A Flexible, Syntax Independent Representation (SIR) for System Level Design Models

Ines Viskic, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, CA 92617
{iviskic, rdoemer}@ics.uci.edu

Abstract

System Level Design (SLD) is widely seen as a solution for bridging the gap between chip complexity and design productivity of Systems on Chip (SoC). SLD relieves the designer from detailed manual implementation by raising the level of abstraction in design models.

There are many different modeling approaches to SLD. With the abundance of design languages and supporting tools, a designer can create a multitude of models of the same system that are difficult to compare and evaluate against each other. Also, it is not unusual for the same system model to comply to specification guidelines in one simulation tool, and exceed them in another.

This paper presents an approach to circumvent this problem by describing a design model using a $Syntax$ $Independent$ $Representation$ (SIR). Such representation of the model complies to modeling guidelines shared by various SLD languages but is not restricted by their syntactic compositions. Further, the same structure can represent models on different levels of abstraction.

Our multidimensional and multipurpose SIR structure can be projected onto planes of modeling languages including SpecC and SystemC, as demonstrated by experimental results shown in this work.

Keywords: *system level design, modeling, simulation, evaluation, model representation, data structure, syntax and semantics.*

1. Introduction

As implementation technology evolves, the idea of integrating system components into a single chip is becoming more attractive. System-on-chip (SoC) designs usually consume less power, have a lower cost and higher reliability than multi-chip systems. However, SoC are becoming too complex to design and program manually. Moreover, time-to-market in SoC design is continually shrinking, increasing the pressure on design productivity. One generally accepted solution to close the productivity gap is to make design decisions at a higher, system level of abstraction. Such system level models (and their components) should be reusable as parts of other system models and simulatable in order to verify that the model indeed complies with its specification before going into production.

Traditional programming languages are not sufficient for SLD modeling, because they lack support for modeling constructs describing the architecture and the functionality of the modeled system. System Level Design Languages (SLDL), on the other hand, support hierarchical decomposition of the model, hardware-software (HW/SW) communication, concurrency and synchronization of system components. However, there are many different approaches to SLD, and each is further supported with a specific design language and simulation engine for evaluation and verification. These simulation engines and other SLDL supported tools have their own internal data structure only used for a strictly defined purpose: simulation, verification, or synthesis.

SystemC [1], [2], SpecC [3], [4], [5] and SystemVerilog [6], [7] are all examples of SLDL, each supported with their own tool set. With these different SLD languages, it is possible to create a multitude of models of the same system that are difficult to compare and evaluate against each other. To one extreme, they can even be unrecognizable as equivalent models.

Also, a designer can model a system with a SLD language with little regards to the modeling guidelines of SLD. It is possible that such a model simulates and even produces the desired simulation output using a simulation engine specific to that SLD language. Such badly structured model seamingly complies to its specification, but would not do so in any other modeling language.

We have created a $Syntax$ $Independent$ $Representation$ (SIR) data structure in the attempt to overcome these and similar problems introduced by

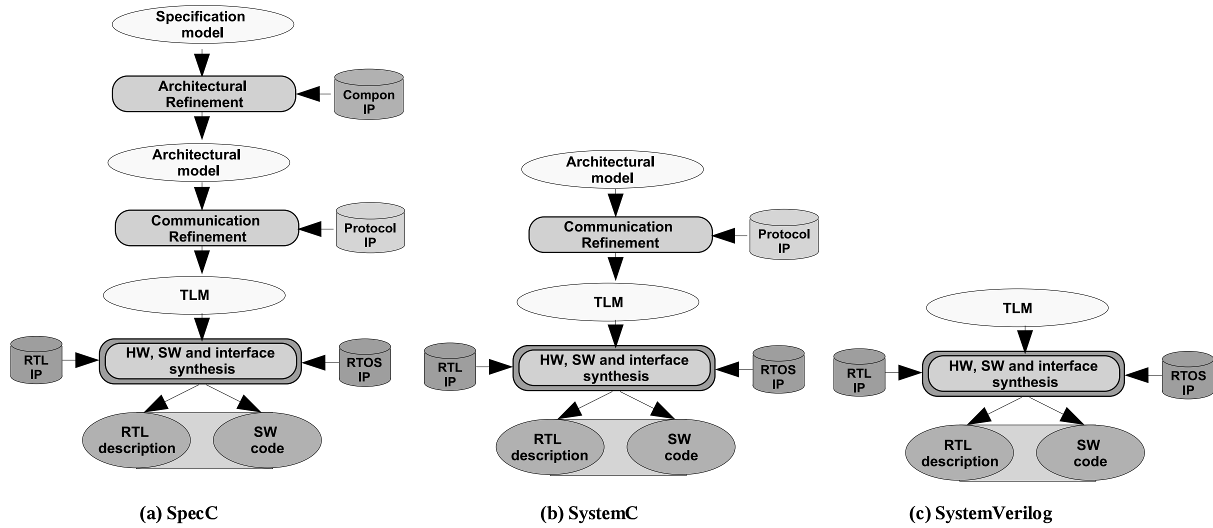

Figure 1. SLD modeling levels of abstraction

syntactic limitations of SLD languages.

The SIR structure presented here is flexible enough to describe models on different levels of abstraction. Higher abstracted models simply omit the modeling constructs which are not visible in the model at that level, and include them in the more detailed description level. This feature of SIR is demonstrated in SIR-supported design tools [9].

Further, the SIR structure adheres to modeling guidelines shared by different SLD languages. A poorly written model in one language can then easily be detected by creating its SIR structure and analyzing it for modeling compliance. Design analysis identifies and informs the designer of any violations of modeling guidelines existing in the model. This feature of the SIR is demonstrated in our SIR-supported design analysis tool (*see section* 4.1).

Finally, the SIR structure is syntax free and therefore independent of SLDL specifics. Once created, the SIR representation of the design can be exported in various formats which are supported by different modeling/simulation tools. This feature of SIR is demostrated in our SIR-supported convertion tool (*see section* 4.2).

The remainder of this paper is organized as follows. In Section 2, we give an overview of SLD languages and how they can be used to model a system at different levels of abstraction. Section 3 describes the SIR data structure and outlines its properties and usage. In Section 4, we use MP3 Decoder and Vocoder design examples to demonstrate the effectiveness of the SIR structure. Finally, we conclude the paper with a summary.

2. SLD Languages and Methodology

SystemC and *SpecC* are C-based representatives of SLDL with strong support in the field of system-level design. *SystemC* [1], [2] and *SpecC* [3], [4], [5] are based on ANSI C/C++ language and as such share all C-related features. However, they each have their own syntactic structure for describing computation and communication entities, concurrency, hierarchy, synchronization and timing. SpecC describes a model with behaviors interconnected with channel and interface constructs, while SystemC models a system with module objects implementing its structure and processes implementing its functionality. The connection and communication between the processing elements is implemented with channels and interfaces, similar to those in SpecC. Also, SpecC and SystemC are both supported by their own simulation engine with which generated models are validated for correctness.

SystemVerilog [6], [7] is a major extension of the *Verilog* [8] language and was ratified as the hardware description and verification language (HDVL) standard. It dramatically improves the productivity in the design of large gate count, IP-based, bus-intensive chips over Verilog. *SystemVerilog* is able to invoke C/C++ functions, thus supporting efficient co-simulation with system-level blocks.

Any SLDL should be able to model a system on various levels of abstraction. The most abstracted model is the *specification level model*, which describes the system functionality and is free of any structural/architectural details. Next, the *architectural level model* provides both the structural and functional description of the design. It consists of multiple processors, custom hardware blocks,

intellectual property (IP) components and memories, interconnected with busses. The $transaction\ level\ model$ implements communication between system components using bus interfaces and protocols for data transfer and process synchronization. Finally, the $implementation\ level\ model$ is synthesized in the back-end, with each component implemented separately, either with its RTL description, or reusing pre-coded library components.

Figure 1 illustrates which levels of abstraction are supported when modeling with aforementioned SLD languages ($SpecC$, $SystemC$ and $SystemVerilog$, respectively).

As seen in Figure 1, $SpecC$ supports modeling at all levels of abstraction. $SystemC$ lacks the constructs with which a design can be described without regards to its structure, so it does not support modeling at $specification\ level$. $SystemVerilog$ supports modeling and verification of $transaction\ level$ models.

This paper describes the SIR structure used to represent models compliant to modeling rules of SLD but without regards to syntactic rules imposed by different SLD languages. At this time, the value of the SIR is demonstrated with an analysis and conversion tool which can successfully translate architectural SpecC models into equivalent SystemC models, using our SIR structure. Future work includes a tool that outputs a SystemVerlog model from an SIR design as well as a GUI with which an SIR structure can be created graphically.

3. Syntax Independent Representation

The SIR structure is a complex data structure which can be viewed as a graph. The nodes of the graph are represented by C++ class objects, whereas the edges are represented by C++ pointers. Figure 2 shows an abstracted view of the hierarchical class order in the SIR structure.

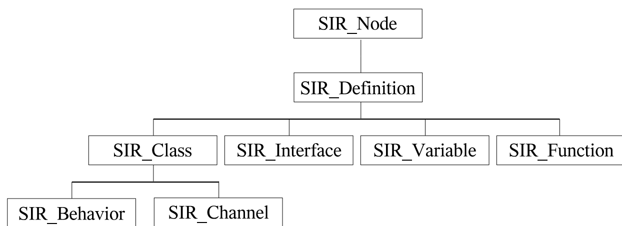

Figure 2. SIR class hierarchy

The nodes in the SIR graph are of different types inherited from one generic class SIR_Node. Class $SIR_Definition$ inherits class SIR_Node and further classifies node types into four categories: SIR_Class, $SIR_Interface$ $SIR_Variable$, and $SIR_Function$. Classes $SIR_Behavior$ and $SIR_Channel$ inherit class SIR_Class. Each node type is implemented with a corresponding C++ class which defines the data members and API methods available for the node.

3.1. SIR features

The nodes in any SIR graph can be classified into two groups, called levels. The nodes at level 1 contain all basic data contained in a SIR file, whereas the level 2 nodes represent a more abstract view of the SIR data. For example, the structural hierarchy of the design is represented directly at level 2, while at level 1, the same hierarchical composition is clouded by details implementing design functionality.

The edges of a SIR graph, representing relations among the nodes, can also be classified into two groups, called pointers and links. A pointer represents a containment relation of two objects. For example, a compound statement contains a list of statements and therefore there exists a pointer from the compound statement object to the header of the statement list (and also from the header of the list to the elements of the list). A link represents a (loose) connection between two objects, that does not imply any containment. For example, expressions and symbols have a link to a node representing their type.

Graphical overviews of the top level SIR data structure object SIR_Design at levels 1 and 2 are given in Figure 3 and Figure 4, respectively.

The general information of the design is contained in the level 1 representation, with objects $SIR_FileInfo$, SIR_Notes and supported SIR_Types. The core of the SIR_Design object at level 1 is a hierarchical list of $SIR_Symbols$ and $SIR_UserTypes$. $SIR_Symbols$ and $SIR_UserTypes$ are lists of generic objects of different types used to implement the structure, computation and communication of the design. The list $SIR_UserTypes$ contains user-defined type declarations (i.e. struct and union types and aliases), named and unnamed enumerator definitions, while $SIR_Symbols$ include classes implementing model behavior as well as other variables and methods.

Figure 4 shows the design structure at level 2. The design is represented with a list of behaviors ($SIR_Behaviors$), a list of channels ($SIR_Channels$), a list of interfaces ($SIR_Interfaces$), a list of global variables ($SIR_Variables$), and a list of global functions ($SIR_Functions$). Members of these lists are interconnected with pointer edges to create a unique representation of a system level model.

Considering the classification of SIR nodes into two levels and the SIR edges into pointers and links, a design represented by its SIR structure can be viewed as a generic graph. The SIR graph becomes a tree, if the edges classified as links are ignored and only pointer edges build the arcs between the nodes. We call such a graph a design tree.

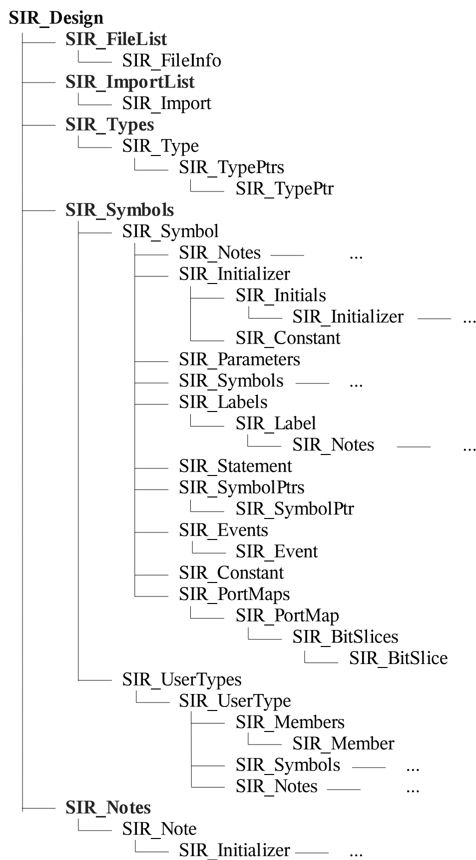

Figure 3. SIR components: SIR_Design object at level 1

3.2. SIR design tree

The SIR structure of a design model at level 2 (Figure 4) is called the design tree and is composed of nodes, representing the computation and communication objects of the model, and edges which connect the nodes into a functional hierarchical system.

The usage of design trees is three-fold. First, refinement steps performed over the SIR data structure usually consist of multiple traversals over the design tree. Traversals follow only the pointers connecting the nodes. With each iteration the model is being described with more details. For example, in the architectural refinement step, all computation gets distributed and mapped to different processing elements (PE), and communication refinement combines end-to-end channels connecting PEs into a central bus implementing a bus protocol. More on SIR-supported refinement of models can be found in [9].

Second, traversing a design tree can easily identify any violations of the modeling rules common to SLD languages.

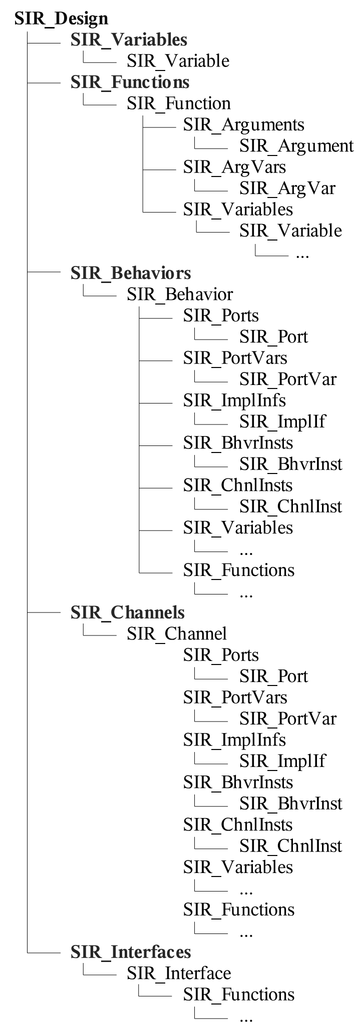

Figure 4. SIR components: SIR_Design object at level 2, i.e. the design tree

Such analysis either informs the user of the compliance of the model, or produces a message listing possible irregularities of the model.

Third, and most importantly, the tree representation frees the model from any notion of language syntax. Once created, an SIR structure can be exported as either a SpecC model or an equivalent SystemC model. Each model can then be simulated with SpecC and SystemC simulation engines, or further refined with their respective modeling tools. Exporting a syntax-free SIR to SystemC and SpecC modeling plane is shown in Figure 5.

As Figure 5 shows, each SLD approach to modeling is represented by a 2-dimensional plane: the SpecC plane is on the bottom, and the SystemC on the right side of the figure. Each modeling plane is restricted with the syntactic

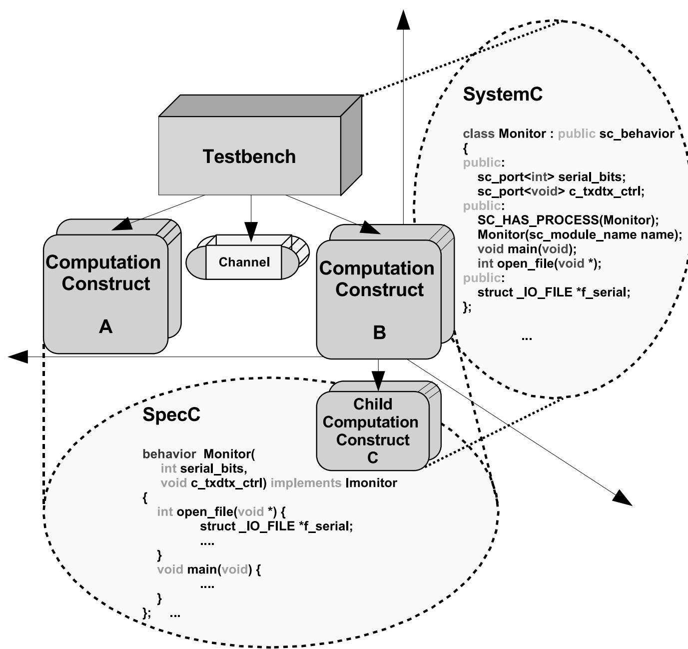

Figure 5. SIR structure in relations to SpecC and SystemC modeling planes

structure the of supported SDL language, modeling guides and design flow. However, our SIR structure, in turn, is not limited by the syntax of any language and can therefore be projected to any of the two modeling planes. This feature of SIR is symbolically shown with a 3-dimensional representation of the SIR structure. In other words, the syntax-free (i.e. ”multi-dimensional”) SIR structure can be ”projected” onto the modeling planes of SystemC and SpecC, which can be thought of as different ”views” on the SIR representation.

4. Experimental Results

The SIR independence from syntax is demonstrated with the following analysis and conversion tool.

Our tool creates an SIR design tree from the SpecC model, traverses the tree's object lists, interprets the semantics of each component, analyzes them, and finally maps them into corresponding SystemC constructs.

The control flow of the SpecC-to-SystemC conversion tool, shown in Figure 6, consists of the following four steps:

1. Read SpecC model description and create corresponding SIR design tree

2. Analyze design. If translation is not possible, abort with appropriate error message

3. Traverse SIR design tree and its object lists to create SystemC header ($filename.h$) file

4. Traverse SIR design tree and its object lists to create SystemC source ($filename.cpp$) file

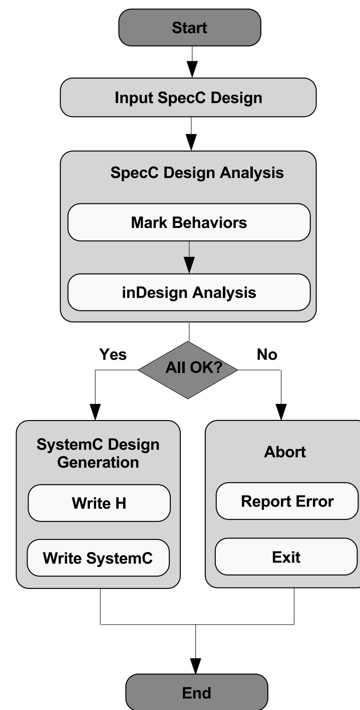

Figure 6. SpecC to SystemC converter diagram flow

Following are two industry-sized examples of real SOC designs, a Vocoder and a MP3decoder that we have used as input to the automatic conversion tool. In each example, a SpecC architectural model was constructed and analyzed by the converter. If the conversion was possible, the tool generated the equivalent SystemC architectural description. The generated model was then compiled and linked with the SystemC library, and executed for equivalence validation.

4.1. MP3 decoder

We first describe an example of a MP3 decoder to show the restrictions to the model of the SIR structure in order to conform to the standard modeling rules shared by both SpecC and SystemC tools. Our design analysis tool correctly identifies the irregularities in the MP3Decoder model used as its input.

The MP3 decoder [11], [12] is a device for decompression of a MP3 stream into audio data. The MP3 stream used as input is organized in frames and encoded using the MP3

compression algorithm. Each frame contains main audio data encoded as a Huffman stream and certain side information necessary for decompression, such as compression parameters and scale factors.

The testbench for the MP3 decoder consists of Stimulus, Design Under Test (DUT) and Monitor components executing in parallel and synchronized with each other by two synchronization channels, C1 and C2, as shown in Figure 7. The DUT component is further separated into Parser, sfilter, IStereo and TopHybrid functions, where frames are created from the input MP3 stream. MP3 decoding and further data processing is implemented in IStereo and TopHybrid.

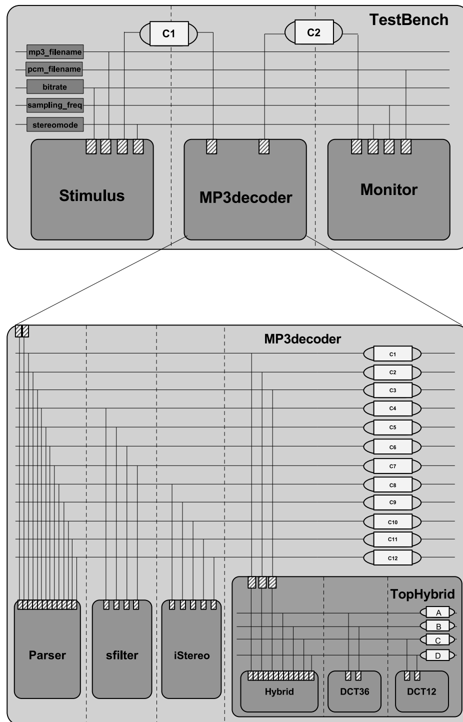

Figure 7. SIR representation of the MP3 Decoder model

The testbench is used as an input of the converter tool. After the Design Analysis, the converter aborts and reports the message informing the designer of the incompatibilities of the design with the clean modeling structure of either SpecC or SystemC.

Found irregularities of the tool are shown in Figure 8. They include the usage of global variables shared by top-level components Stimulus, DUT and Monitor, as well as an access to an out of bounds element of an array (i.e. frame). Figure 7 shows the MP3Decoder testbench with the mentioned global variables $mp3_filename$, $pcm_filename$, $bitrate$, $sampling_freq$ and $stereomode$. Access to out of bounds elements of the frame, in turn, is performed during data processing and decoding.

ERROR: '*Design*' cannot contain port '*mp3file_name*' for communication
(use interface/signal instead)
ERROR: '*Design*' cannot contain port '*pcmfile_name*' for communication
(use interface/signal instead)
ERROR: '*Design*' cannot contain port '*sampling_frequency*' for communication
(use interface/signal instead)
ERROR: '*Design*' cannot contain port '*bitrate*' for communication
(use interface/signal instead)
ERROR: '*Design*' cannot contain port '*stereomode*' for communication
(use interface/signal instead)
ERROR: Array access '*window[-1]*' out of bounds!
ERROR: Array access '*window[-2]*' out of bounds!
ERROR: Array access '*window[-3]*' out of bounds!
...

Figure 8. Output of the Converter tool for the MP3Decoder model

4.2. Vocoder

Our Vocoder example shows that the SIR structure of a design can correctly output both SpecC and SystemC architectural models, providing that the model does not contain any modeling irregularities.

The voice encoder/decoder (vocoder) application used in the Vocoder design [13], is part of the European GSM standard for cellular telephone networks. Figure 9 shows the architectural model of the Vocoder. The testbench consists of Stimulus, Monitor and Vocoder running in parallel, as well as a simple C/C++ function (Argument Handler) that reads parameters from the console. The Vocoder structure is partitioned into the DSP processor core and an ASIC hardware component for the codebook search. The functionality of the DSP processor is to prepare the next speech block while the hardware component operates on the previous block. Communication channels between DSP and Hardware are used to synchronize their execution.

A SystemC model has been successfully generated by our converter from the input SpecC model in only 0.112 seconds. Table 1 compares the features of the SpecC and SystemC models of the Vocoder example. The execution of the generated SystemC model simulates only slightly slower (7.05%) than the one of the SpecC model.

Model properties	SpecC	SystemC
Size/lines of code (source)	8974	8395
Size/lines of code (header)	5956	2879
Simulation time (sec)	0.959	0.896

Table 1. Comparison of SpecC and SystemC models

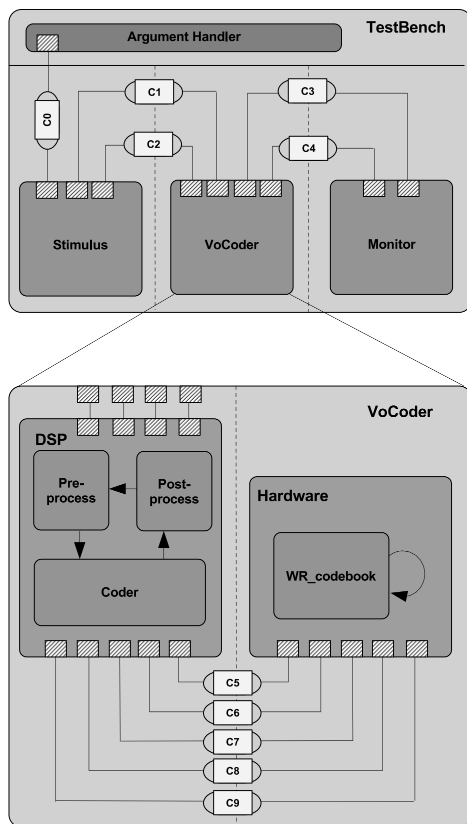

Figure 9. SIR representation of the Vocoder model

5. Conclusion

This paper presented the Syntax Independent Representation of system models, a complex data structure that adheres to the modeling guidelines of SLD but is not dependent on the syntax of a particular SLD language. The described SIR structure is flexible enough to describe models on different levels of abstraction.

The proposed SIR structure is applicable to a multitude of design tools. For example, the SIR structure can be used as an input to various design analysis tools, such as SLD language parsers and profilers. In our example, the SIR structure of a MP3 Decoder was used as an input to our Design Analysis tool and then analyzed for modeling compliance.

Further, the SIR structure can be used as a central data structure for modification and refinement tools, since it is flexible enough to represent models at different abstraction levels, starting from the most abtracted specification level, down to the detailed implementation level.

Finally, the SIR representation of a model can be easily exported to different SDL langage formats, such as SystemC and SpecC, as was shown with the conversion of the SpecC model of the Vocoder to its equivalent SystemC model.

In future work, we plan to implement a conversion tool that outputs a SystemVerilog model from a SIR design as well as a GUI with which our SIR structure can be created and maintained graphically.

References

[1] OSCI. http://www.systemc.org/.

[2] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[3] Jianwen Zhu, Rainer Dömer, and Daniel D. Gajski. ”Syntax and semantics of the SpecC language. *In Proceedings of the International Symposium on System Synthesis*, Osaka, Japan, December 1997.

[4] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[5] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[6] Accellera. SystemVerilog 3.0, extensions to verilog. In http://www.accellera.org.

[7] http://www.systemverilog.org/.

[8] Samir Palnitkar. *Verilog HDL : A Guide to Digital Design and Synthesis*. Prentice Hall, 1996.

[9] System on Chip Environment. In http://www.cecs.uci.edu/ cad/sce.html

[10] R. Salambi et. al. ”Design and description of cs-acelp: a toll quality 8 kb/s speech coder”. *In IEEE Transactions on Speech and Audio Processing*, Vol. 6, No. 2, pp. 116-130, March 1998.

[11] Pramod Chandraiah. Master of Science Thesis: ”Specification and Design of a MP3 Audio Decoder”. 2005.

[12] Pramod Chandraiah, Hans Gunar Schirner, Nirupama Srinivas, and Rainer Dömer. ”System-On Chip Modeling and Design: A Case Study on MP3 Decoder”. Center for Embedded Computer Systems, 2004.

[13] European Telecommunication Standards Institute (ETSI). *Enhanced full rate (efr) speech transcoding (gsm 06.60)*. In Final Draft, November 1996.