# C/C++ Based System Design Flow Using SpecC, VCC and SystemC

Technical Report CECS_02_30
June, 14, 2002

Lukai Cai, Daniel D. Gajski
Center for Embedded Computer Systems
University of California
Irvine, CA 92697, USA
{lcai, gajski} @cecs.uci.edu

Mike Olivarez, Paul Kritzinger
Motorola  Semiconductor Products Sector
Wireless and Broadband Systems Group
Austin, Texas 78729, USA
{M.Olivarez, paul.kritzinger }@ motorola.com

# C/C++ Based System Design Flow Using SpecC, VCC and SystemC

Technical Report CECS-02-30
June, 14, 2002

Lukai Cai, Daniel D. Gajski
Center for Embedded Computer Systems
University of California
Irvine, CA 92697, USA
{lcai, gajski} @cecs.uci.edu

Mike Olivarez, Paul Kritzinger
Motorola Semiconductor Products Sector
Wireless and Broadband Systems Group
Austin, Texas 78729, USA
{M.Olivarez, paul.kritzinger }@motorola.com

**Abstract**

This report presents a C/C++ based system design flow that uses SpecC, VCC and SystemC tools. The design starts with a pure C model that is then converted into a SpecC model. A so-called behavior exploration task then takes place to analyze and optimize the system behavior. We then perform architectural exploration using VCC. Once this is complete, the behavior model is refined to an architecture model utilizing the SpecC methodology and the SpecC refinement tool. Finally, the design is linked to implementation using SystemC. We utilize this design flow to achieve the design from C to silicon in an efficient manner. An example of the JPEG encoder is utilized to prove this methodology.

## Index

## List of Figures

## List of Tables

# C/C++ Based System Design Flow Using SpecC, VCC and SystemC

Lukai Cai
*University of California, Irvine*

Mike Olivarez
*SPS,Motorola*

Paul Kritzinger
*SPS,Motorola*

Dan Gajski
*University of California, Irvine*

Abstract

*This report presents a C/C++ based system design flow that uses SpecC, VCC and SystemC tools. The design starts with a pure C model that is then converted into a SpecC model. A so-called behavior exploration task then takes place to analyze and optimize the system behavior. We then perform architectural exploration using VCC. Once this is complete, the behavior model is refined to an architecture model utilizing the SpecC methodology and the SpecC refinement tool. Finally, the design is linked to implementation using SystemC. We utilize this design flow to achieve the design from C to silicon in an efficient manner. An example of the JPEG encoder is utilized to prove this methodology.*

## 1 Introduction

System-level design issues are becoming increasingly critical as implementation technology involves more and more complex integrated circuits and software programs. In addition, time-to-market pressures are increasing. A potential solution to improve the time and quality of a complex system design is to make design decisions at higher levels of abstractions. This would also allow for the reuse of larger design components.

There appears to be an increasing trend towards the use of the C/C++ language as a basis for the next generation modeling tools and platform based design methodology to encompass design reuse. However, even with this convergence, industry is suffering the pain that there is no one tool and no one complete design methodology that can implement a top-down design methodology from C to silicon. Even more mature modeling environments such as VCC face significant hurdles in achieving a complete top-down flow [3][4][5].

In this report we suggest a C/C++ based top-down design flow from C to silicon, to make the system design smooth and efficient, by using currently available tools. We developed our design methodology by using SpecC [1][2], VCC [3][4][5], and SystemC [6][7][8]. Although there are some other tools and methodologies, such as SPADE [9], Ptolemy [10], and Polis [11], we choose SpecC, VCC and SystemC because they are all C-related and each has strong support in at least one field of design. We made the design flow based on our experiences of attempting to model the JPEG encoder with SpecC, SystemC and VCC, and one internal project, attempting to implement architecture exploration for MPEG encoding and decoding using VCC.

The report is organized as follows: section 2 introduces the design flow; section c) describes the behavior exploration; section4 describes the architecture exploration; in section 5 and section 6, model refinements are introduced. Finally, conclusions are given in section 7.

## 2 Top-down design flow

As shown in Figure 1, we describe the system on 3 different levels of abstraction:

*Behavior level*: system model at the behavior level only represents system functionality, without any timing information.

*Architecture level*: system at the architecture level is described with a set of interconnected components, each of which represents a computation component or a storage component of architecture. Wires representing system buses connect the ports of architecture components. The model of each architecture component is functionally correct with added abstract timing information.

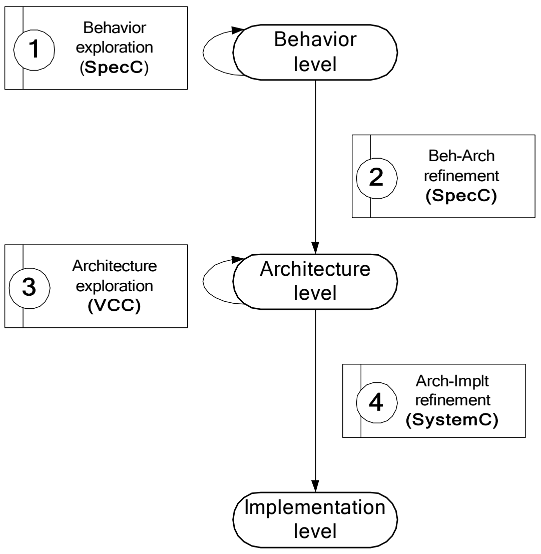
1
Behavior exploration (SpecC)
Behavior level
2
Beh-Arch refinement (SpecC)
3
Architecture exploration (VCC)
Architecture level
4
Arch-Implt refinement (SystemC)
Implementation level

Figure 1 Top-down design flow overview.

*Implementation level*: the implementation level describes computation components in terms of register transfers executed in each clock cycle for custom hardware or in terms of instruction sequence for software.

To start design from the behavior level and to end design at the implementation level, designers should accomplish four tasks.

a) *Behavior exploration:* it analyzes and optimizes the behavior model.
b) *Behavior-architecture refinement*: it refines the behavior model to the architecture model.
c) *Architecture exploration:* it allocates system components, maps the behavior to the architecture, and schedule different behavior blocks.
d) *Architecture-communication refinement*: it refines the architecture model to the implementation model.

Ideally, the entire design process should be accomplished by using one language to model three different levels of abstraction and using tools to do four tasks. However, currently there is no one tool that can go through the entire design flow efficiently. A variety of tools supporting various languages must be used to get the greatest return on tool capabilities and people's skills. Therefore, we use SpecC, VCC, and SystemC together for our design flow, which is shown in Figure 1 and Figure 2.

The SpecC methodology is a top down methodology. It provides four well-defined levels of abstraction (models). Among the four models, SpecC *specification model* is at the behavior level while SpecC *communication model* is at the architecture level. SpecC methodology also provides a well-defined method for moving down these successive levels. In terms of assisting in this process, SpecC today provides a profiler at the *specification model*, and a model refinement tool to help in the conversion from *specification model* to *communication model*. We use SpecC for behavior exploration and behavior-architecture refinement.

VCC [3][4][5] is a behavior/architecture co-design and design export tool. VCC models a system at the behavior level by the use of *whitebox C,* a tool-specific C based language. By mapping the behavior blocks to virtual architecture components saved in the VCC library, VCC estimates the system performance, which helps designers to select the suitable architecture and behavior-architecture mapping solution with the best performance. Therefore, we use VCC to implement architecture exploration.

SystemC is a C++ class library that can be used to create a cycle-accurate model for software algorithms, hardware architectures, and interfaces, related to system-level designs [6][7][8]. Compared with SpecC, it has more powerful support for the RTL model of hardware design. Moreover, SystemC co-simulation and synthesis tools can help to generate RTL level design model from the architecture level. Thus, we use SystemC for architecture-implementation refinement.

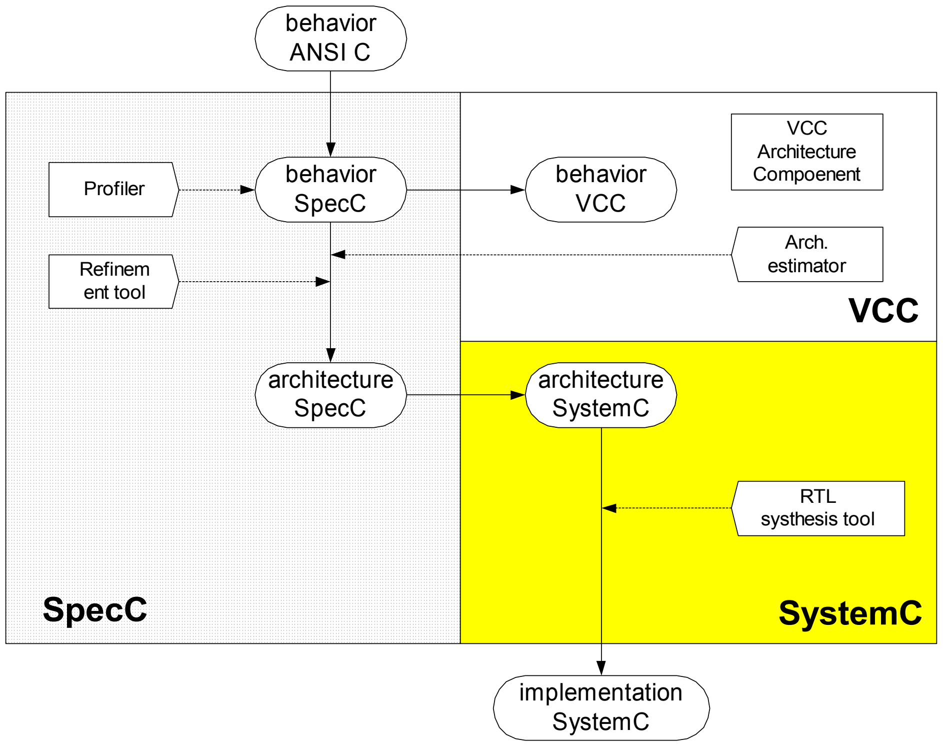

Figure 2: Detailed system design flow

Note that in the design flow, we model the system by four different languages: standard ANSI-C as input, SpecC as behavior model and architecture model, VCC as behavior model, and SystemC as architecture model and output implementation model. Converting ANSI-C behavior model to SpecC behavior model is relatively easy because SpecC is a C-based language. Converting SpecC architecture model to SystemC architecture model is also smooth because SpecC is C based and SystemC is C++ based, which also supports ANSI C. Furthermore, both of them support similar system design concepts, such as *event* and *wait*. However, we met difficulties when converting SpecC behavior model to VCC behavior model because VCC behavior modeling uses a graphical integration scheme. Thus, we generated a SpecC-VCC converter to automatically implement the necessary language conversion.

## 3 Behavior exploration

Current platform based design methodologies, such as VCC's, focus on performing behavior to architecture mapping and exploring various combinations of mapping [3][4]. This provides a limited opportunity for system performance optimizations. Purely architectural exploration has the disadvantage that system performance can only be estimated after architecture mapping has taken place. This limits the opportunity to see more global, inter-behavioral optimization opportunities.

We strongly believe that the system behavior and the interaction between behavior blocks should be completely understood and explored before behavior to architecture mapping and exploration takes place. An initial behavior based exploration, called behavior exploration, allows for better heuristics when selecting the behavior to architectural component mapping. For example, since system level design consists of many architectural components (processors, ASICs, and IP blocks), parallel and pipelined execution possibilities among behavior blocks at the behavior level needs to be exploited. However, when using pure platform based design, behavioral exploration is done within the constraints of the chosen architectural components. Because of this, opportunities for performance optimization would be missed by an exploration of different architectural mappings before performance estimation. (In VCC, performance profiling capability is provided by using a built in microprocessor model with associated compiler and profiling tool). These missed opportunities reduce functionality or speed capabilities and optimizations in the final design.

Behavior modeling is not a straightforward task because it must take into account later architecture exploration. Furthermore, behavior modeling for optimized system level design is different from the algorithm modeling associated with pure software design. We also believe a good top-down model for system level design should allow for performance estimation of the design is understood before and after the mapping process takes place.

In this section, we first discuss some confusion between behavior and architecture, and then we discuss the difference between a pure software model and a system level design model. Finally, we introduce the behavior exploration.

## 3.1 Sequential programming model vs. parallel programming model

ANSI-C programs consist of a number of functions. The executing sequence among function calls is sequential. Therefore, this is the sequential programming model.

In general, hardware language consists of a number of components executing in parallel, which can be called the parallel programming model.

One step of behavior exploration is the conversion from a sequential programming model to a parallel programming model.

## 3.2 Clean model and non-clean model

For the purposes of further discussion we need to define two terms:

a) *Clean computation*: Behaviors are defined hierarchically; each behavior can also contain a number of behavior instantiations of other behaviors. For example, in the C language, behaviors are represented by functions, and the behavior instances are represented by function calls. In a clean computation behavior, only two types of behaviors, leaf behavior and non-leaf behavior, are allowed. Leaf behavior contains a sequence of statements without any behavior instances. Non-leaf behavior contains only behavior instances without any statement execution. Figure 3(a) is a leaf behavior, Figure 3(b) is a non-leaf behavior. Figure 3(c) is a non-clean computation behavior.

b) *Clean communication*: In a clean communication model, parameters are passed by value among behavior instances. Figure 3(d) is a non-clean communication model; Figure 3(e) is a clean communication model.

If a model is both communication-clean and computation-clean, it is a clean model. Otherwise, it is a non-clean model. In general, C is a non-clean modeling language. Hardware languages are clean modeling languages. Thus, one step of behavior exploration is the transition of a model from non-clean to clean.

## 3.3 Behavior parallel, behavior pipeline vs. architecture parallel

Neither the concept of pipelining nor parallelism exists within the C language. However, to efficiently perform behavior modeling, system level design language must supports these concepts. Two terms are defined for this purpose:

a) *Behavior-parallel*: Two behaviors are defined as behavior-parallel if the execution sequence of the two behaviors does not influence the simulation result. Otherwise, the two behaviors are defined as behavior-sequential.
b) *Behavior-pipeline*: If, within a sequential programming model, a number of behaviors are executed one after another in a loop body, and behavior communicates only with the next behavior, then the execution relation between these behaviors can be termed as: behavior-pipeline.

```
void A() {
  int a;
  a = 0;
  a++;
  a--;
}
```

(a)

```
void B() {
  A();
  A();
}
```

(b)

```
void C (){
  int a;
  A();
  a++;
}
```

(c)

```
void D (int *d)
{
  int a;
  a = *d;
}
```

(d)

```
void E (int d)
{
  int a;
  a = d;
}
```

(e)

Figure 3: Examples of clean computation and communication in C language.

Another term, namely *architecture-parallel* can be defined as: if, during behavior to architecture mapping, behavior A and behavior B are mapped to different architecture components, then the implementation relation between A and B is called architecture-*parallel*. Otherwise it is called *architecture-sequential*.

During behavior-architecture mapping, if we map either a set of behavior-parallel or behavior-pipeline behaviors to different architecture components to form *architecture-parallel*, then we reach a *parallel matching*. *Parallel matching* is a necessary but not a sufficient condition of *parallel execution*.

## 3.4 Choosing system level design language (SLDL)

First, we need to choose a system level design language for behavior exploration. To understand behaviors by behavior exploration, the chosen SLDL must satisfy three conditions:

a) Ease and accuracy of profiling.
b) Ease of converting from original C language.
c) Ease of converting from one model to another when the execution sequence of behaviors changes.

We evaluated both the SpecC and SystemC languages for behavior exploration by using the JPEG encoder example. SpecC is considered the better language for this task.

### 3.4.1 Ease and accuracy of profiling

SpecC is a C syntax extension language. SystemC is C++ library extension. The SpecC behavior model can be profiled relatively easily. It is difficult to accurately profile the SystemC model because of the C++ class library burden, which does not allow for the splitting of system computational needs from the SystemC simulator's computational needs. Furthermore, SpecC provides a behavior profiler.

### 3.4.2 Ease of model converting

Since SpecC is an extension of the C-language, it inherently supports sequential modeling. In addition, SpecC has the added *par* and *pipe* keywords. This allows for the explicit definition and modeling of behavior-parallel, and behavior-pipeline execution. Converting a C model to a sequential SpecC model is simple because it only contains the syntax change, which is illustrated by the example in Table 1. Converting a sequential SpecC model to a parallel SpecC model is also simple because it only contains keyword (*par/pipe)* adding shown in Table 1. Similarly, by adding or deleting *par/pipe* keyword, designers can convert one model to another to explore different execution sequences among behaviors.

| Original C model | SpecC model (sequential execution) | SystemC model (sequential execution) |
|---|---|---|
| main(){<br>F1();<br>F2();<br>F3(); } | `main() {`<br>`F1.main();`<br>`F2.main();`<br>`F3.main(); }` | sc_main() {<br>F1 F2_inst(“inst1”);<br>F2 F2_inst(“inst2”);<br>F3 F3_inst(“inst3”); } |
| | SpecC model (parallel execution) | SystemC model (parallel execution) |
| | main() {<br>`par {`<br>F1.main();<br>F2.main();<br>}<br>F3.main(); } | sc_main() {<br>F1 F2_inst(“inst1”);<br>F2 F2_inst(“inst2”);<br>F3 F3_inst(“inst3”); } |

Table 1: Model example with SpecC and SystemC

On the other hand, the SystemC language only supports parallel programming. Since SystemC *processes* are executed in a parallel fashion, it does not support explicit sequential execution. The executing sequences of behaviors are determined by a signal-trigger mechanism. This introduces problems when converting a C model to SystemC and converting from one model to another.

To convert a C model to a sequential SystemC model, designers must keep adding trigger signals and wait statements for each behavior. This is true, although the top-level behavior of the SystemC shown in Table 1 does not change much from the C model. For example, to make F1 execute after the execution of F2, designers must declare a signal *F1_done* at the top level behavior, add a statement to trigger *F1_done* at the end of the execution of *F1,* and add a statement to wait *F1_done* at the beginning of the execution of *F2.*

To convert a sequential SystemC model to a parallel SystemC model, designers must delete old trigger signals and add new signals. For example, parallelization of *F1* and *F2* needs four steps. First, designers delete the signal *F1_done* and related statements inserted in the previous step. Second, designers declare a new signal *F1_F2_start.* Third, designers add a statement to trigger *F1_F2_start* before the execution of the top-level behavior. Finally, designers add statements to wait *F1_F2_start* at the beginning of the execution of *F1* and *F2,* respectively. Since the execution sequence change is implemented by adding/deleting trigger signals and related trigger/wait statements, the processes of converting from one model to another is tedious using SystemC.

Therefore, we choose SpecC rather than SystemC for the behavior exploration.

## 3.5 Behavior exploration process

The tasks of behavior exploration are:

a) Convert a non-clean C model to a non-clean sequential SpecC model.
b) Convert the non-clean sequential SpecC model to a clean sequential SpecC model.
c) Determine the granularity of behaviors, merge small behaviors, and split big behaviors based on the profiling results.
d) Explore the specification in order to find *behavior-parallel* and *behavior-pipeline* attributes among behaviors and converting the SpecC model by specifying parallel/pipeline execution relationships among behaviors.
*e)* Determine the most time-consuming behaviors by analyzing the profiling results.
*f)* Change the communication models among behaviors in order to enable *parallel execution.*

In task c, we use the SpecC profiler to produce the characteristics of behaviors. The SpecC profiler estimates characteristics of behaviors by computing sum of weighted operations based on a testbench simulation utilizing a designer-provided weight table. By analyzing these profiling results and the structure of the code, the system designer can determine the granularity of the design. Then behaviors are merged and split accordingly.

In task d, we find the behaviors containing *behavior-parallel* and *behavior-pipeline* in the specification by using SpecC profiler. The SpecC profiler provides statistics representing traffic and connection between behaviors. Therefore, sets of behavior-parallel and behavior-pipeline behaviors are found.

After finding *behavior-parallel* and *behavior-pipeline*, designers then update the original sequential SpecC to specify parallel and pipeline execution, by simply adding the par/pipe keywords. The SpecC profiler is again applied to the updated specification model in order to compute the performance improvement given by parallel/pipeline execution. The model is refined accordingly, with the refinement being repeated until an optimal performance is reached.

Finally, we re-model the communication. As we mentioned before, *parallel-match* is the sufficient condition of the *parallel execution. Parallel-match* can guarantee the *parallel execution* if and only if the communication among behaviors is modeled correctly.

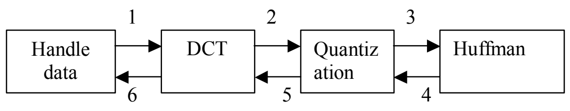

Figure 4: Sequential JPEG encoder model

Figure 4 shows an error-modeling example for a JPEG encoder. After the *Handle_data* completes its execution, it sends the output(1) to *DCT*. Similarly, *DCT* sends the output(2) to the *Quantization* block, *Quantization* block sends the output(3) to *Huffman,* and *Huffman* sends the output(4) back to *Quantization*, after their execution. Then the output(4) triggers the output(5) produced by *Quantization*, and the output(5) triggers the output(6) produced by *DCT*. As long as *Handle_data* receives output(6), it starts the execution on the next frame of data. Modeling the communication in this way, the four behaviors are executed sequentially, even they are mapped to four different architectural components and have gained *parallel match*.

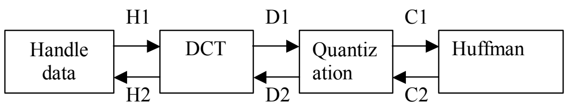

Figure 5: Pipeline JPEG encode model

To make four behaviors run in a pipeline fashion, we remodel the communication among behaviors shown in Figure 5. After successor behavior entity receives input from its predecessor behavior entity, it will immediately send an acknowledge back to the predecessor behavior entity, which will trigger the execution of its predecessor. For example, when *DCT* receives input *H1* from *Handle_data,* it immediately sends the acknowledge *H2* to *Handle data.* As long as *Handledata* receives *H2,* it starts execution on the next frame of data.

The resulting behavior model is clean and has suitable granularity. It can be imported into the VCC architectural exploration tool directly using VCC's graphical input capabilities, or indirectly using the SpecC-VCC translator.

## 4 Architecture exploration

In this project, we use VCC as our architecture exploration tool. We have built a translator to convert from our SpecC behavioral model to a VCC behavior model for importing into VCC.

VCC can estimate the system performance for the implementation on different target architectures. During performance estimation, it can handle bus competition as well as different memory/cache structures. We use VCC to evaluate the performance of the architecture exploration result. In stead of VCC, other third party tools can also be used for this purpose.

During architecture exploration, we first choose the system architecture. According to the heuristics provided by behavior exploration under SpecC, we then map behaviors to the architecture in order to achieve *parallel match*. We also map time-consuming functions to specific hardware components for the fast execution time. After behavior-architecture mapping, we evaluate the performance of the implementation by using VCC. If the design requirements are not met, then the process of system architecture selection or the behavior-architecture mapping will be repeated.

Because behaviors can be matched to different components and platforms, a variety of Intellectual Property (IP) components can be used in a mix and match fashion from a variety of vendors. IP can be exchanged at different levels of abstraction, from behavior definition for IP protection, to implementable hard and soft IP. This allows system designers and integrators to take advantage of IP reuse and exchange to reduce design cycle time with greater quality of designs.

## 5 Behavior-architecture refinement

After we derive the target architecture and behavior-architecture mapping solution, we use the SpecC refinement tool to refine the SpecC behavior model to SpecC architecture model. The SpecC refinement tools can automatically refine the behavior model to the architecture model with abstract communication (*channel)*. Designers need to refine *channels* to *wires* manually following the guidelines of the SpecC Methodology [1].

## 6 Architecture-implementation refinement

After behavior-architecture refinement, the SpecC architecture model is then translated into a SystemC architecture model. This translation allows for the use of SystemC based synthesis tools. This capability allows designers to take advantage of the vast selection of design and verification tools that currently exist to take care of the structural RTL down to mask creation steps of the flow. Though this is true, SystemC currently has and is working on more capabilities that will allow designs done at higher levels of abstraction to be more effective in creating new products more quickly.

SystemC already supports a well-defined HDL model and will support an RTOS and analog model in the future [6]. Using SystemC, implementation modeling and verification can be completed. Furthermore, the SystemC behavior to RTL synthesis tools can be used to generate an RTL model. Clearly defined constructs for synthesis have been developed for both behavioral and structural SystemC. Upon ensuring the model follows these guidelines, synthesis to the final product can be achieved.

By using the defined guidelines, models created in SpecC or VCC can be translated to SystemC code that will ensure the ability to synthesize to final products. SystemC allows designers to make tweaks in the translated design as needed at the RTL level. This is helpful when creating minor changes that will result in cost reduction re-spins of a product, as well as ensuring better power utilization techniques are used.

Because SystemC is C based, it is possible to easily implement mixed mode simulations between SpecC and SystemC models. This will allow for quickly adding new functionality for product derivatives, and verifying the product behavior before committing to the complete design. The down side remains that the simulation will only run as fast as the lowest abstraction model level will allow. This method should only be used if the original higher abstraction level model is not available and will take too long to create.

## 7 Conclusion

In this report, we provide C/C++ based system design flow based on the use of the SpecC, VCC, and SystemC. The report introduced the concepts of behavior exploration under SpecC. These concepts can be utilized to provide a suitable model and heuristics for later architecture exploration under VCC.

Our methodology is a C language-based methodology, from specification to implementation. Conversion from C to SpecC to SystemC is a logical and straightforward process. IP can be modeled at different levels of abstraction allowing for IP reuse, exchange, and integration to be possible. SpecC and SystemC languages enable tools to have a common framework for interoperability. This allows designers to utilize the best "point tool" solution for the implementation of the methodology. Since both languages use C as the underlying technology, interoperability can be achieved. This method quickly converts the C model to an implementation, resulting in decreased design cycle time.

In the design flow, the SpecC profiler, VCC architectural exploration tool, SpecC refinement tool and SystemC synthesis tools, all help to complete an optimized system design. Using this methodology, a JPEG encoder has been created and can be placed into current synthesis tools. Although more automation is needed, the methodology proves design decisions and trade-offs can be easily made at higher levels of abstraction, resulting in an easing of the time to market pressure.

Reference:

[1] D. Gajski, J. Zhu, et al., "SpecC: Specification Language and Design Methodology", Kluwer Academic Publishers, 2000
[2] www.ics.uci.edu/~specc
[3] www.cadence.com/products/vcc.html
[4] Cadence, "VCC2.1 Production documentation"
[5] Andreas Kanstein, "High-level architecture modeling for a top-down HW/SW co-design Flow", SMS 2000 Symposium, Phoenix.
[6] www.systemc.org
[7] Synopsys, CoWare, "SystemC version 1.21Beta Designer's Guide", 2001
[8] "Functional specification for SystemC 2.0", January, 2001
[9] P. Lieverse, et al., "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems", Proc. 1999 Workshop on Signal Processing Systems, 1999.
[10] E.A. Lee, et al., "Overview of The Ptolemy Project", UC, Berkeley, March, 2001
[11] F. Balarin, et al., "Hardware-Software Co-design of Embedded Systems, The POLIS Approach", Kluwer Academic Publishers, April 1997.
[12] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", in Information Processing, J.L. Rosenfeld, Ed, North-Holland Publishing Co., 1974
[13] K. Keutzer, S. Malik, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design" Transactions on Computer-Aided Design of Integrated Circuits and Systems 12, 2000
[14] A. Gerstlauer, R. Dömer, et al., "System Design: A Practical Guide of with SpecC. Kluwer Academic Publishers 2001