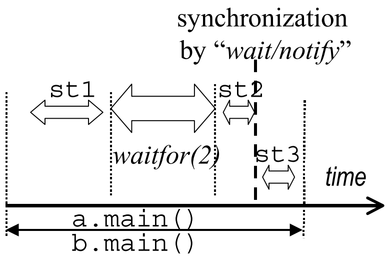


Figure 15. Scheduling for the example of Figure 8.

5. Concluding Remarks

In this paper we have reviewed the current status of SpecC languages emphasizing the issues being discussed by LSWG of STOC. We plan to formally release SpecC language version 2.0 in the beginning of 2002. Based on the formal semantics, many supporting tools for SpecC are expected to be available soon.

6. REFERENCES

[1] R. Doemer, A. Gerstlauer, and D. Gajski, "SpecC Language Reference Manual Version 1.0," http://www.specc.gr.jp/eng/tech/SpecC_LRM.pdf.

[2] http://www.ics.uci.edu/~specc/reference/

[3] http://www.specc.gr.jp/eng/wg_lang