System-on-Chip Environment (SCE)

Tutorial

Samar Abdi
Junyu Peng
Rainer Doemer
Dongwan Shin
Andreas Gerstlauer
Alexander Gluhak
Lukai Cai
Qiang Xie
Haobo Yu
Pei Zhang
Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
+1 (949) 824-8919
http://www.cecs.uci.edu

System-on-Chip Environment (SCE): Tutorial

by Samar Abdi, Junyu Peng, Rainer Doemer, Dongwan Shin, Andreas Gerstlauer, Alexander Gluhak, Lukai Cai, Qiang Xie, Haobo Yu, Pei Zhang, and Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
+1 (949) 824-8919
http://www.cecs.uci.edu

Published September 23, 2002

Table of Contents

Chapter 1. Introduction

The basic purpose of this tutorial is to guide a user through our System-on-Chip design environment (SCE). SCE helps designers to take an abstract functional description of the design and produce an implementation. We begin with a brief overview of our SoC methodology, describing the design flow and various abstraction levels. The overview also covers the user interfaces and the tools that support the design flow.

We then describe the example that we use in this tutorial. We selected the GSM Vocoder as an example for a variety of reasons. For one, the Vocoder is a fairly large design and is an apt representative of a typical component of a system on chip design. Moreover, the functional specification of the Vocoder is well defined and publically available from the European Telecommunication Standards Institute (ETSI).

The tutorial gives a step by step illustration of using the System-on-Chip Environment. Screenshots of the GUI are presented to aid the user in using the various features of SCE. Over the course of this chapter, the user is guided on synthesizing the Vocoder model from an abstract specification to a clock cycle accurate implementation. The screenshots at each design step are supplemented with brief observations and the rationale for making design decisions. This would help the designer to gain an insight into the design process instead of merely following the steps. We wind up the tutorial with a conclusion and references.

1.1. Motivation

System-on-Chip capability introduces new challenges in the design process. For one, codesign becomes a crucial issue. Software and Hardware must be developed together. However, both Software and Hardware designers have different views of the system and they use different design and modeling techniques.

Secondly, the process of system design from specification to mask is long and elaborate. The process must therefore be split into several steps. At each design step, models must be written and relevant properties must be verified.

Thirdly, the system designers are not particularly fond of having to learn different languages. Moreover, writing different models and validating them for each step in the design process is a huge overkill. Designers prefer to create solutions rather than write several models to verify their designs.

It is with these aspects and challenges in mind that we have come up with a System-on-Chip Environment that takes off the drudgery of manual repetitive work from the

designer by generating each successive model automatically according to the decisions made by the designers.

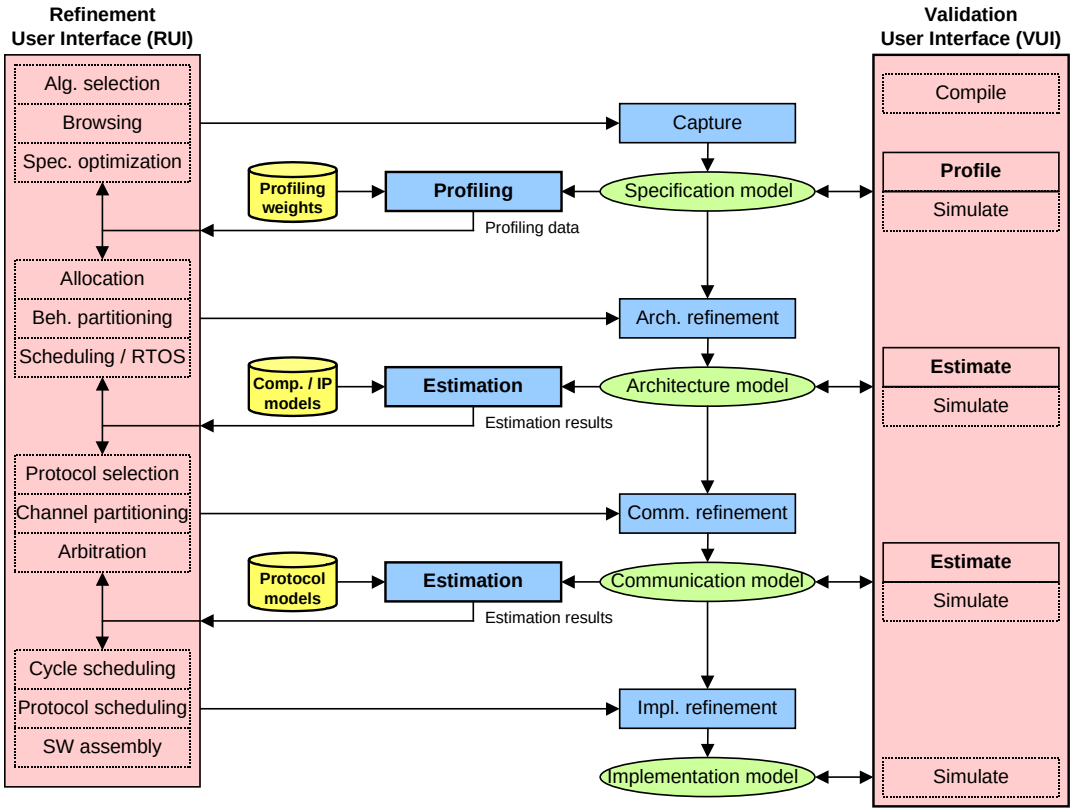
1.2. SCE Goals

SCE represents a new technology that allows designers to capture system specification as a composition of C-functions. These are automatically refined into different models required on each step of the design process. Therefore designers can devote more effort to the create part of designing and the tools can create models for validation and synthesis. The end result is that the designers do not need to learn new system level design languages (SystemC, SpecC, Superlog etc.) or even the existing Hardware Description Languages (Verilog, VHDL).

Consequently, the designers have to enter only the golden specification of the design and make design decisions interactively in SCE. The models for simulation, synthesis and verification are generated automatically.

1.3. Models for System Design

Figure 1-1. SCE Exploration Engine



The SoC system-level design methodology is shown in the figure. It consists of 4 levels of model abstraction viz.specification, architecture, communication and implementation models.Consequently, there are 3 refinement steps viz. architecture refinement, communication refinement and implementation refinement. These refinement steps are preformed in the order as given. As shown in the figure, we begin with an abstract specification model. The specification model is untimed and has only the functional description of the design. Architecture refinement transforms this specification to an architecture model. It involves partitioning the design and mapping the partitions onto the selected components. The architecture model thus reflects the intended architecture for the design. The next step of Communication refinement adds system busses to the design and maps the abstract communication between components onto the busses. The

result is a timing accurate communication model (bus functional model). The final step is implementation refinement which produces clock cycle accurate RTL model for the hardware components and instruction set specific code for the processors. All models are executable and can be validated through simulation. Moreover, all transformations are formally defined and proven for refinement of one model to another.

1.4. System-on-Chip Environment

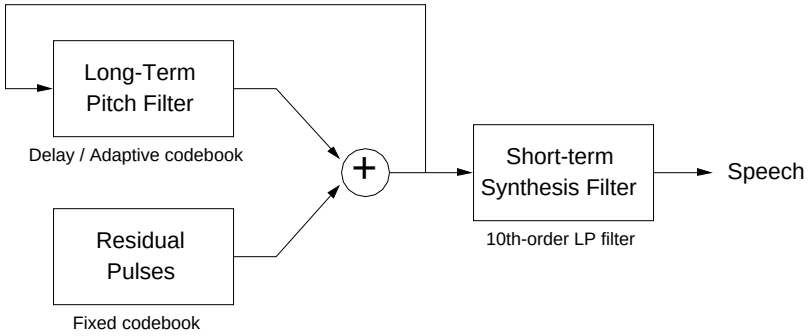
The SCE provides a environment for modeling, synthesis and validation. It includes a graphical user interface (GUI) and a set of tools to facilitate the design flow and perform the aforementioned refinement steps. The two major components of the GUI are the Refinement User Interface (RUI) on the left and the Validation User Interface (VUI) on the right. The RUI allows designers to make and input design decisions, such as component allocation, specification mapping. With design decisions made, refinement tools can be invoked inside RUI to refine models. The VUI allows the simulation of all models to validate the design at each stage of the design flow.

Each of the boxes is corresponding to a tool which performs a specific task automatically. Profiling tool is used to obtain the characteristics of the initial specification, which serves as the basis for architecture exploration. Refinement tool set automatically transforms models based on relevant design decisions. Estimation tool set produces quality metrics for each intermediate models, which can be evaluated by designers.

With the assistance of the GUI and tool set, it is relatively easy for designer to step through the design process. With the editing, browsing and algorithm selection capability provided by RUI, a specification model can be efficiently captured by designers. Based on the information profiled on the specification, designers input architectural decisions and apply the architecture refinement tool to derive the architecture model. If the estimated metrics are satisfactory, designers can focus on communication issues, such as protocol selection and channel partitioning. With communication decisions made, the communication refinement tool is used to generate the communication model. Finally, the implementation model is produced in the similar fashion. The implementation model is ready for RTL synthesis.

1.5. GSM Vocoder

Figure 1-2. GSM Vocoder



The example design used through out this tutorial is the GSM Vocoder system , which is employed worldwide for cellular phone networks. The figure shows the GSM Vocoder speech synthesis model. A sequence of pulses is combined with the output of a long term pitch filter. Together they model the buzz produced by the glottis and they build the excitation for the final speech synthesis filter, which in turn models the throat and the mouth as a system of lossless tubes.

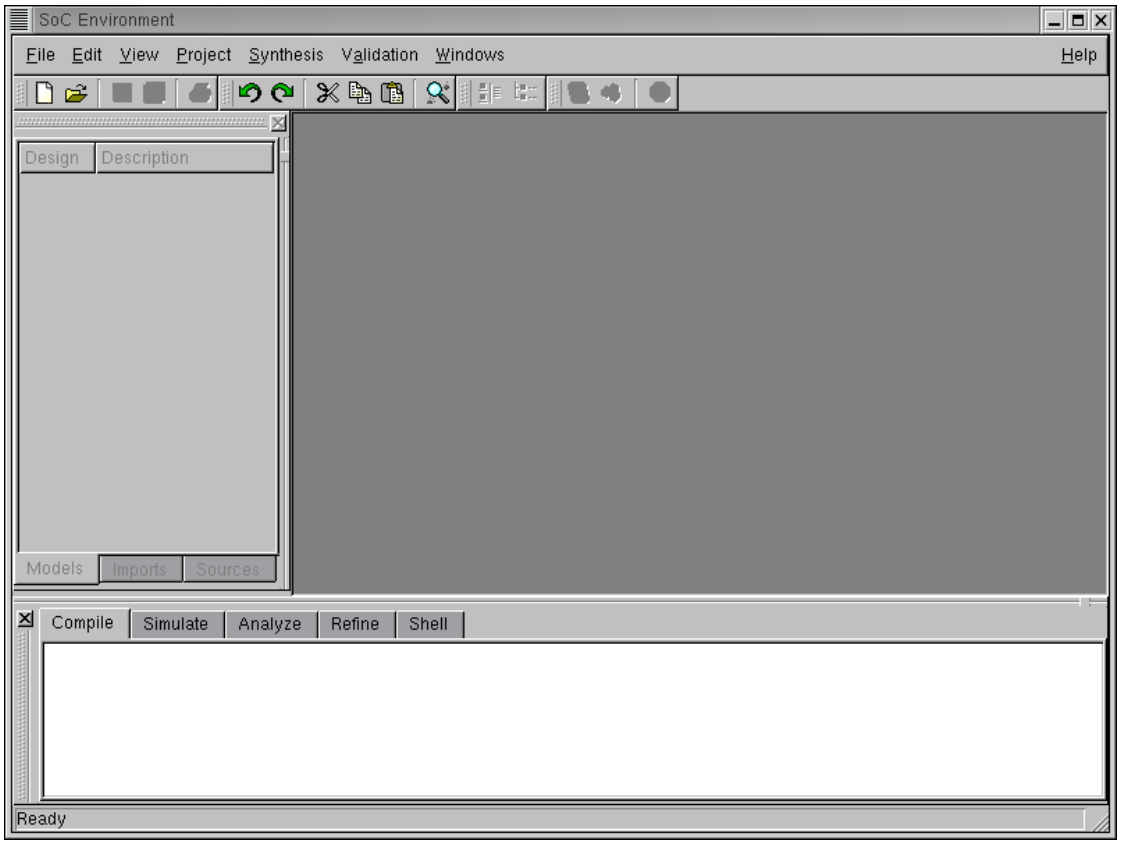
The example used in this tutorial encodes of speech data comprised of frames. Each frame in turn comprises of 4 subframes. Overall, each sub-frame has 40 samples which translate to 5 ms of speech. Thus each frame has 20 ms of speech and 160 samples.Each frame uses 244 bits. The transcoding constraint (ie. back to back encoder/decoder) is < 10 ms for the first subframe and < 20 ms for the whole frame (4 subframes).

The vocoder standard, published by the European Telecommunication Standards Institute (ETSI), contains a bit-exact reference implementation of the standard in ANSI C. This reference code was taken as the the basis for developing the specification model. At the lowest level, the algorithms in C could be directly reused in the leaf behaviors without modification. Then the C function hierarchy was converted into a clean and efficient hierarchical specification by analyzing dependencies, exposing available parallelism, ets. The final specification model is composed of 13,000 lines of code, which contains 43 leaf behaviors.

Chapter 2. Getting Started with Specification

The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to the series of exploration and refinement steps in the SoC design methodology. Moreover, the specification model defines the granularity for exploration through the size of the leaf behaviors. It exposes all available parallelism and uses hierarchy to group related functionality and manage complexity. In this chapter, we go through the steps of creating a project in SCE and intiating the system design process. The various aspects of the specification are observed through simulation and profiling. Also, the model is graphically viewed with the help of SCE tools.

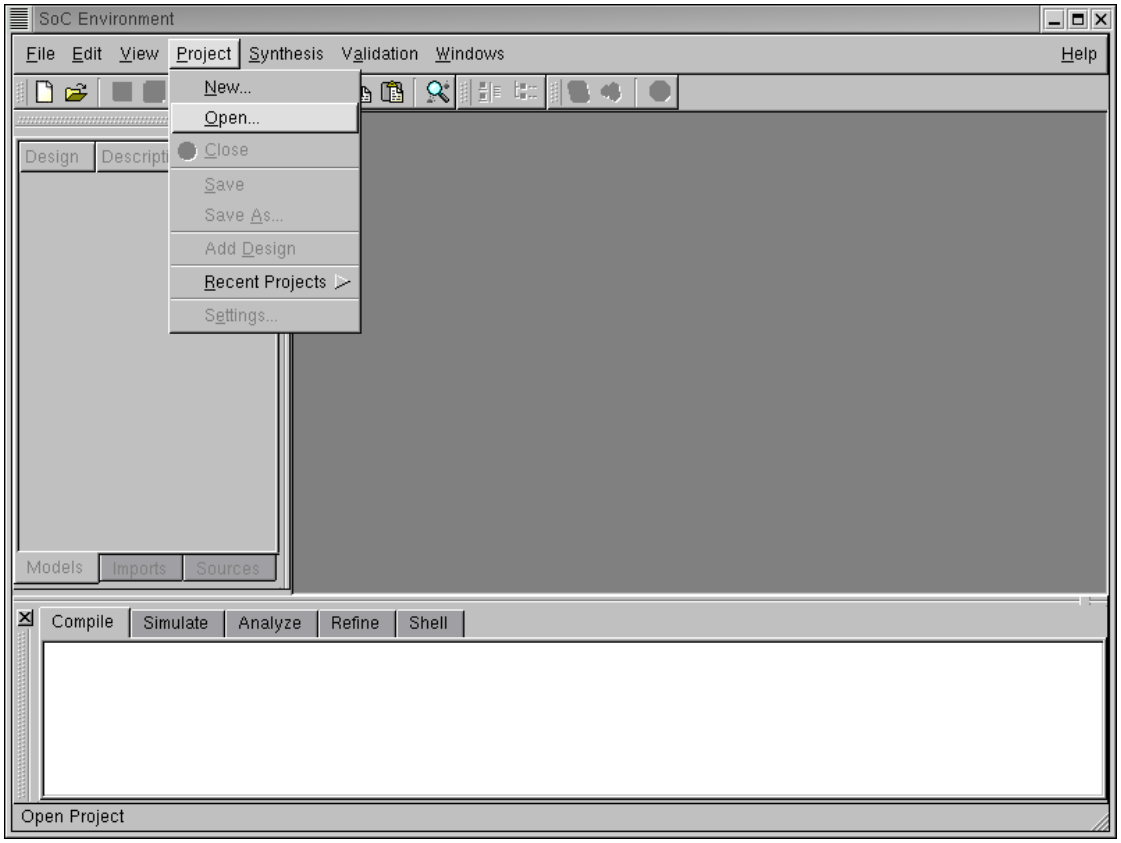
2.1. SCE window



On launching the System on Chip Environment (SCE), we see the above GUI. The GUI is divided broadly into three parts. First is the Project Management window on the top left part of the GUI, which maintains the set of Models in the open projects. This window becomes active once a project is opened and a design is added to it. Secondly, we have the design management window on the top right where the currently active design is maintained. It shows the hierarchy tree for the design and maintains various statistics associated with it. Thirdly, and finally, we have the logging window at the bottom of the GUI, which keeps the log of various tools that are run during the course of the demo. We keep logs of compilation, simulation, analyses and refinement of models.

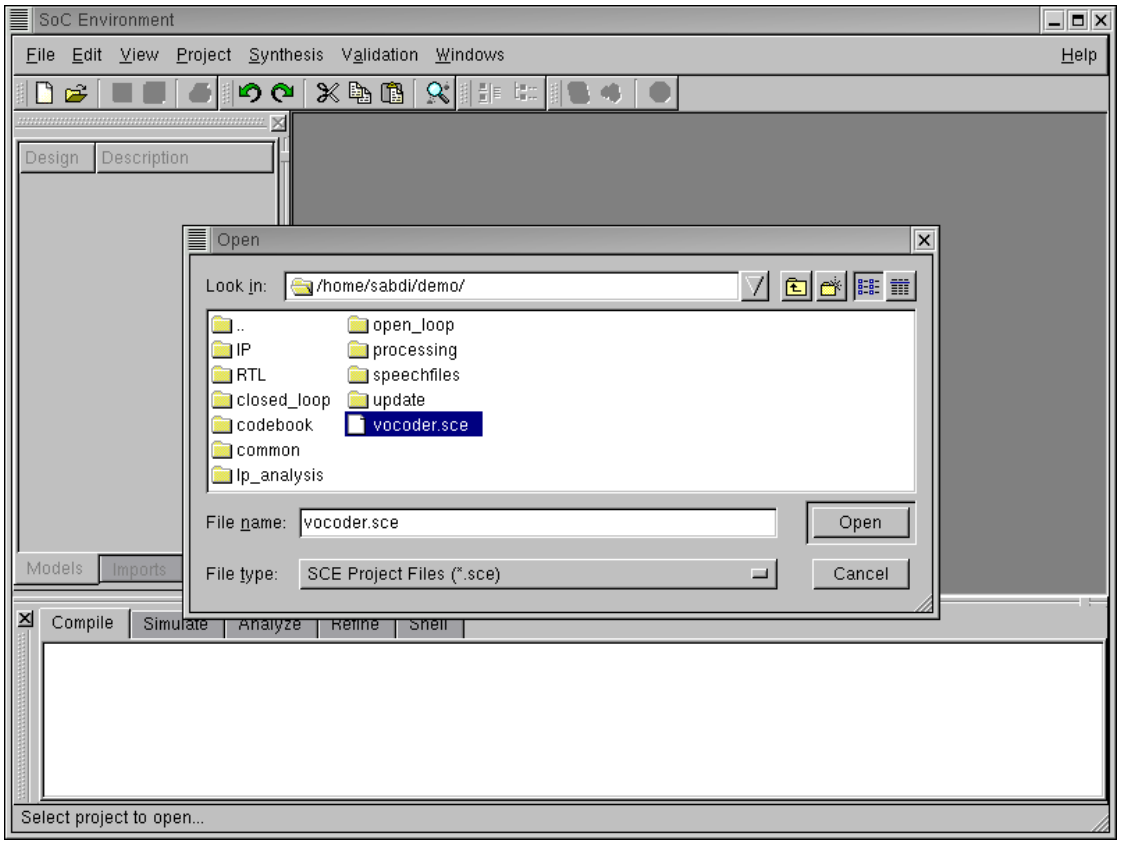
The GUI also consists of a tool bar and shortcuts for menu items. The File menu handles file related services like opening designs, importing models etc. The Edit menu is for editing purposes. The View menu allows various methods of graphically viewing the design. The Project menu manages various projects. The Synthesis menu provides for launching the various refinement tools and making synthesis decisions. The Validation menu is primarily for compiling or simulating models.

2.2. Open project



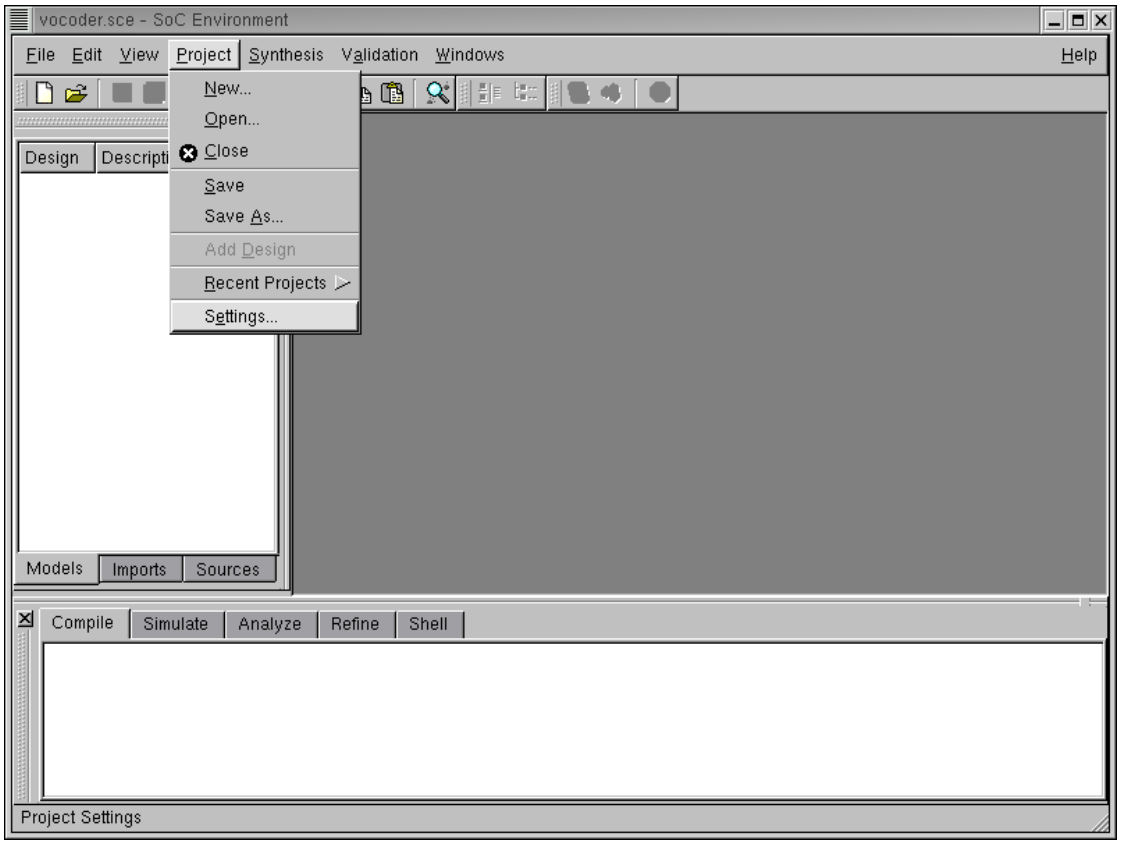
The first step in working with SCE is opening a project. A project is associated with every design process since each design might impose a different set of databases or dependencies. The project is hence used by the designer to customize the environement for a particular design process. We begin by selecting Project—→Open from the menu bar.

2.2.1. Open project (cont'd)



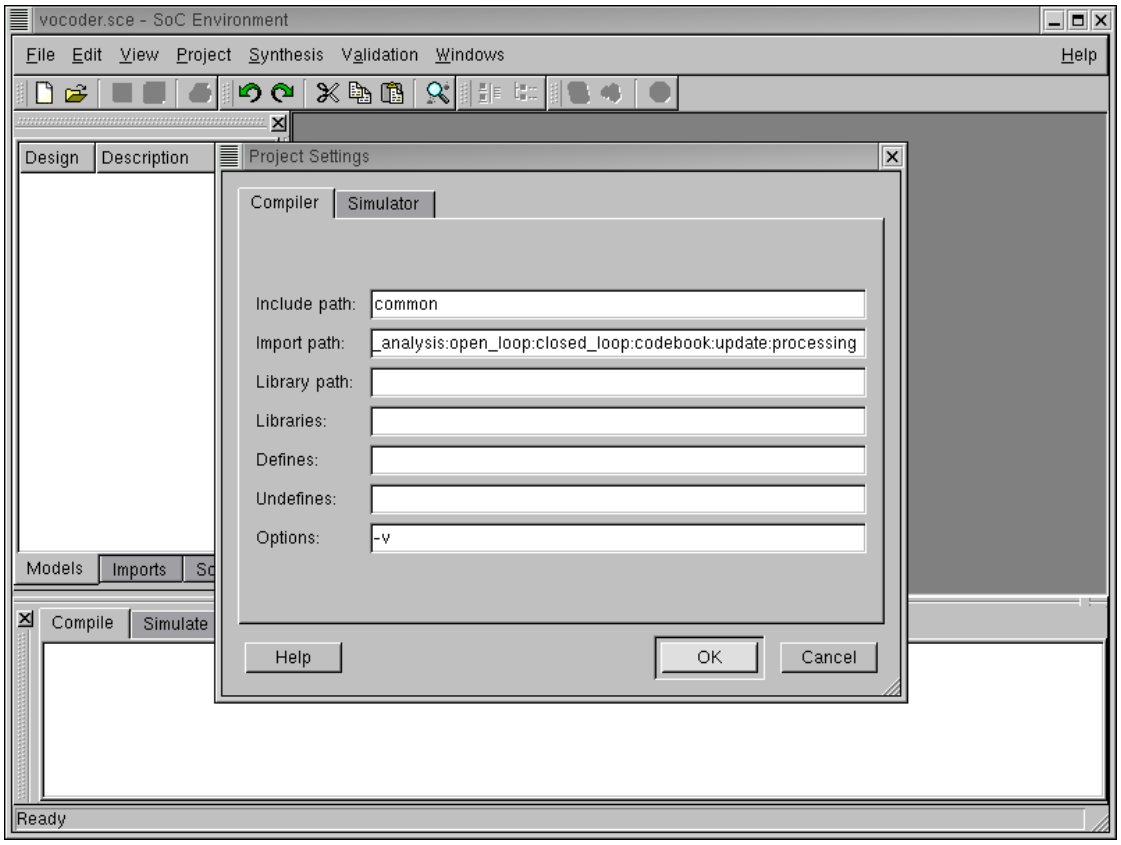
A file selection window pops up. For the purpose of the demo, a project is pre-created. We simple open it by selecting the project "vocoder.sce" and Left click on Open on right corner of the the pop-up window.

2.2.2. Open project (cont'd)



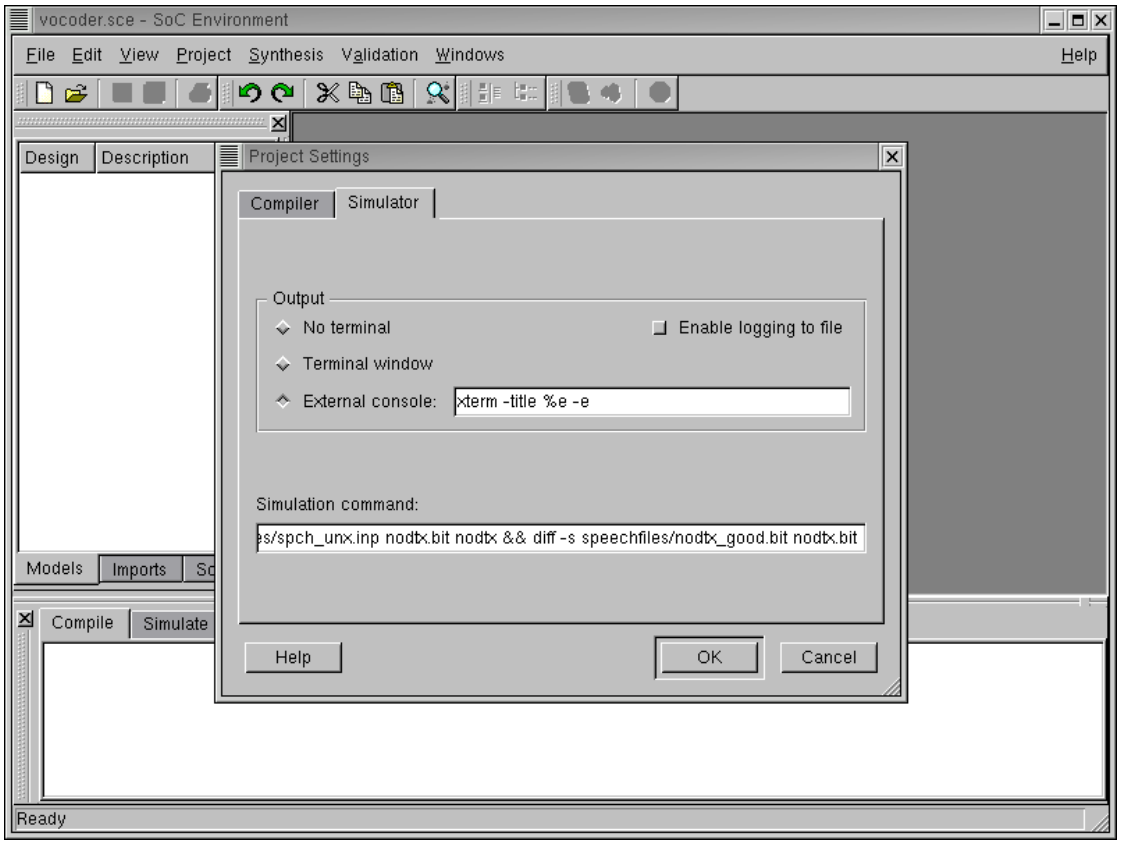
Since we need to ensure that the paths to dependencies are correctly set, We now check the settings for this precreated Vocoder project by selecting Project$\longrightarrow$Settings from the top menu bar.

2.2.3. Open project (cont'd)



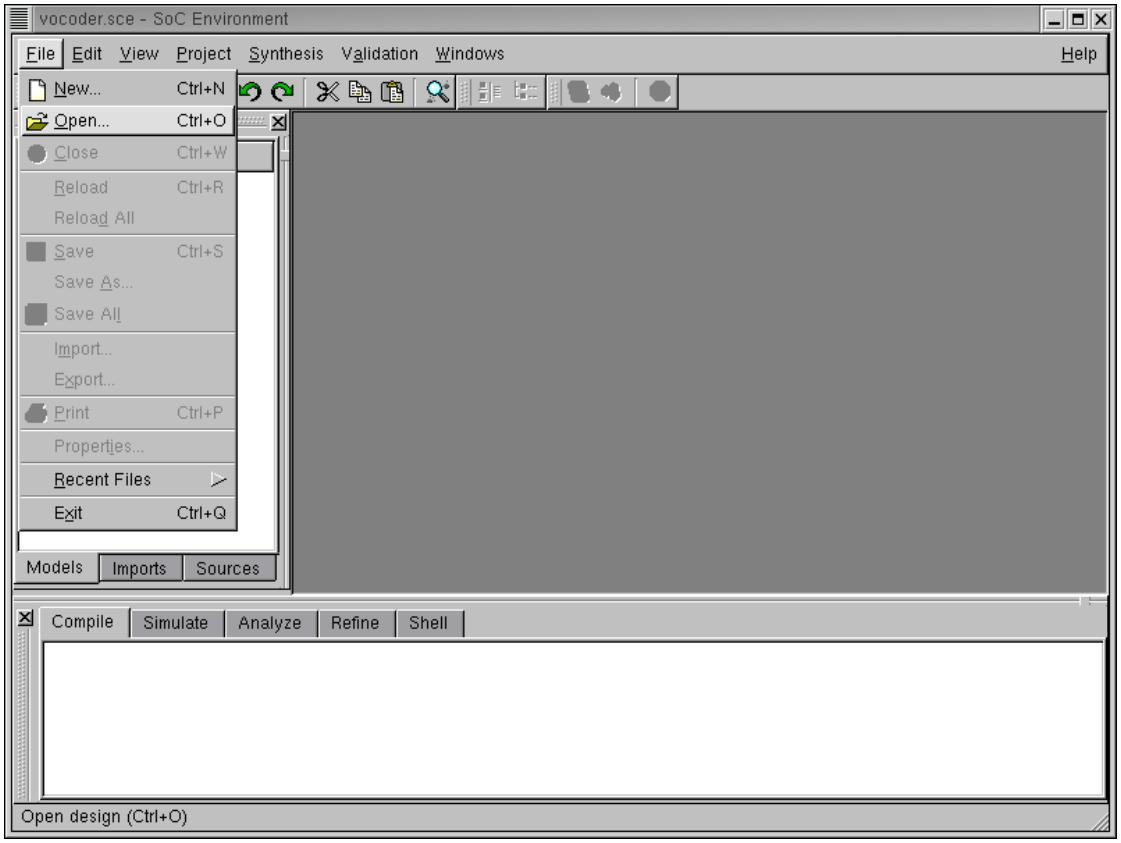
We now see the compiler settings showing the import path for the model's libraries and the '-v' (verbose) option. The Include path setting gives the path which is searched for header files. The import path is searched for files imported into the model. The library path is used for looking up the libraries used during compilation. There are also settings provided for specifying which libraries to link against, which macros to define and which to undefine. These settings basically form the compilation command. To check the simulator settings, Left click on the Simulator tab.

2.2.4. Open project (cont'd)



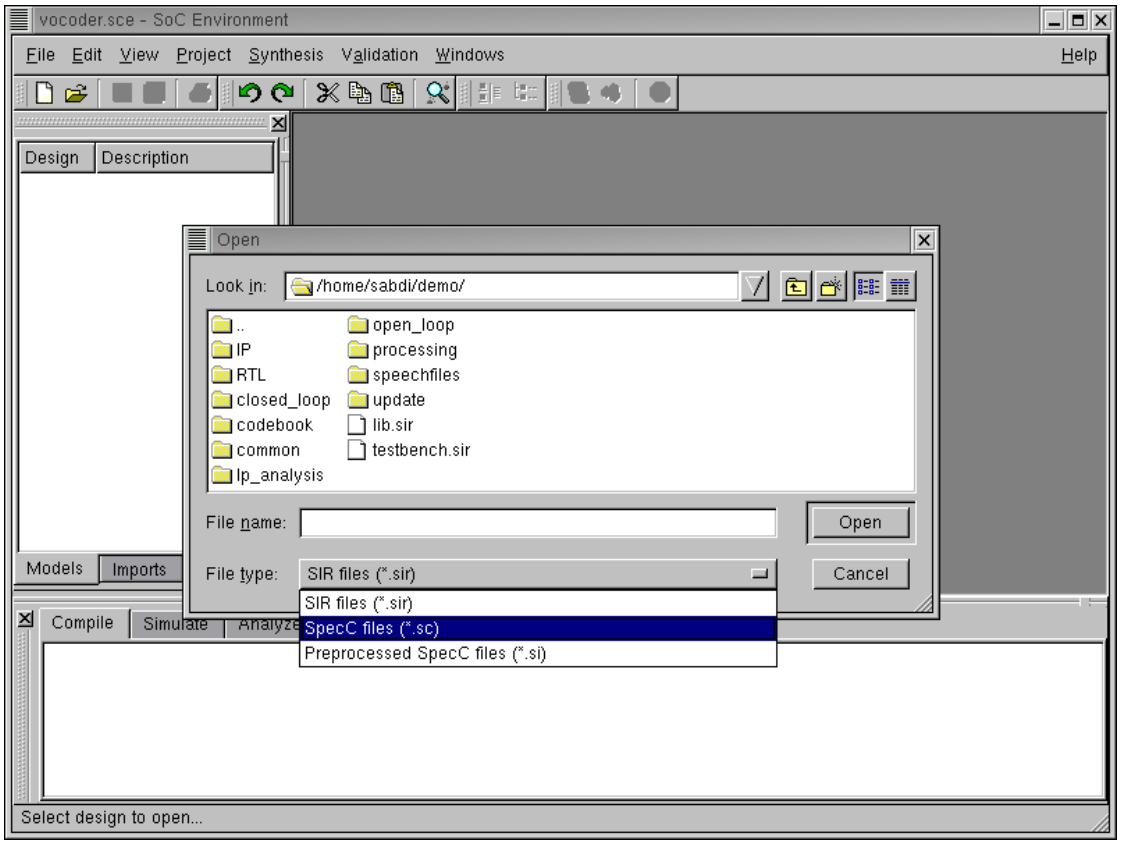
We now see the simulator settings showing the simulation command for the Vocoder model. There are settings available to direct the output of the model simulation. As can be seen, the simulation output may be directed to a terminal, logged to a file or dumped to an external console. For the demo, we direct the output of the simulation to an xterm. Also note that the simulation command may be specified in the settings. This command is invoked when the model is validated after compilation. The vocoder simulation processes 163 frames of speech and the output is matched against a golden file. Press OK to proceed.

2.3. Open specification model



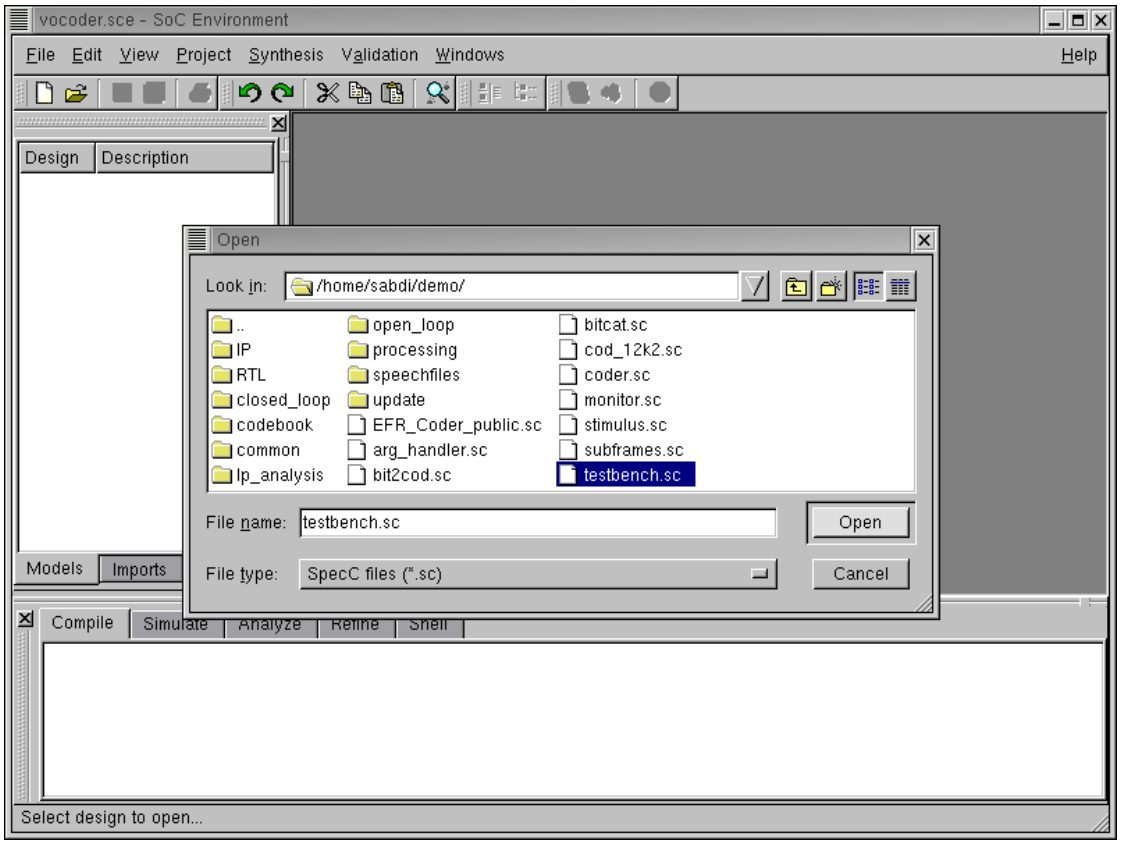
We start with the specification that was already captured as a model. We open this model to see if it meets the imposed constraints. Once the model is validated to be "golden", we will start refining it and adding implementation details to it. We open the specification model for the Vocoder example by selecting File—→Open from the menu bar.

2.3.1. Open specification model (cont'd)



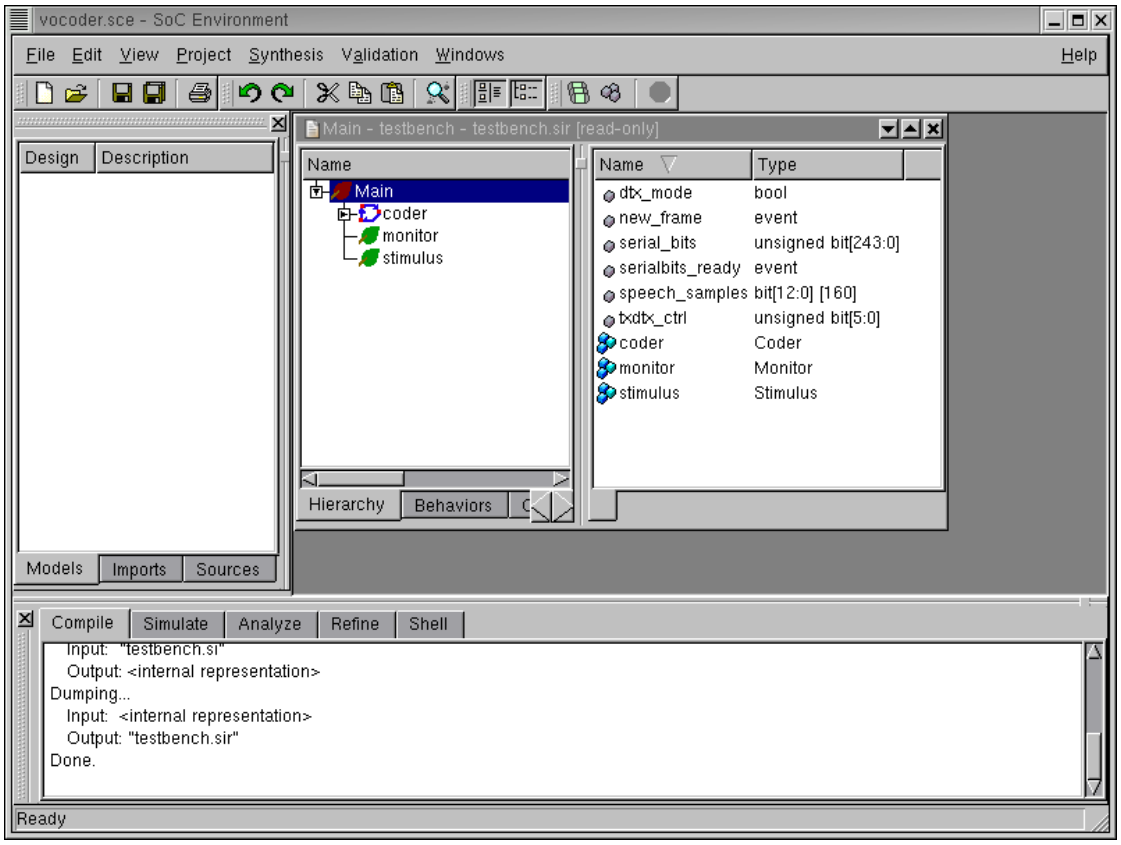
A file selection window pops up showing the SpecC internal representation (SIR) files. The internal representation files are a collection of data structures used by the tools in the environment. They uniquely identify a SpecC model. At this time however, the design is available only in its source form. We therefore need to start with the sources. Select "SpecC files (*.sc)" to view the source files.

2.3.2. Open specification model (cont'd)



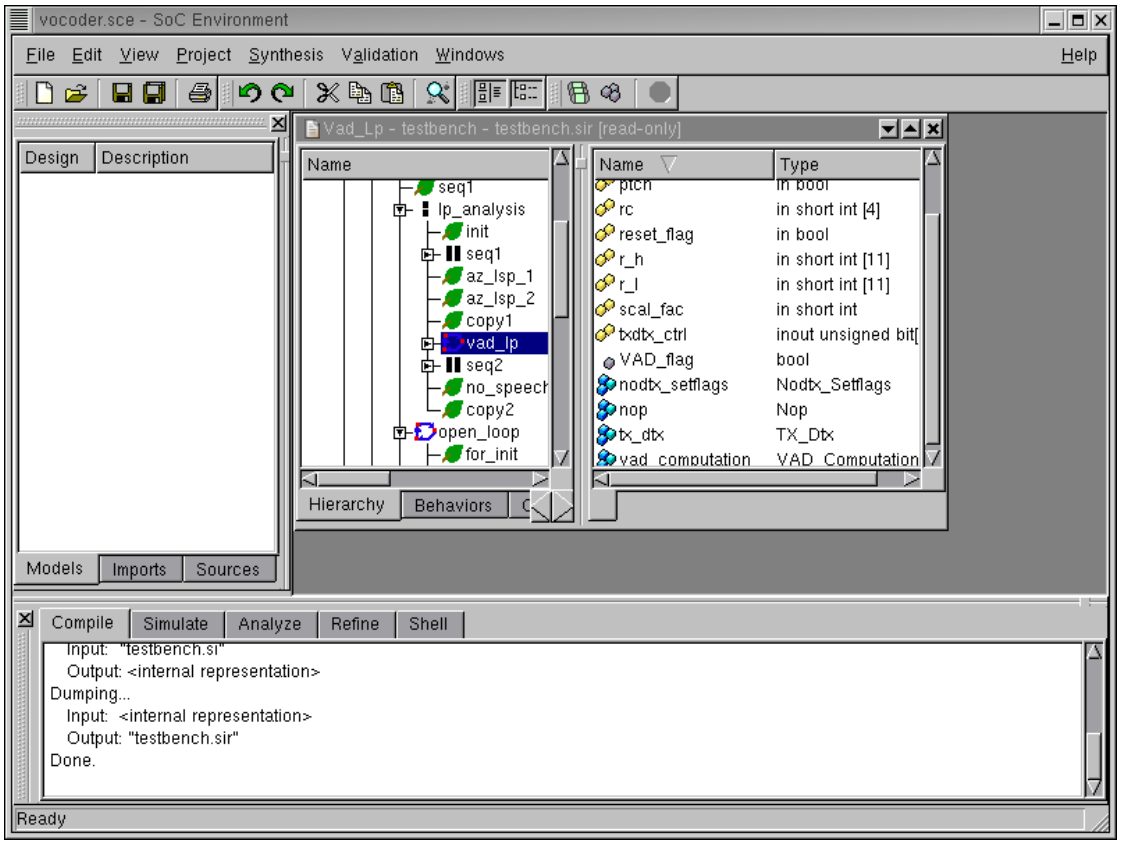
A new window pops up showing the available source files of the GSM Vocoder design specification. Select the file containing the top hierarchy of the model. In this case, the file is "testbench.sc." The testbench instantiates the Design under Test (DUT) and the corresponding modules for triggering the testvectors and for observing the outputs. To open this file Left click on Open.

2.3.3. Open specification model (cont'd)



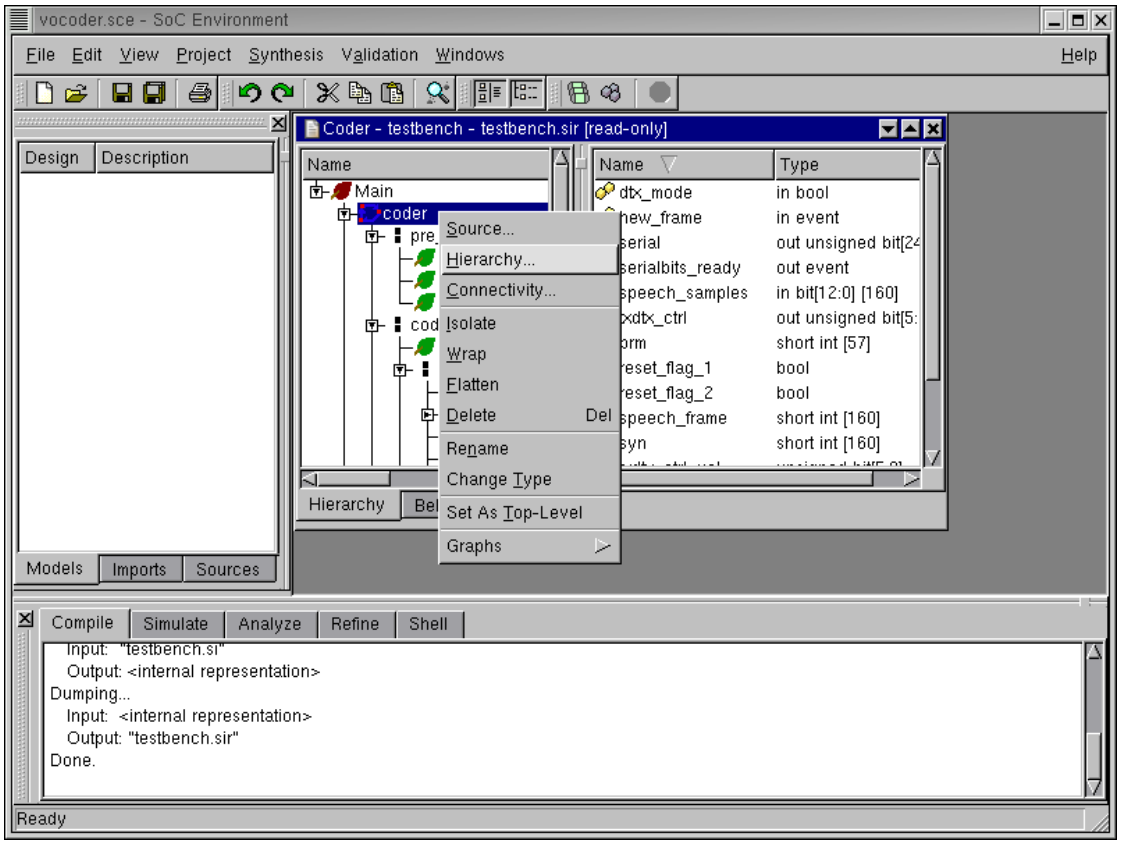
Note that a new window pops up in the design management area. It has two sub-windows. The sub-window on the left shows the Vocoder design hierarchy. The leaf behaviors are shown with a leaf icon next to them. For instance, we see two leaf behaviors: "stimulus" which is used to feed the test vectors to the design and "monitor" which validates the response. "Coder" is the top behavior of the Vocoder model. It can be seen from the icon besides the "Coder" behavior that it is an FSM composition. This means the Vocoder specification is captured as a finite state machine. Also note in the logging window that the SoC design has been compiled into an intermediate format. Upon opening a source file into the design window, it is automatically compiled into its unique internal representation files (SIR) which in turn is used by the tools that work on the model.

2.3.4. Open specification model (cont'd)



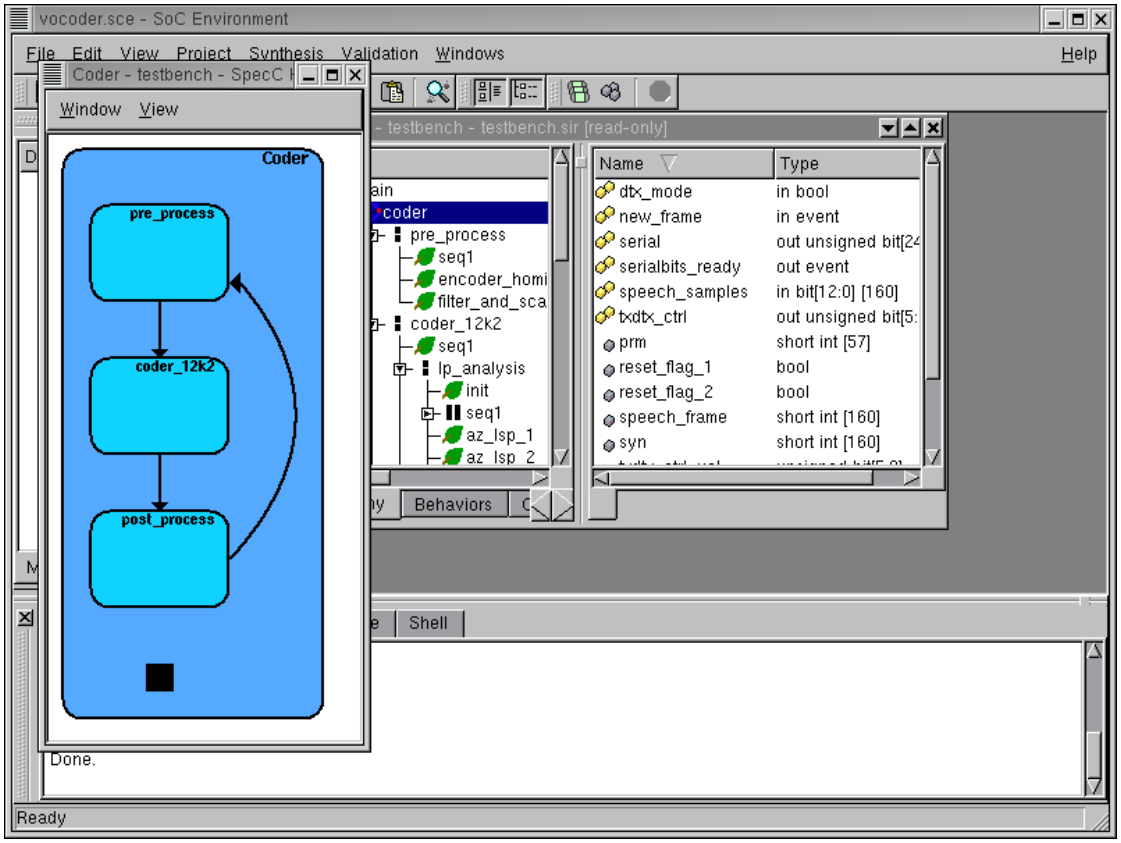
The model may be browsed using the design hierarchy window. Parallel composition is shown with || shaped icons and sequential composition with ':' shaped icons.On selecting a behavior in the design hierarchy window, we can see the behavior's characterstics in the right sub-window. For instance, the behavior "vad_lp" has ports shown by yellow icons, variables with grey icons and sub-behaviors with blue icons.

2.3.5. Open specification model (cont'd)



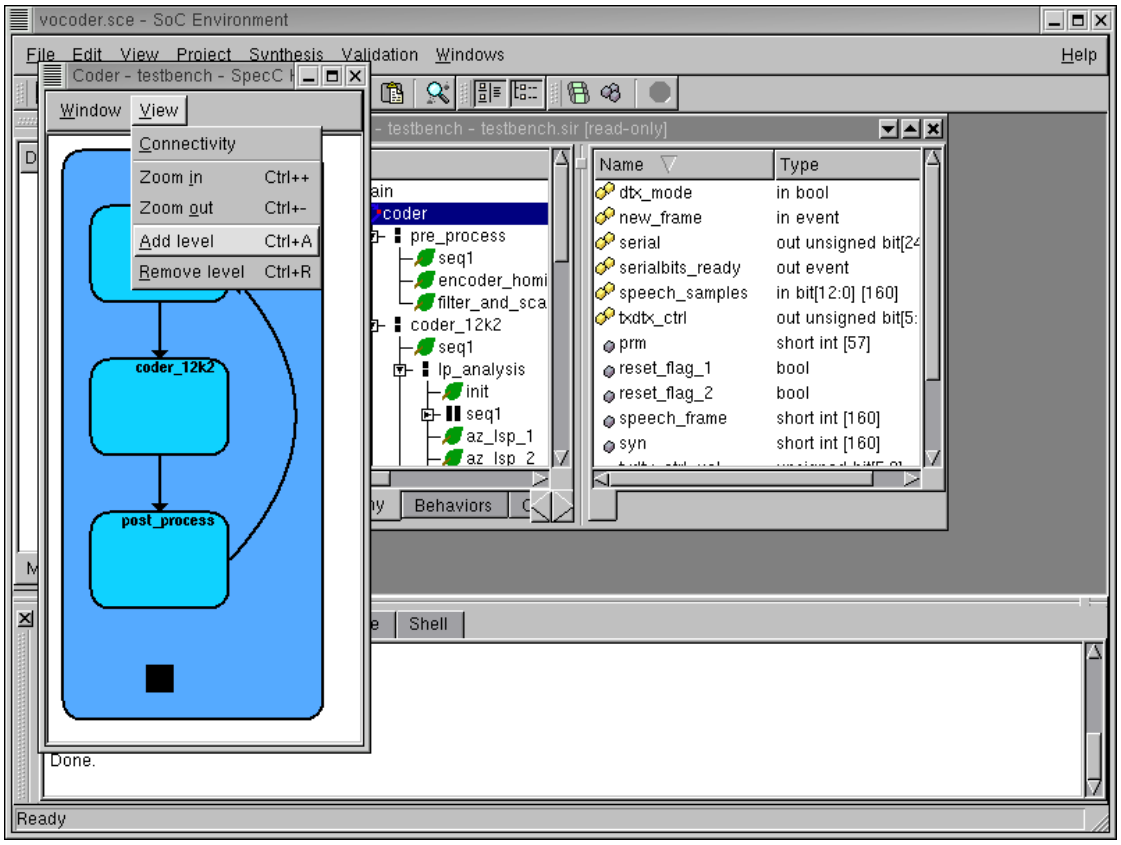
Before making any synthesis decisions, it is important to understand the composition of the specification model. It is useful because the composition really tells us which features of the model may be exploited to gain maximum productivity. Naturally, the most intuitive way to understand a model's structure is through a graphical representation. Since system models are typically very complex, its more convenient to have a hierarchical view which may be easily traversed. SCE provides for such a mechanism. To graphically view the hierarchy, from the design hierarchy window, select "Coder". Right click and select Hierarchy. Notice that the menu provides for a variety of services on individual behaviors. We shall be using one or more of these in due course.

2.4. Browse specification model



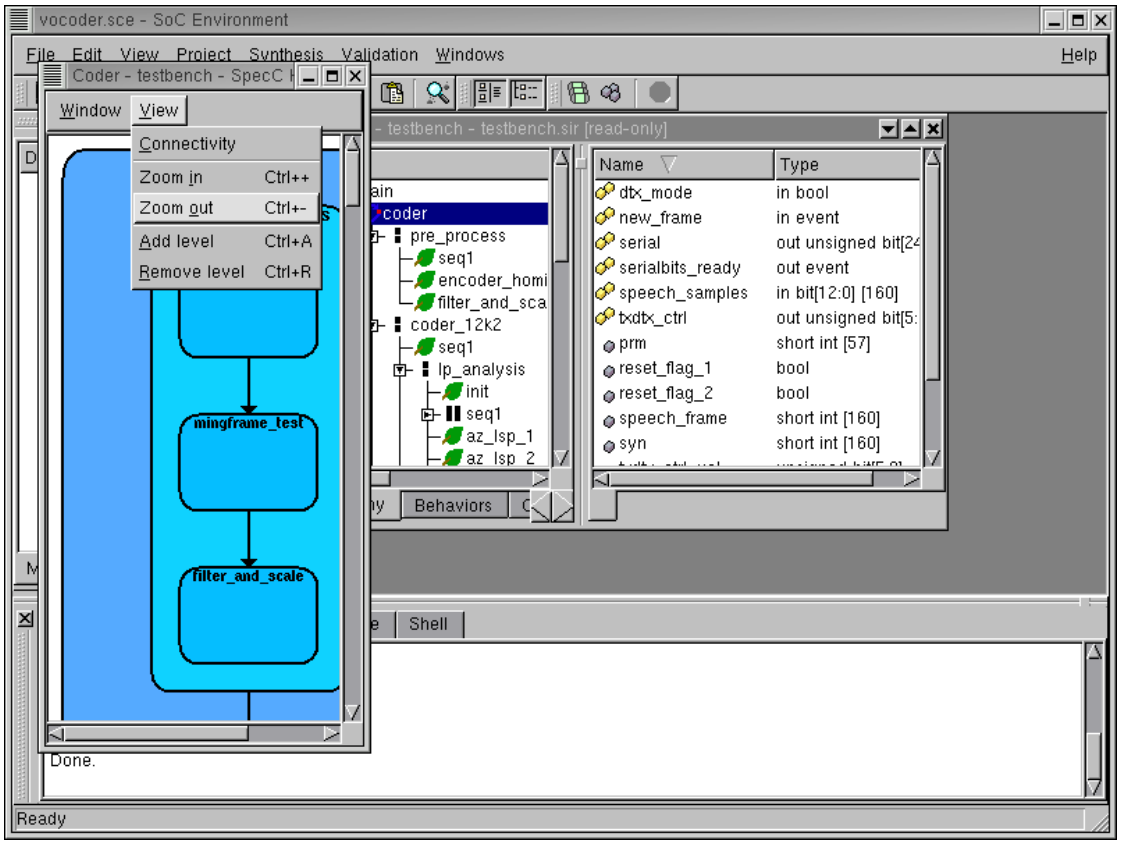
A new window pops up showing the Vocoder model in graphical form. As noted earlier, the specification is an FSM at the top level with three states of pre-processing, the bulk of the coder functionality itself and finally post-processing.

2.4.1. Browse specification model (cont'd)



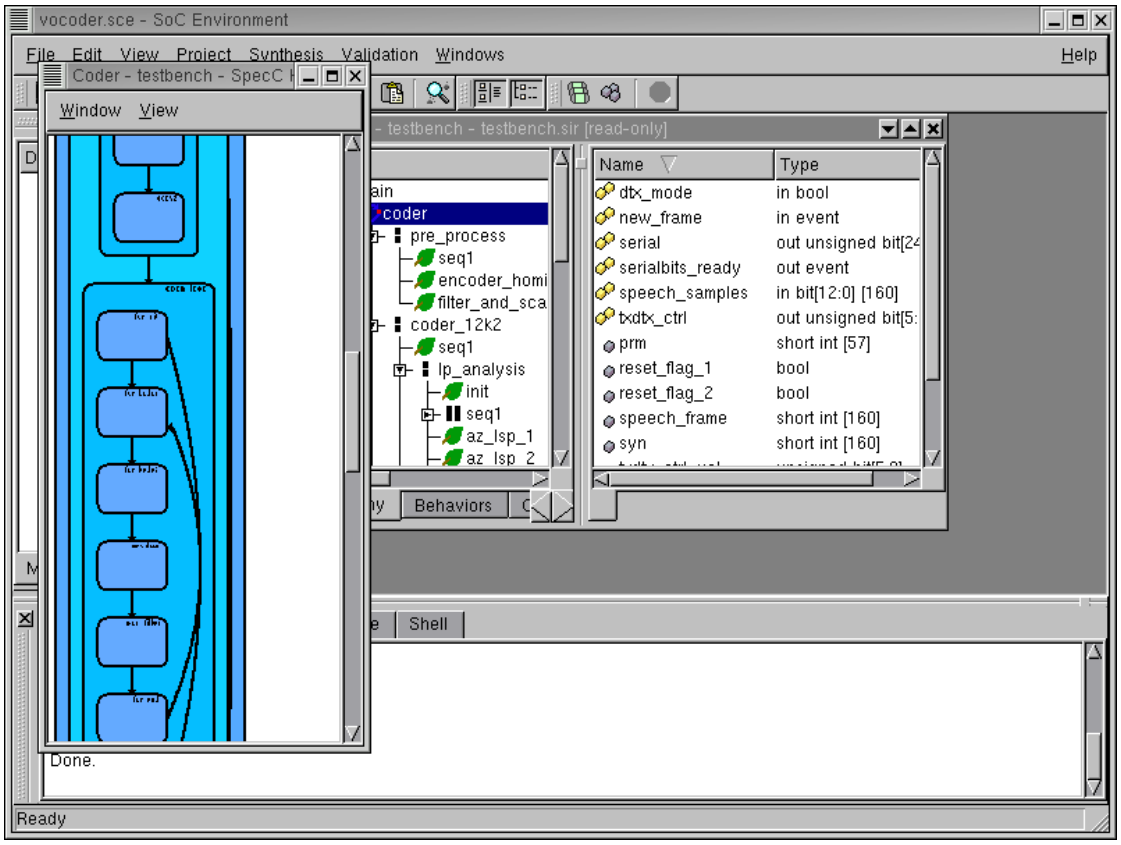
At this stage, we would like to delve into greater detail of the specification. To view the model graphically with higher detail, select View—→Add level Perform this action TWICE to get a more detailed view. As can be seen, the View menu provides features like displaying connectivity of behaviors, modifying detail level and zooming in and out to get a better view.

2.4.2. Browse specification model (cont'd)



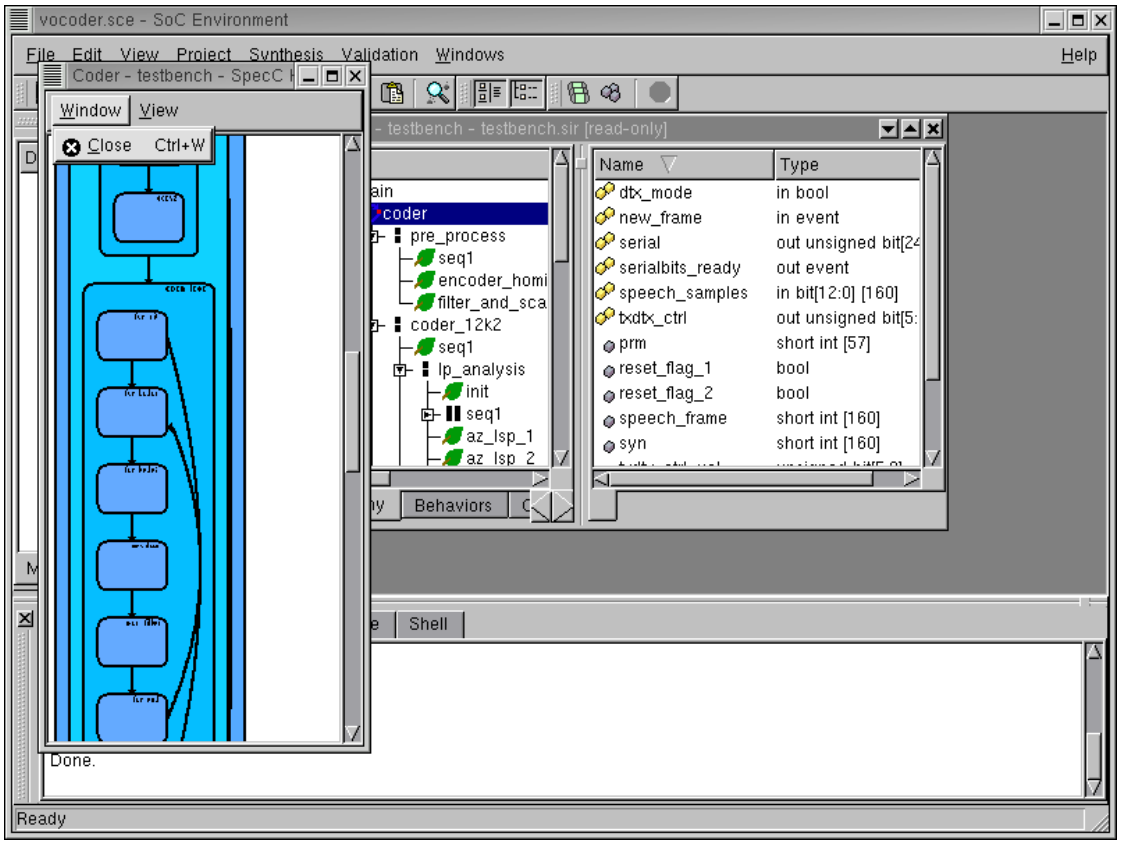
Zoom out to get a better view by selecting View⟶Zoom out

2.4.3. Browse specification model (cont'd)



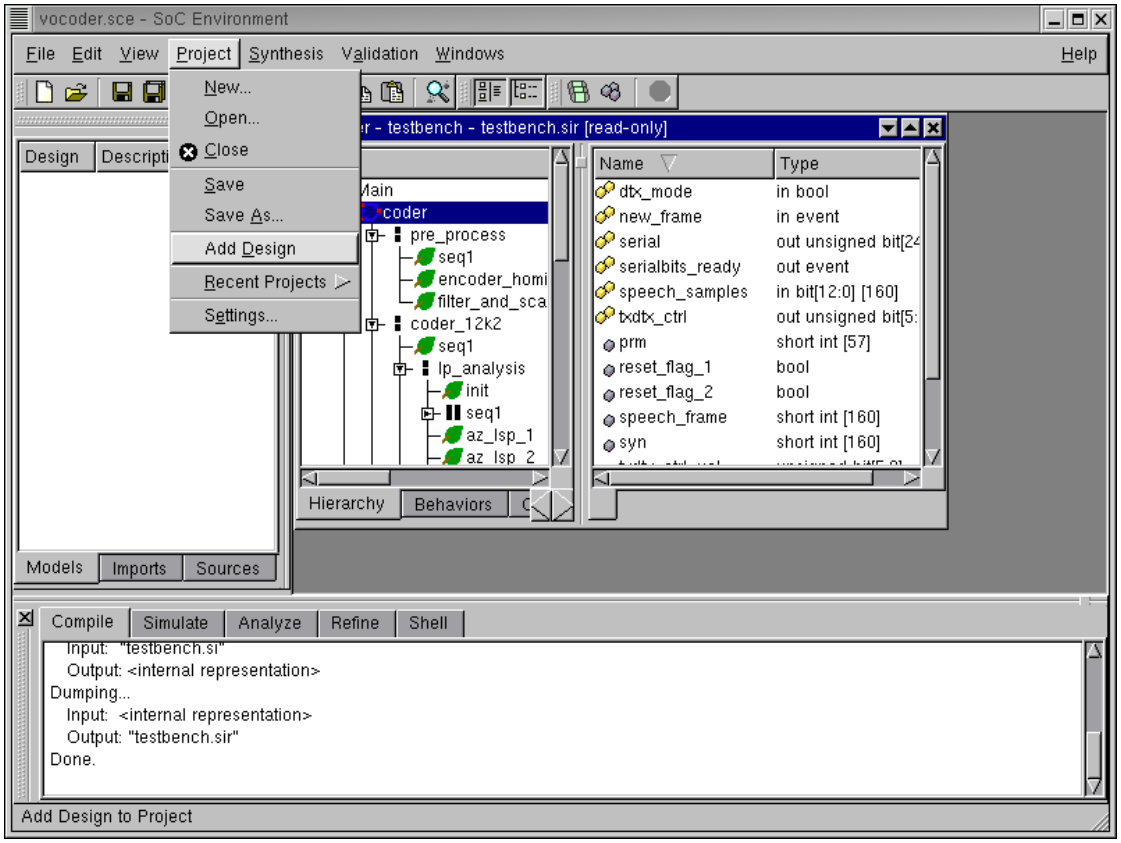
Scroll down the window to see the FSM and sequential composition of the Vocoder model. Note that the specification model of the GSM vocoder does not contain much parallelism. Instead, many behaviors are sequentially executed. This is due to the several data dependencies in the code. For our implementation, this is an important observation. Since there is not much parallelism in the code to exploit, speedup can be achieved only by use of faster components. One way to speed up is to use dedicated hardware units.

2.4.4. Browse specification model (cont'd)



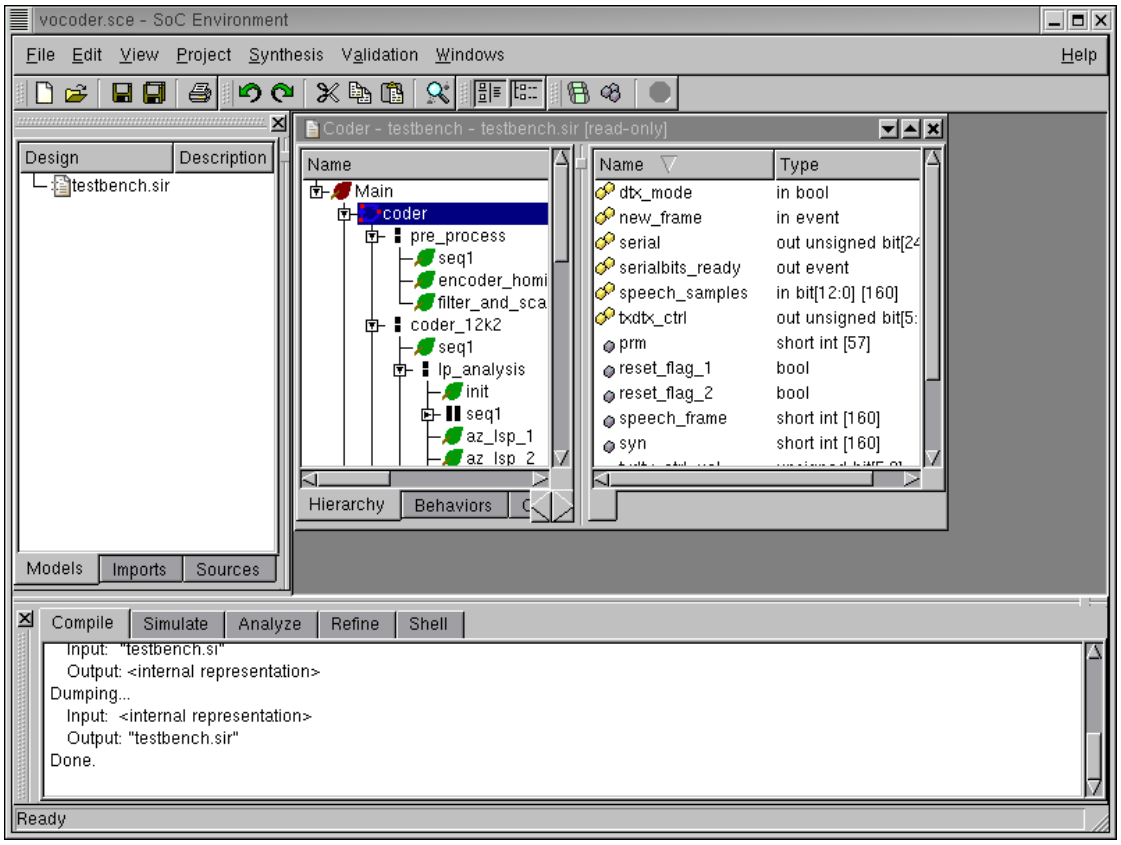
Exit the hierarchy browser by selecting Window⟶Close

2.5. Validate specification model



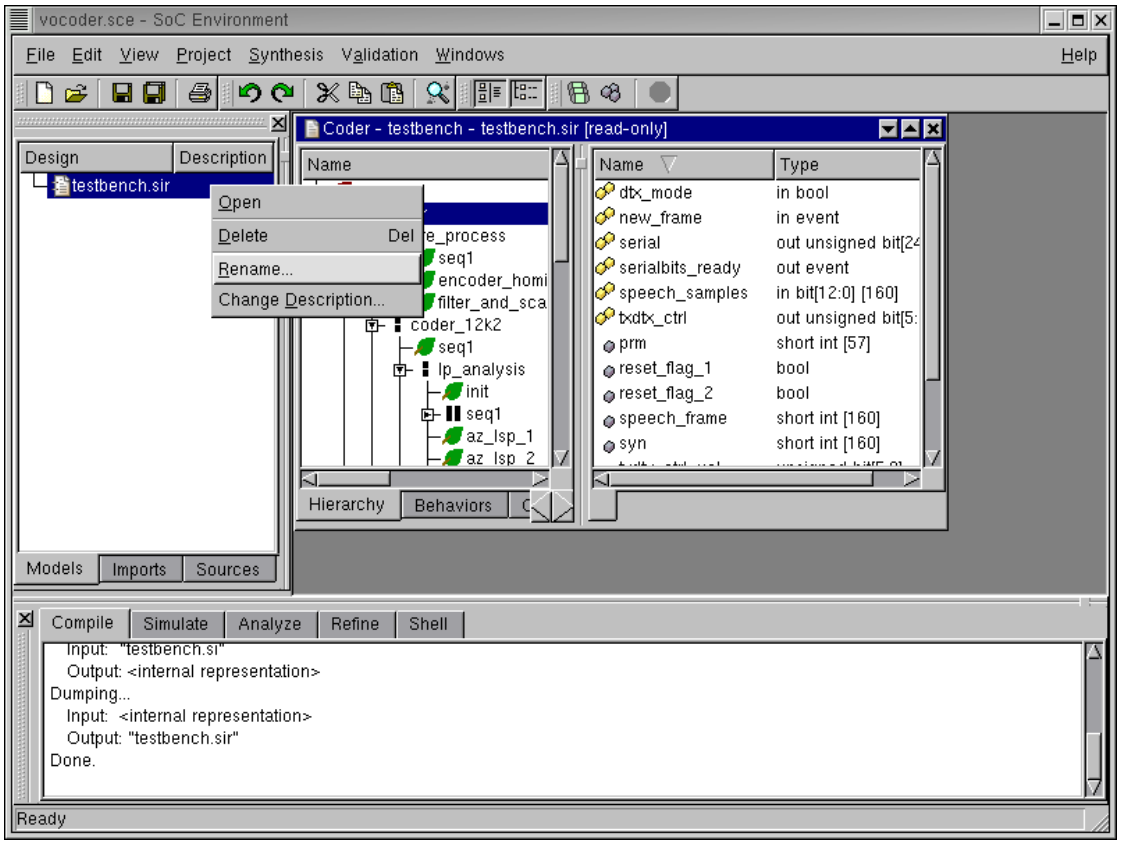
We must now proceed to validate the Specification Model. Remember that we have a "golden" output for encoding of the 163 frames of speech. The specification model would meet its requirements if we can simulate it to produce an exact match with the golden output. In practice, a more rigorous validation process is involved. However, for the purpose of the tutorial, we will limit ourselves to one simulation only. Start with adding the current design to our Vocoder project by selecting : Project—→Add Design

2.5.1. Validate specification model (cont'd)



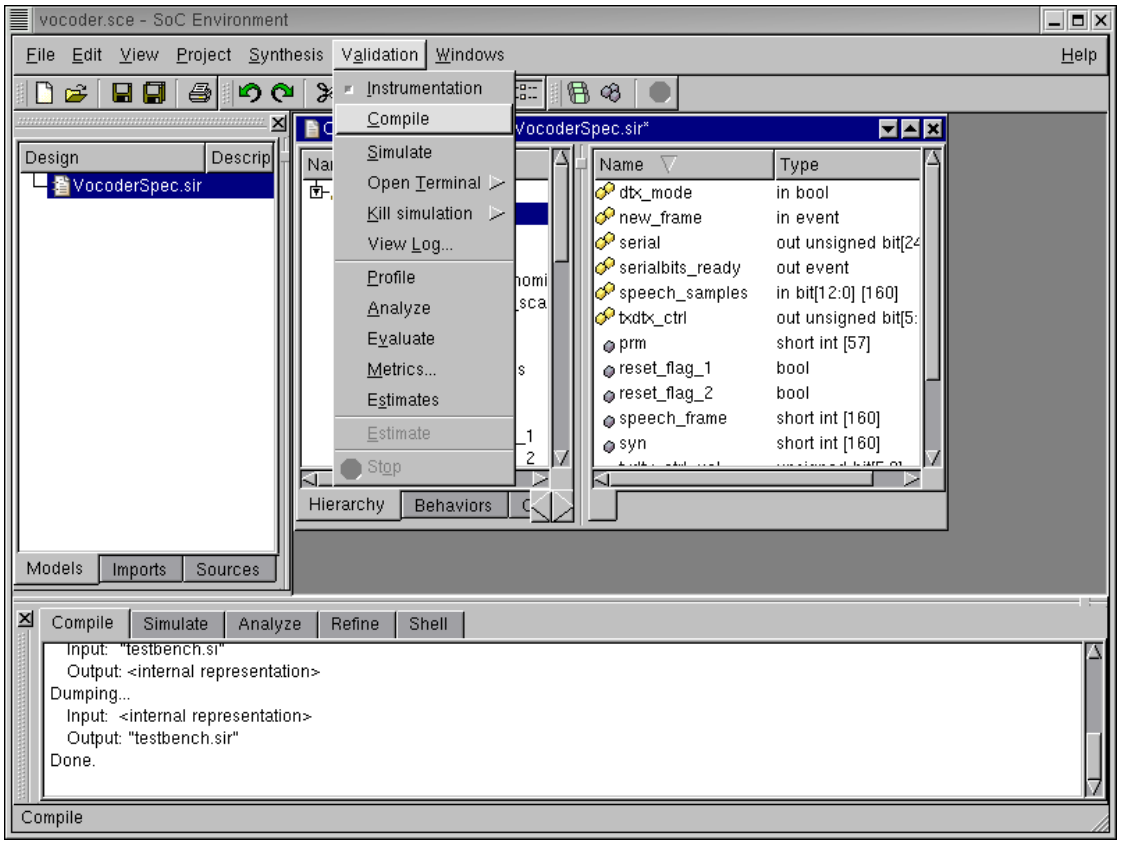
The project is now added as seen in the Project Management workspace on the left in the GUI.

2.5.2. Validate specification model (cont'd)



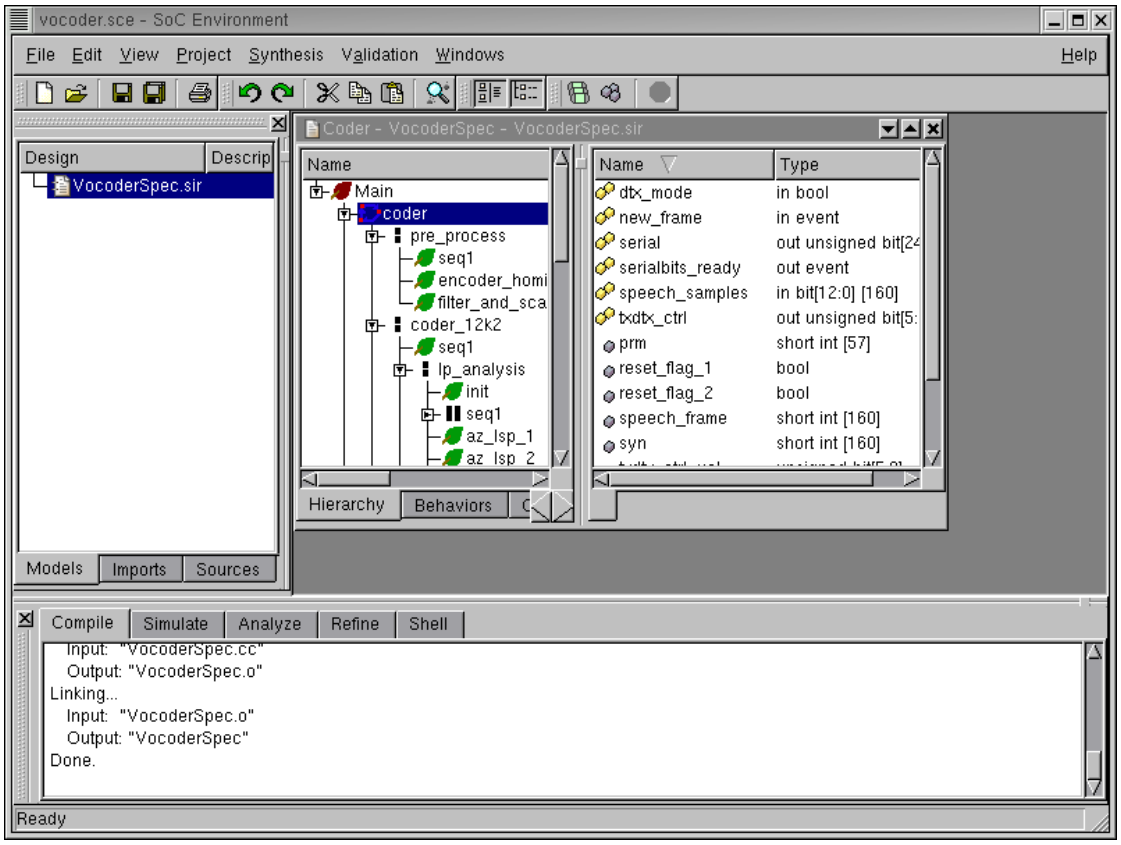
We must now rename the project to have a suitable name. Remember that our methodology involved 4 models at different levels of abstraction. As these new models are produced, we need to keep track of them. Right click on "testbench.sir" and select Rename Rename the design to VocoderSpec. This indicates that the current model corresponds to the topmost level of abstraction namely the Specification level. Note that the extension ".sir" would be automatically appended. Also note that a model may be made active (Open), Deleted, Renamed and described by Right click on its name in the Project window.

2.5.3. Validate specification model (cont'd)



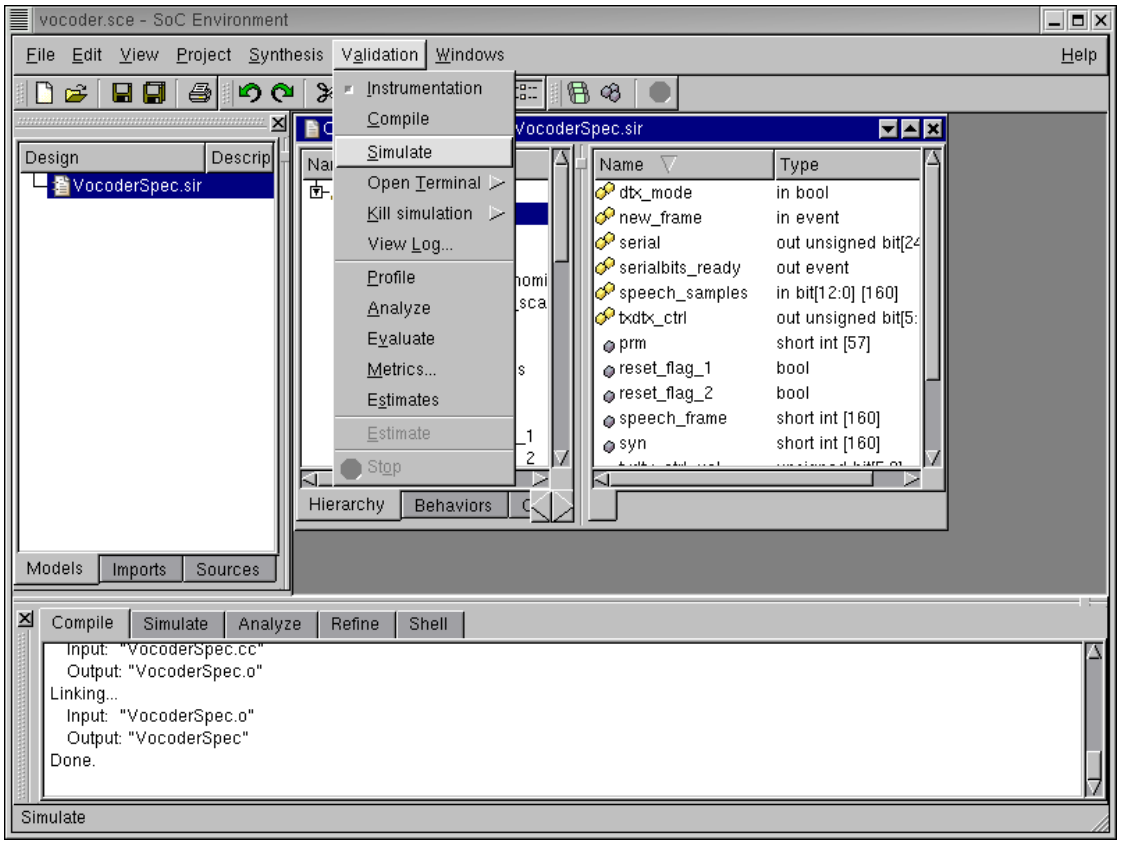
After the project is renamed to VocoderSpec.sir, we need to compile it to produce an executable. Since we are in the phase of validating the specification model, we should make it executable by compiling it. This may be done by selecting : Validation$\longrightarrow$Compile Note that the validation menu also provides for Code instrumentation which is used for profiling. Moreover, we have choices for simulating the model, Opening a simulation terminal. Killing an existing simulation, Viewing the log, Profiling, Analyzing simulation results, Model Evaluation, displaying Metrics and Estimates etc. All these features will be used in due course of our system design process.

2.5.4. Validate specification model (cont'd)



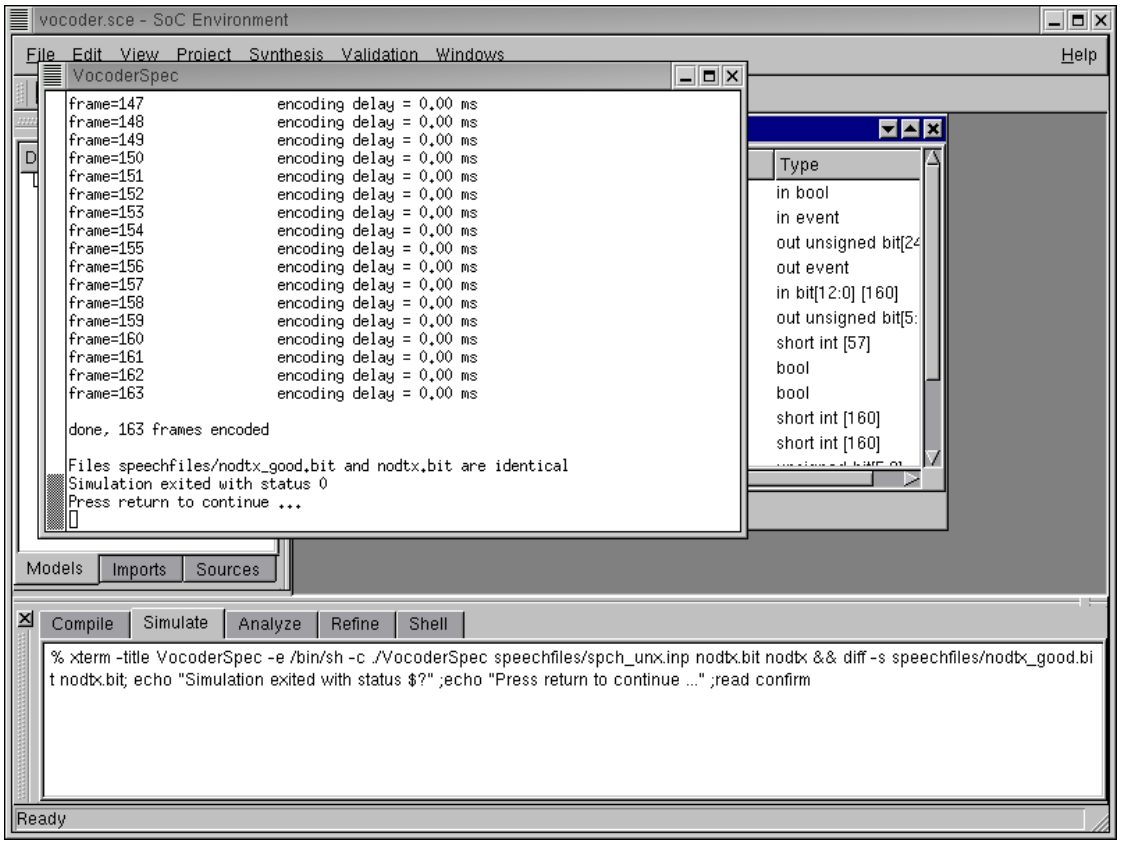
Note that in the logging window we see the compilation messages and an output executable "VocoderSpec" is created.

2.5.5. Validate specification model (cont'd)



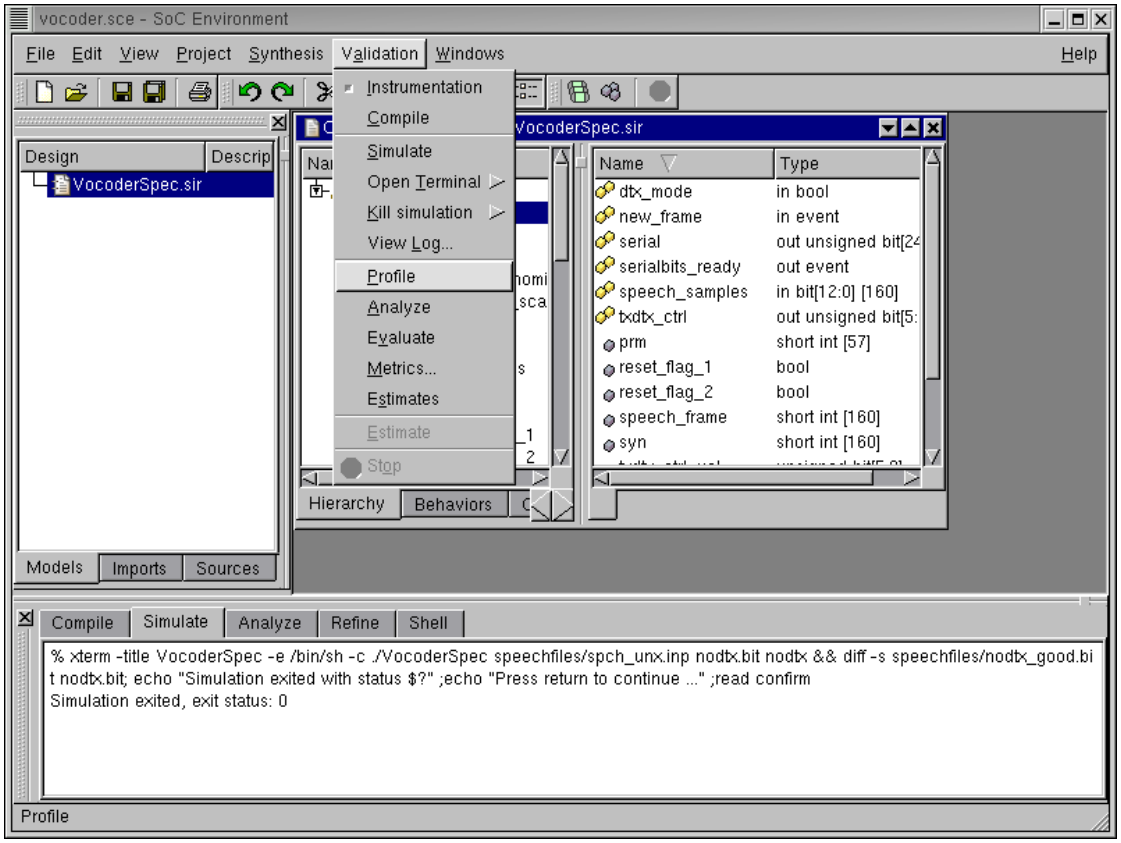
The next step is to simulate the model to verify whether it meets our requirements or not. This may be done by selecting : Validation$\longrightarrow$Simulate

2.5.6. Validate specification model (cont'd)



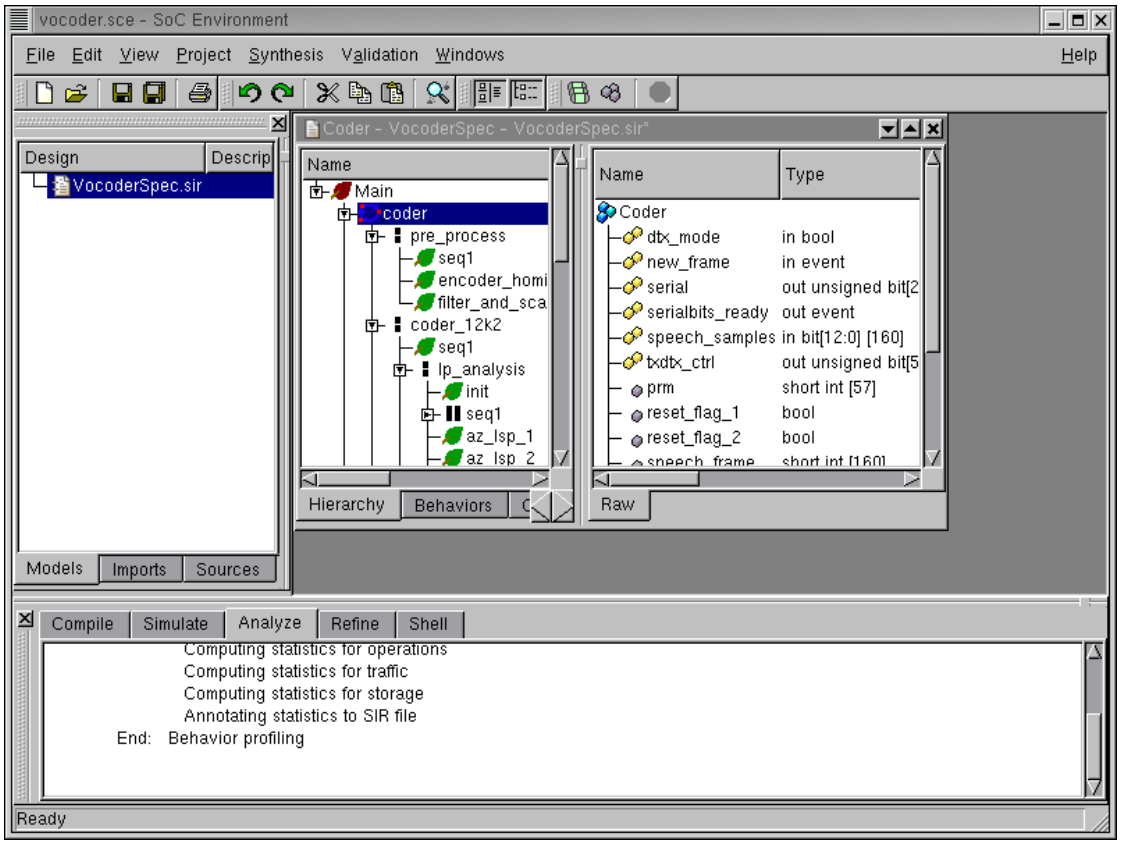
Note that an xterm pops up showing the simulation of the Vocoder specification model on a 163 frame speech sample. The simulation should finish correctly which is indicated by the exit status being '0'. It can be seen that 163 speech frames were correctly simulated and the resulting bit file matches the one given with the vocoder standard. It may be noted that each frame has an encoding delay of 0 ms. This is a consequence of the fact that our specification model has no notion of timing. As explained in the methodology, the specification is a purely functional representation of the design and is devoid of timing. For this reason, all behaviors in the model execute in 0 time thereby giving and encoding delay of 0 for each frame. Press RETURN to close this window and proceed to the next step.

2.6. Profile specification model



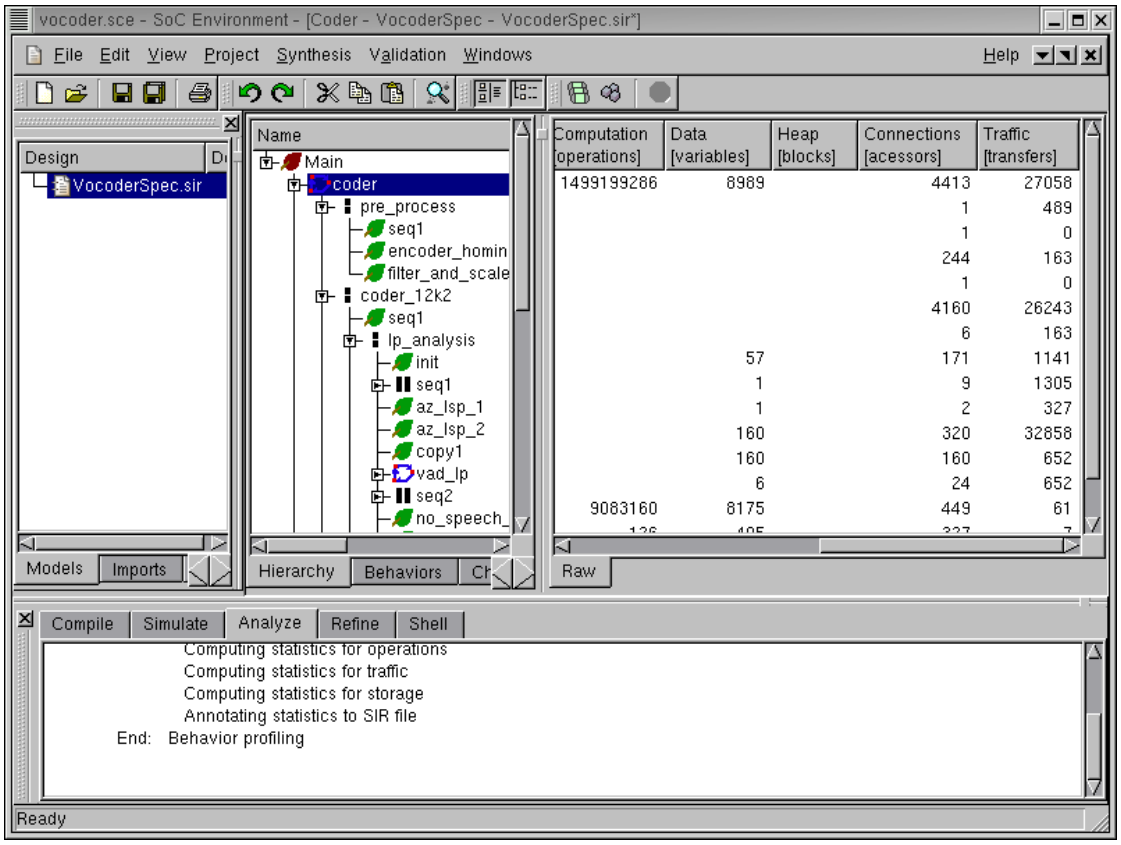
In order to select the right architecture for implementing the model, we must begin by profiling the specification model. Profiling provides us with useful data needed for comparative analysis of various modules in the design. It also counts the various metrics like number of operations, class and type of operation, data exchanged between behaviors etc. These statistics are collected during simulation. Profiling may be done by selecting : Validation$\longrightarrow$Profile

2.6.1. Profile specification model (cont'd)



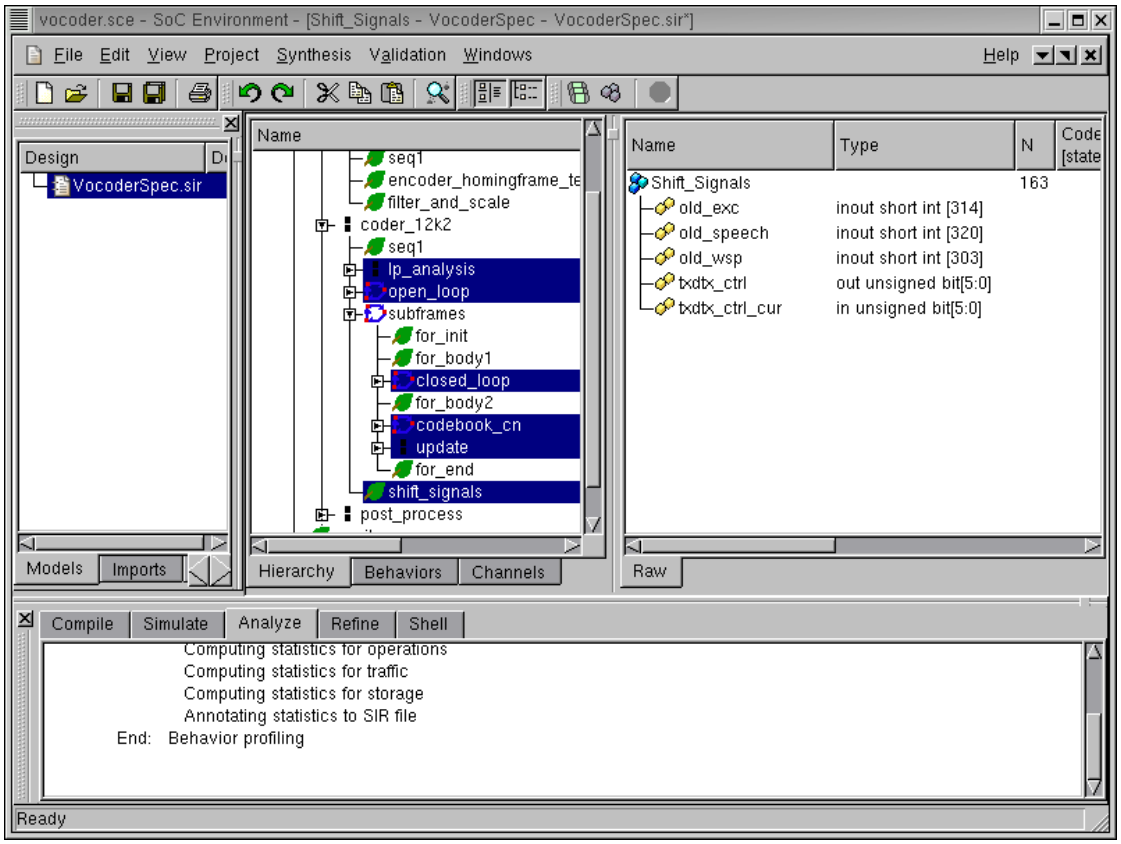
The logging window now shows the results of the profiling command. Note that there is a series of steps for computing statistics for individual metrics like operations, traffic, storage etc. Once these statistics are computed. they are annotated to the model and displayed in the design window.

2.6.2. Profile specification model (cont'd)



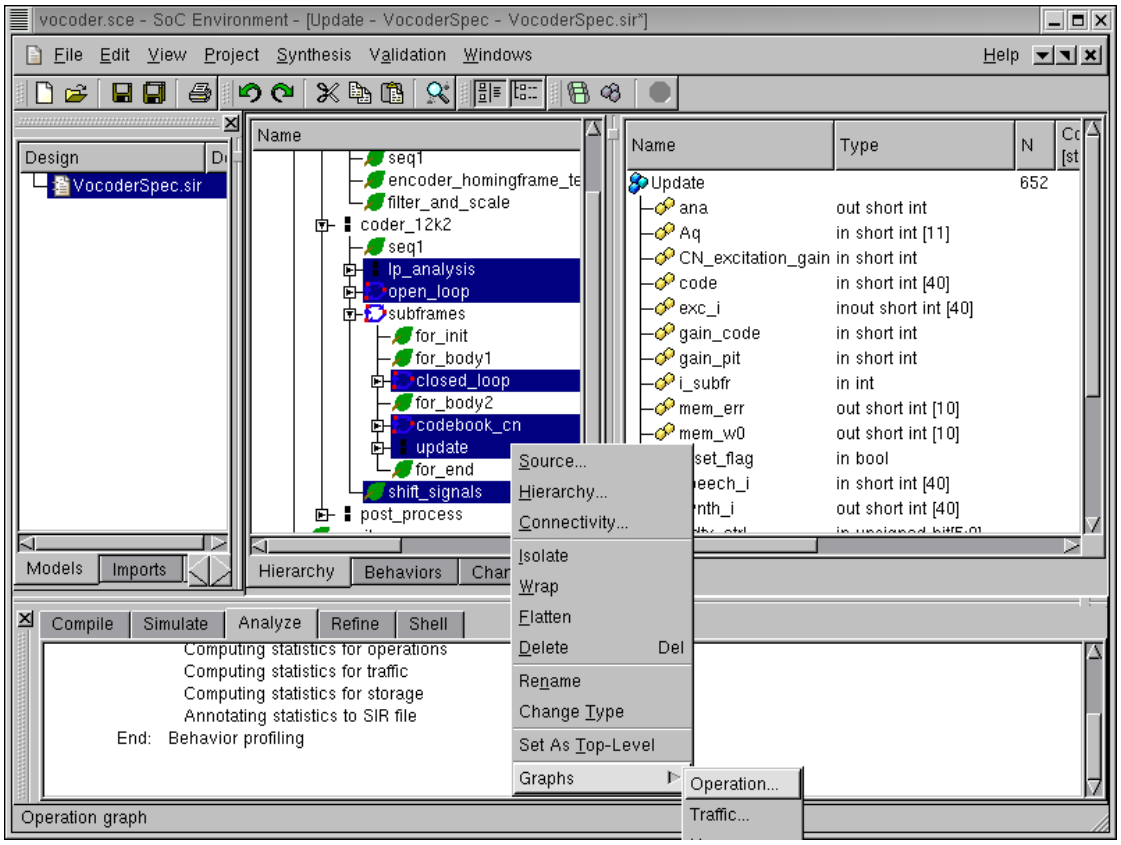
It may also be noted that the design management window now has new column entries that contain the profile data. Maximize this window and scroll to the right to see various metrics for behaviors selected in the design hierarchy. The current screenshot shows "Computation", "Data", "Connections" and "Traffic" for the top behavior "Coder". Computation essentially means the number of operations in each of the behaviors. Data refers to the amount of memory required by the behaviors. Connections indicate the presence of inter-behavior channels or connection through variables. Traffic refers to the actual amount of data exchanged between behaviors. The metrics may also be obtained for other behaviors in the design besides the coder.

2.7. Analyze profiling results



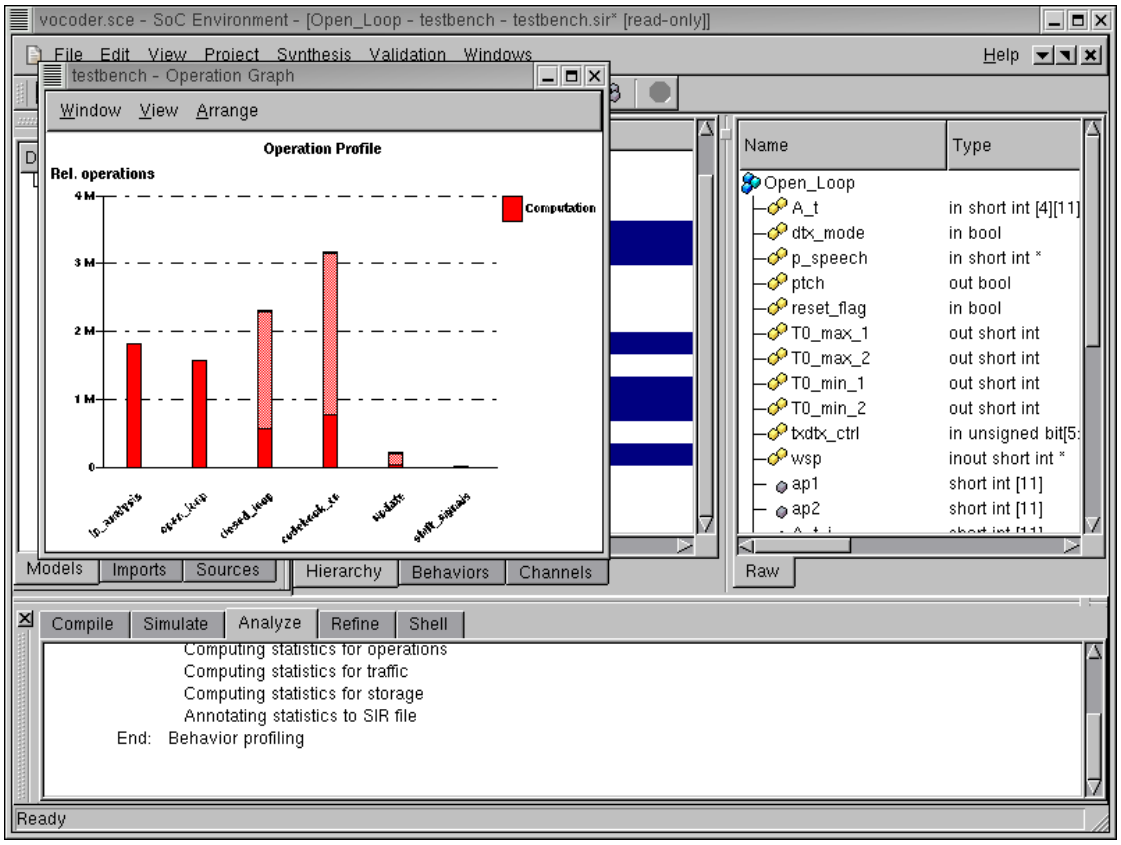
Once we have the profiling results, we need a comparative analyses of the various behaviors to enable suitable partitioning. Here we analyze the six most computationally intensive behaviors viz. "lp_analysis", "open_loop", "closed_loop", "codebook_cn", "update" and "shift_signals." They may be multi-selected in the design hierarchy by pressing CNTRL key and Left clicking on them. These particular behaviors were selected because these are the major blocks in the coder_12k2 behavior, which in turn is the central block of the entire coder. Thus the selected behaviors show essentially the major part of the activity in the coder. We ignore the pre-processing and the post-processing blocks, because they are of relatively lower importance.

2.7.1. Analyze profiling results (cont'd)



In order to select a suitable architecture for the Vocoder, we must perform not only an absolute but a comparative study of the computation requirements of the selected behaviors. SCE provides for graphical view of profiling statistics which may be used for this purpose. After the multi-selection, we Right click and select: Graphs$\longrightarrow$Operation

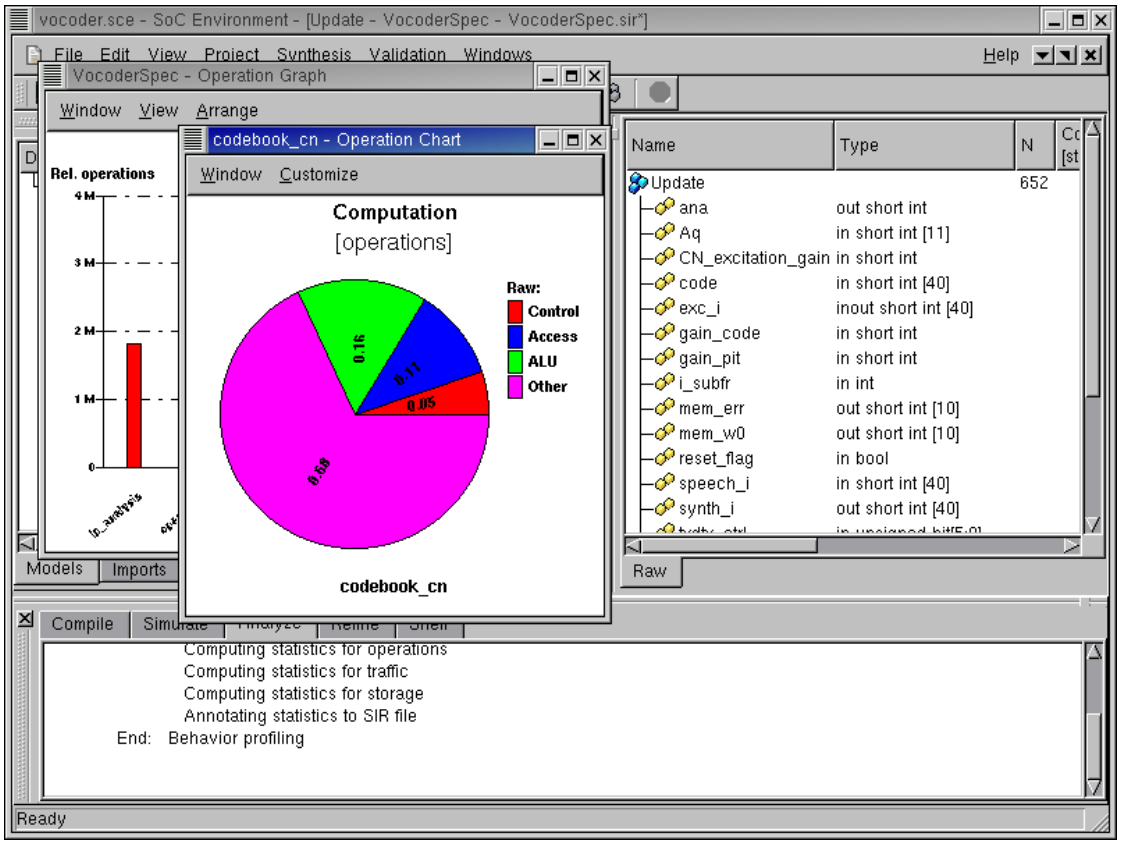
2.7.2. Analyze profiling results (cont'd)



We now see a bar graph showing the relative computational intensity of the various behaviors in the selected behaviors. Essentially, the graph shows the number of operations on the Y-axis for the individual behaviors on the X-axis. Double Left click on the bar for codebook_cn to view the distribution of its various operations. Note that we select "codebook_cn" because it is the behavior with the most computational complexity.

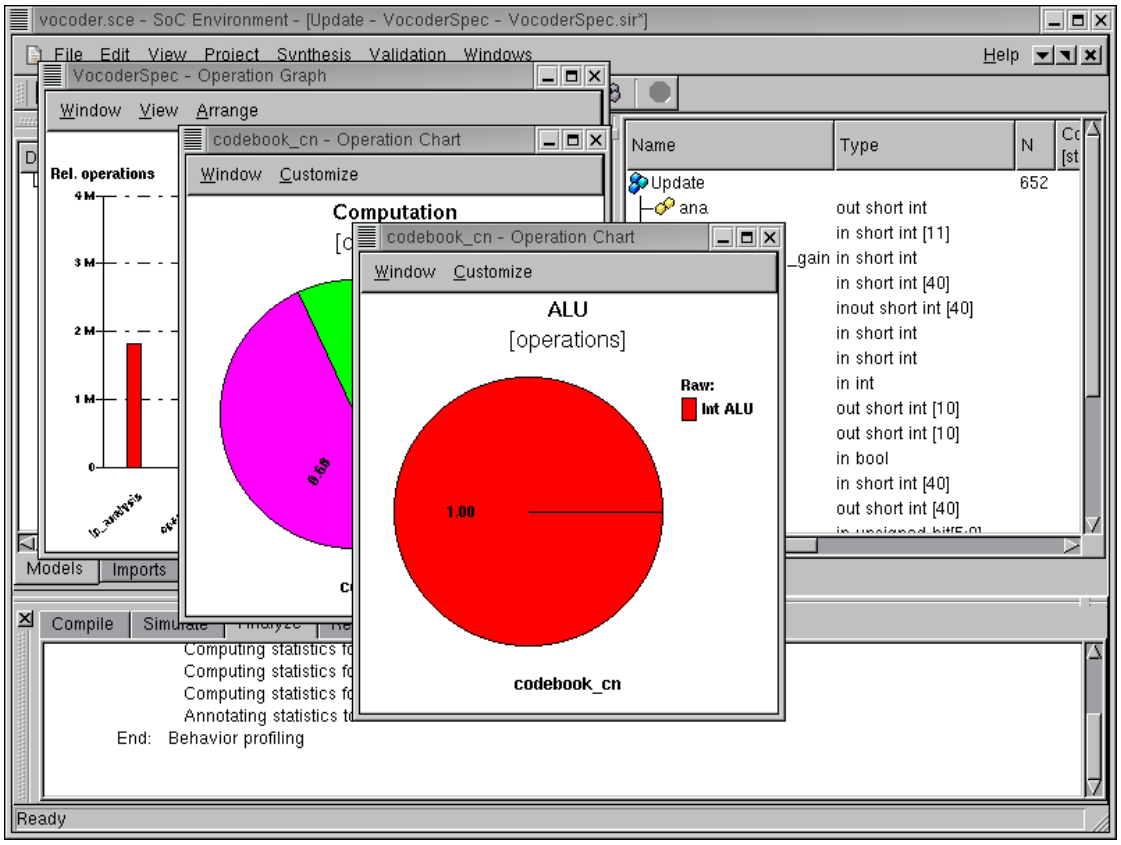
Note that the bars representing the computation for "codebook_cn" and "closed_loop" have two sections. The lower section is filled with red color and the upper section is partially shaded. Each speech frame consists of four sub-frames and the behaviors "codebook_cn" and "closed_loop" are executed for each subframe in contrast to other behaviors in the graph, which are executed once. Hence the filled section of the bar represents computation for each execution of behavior and the complete bar (including the shaded section) represents computation for the entire frame.

2.7.3. Analyze profiling results (cont'd)



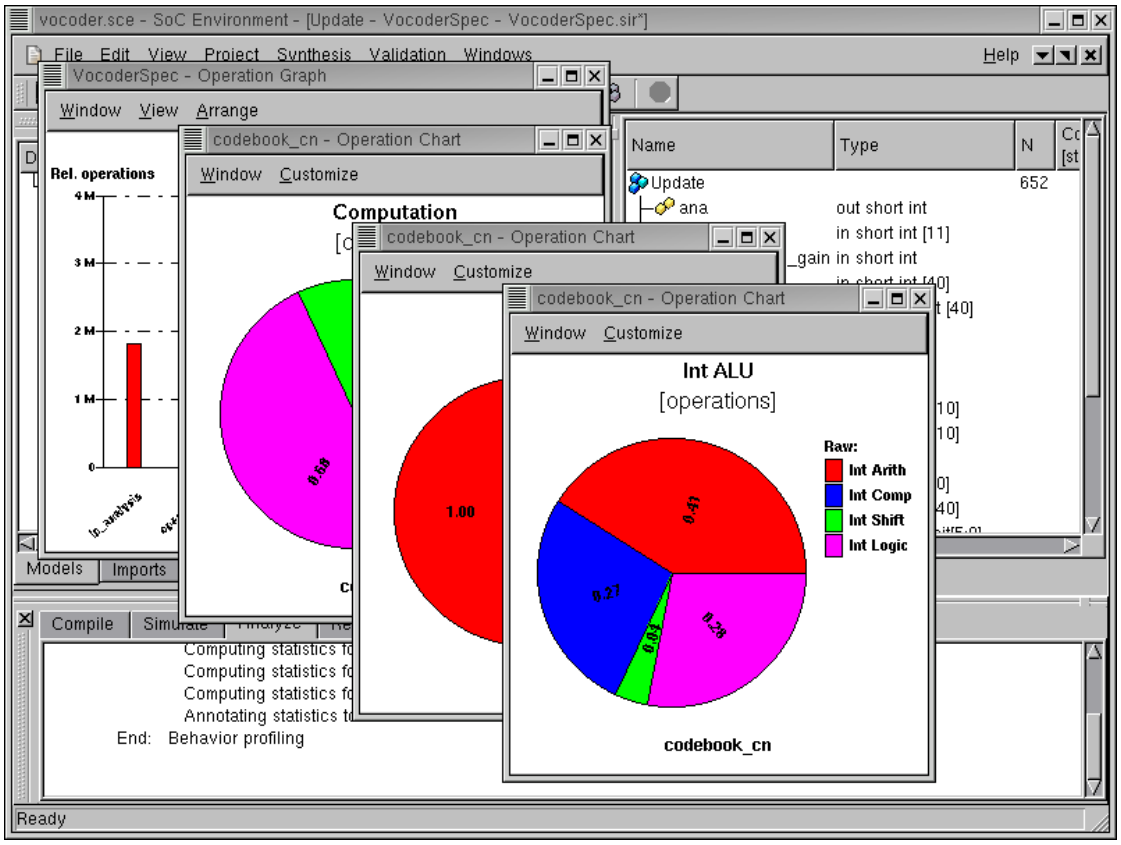
A new window pops up showing a pie chart. This pie chart shows the distribution of various operations like "ALU", "Control", "Memory Access" etc. We are interested in seeing the types of ALU operation for this design. To do this double Left click on the ALU (green) sector of the pie chart.

2.7.4. Analyze profiling results (cont'd)



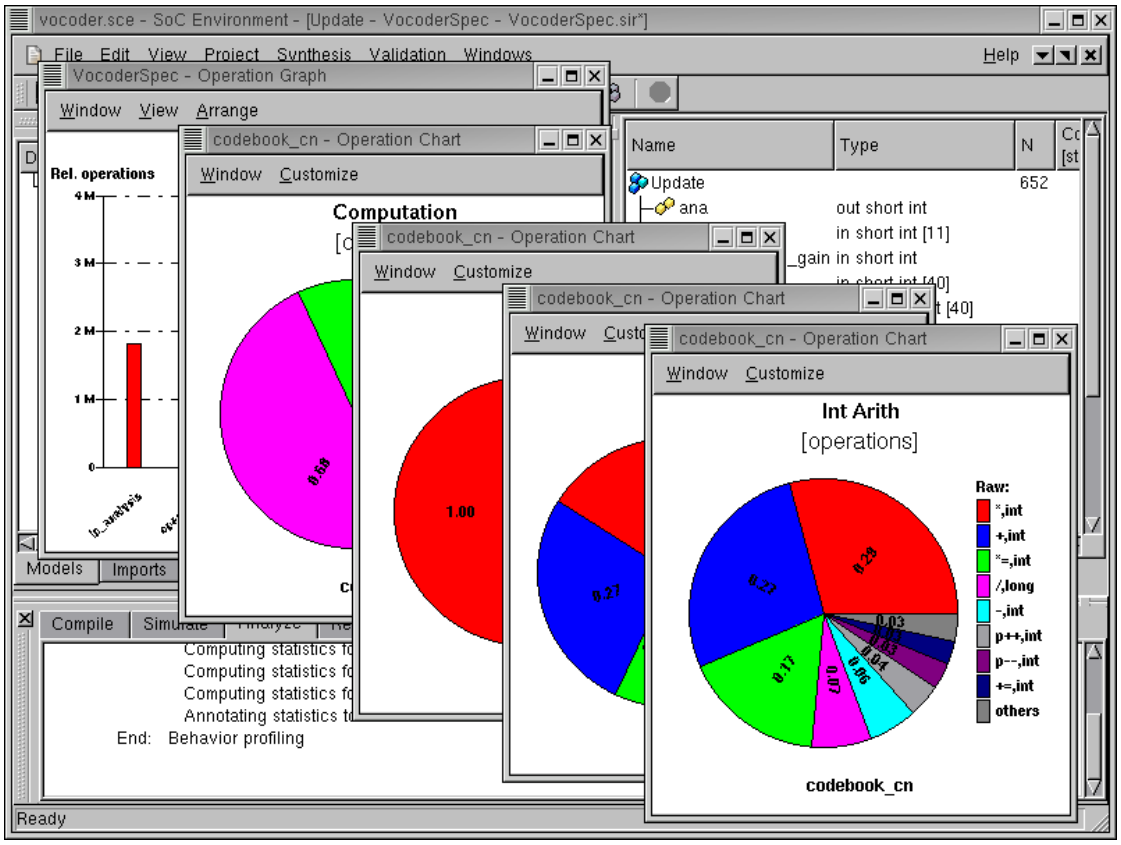
A new window pops up showing another pie chart. This pie chart shows the distribution of ALU operations. It can be seen that all the operations are integer operations, which is typical for signal processing application like the Vocoder. Since all the operations are integral, it does not make sense to have any floating point units in the design. Instead, we need a component with fast integer arithmetic like a DSP. To see the distribution of these integer operations, again double Left click on the pie chart.

2.7.5. Analyze profiling results (cont'd)



A new window pops up showing another pie chart. This pie chart shows the distribution of the type of integer operations. We can see that the majority of the operations is integer arithmetic. To view the distribution of the arithmetic operation types, again double Left click on the sector for "Int Arith."

2.7.6. Analyze profiling results (cont'd)



We can now observe the distribution of arithmetic operations like "multiplication", "addition", "increment", "decrement", etc. on a new pie chart. Note that 3/4 ths of the operations are additions or multiplications, thus it would be a good idea to have these two operations directly supported by a specific hardware unit.

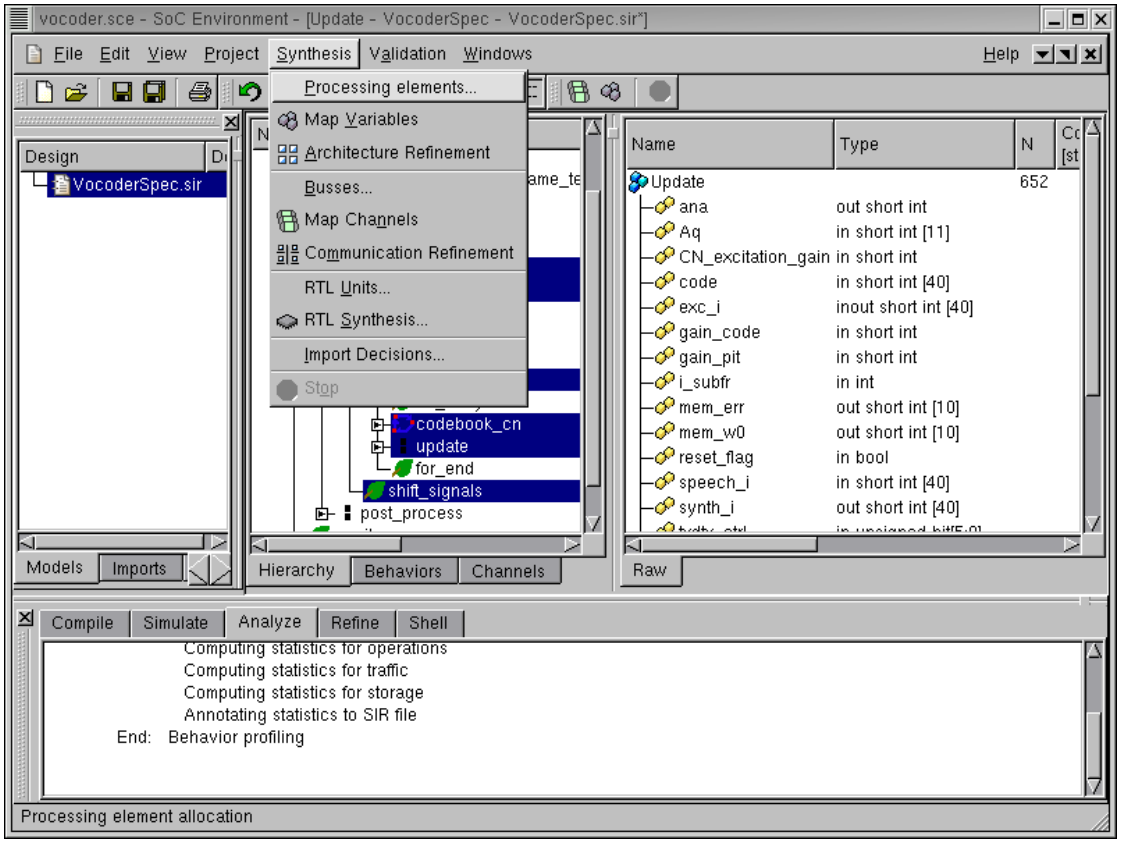
Chapter 3. Architecture Exploration

Architecture exploration is the design step to find the system level architecture and map different parts of the specification to the allocated system components under design constraints. It consists of the tasks of selecting the target set of components, mapping behaviors to the selected components and implementing correct synchronization between the components. Note that the components themselves are independent entities that execute in a parallel composition. In order to maintain the original semantics of the specification, the components need to be synchronized as neccessary. Architecture exploration is usually an iterative process, where different candidate architectures and mappings are experiemented to search for a satisfactory solution.

As indicated earlier, the timing constraint for the Vocoder design is the real time response requirement, i.e., the time to encode and decode the speech should be less than the speech time. The test speech has a 3.26 seconds duration. Therefore, the final implementation must meet this time constraint. In this chapter we see how we arrive at a suitable architecture with keeping this requirement in mind and using the refinement tool.

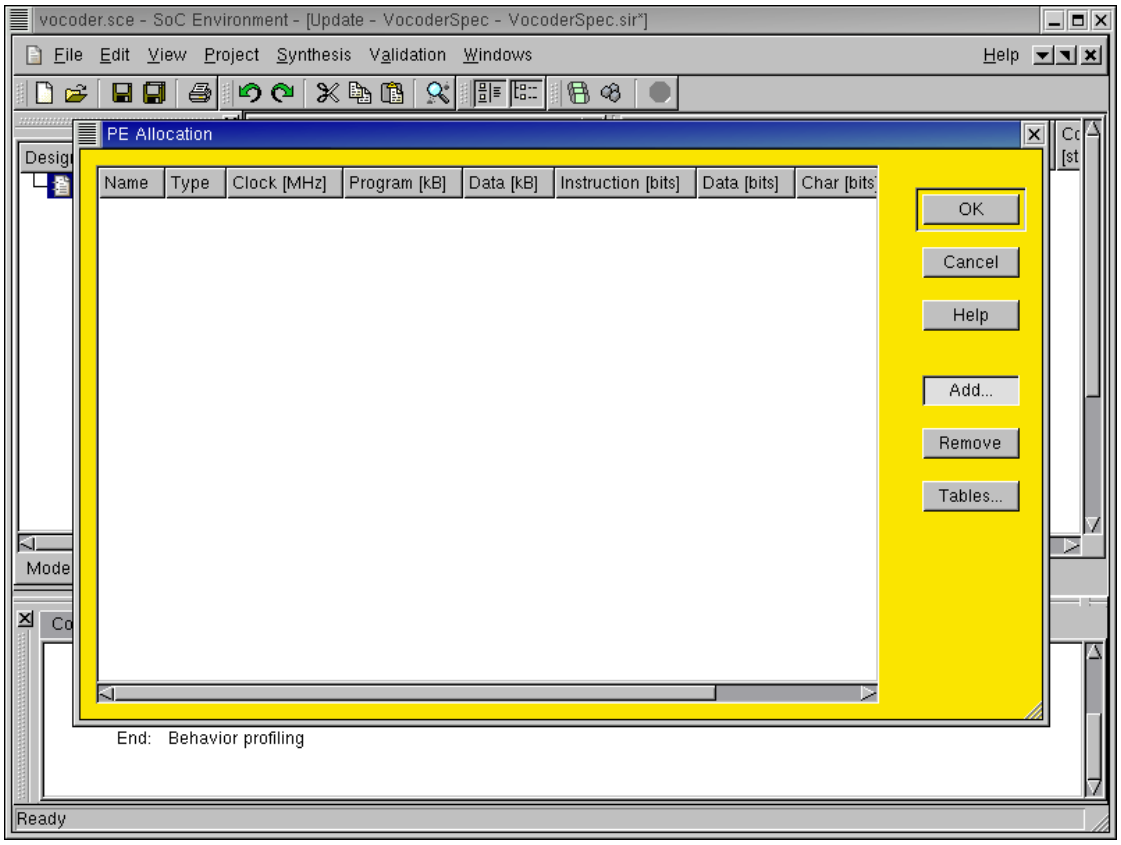
3.1. Try pure software

The goal of our exploration process is to implement the given functionality on a minimal cost architecture and still meet the timing constraint. The first approach is to implement everything in software so that we do not have the overhead of adding extra hardware and associated interfaces. To accomplish this, we first select a processor out of our component database. Thereafter, we map the entire specification on to this processor. Once the mapping is done, we invoke the analysis tool to see if the processor alone is sufficient to implement the system.



We begin by exploring the available set of components in the database. This is required to select a suitable processor. To view all available components and select the desired processor, go to the Synthesis menu on the menu bar and select Processing elements .

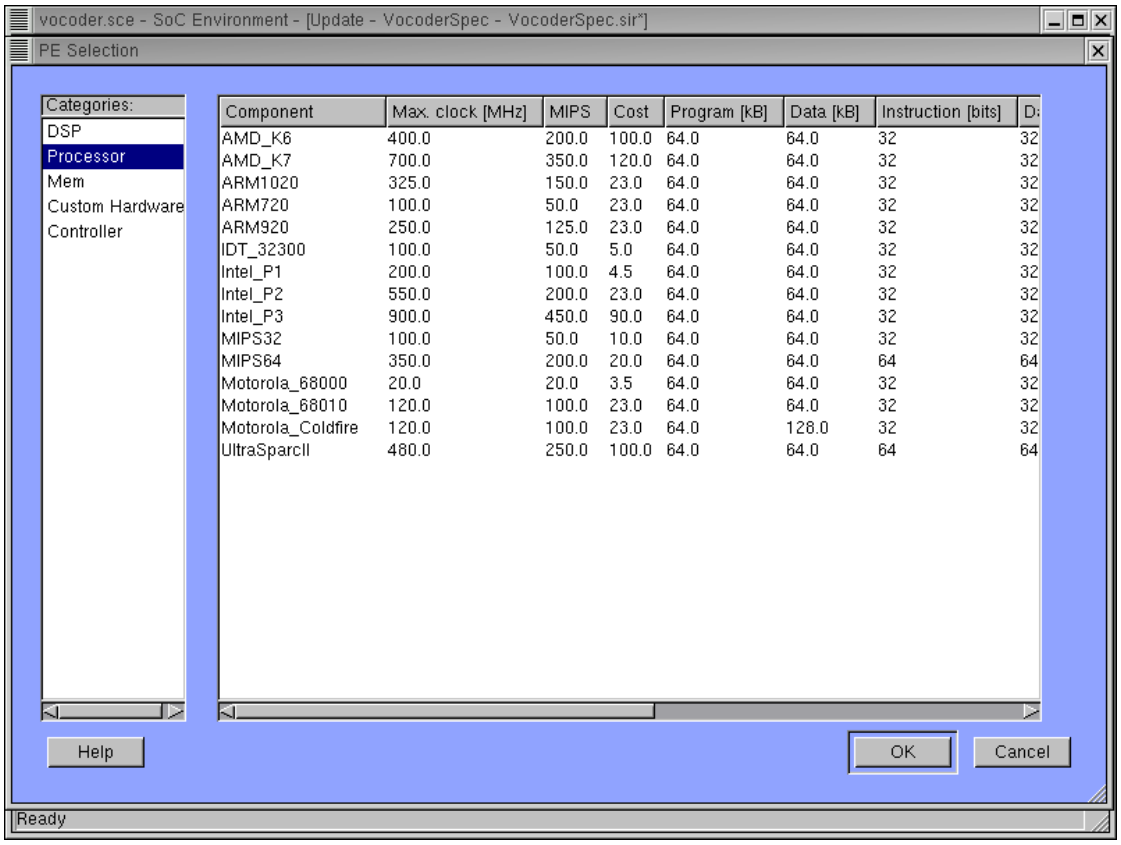
3.1.1. Try pure software (cont'd)



Now a PE Allocation window pops up. This window includes a table to display important characteristics of components selected for the design. In addition, it also provides a number of buttons (on the right side) for user actions, such as adding a component, removing a component, and so on. Since we have not allocated any component at this point, the table has no entry.

To view the component database and select the desired component, press the Add button.

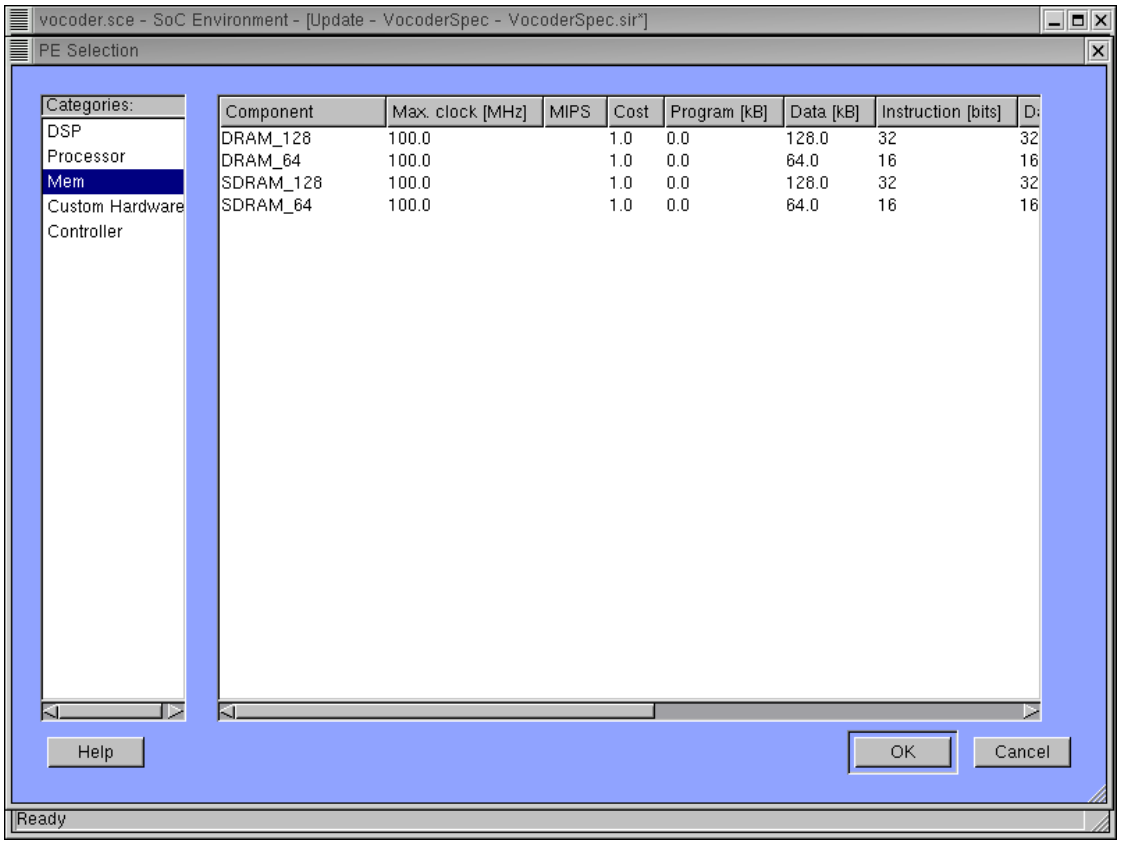
3.1.2. Try pure software (cont'd)



Now a PE Selection window is brought up. The left side of the window (Categories) lists five categories of components stored in the database. The right side of the window displays all components within a specific category along with their characteristics. As shown in the above figure, since the Processor category is selected on the left side, 15 commonly used processor components are displayed in detail on the right side.

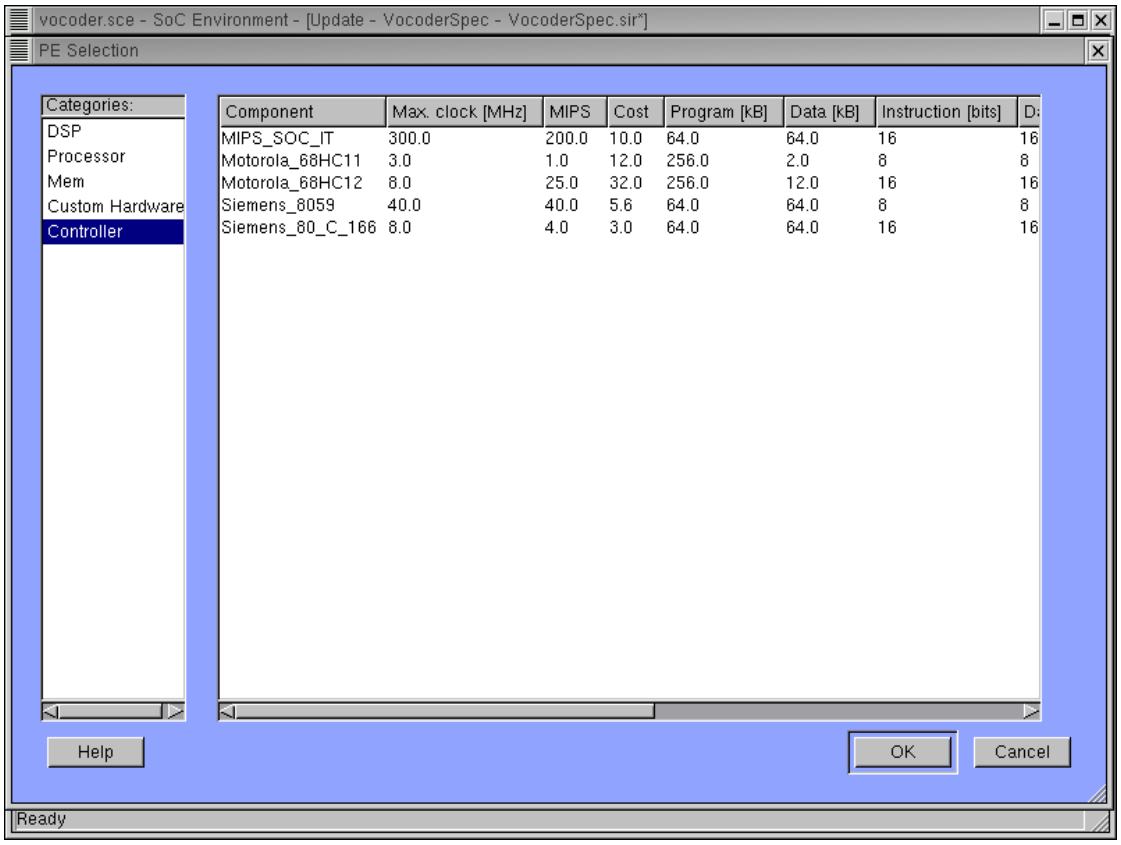
The Component description includes features like maximum clock speed, measure of the number of instructions per second, a cost metric, cache sizes, instruction and data widths and so on. These metrics are used for selecting the right component. Remember that the profiling data has given us an idea of what kind of component would be suitable for the application at hand.

3.1.3. Try pure software (cont'd)



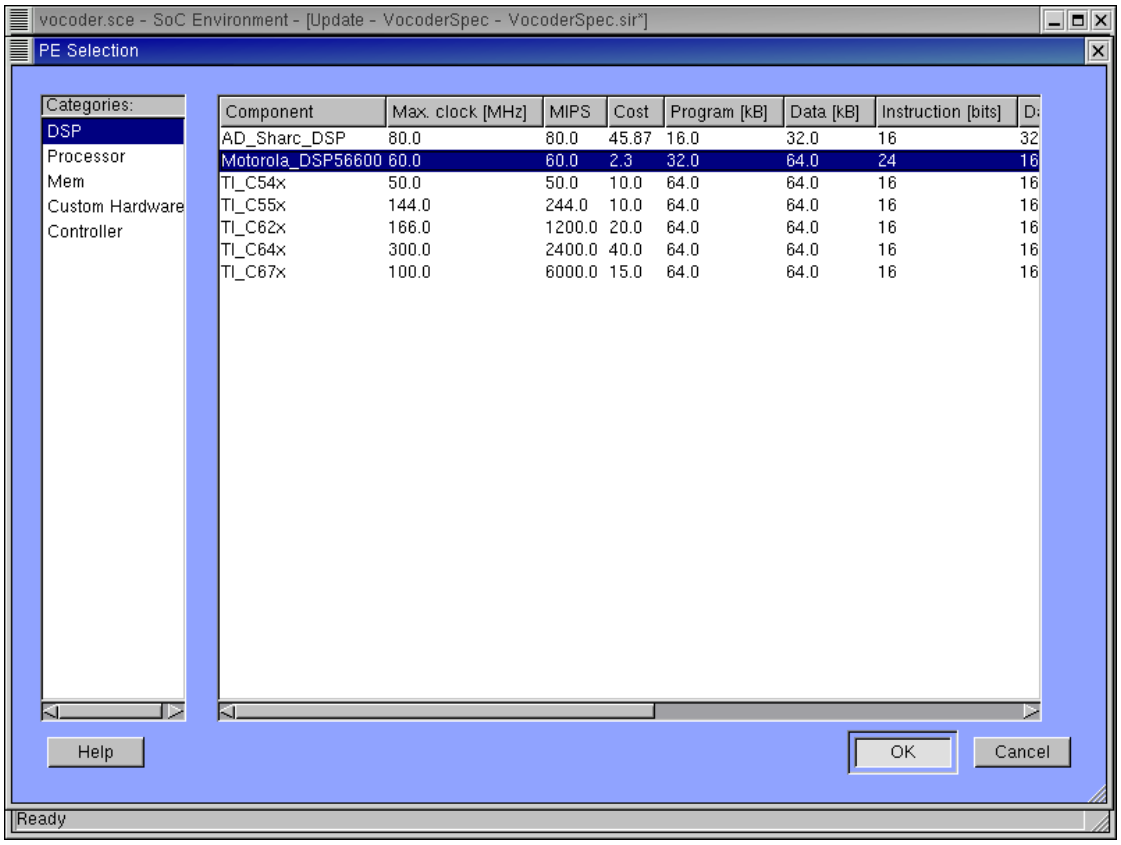
Now if we go to the Mem category, a number of memory components will be displayed in detail on the right side of the window. If the memory in the processor is insufficient for the application, we can add external memory components from this table.

3.1.4. Try pure software (cont'd)



Now if we go to the Controller category, a number of widely used microcontroller components will be displayed in detail on the right side of the window.

3.1.5. Try pure software (cont'd)

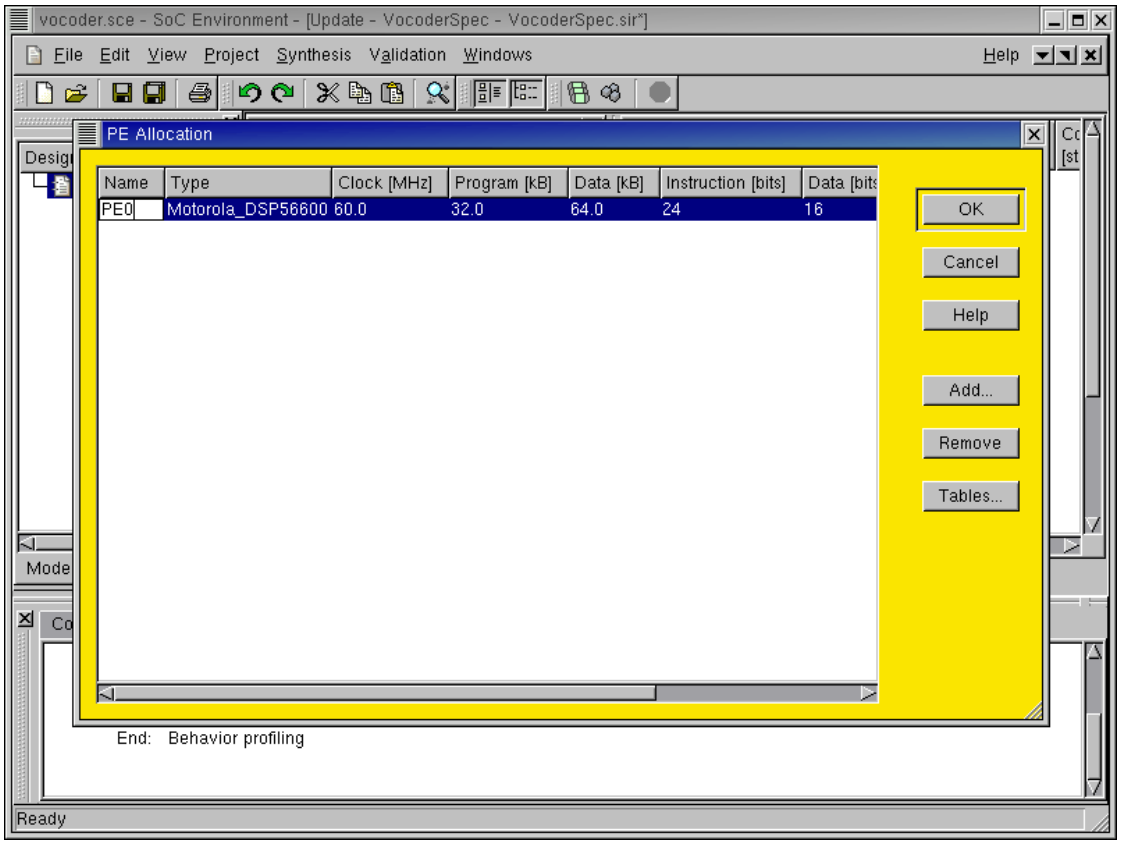


Through earlier profiling and analyzing, we found out that integer multiplication is the most significant operations in the original specification. Therefore, a fixed-point DSP would be desirable for this design.

Under the DSP category, a number of commercially available DSPs are displayed. These DSP components are maintained as part of the component library and may be imported into the design upon requirement. Since the Vocoder design project was supported by Motorola, our first choice is DSP56600 from Motorola.

Left click the Motorola_DSP56600 row to select it. Then click OK button to confirm the selection.

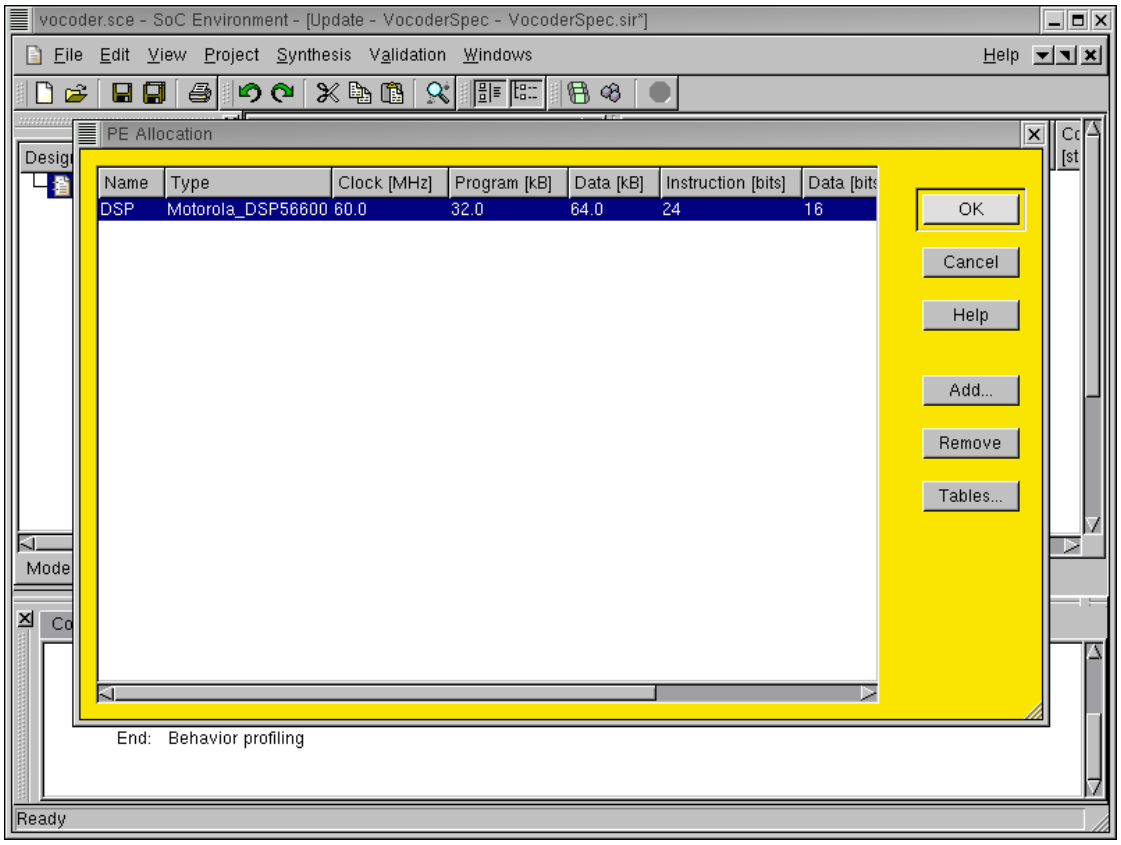
3.1.6. Try pure software (cont'd)



Now the PE Selection goes away and the PE Allocation table has one row that corresponds to our selected component, which has a type of Motorola_DSP56600. This new component was named as PE0 by default. To make it more descriptive for later reference, it is desirable to rename it.

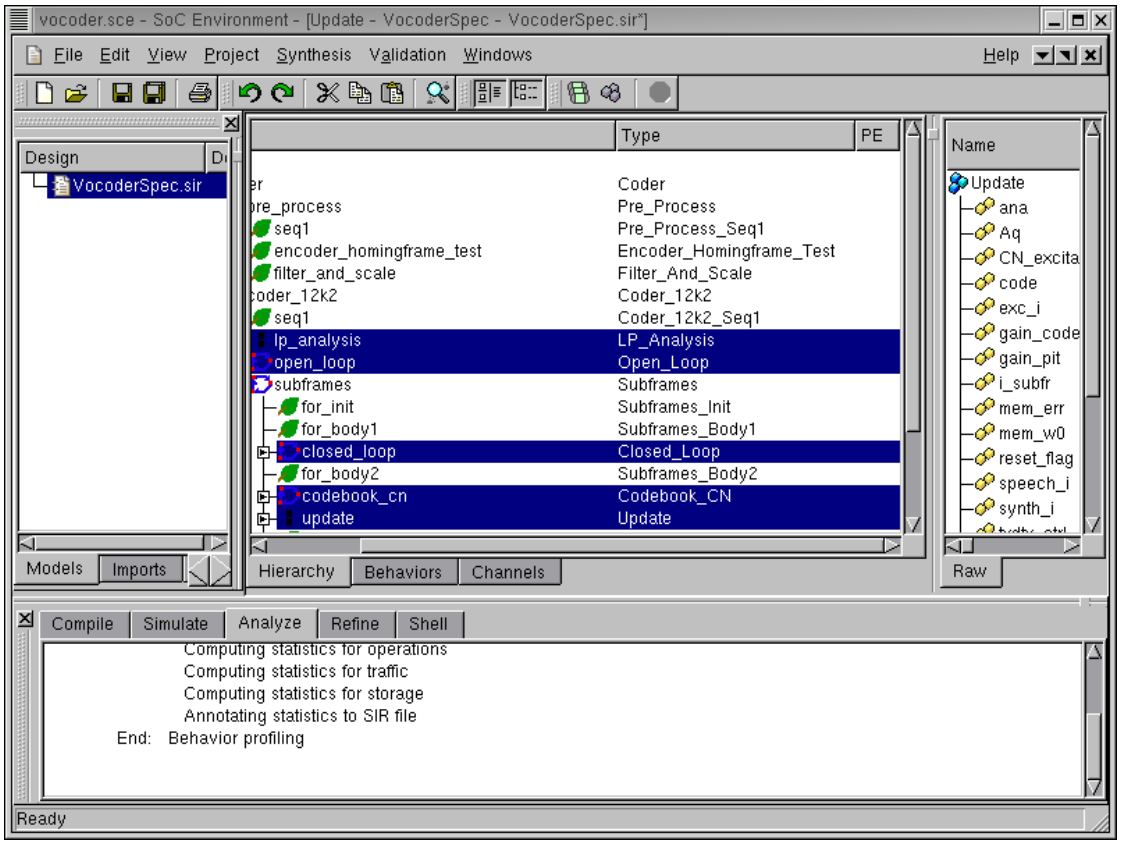
To rename it, just left click in the Name column of the row. The cursor will be blinking to indicated that the textfield is ready for editing.

3.1.7. Try pure software (cont'd)



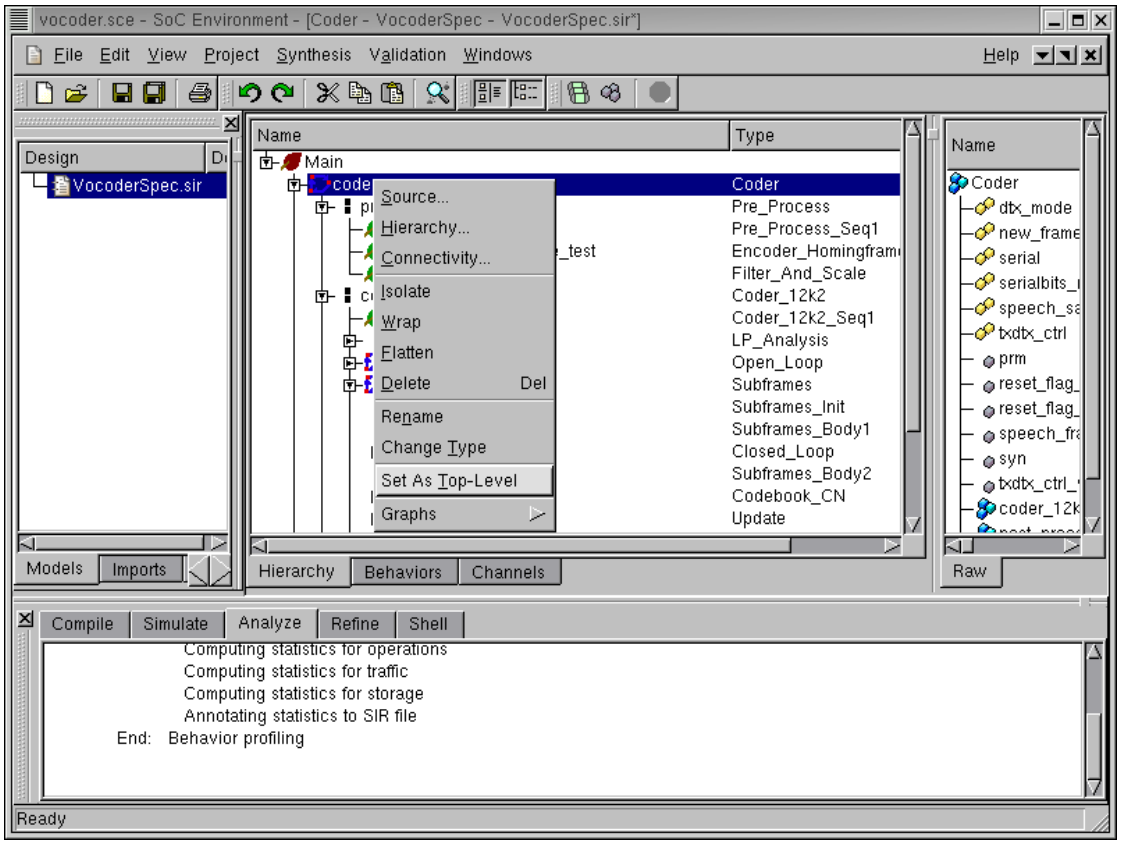
We will simply name the component as "DSP" since it is the only component used in the design at this instance. Proceed by typing "DSP" in the textfield and press return to complete the editing. Then press the OK to finish component allocation.

3.1.8. Try pure software (cont'd)



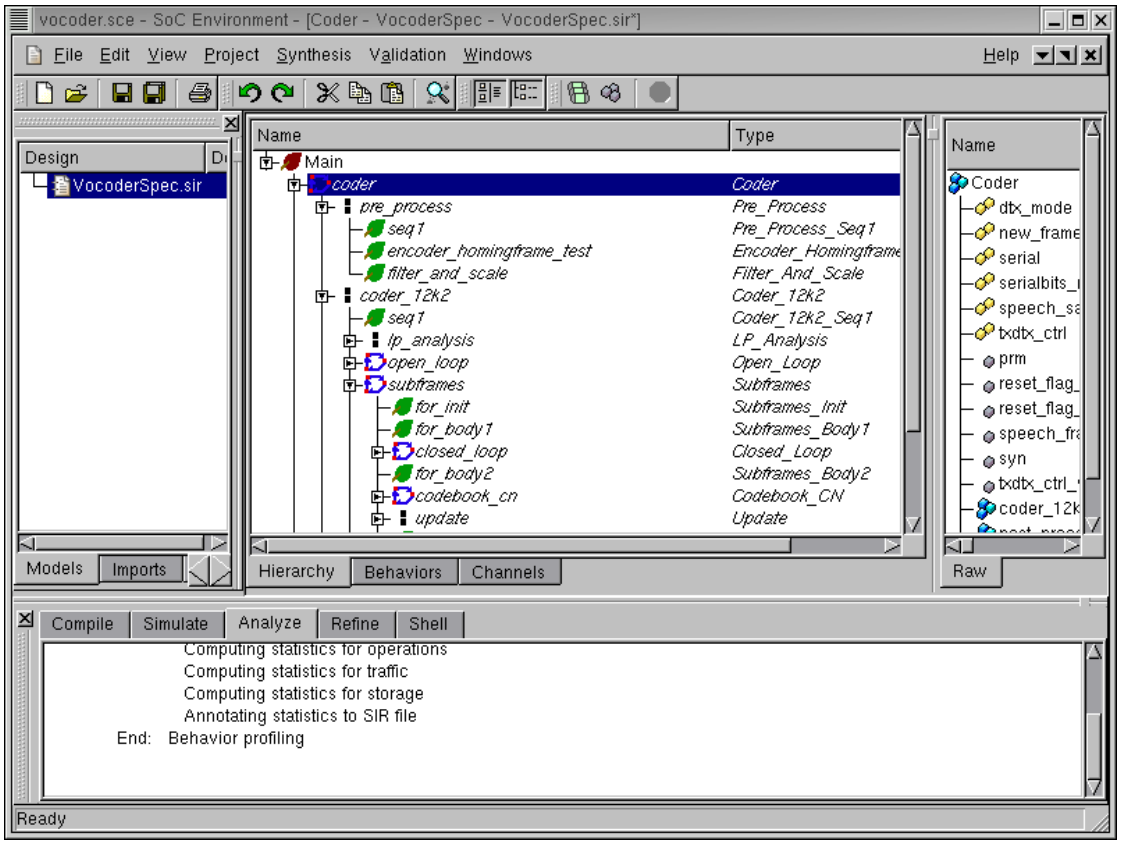
Now that the allocation phase is complete, the PE Allocation table goes away and we return to the design window showing the design hierarchy tree.

3.1.9. Try pure software (cont'd)



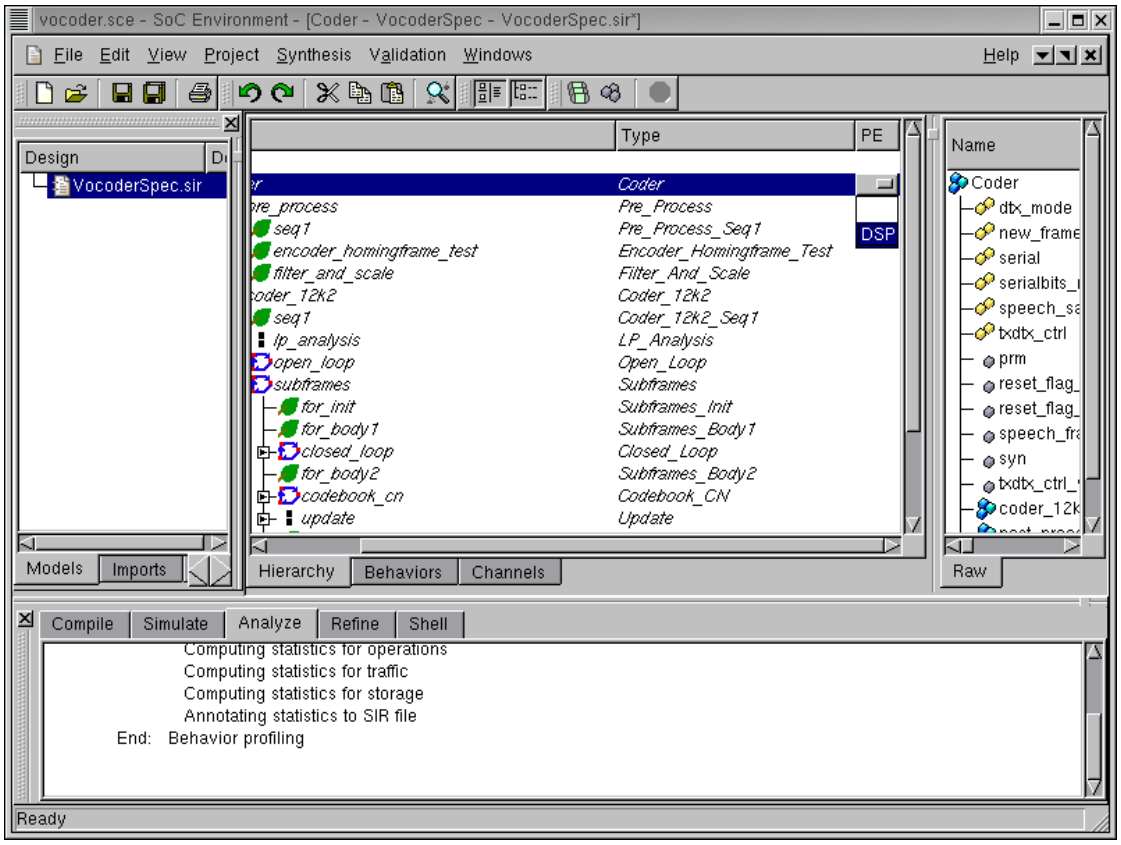
Before we move on, the top level behavior of the design needs to be specified. This is necessary because the specification model may have some test bench behaviors, which are not going to be included in the final design. It may be recalled that the project we are working with invloves not only the design under test(DUT) but also the behaviors that drive it. For example, the Monitor and Stimilus behaviors are just testbench behaviors while the Coder behavior is the real top level behavior of the design. To specify the Coder as the top level behavior, right click on coder to bring up the drop box menu then left click on Set As Top-Level .

3.1.10. Try pure software (cont'd)



As shown in the figure, when the top level behavior Coder is specified, the names of all its child behaviors are italicised to distinguish them from the test bench behaviors. In general, any behavior which needs to be tested can be set as top level. So, in a generic sense, the design under test can be identified by the italicized font.

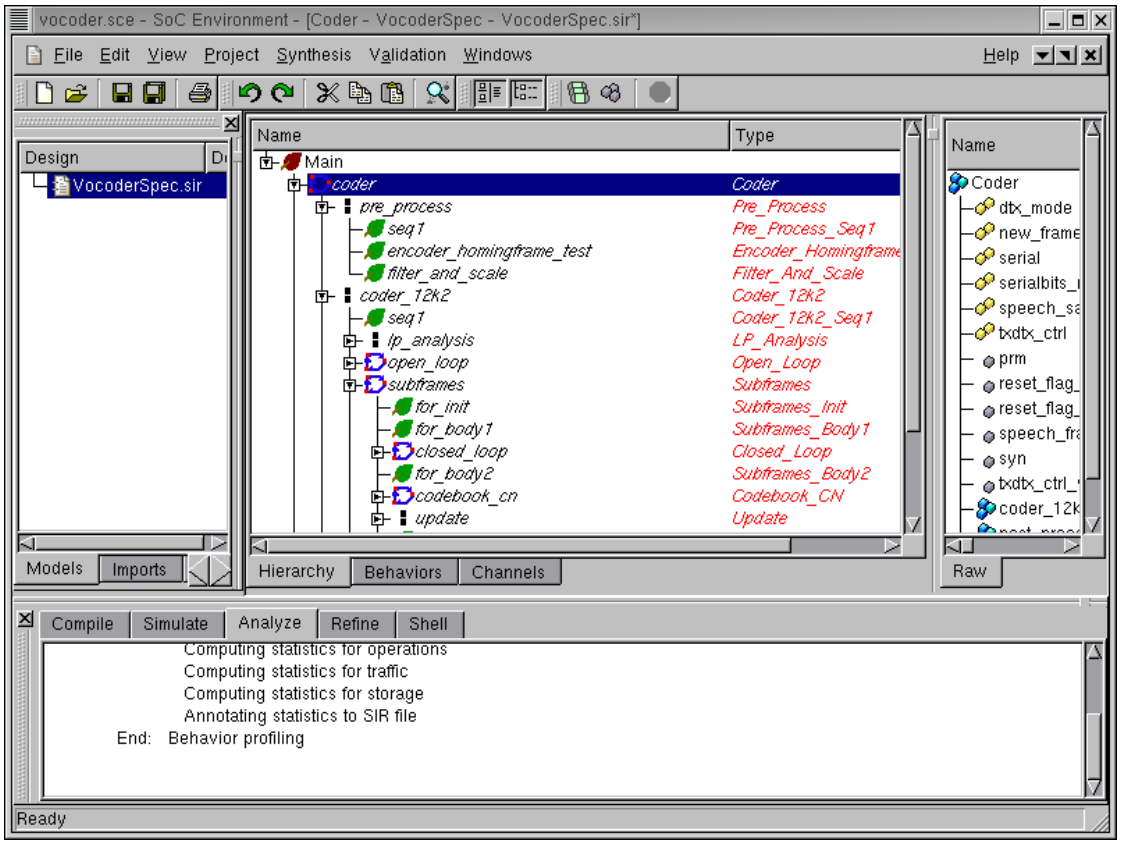
3.1.11. Try pure software (cont'd)



As mentioned earlier, we will map the whole design to the selected processor. This is done by assign the top level behavior Coder to DSP. Left click in the PE column in the row for the Coder behavior. A drop box containing allocated components comes up. Left click on DSP to map behavior Coder to DSP.

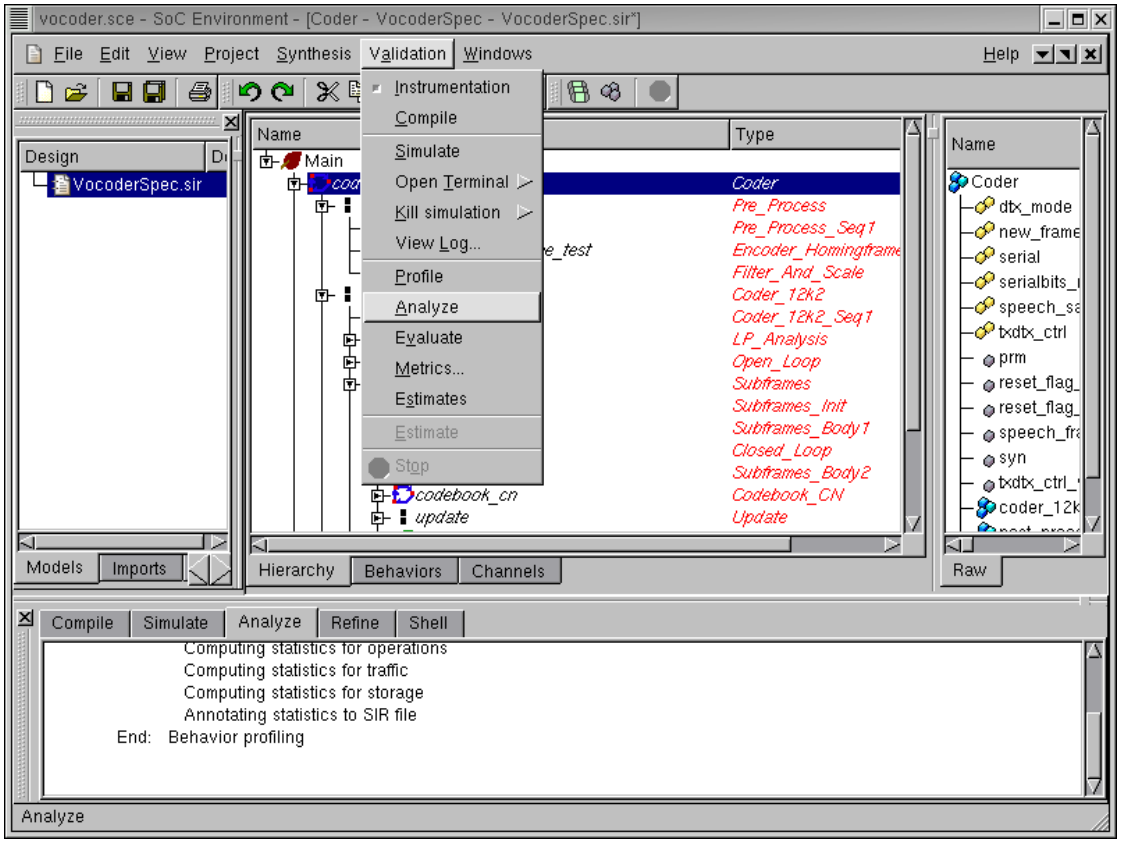
It should be noted that any kind of mapping is allowed. However, since we are investigating a purely software implementation, everything in the design gets mapped to the DSP.

3.1.12. Try pure software (cont'd)



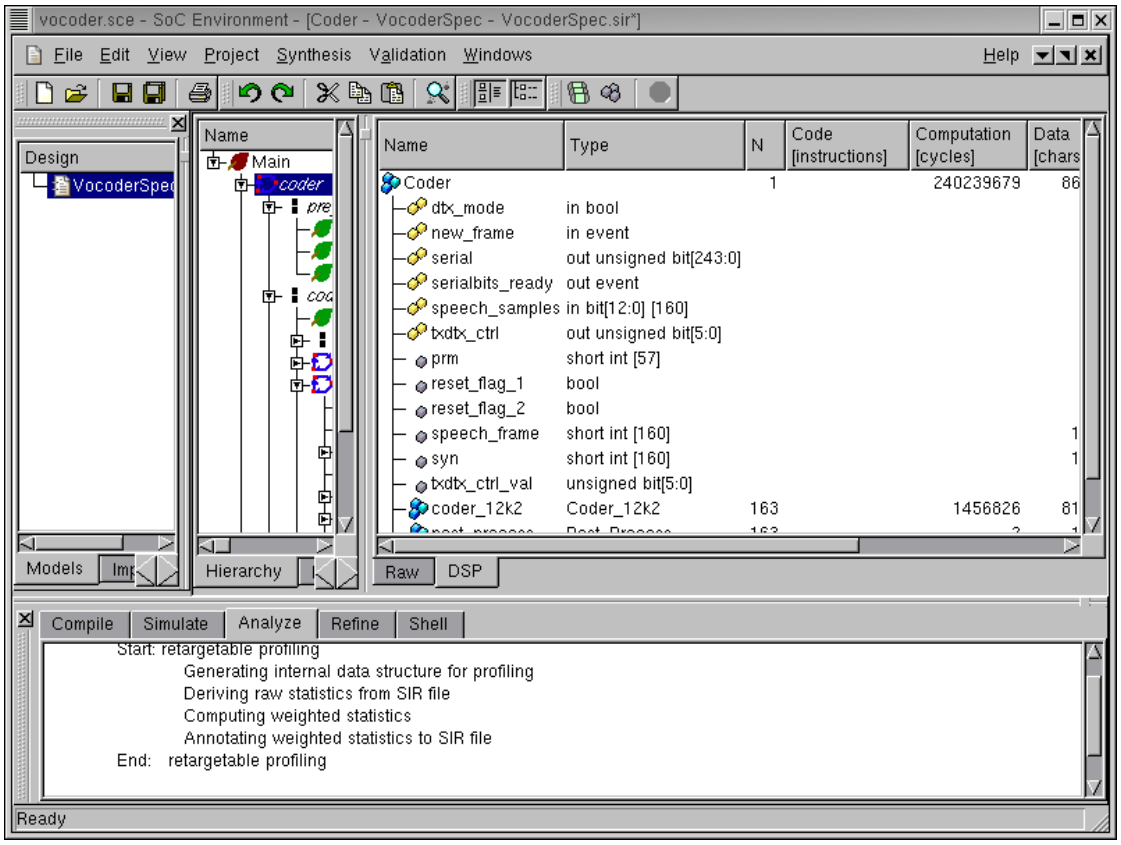
As we can see now, the descendant behaviors are all highlighted in red to indicated that they are mapped to the DSP component.

3.2. Estimate performance



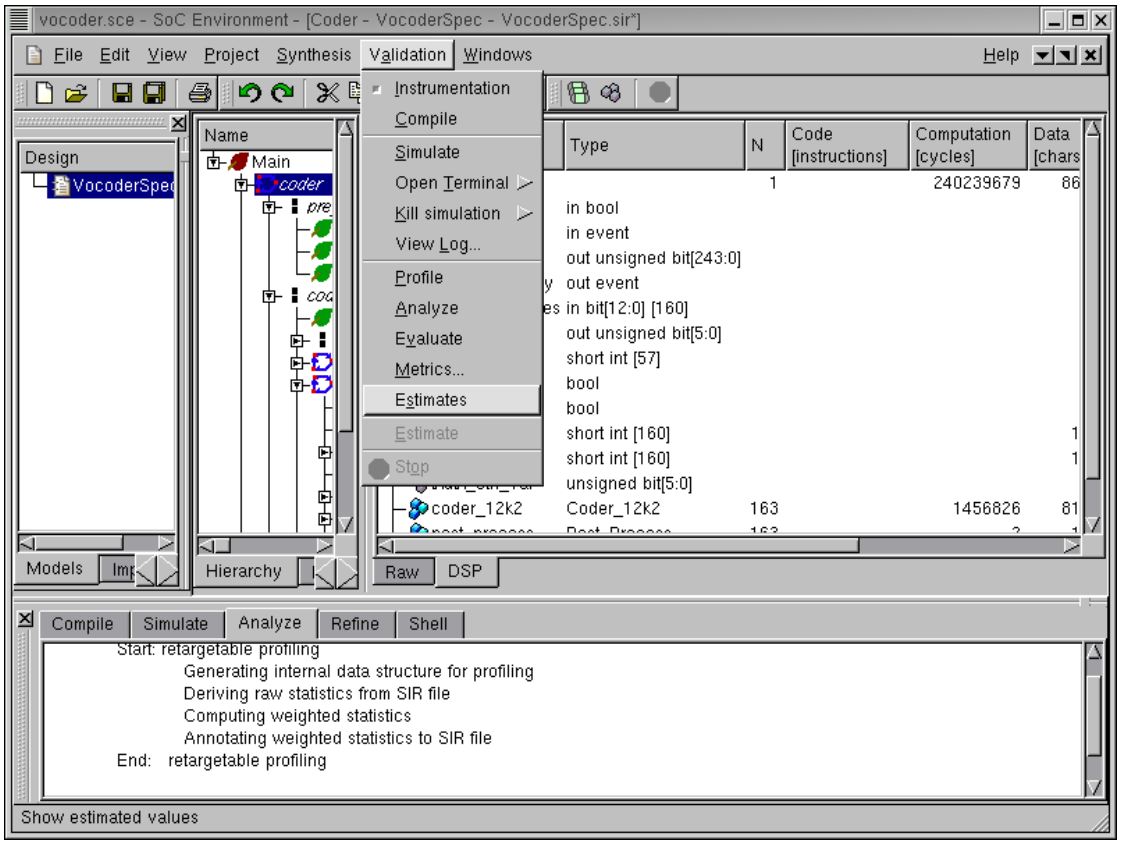
The next step is to analyze the performance of this architecture. Recall that we have a timing constrint to meet. We must therefore check if a purely software implementation would still suffice. If not, we will try some other architecture. Now we can analyze the performance of this pure software mapping by selecting Validation menu and select Analyze .

3.2.1. Estimate performance (cont'd)



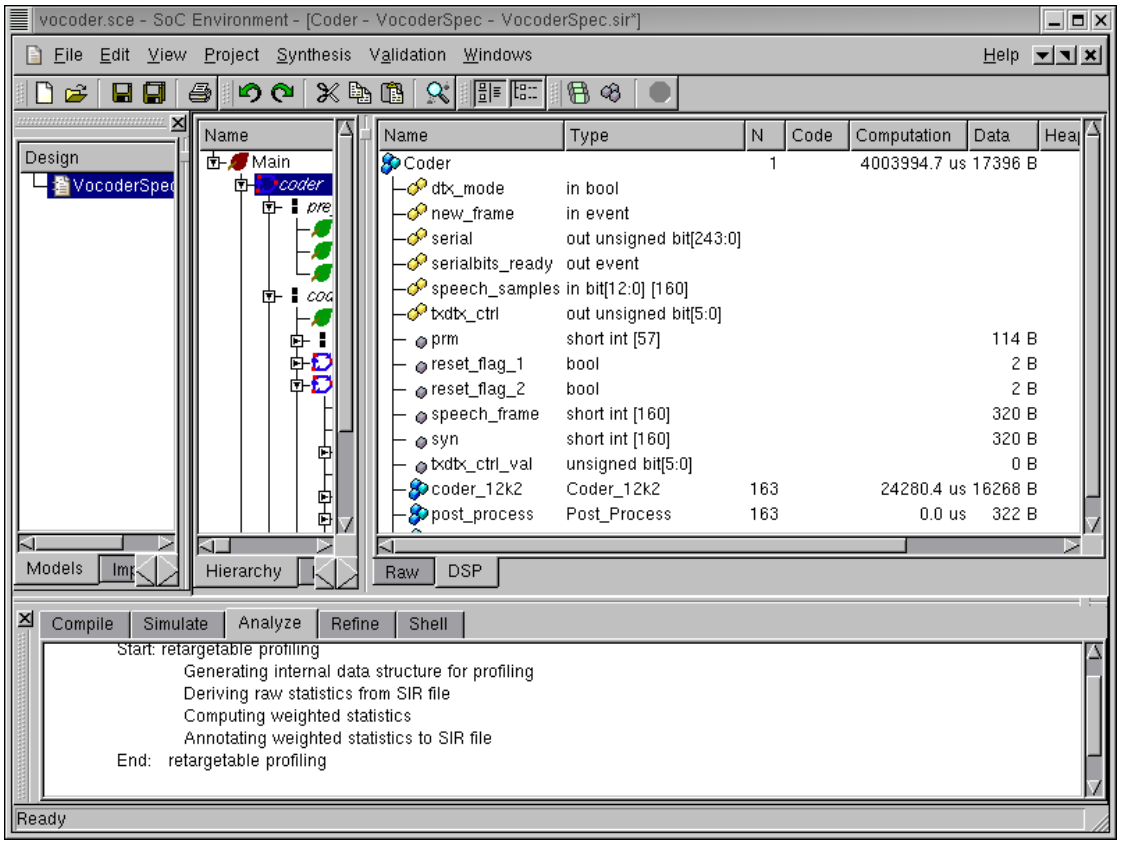
As we can see in the logging window, a retargeted profiling is being performed. Notice in the log information that raw statistic generated during profiling are used here. The raw statistics are take as an input to the analysis tool that generates statistics for the current architecture. Since, we know the parameters of the DSP, the analysis tool can provide a more accurate measure of actual timing. When that is done, the profiled data is displayed in the design window with the DSP tab. Notice that this tab has appeared at the bottom of the design data. The total computation time is shown in terms of number of DSP clock cycles.

3.2.2. Estimate performance (cont'd)



The number of computation cycles is a relevant metric for observation. However, it must be converted to an absolute measure of time so that we may directly verify if this architecture meets the demands. To find out the real execution time in terms of seconds, we turn on the option for estimation as follows. Go to Validation menu and select Estimates .

3.2.3. Estimate performance (cont'd)

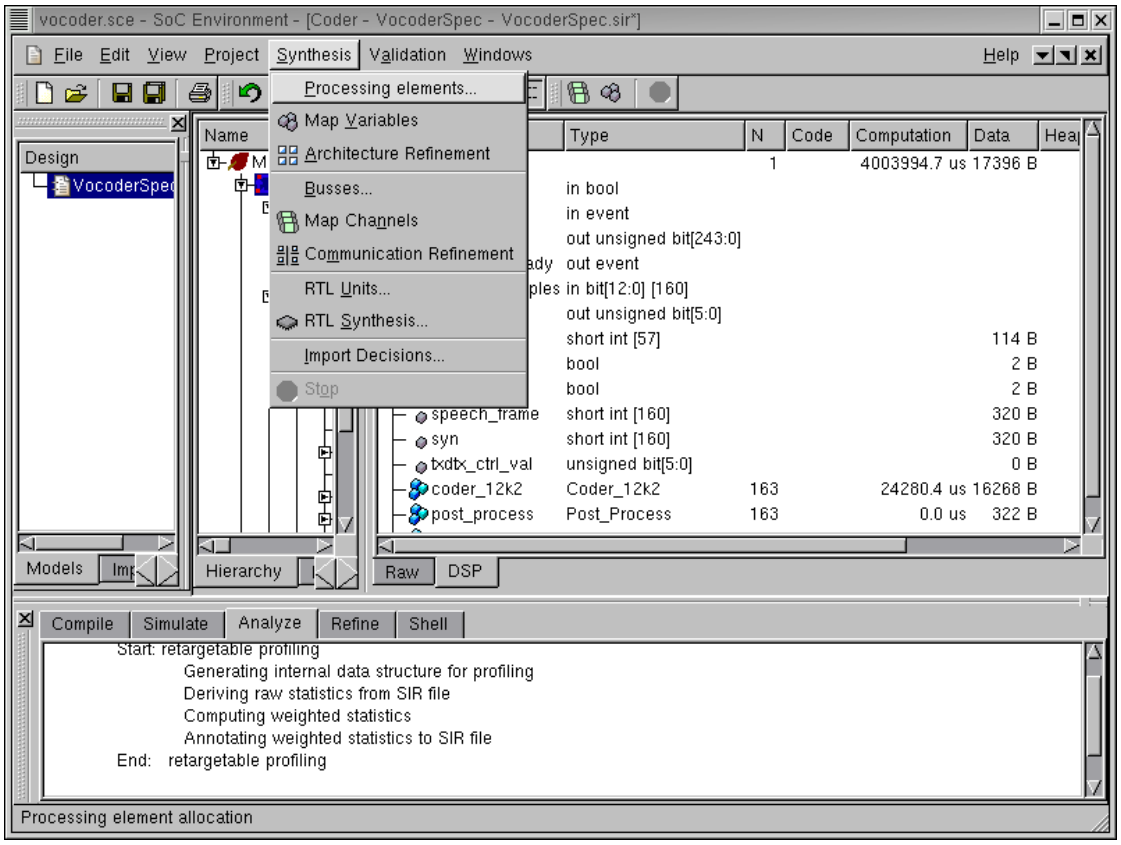


As seen in the design window, the computation time is in terms of us . As we can see in the row of behavior Coder , the estimated execution time (~ 4.00 seconds) exceeds the timing constraint of 3.26 seconds. Therefore, the pure software solution with a single Motorola_DSP56600 does not work. We, therefore, need to experiment with other architectures.

3.3. Try software/hardware partition

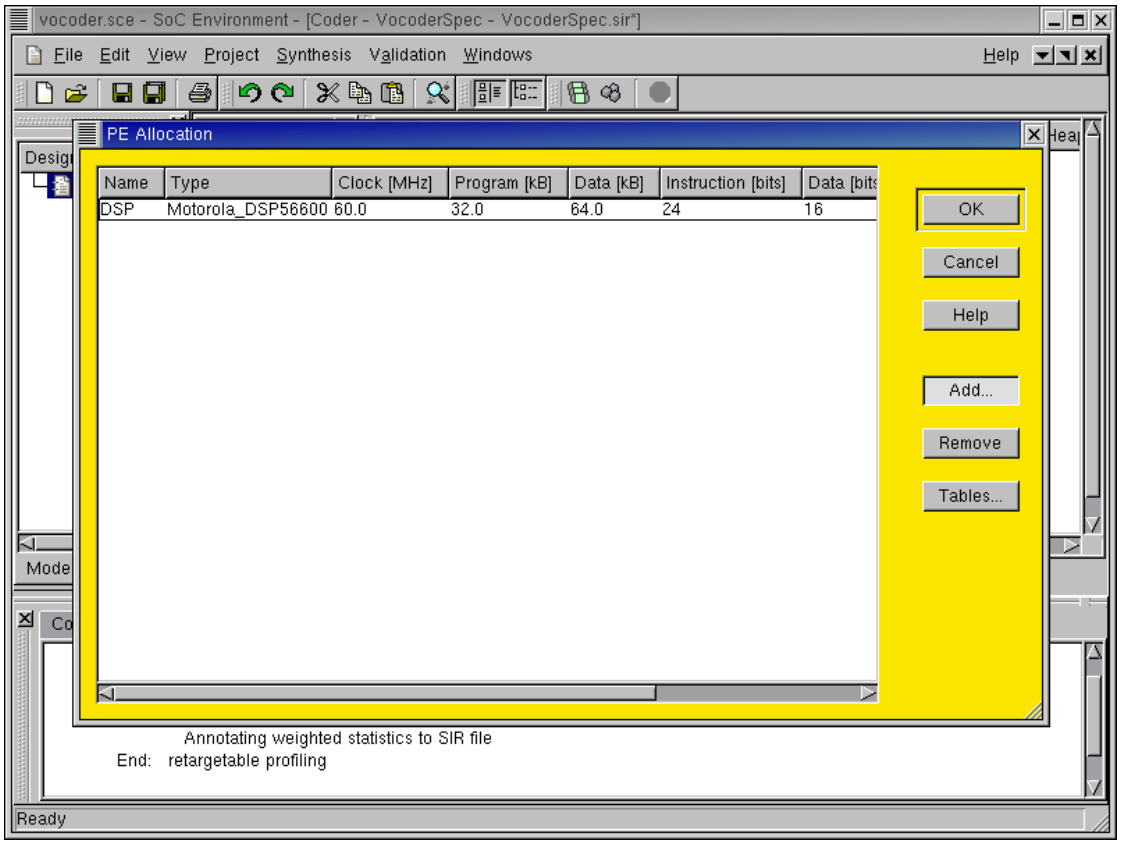
From what we observed while studying the vocoder specification, the design is mostly sequential. There is not much parallelism to exploit. What we need to reduce the execution time is a much faster component than the DSP we used. Some of the critical time consuming tasks may be mapped to a fast hardware. In this iteration, we will try to add one hardware component along with the DSP to implement the design. As we found out earlier, one of the computationally intensive and critical part in the Vocoder is the Codebook behavior. We hope to speed it up by mapping it to a custom hardware component and execute the remaining behaviors on the DSP.

3.3.1. Try software/hardware partition (cont'd)



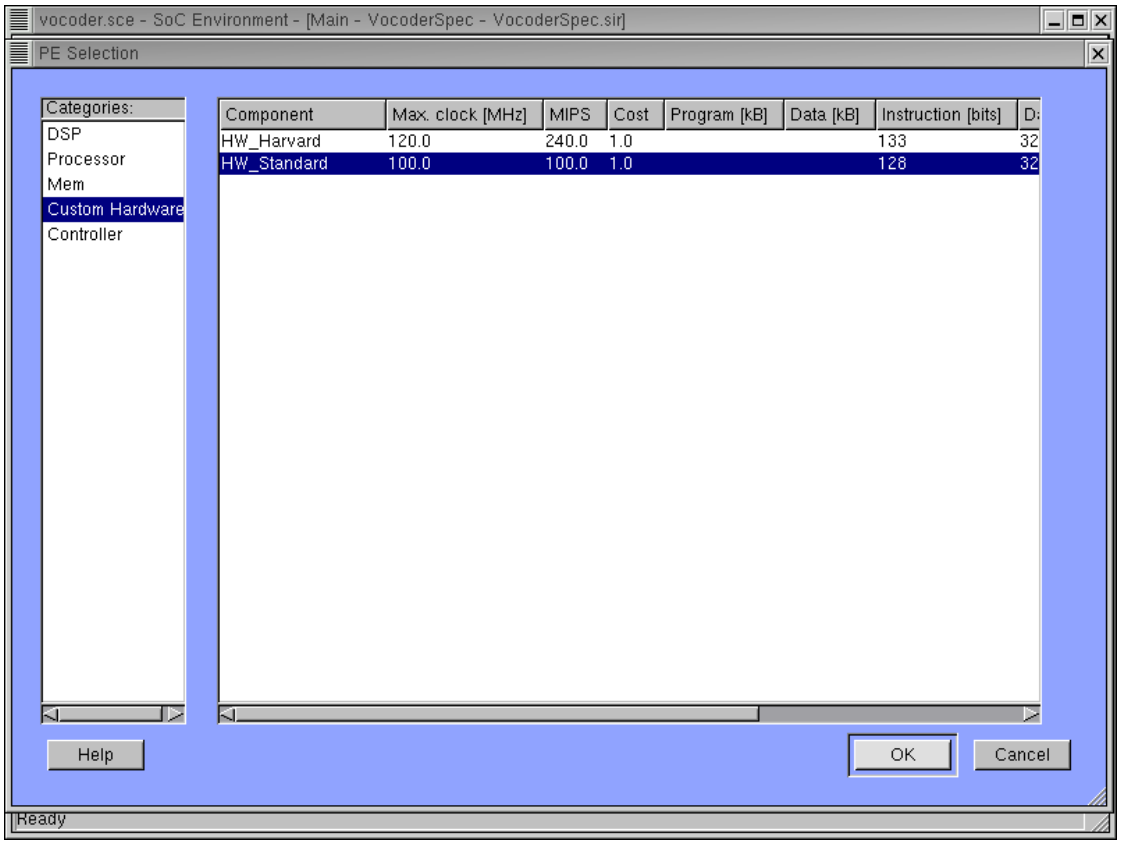
As we did earlier, while selecting the processor, go to the Synthesis menu and select Processing elements .

3.3.2. Try software/hardware partition (cont'd)



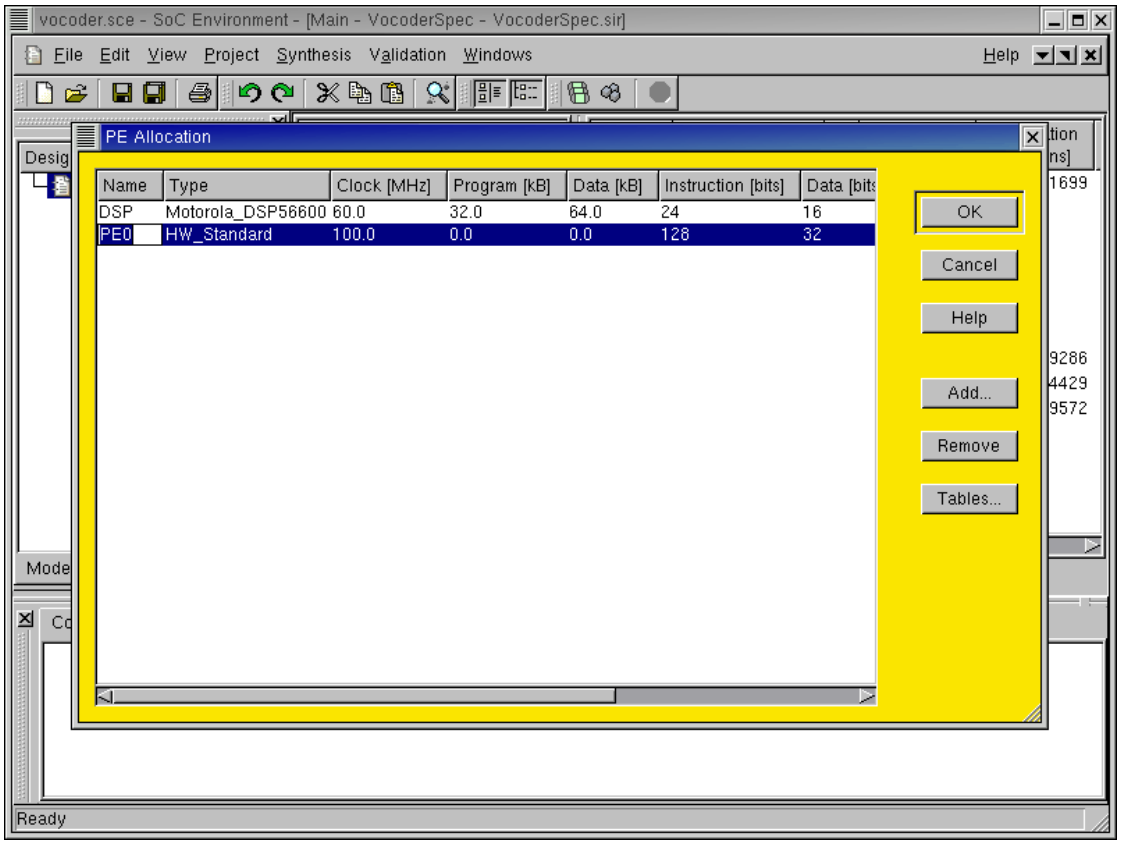
This time, the PE Allocation table pops up. As we can see, the previously allocated DSP is displayed. To insert the hardware component, press Add button to go to our database.

3.3.3. Try software/hardware partition (cont'd)



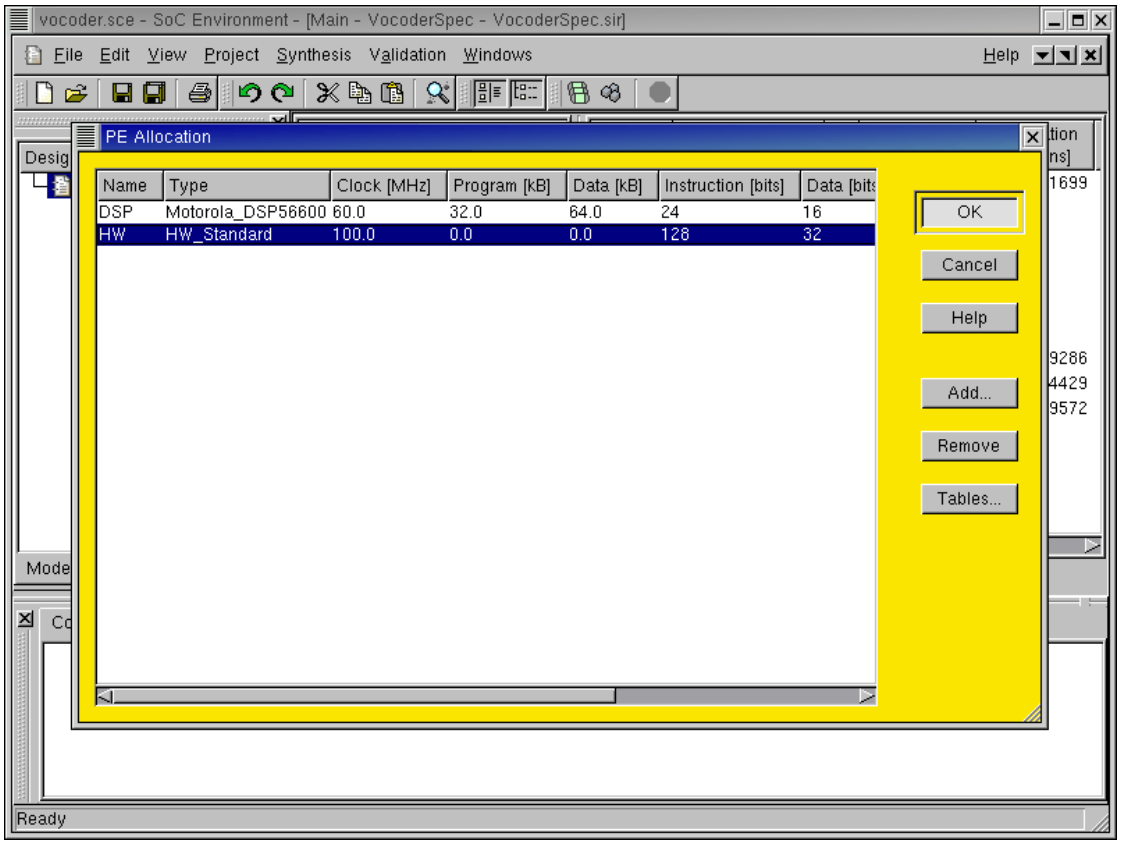
In the Custom Hardware category, two general types of hardware components are displayed. Here we will use the standard hardware design with a datapath and a control unit. Select the HW_Standard and press OK to confirm the selection.

3.3.4. Try software/hardware partition (cont'd)



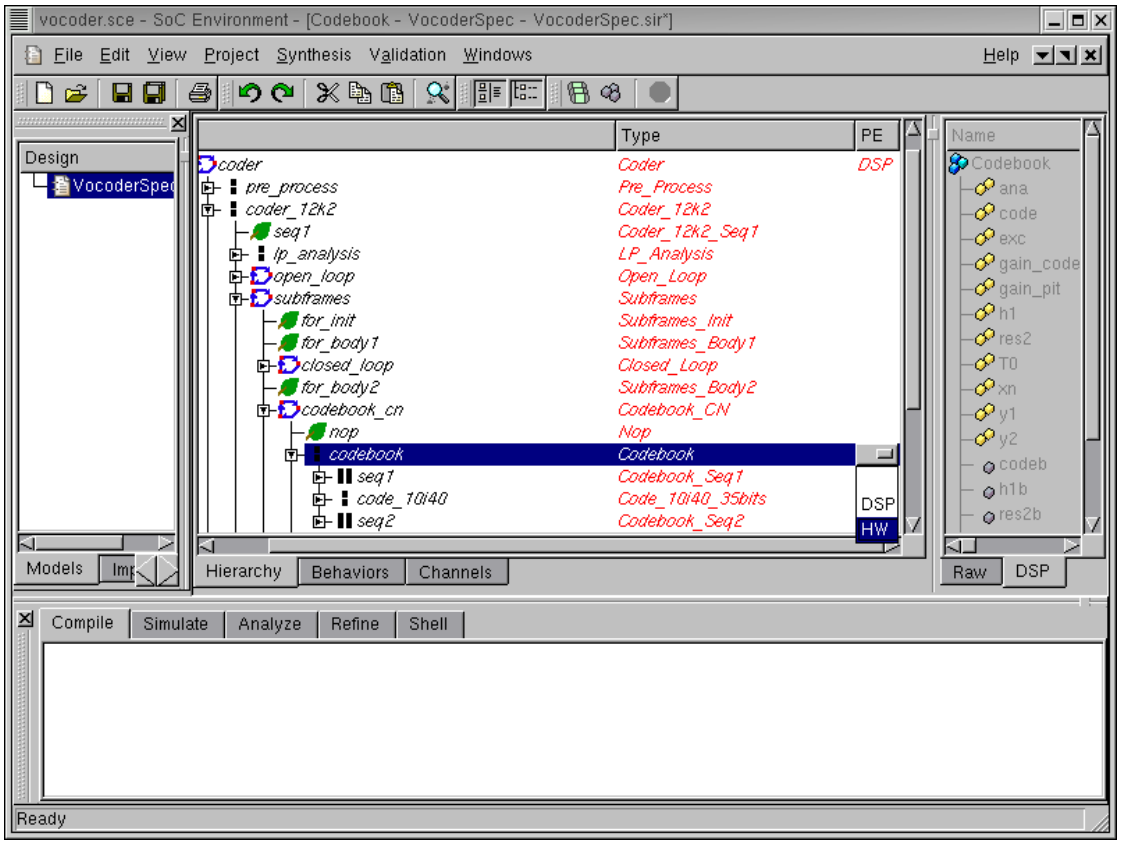
Now the HW_Standard component is added to the PE Allocation table. In the same way we did for the DSP component, we simply rename it to "HW" to distinguish it. Notice that for the hardware component, some metrics are flexible. For instance, the clock period may be changed. However, we stay with the current speed of 100 Mhz for demo purpose.

3.3.5. Try software/hardware partition (cont'd)



After we renamed it, press OK button to complete component allocation.

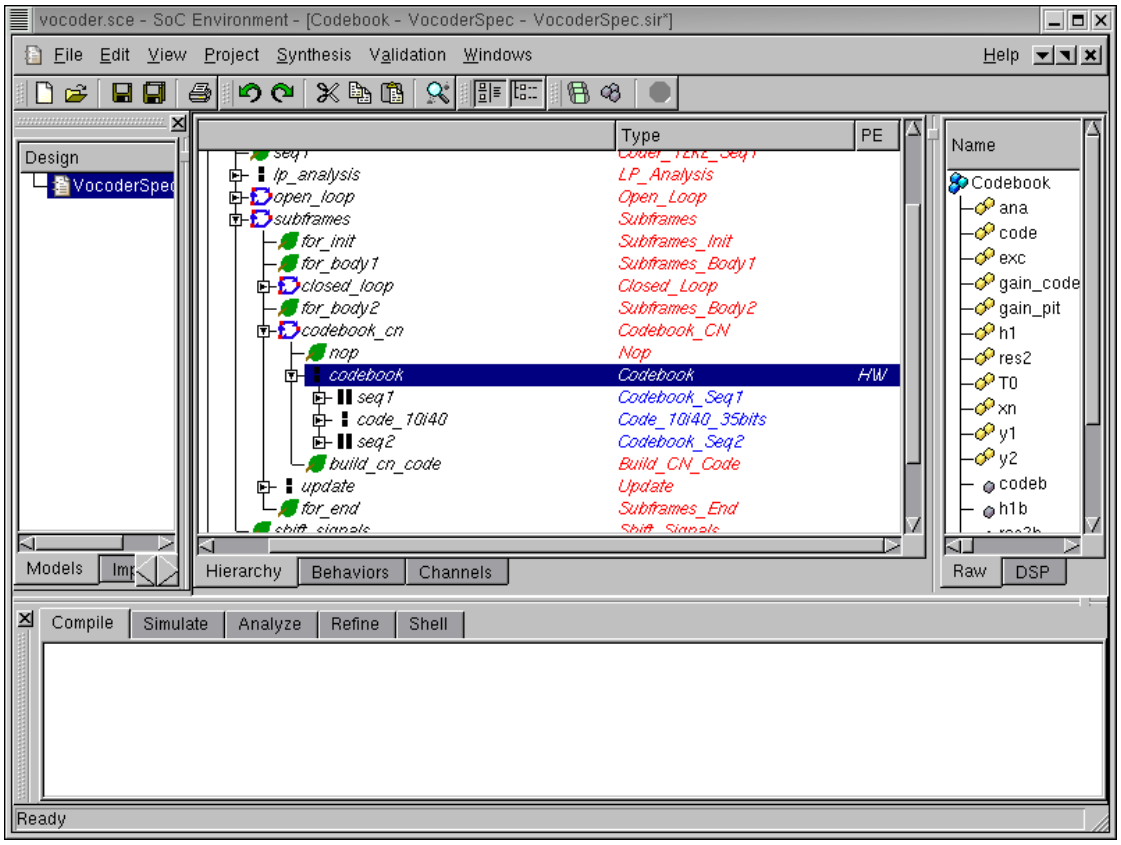
3.3.6. Try software/hardware partition (cont'd)



Remember we have already specified the top level behavior and mapped all behaviors to DSP in the first iteration. That information is still there and we do not have to specify it again. We only need to map the Codebook behavior to the HW component, as suggested earlier.

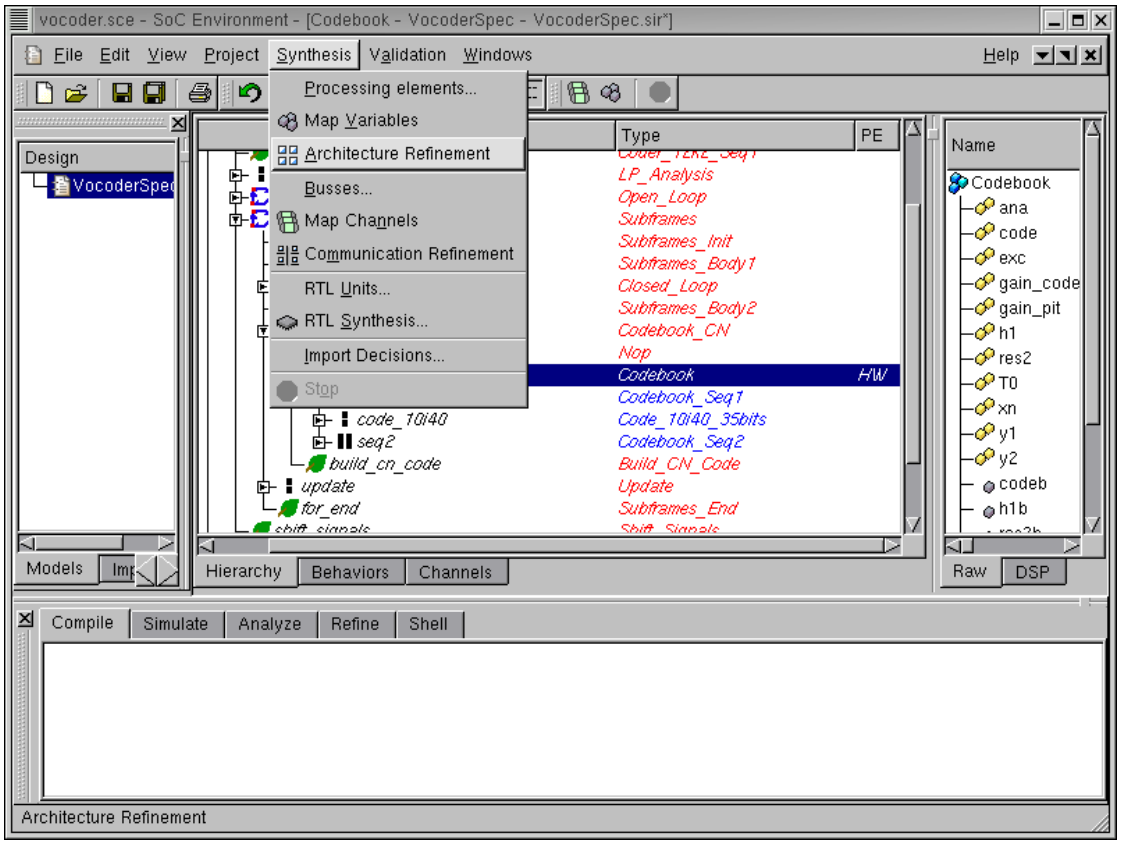
Browse the hierarchy tree to locate behavior Codebook . Click on Codebook in the "PE" column. Click on HW in the drop box to map Codebook to HW. This would map the entire subtree of behaviors under Codebook to custom hardware.

3.3.7. Try software/hardware partition (cont'd)



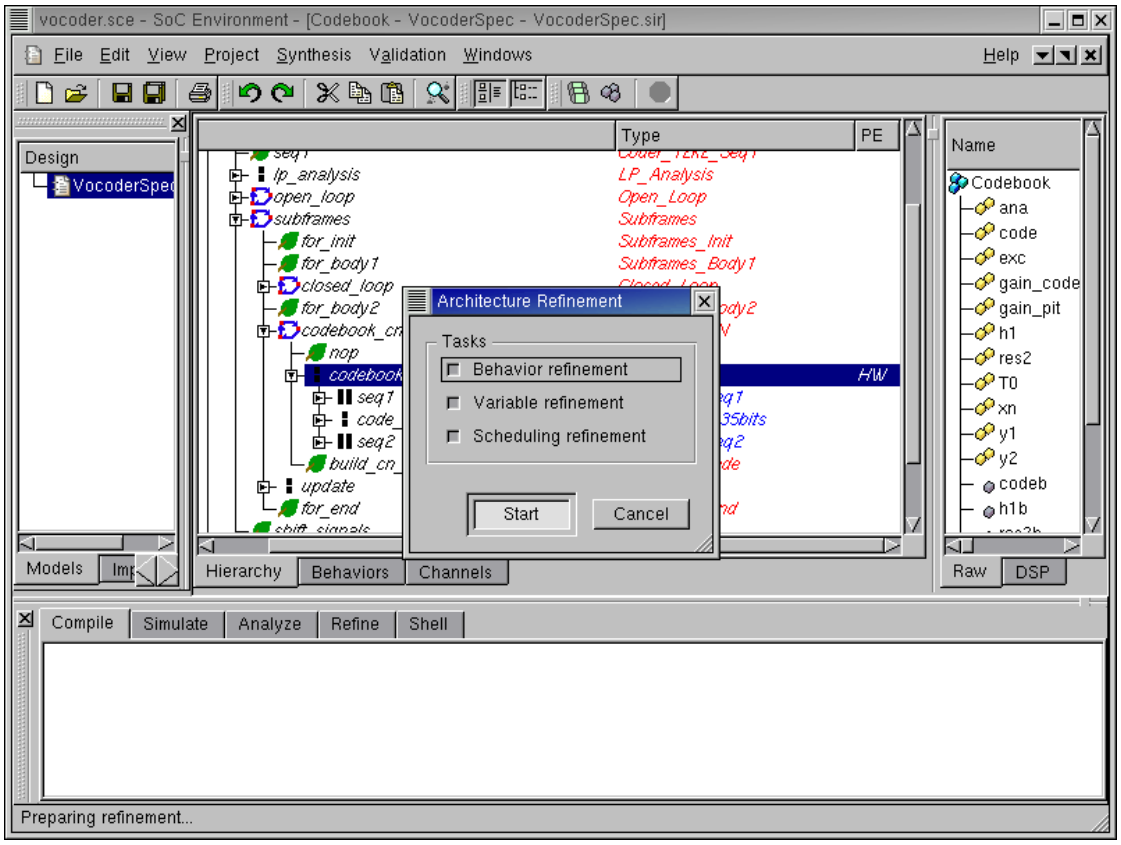
After the mapping, we will see the subtree rooted at Codebook is highlighted in blue in contrast to the rest behaviors in red that are mapped to DSP.

3.4. Architecture refinement



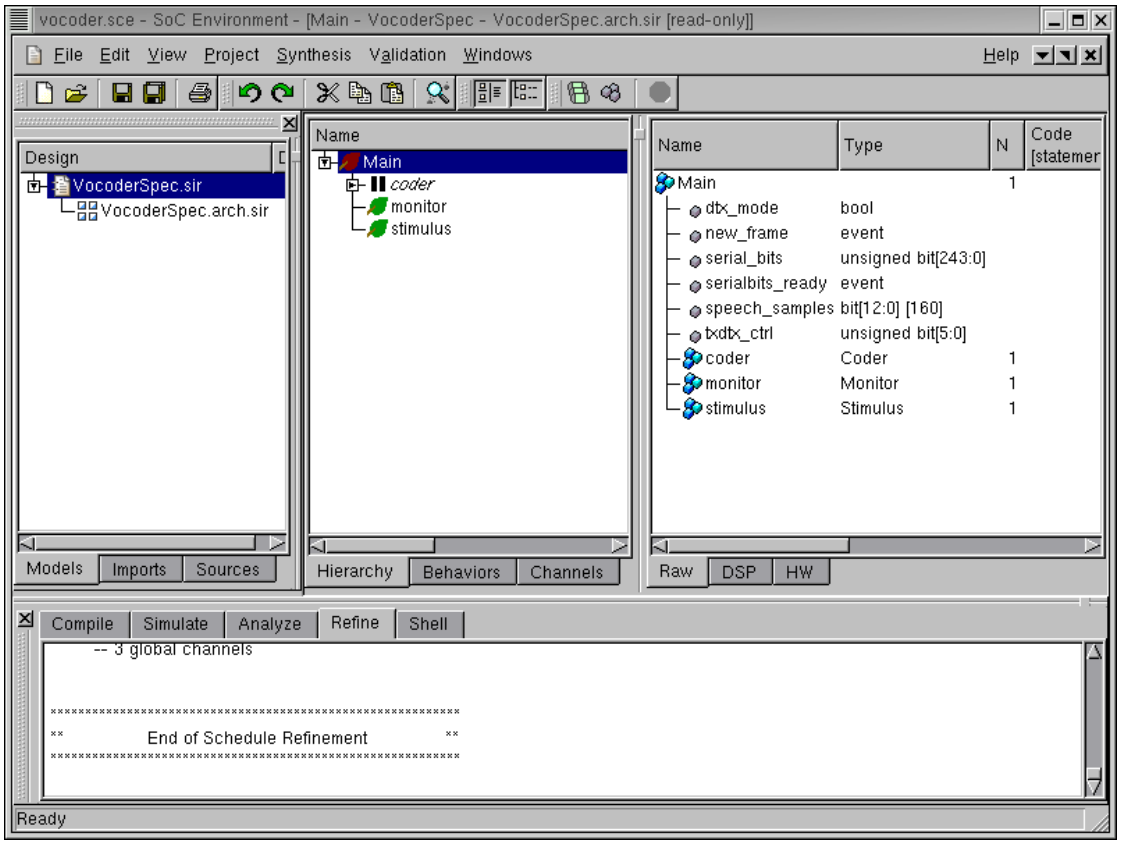
Now we can refine the specification model into an architecture model, which will exactly reflect the this architecture and mapping decisions. This can be done either manually or automatically. As we mentioned earlier, an architecture refinement tool is integrated in SCE. To invoke the tool, go to Synthesis menu and select Architecture Refinement . The tool changes the model to reflect the partition we created and also introduces synchronization between the parallely executing components. Note that we have not decided to Map variables explicitly to components. For demo purposes, we will leave this decision to be made automatically by the refinement tool. However, it needs to be mentioned that the designer may choose to map variables in the design as deemed suitable.

3.4.1. Architecture refinement (cont'd)



A dialog box pops up for selecting specific refinement tasks of architecture refinement. By default, all tasks will be performed in one go. Now press the Start buttonto start the refinement. It must be noted that the user has an option to do the architecture refinements one step at a time. For instance, a designer may want to stop at behavior refinement if he is not primarily concerned about observing the memory requirements or the schedule on each component. Nevertheless, in our demo we perform all steps to generate the final architecture model.

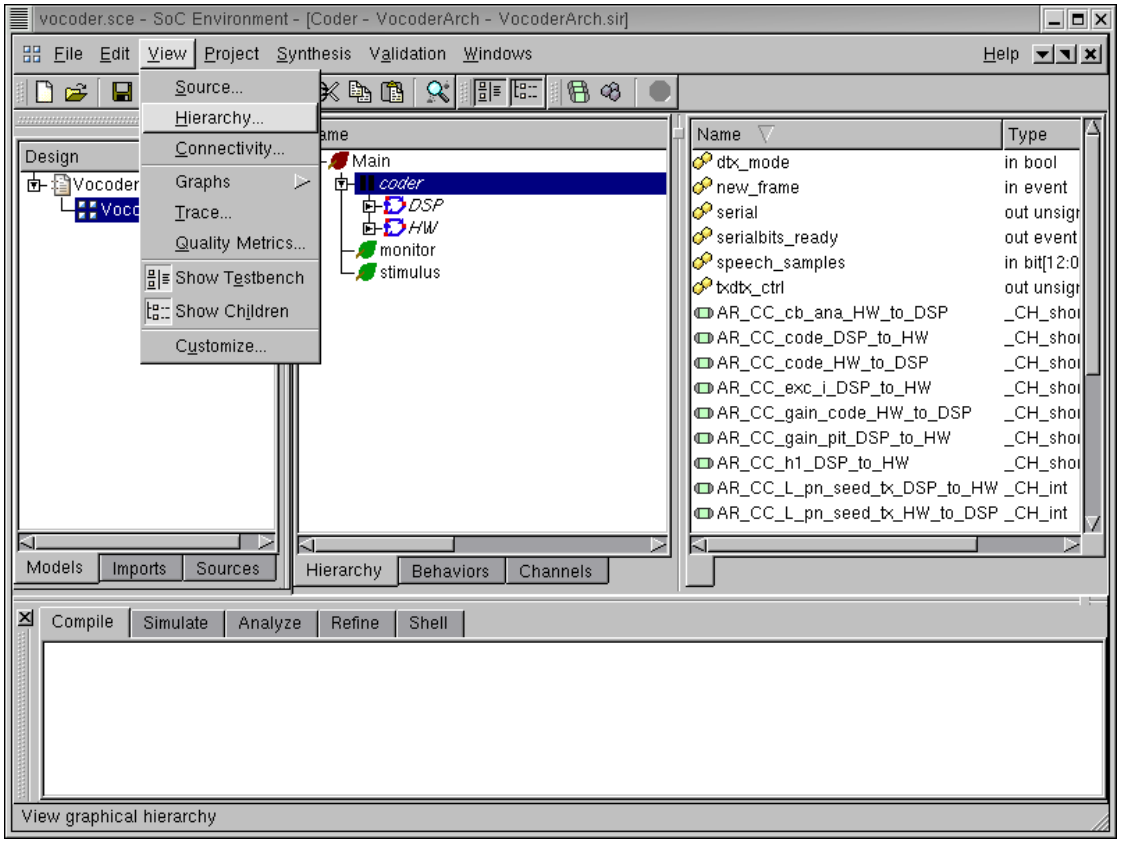
3.4.2. Architecture refinement (cont'd)



As displayed in the logging window, the architecture refinement is being performed. After the refinement, the newly generated architecture model VocoderSpec.arch.sir is displayed to the design window. It is also added to the current project window, under the specification model VocoderSpec.sir to indicate that it was derived from VocoderSpec.sir . Please note that, while the architecture refinement only took a few seconds to generate, a whole new model has been created. For example, 1693 lines of code were deleted, 2290 lines of new code were added and 387 lines of code were modified by the refinement tool.

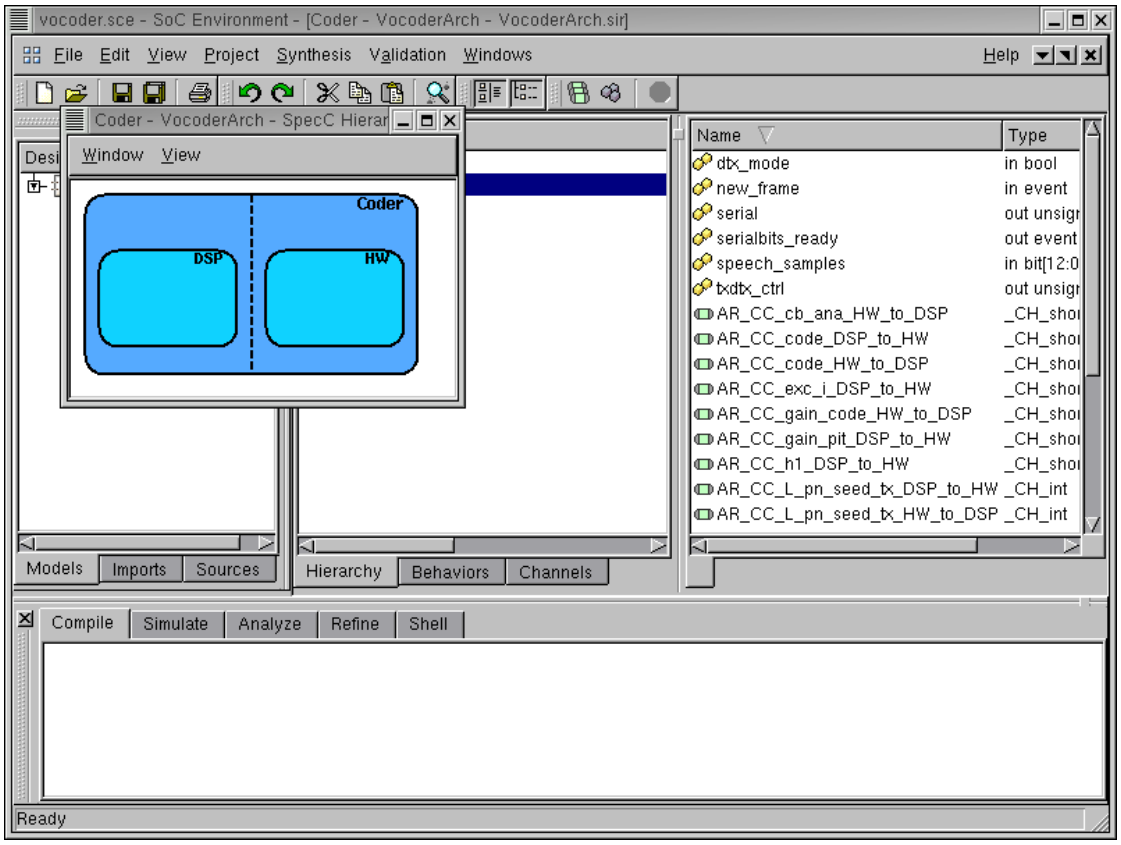
3.5. Browse architecture model

In this section we will look at the architecture model to see some of its characteristics.



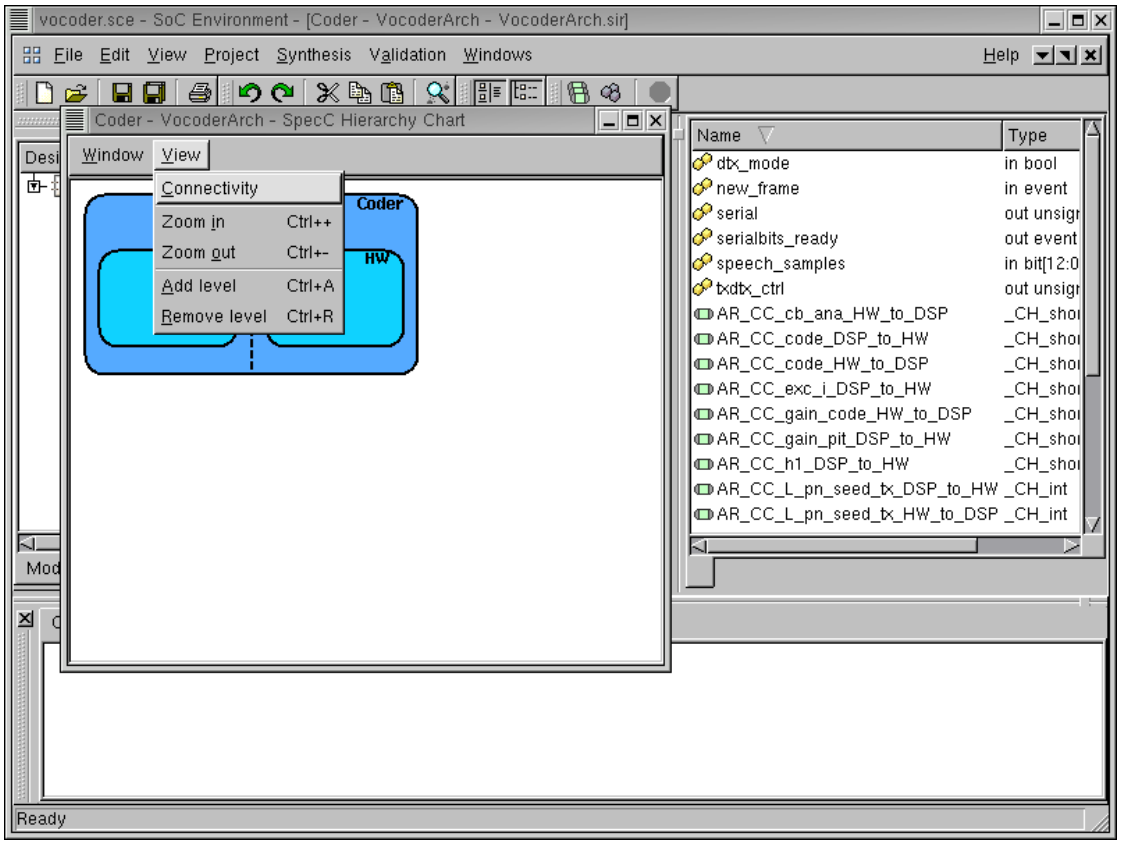
Since the top level behavior is Coder , the test bench behaviors are not changed during architecture refinement. Therefore let's select the Coder by clicking in the corresponding row in the design window. We would like to see how the design looks when it is mapped to the selected architecture. To view the hierarchy of the new Coder behavior, go to view menu and select Hierarchy .

3.5.1. Browse architecture model (cont'd)



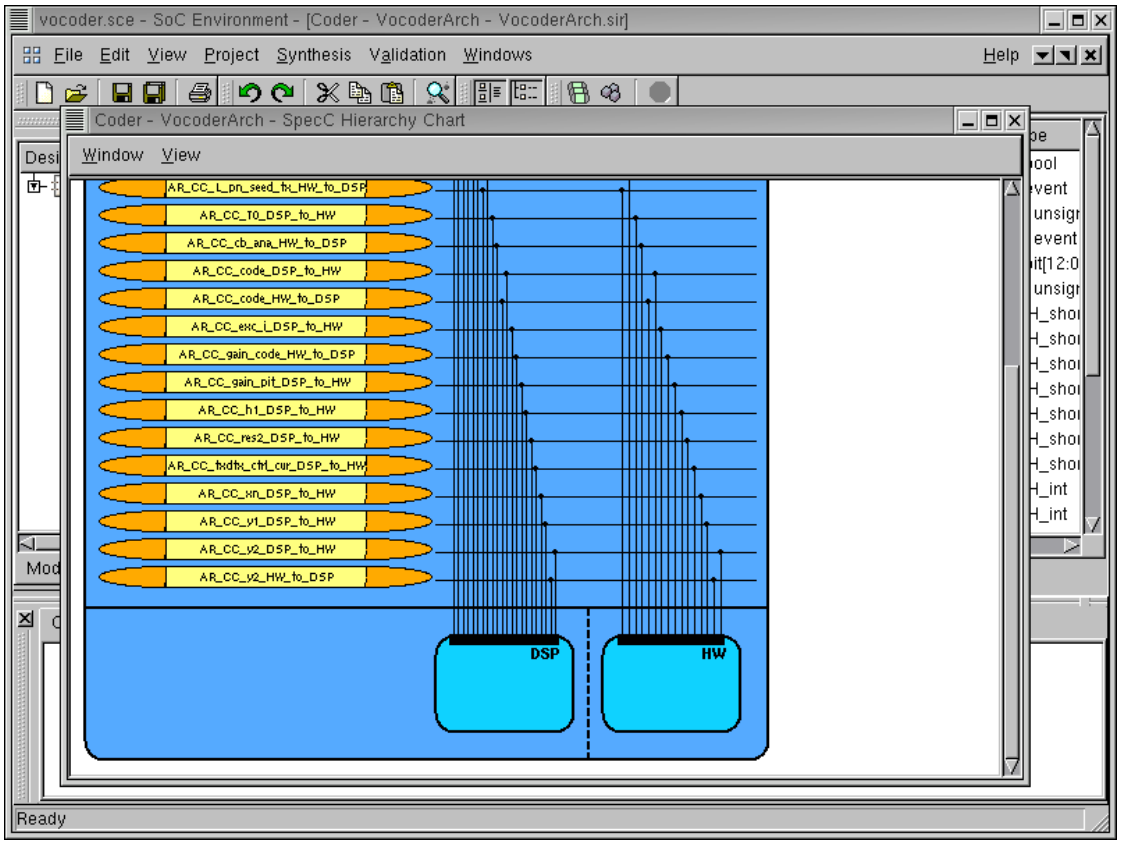
A window pops up, showing all sub-behaviors of the Coder behavior. As we can see, the top level behavior Coder in the architecture model is composed of two new behaviors, DSP and HW, which were constructed and inserted during architecture refinement. These behaviors at the top level indicate the presence of two componenets selected in the architecture. Note that they are also composed in parallel, which represents the actual semantics of the architecture model.

3.5.2. Browse architecture model (cont'd)



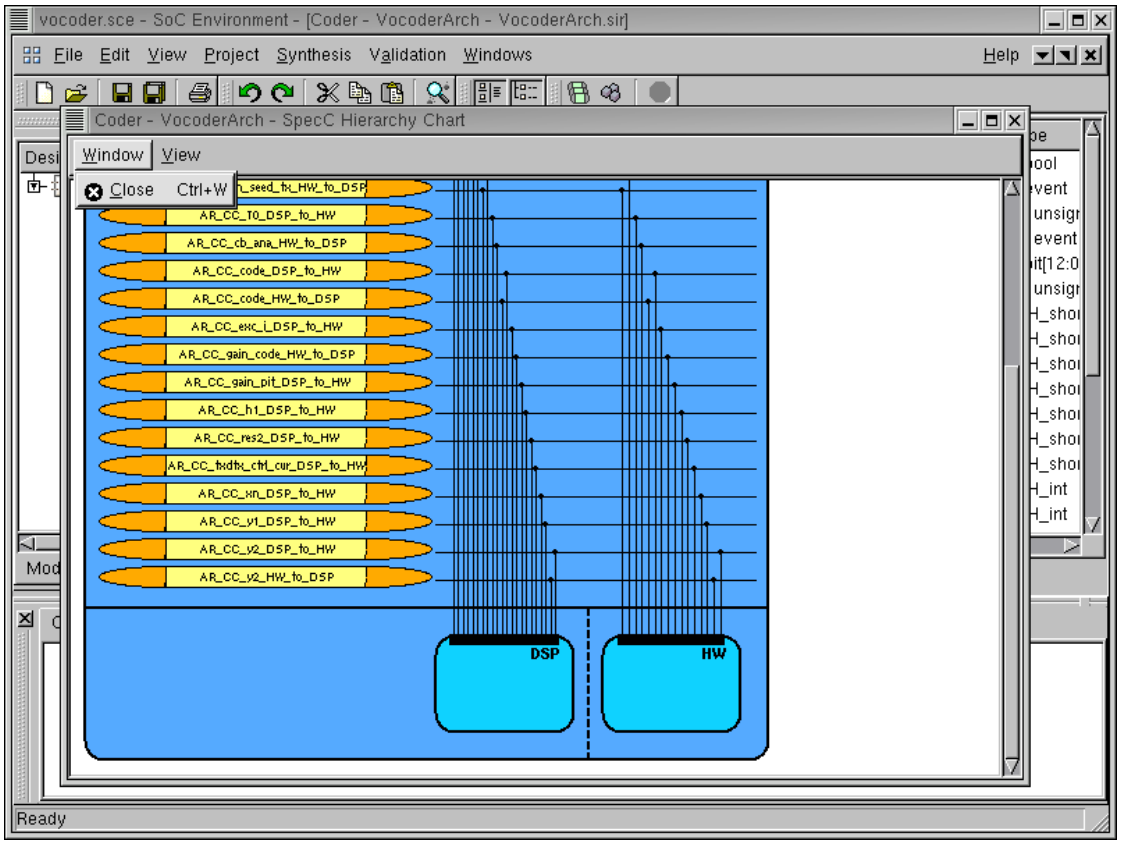
We would now like to see how the DSP and HW behaviors are communicating. This will verify if the refinement process was correctly executed. Go to View menu and select Connectivity to see the connectivity between DSP and HW components.

3.5.3. Browse architecture model (cont'd)



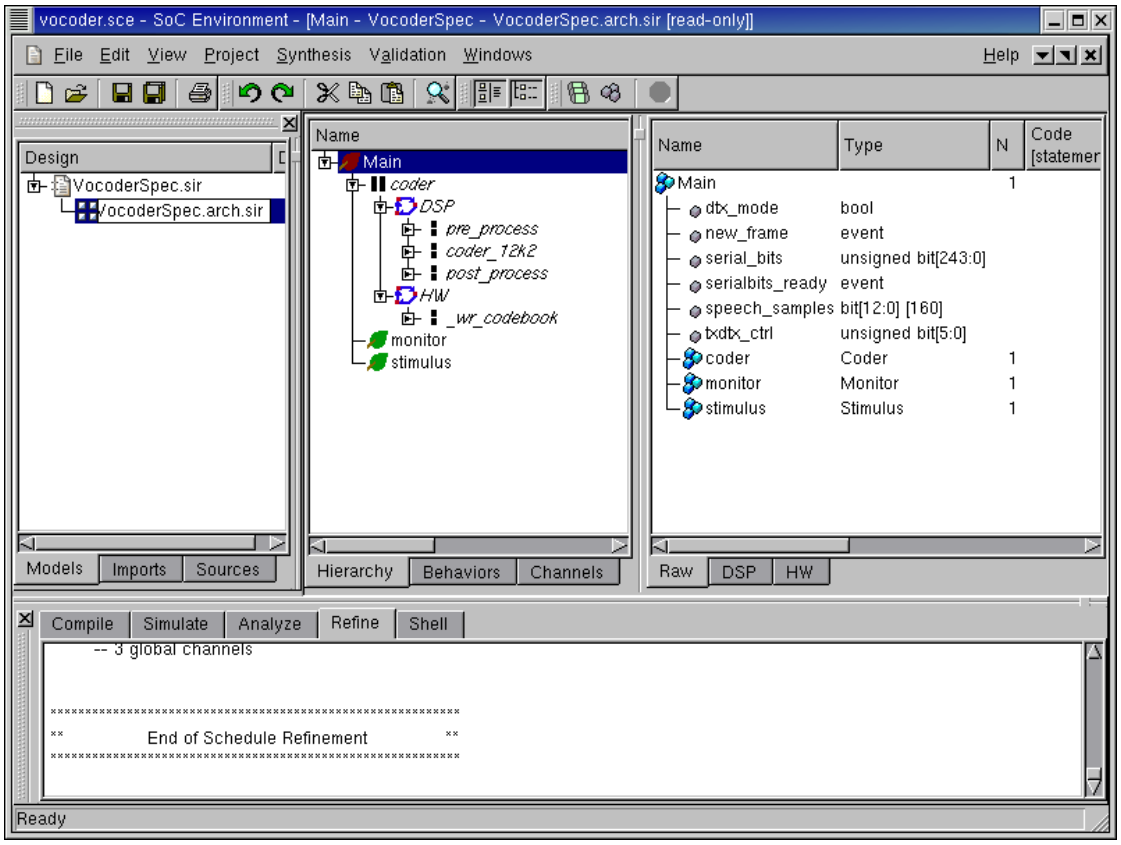
Enlarge the new window and scroll down to view the connectivity of the two components. We can see that DSP and HW components are connected through global variable channels, which were inserted during the architecture refinement. This is different from the original specification model, where only global variables were used for communication.

3.5.4. Browse architecture model (cont'd)



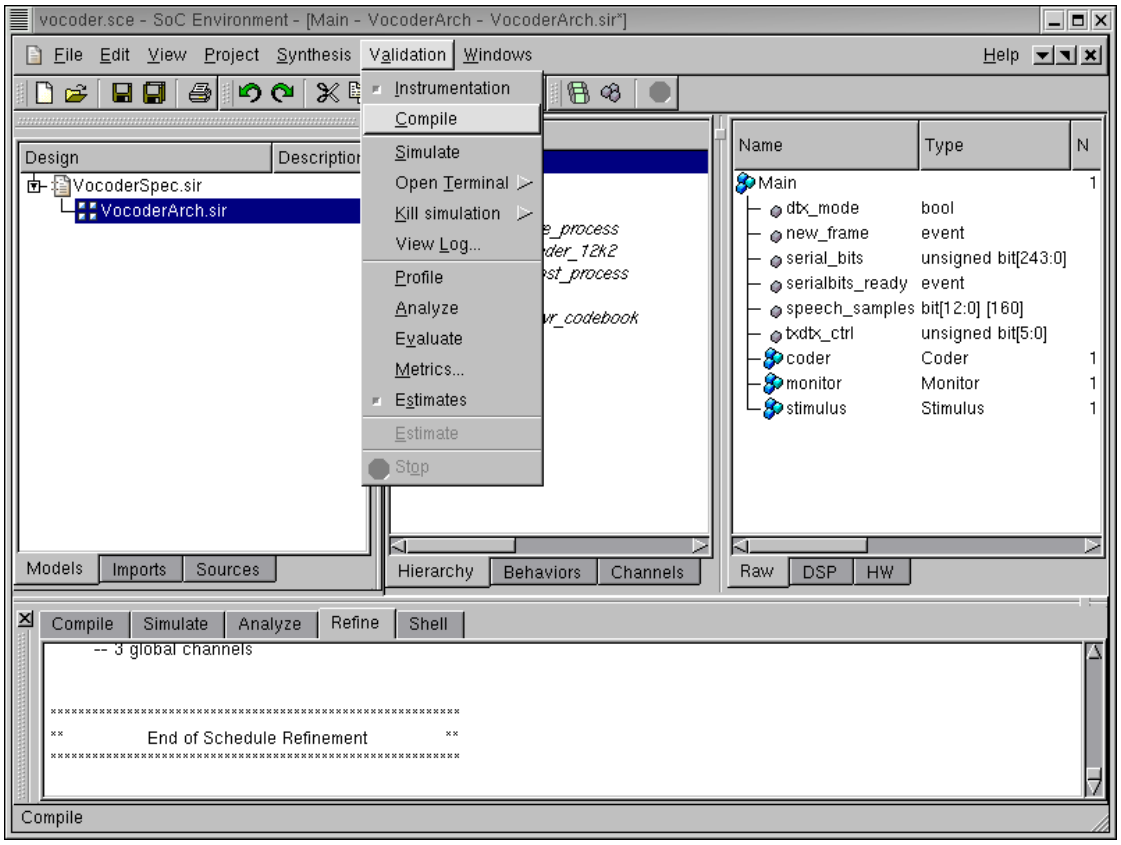
After checking the new architecture model, we can close the pop up window and go back to the design window.

3.6. Validate architecture model



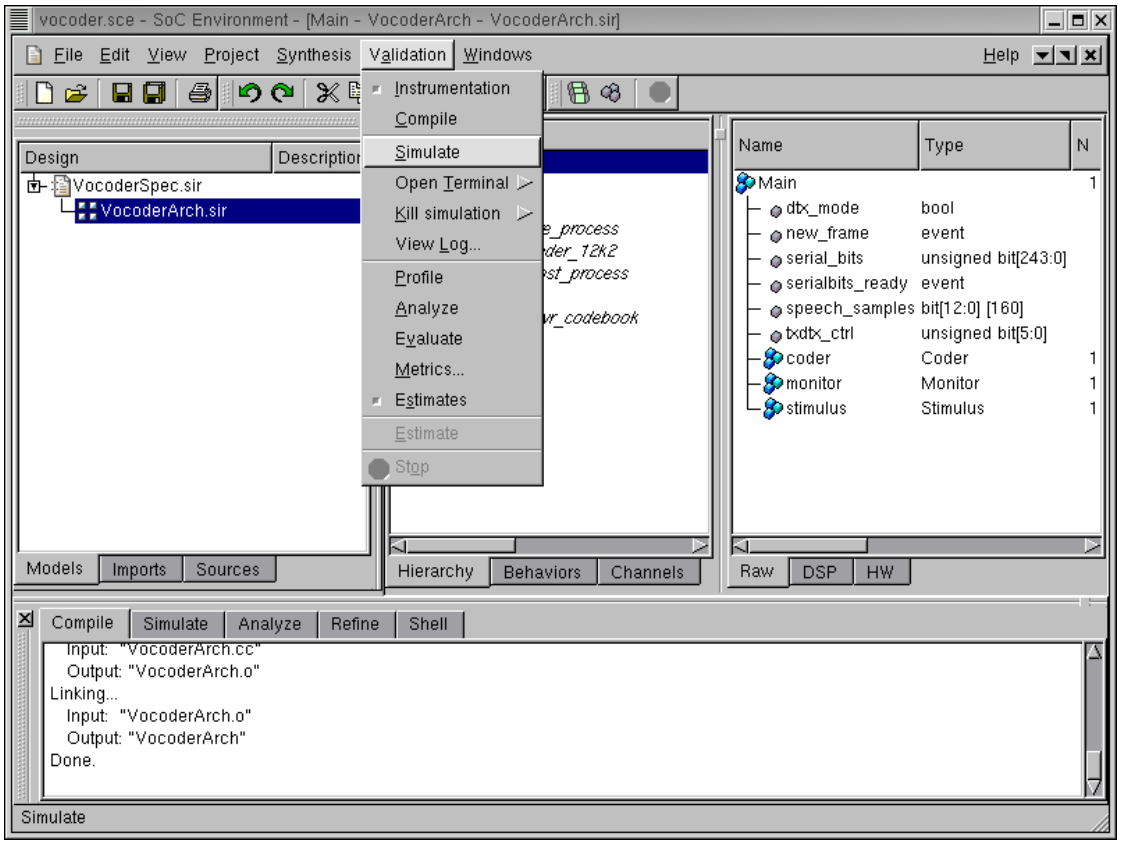
Like we did for the specification model, we also change the name of the new model to be VocoderArch.sir in the project window. The renaming is just for the purpose of maintaining a nomenclature schema and to correctly identify the individual models.

3.6.1. Validate architecture model (cont'd)



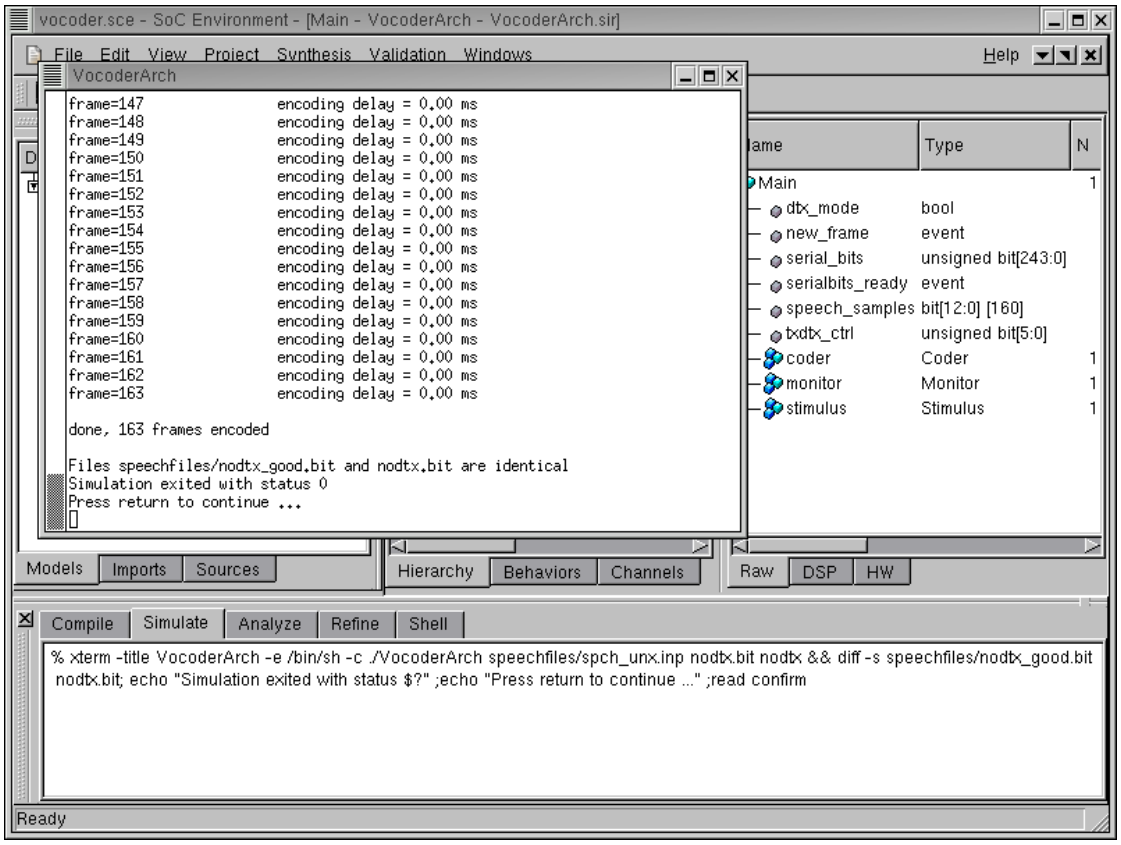
So far we have graphically visualized the automatically generated architecture. We have seen that in terms of its structural composition, the model meets the semantics of an architecture level model in our SoC methodology. However, we also need to confirm that the model has not lost any of its functionality in the refinement process. In other words the new model must be functionally equivalent to the specification. We will validate the architecture model through simulation. But first we need to compile the model into an executable. To compile the architecture model to executable, go to Validation menu and select Compile .

3.6.2. Validate architecture model (cont'd)



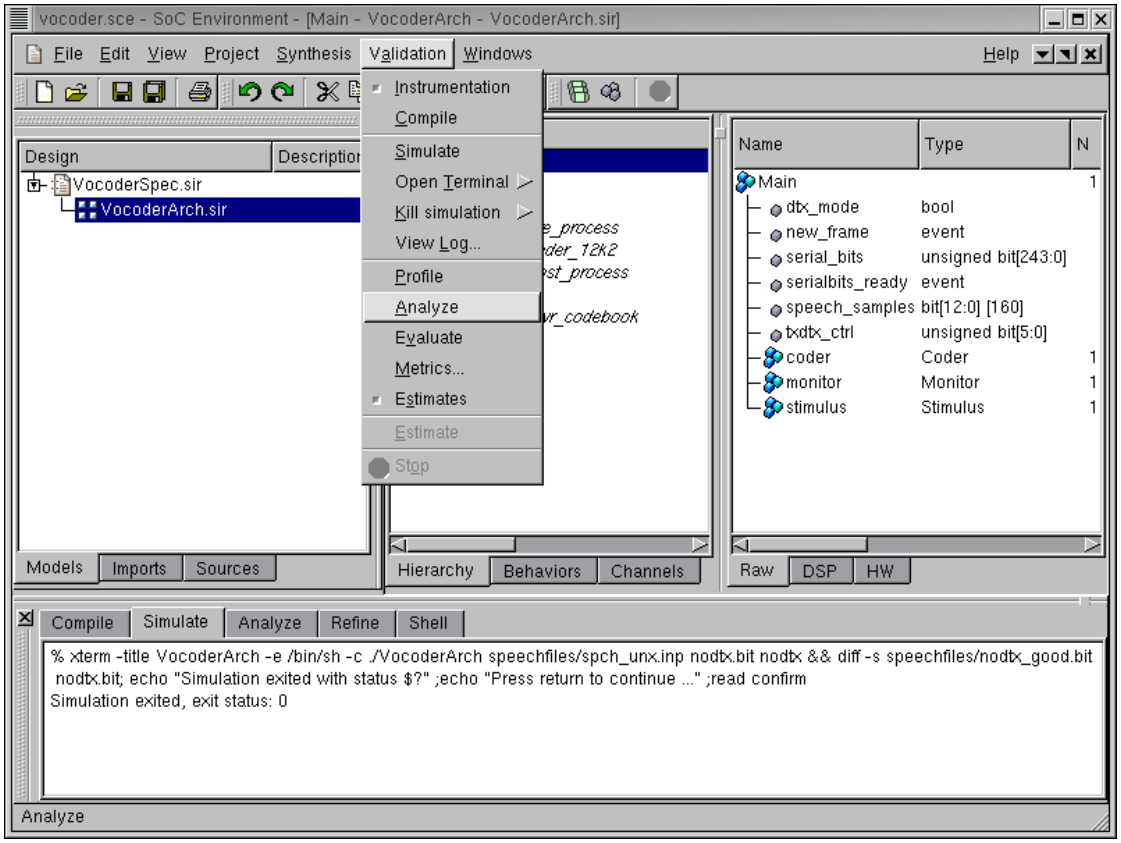
The messages in the logging window shows that the architecture model is compiled successfully without any syntax error. Now in order to verify that it is functionally equivalent to the specification model, we will simulate the compiled architecture model on the same set of speech data used in the specification validation. Go to Validation menu and select Simulate .

3.6.3. Validate architecture model (cont'd)



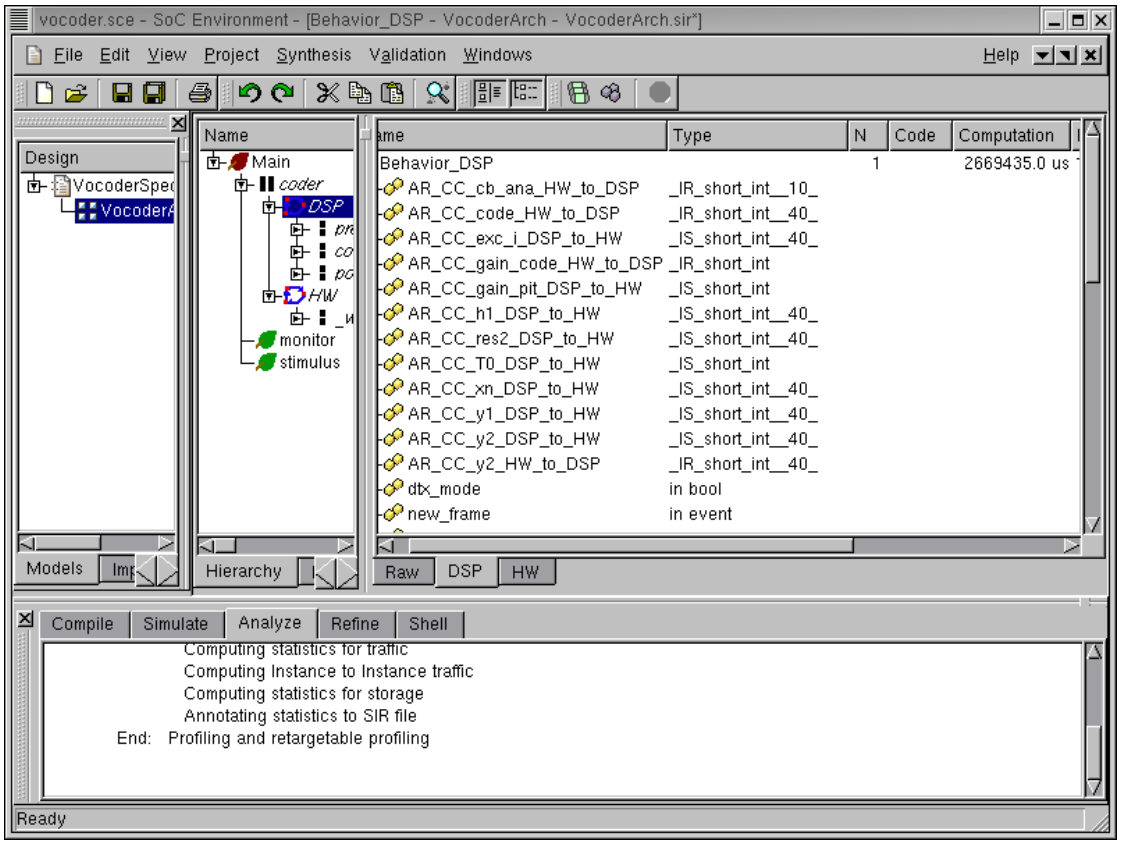
The simulation run is displayed in a new terminal window. As we can see, the architecture model was simulated successfully for all 163 frames speech data. The result bit file is also compared with the expected golden output given with the Vocoder standard. We have thus vreified that the generated architecture model is functionally correct. It must be noted as before that the testing process requires fairly intensive execution, but for the demo purposes we will omit multiple simulations and just show the concept.

3.7. Estimate performance



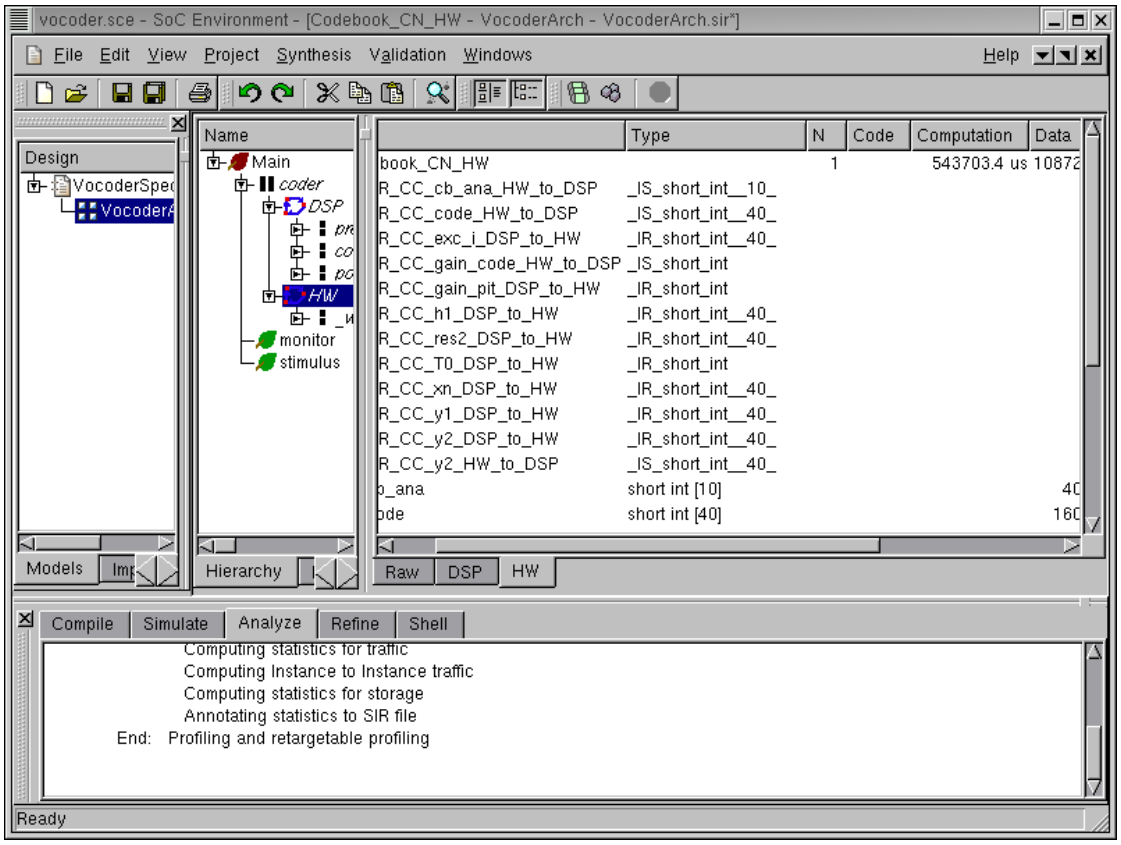
So far we have verified that the generated architecture model is semantically and functionally correct. However, the whole idea behind the architecture exploration process is to ensure that design constrints are met. If may be recalled that we abondoned the pure software implementation because it failed on meeting the timing constraint. It is now time for us to verify if the timing is met by using the combined software/hardware design. To evaluate this software and hardware partitioning, go to Validation menu and select Analyze .

3.7.1. Estimate performance (cont'd)



As we can see in the logging window, a profiling retargeted at the DSP is being performed. When it finishes, the profiled data is presented in the design window. Clicking DSP in the hierarchy tree, we find out that the execution of software part (behavior Behavior_DSP) takes 2.67 seconds.

3.7.2. Estimate performance (cont'd)



Clicking HW in the hierarchy tree, we can see that the execution time for the hardware part (behavior Codebook_CN_HW) is 0.54 seconds. Since the Codebook behavior was executed in sequential composition with the rest of the design, the latency of the design is the sum of DSP and HW execution time, which is 3.21 seconds. Recall that the timing requirement is to be less than 3.26 seconds for the given speech data. Therefore, the current architecture and mapping are acceptable. We have therefore found a suitable architecture that we may map our specification to This concludes the step of architecture exploration.

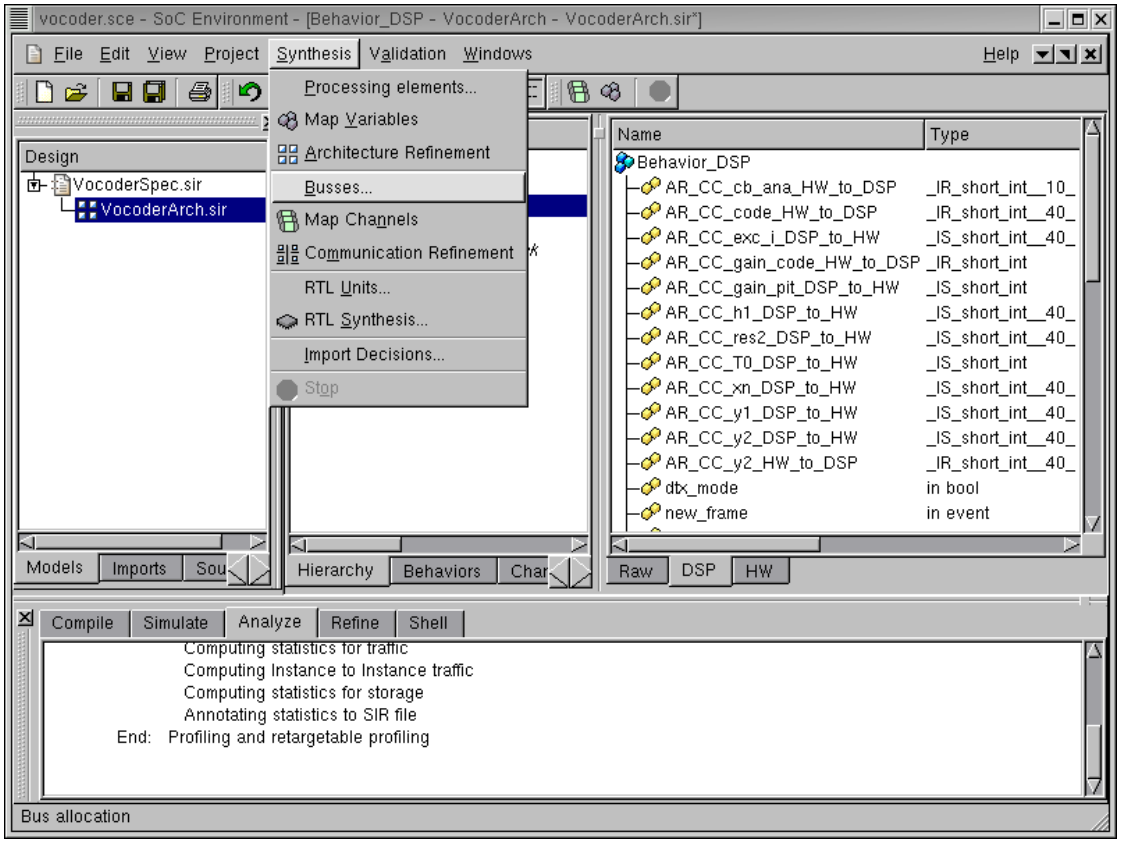
Chapter 4. Communication Synthesis

Communication synthesis is the second part of the system level synthesis process. It refines the abstract communication between components in the architecture model. Specifically, the communication with variable channels is refined into an actual implementation over wires of the system bus. The steps involved in this process are as follows.

We begin with allocation of system buses and selection of bus protocols. A set of system buses is selected out of the bus library and the connectivity of the components with system buses is defined. In other words, we determine a bus architecture for our design.

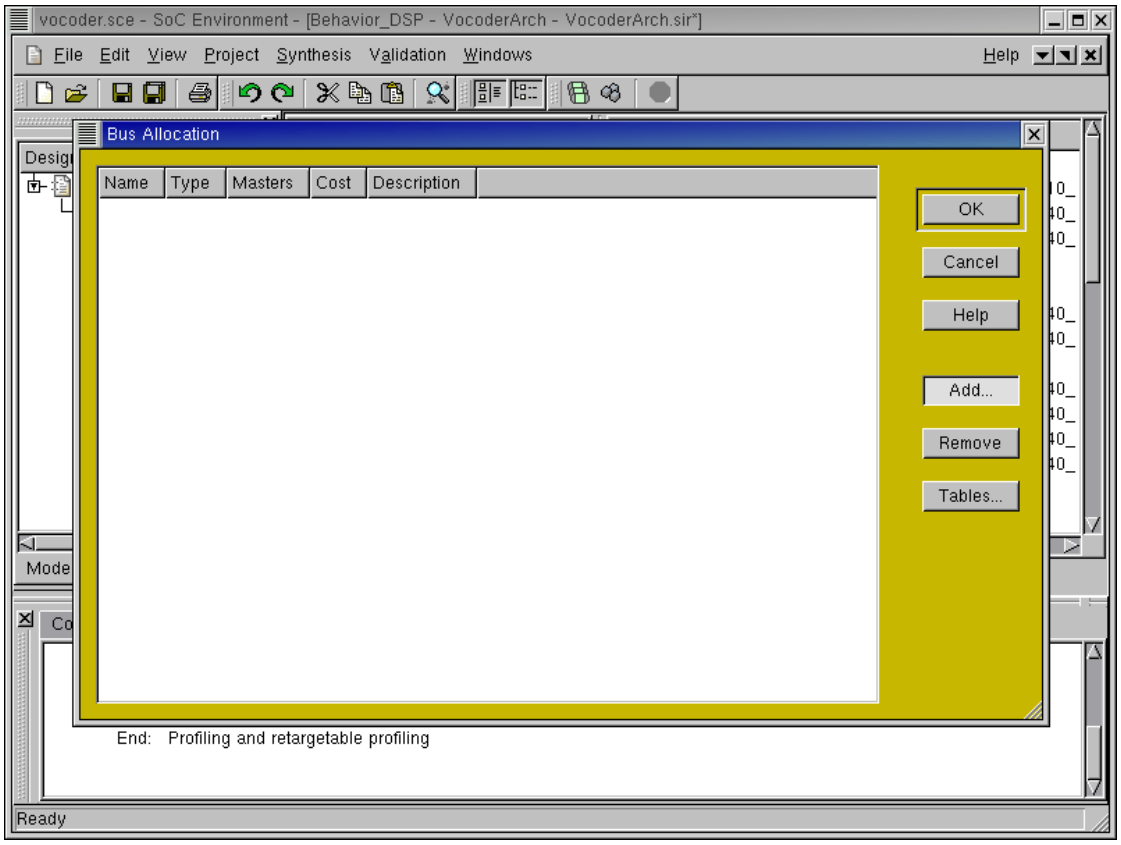
This is followed by grouping of abstract variable channels . The communication between system components has to be implemented with busses instead of variable channels. Thus these channels are grouped and assigned to the chosen system busses. Once this is done, the automatic refinement tool produces the required bus drivers for each component. It also divides variables into slices whose size is the same as width of the data bus. Therefore that each slice can be sent or recieved using the bus protocol. The entire variable is sent or recieved using multiple transfers of these slices.

4.1. Select bus protocols



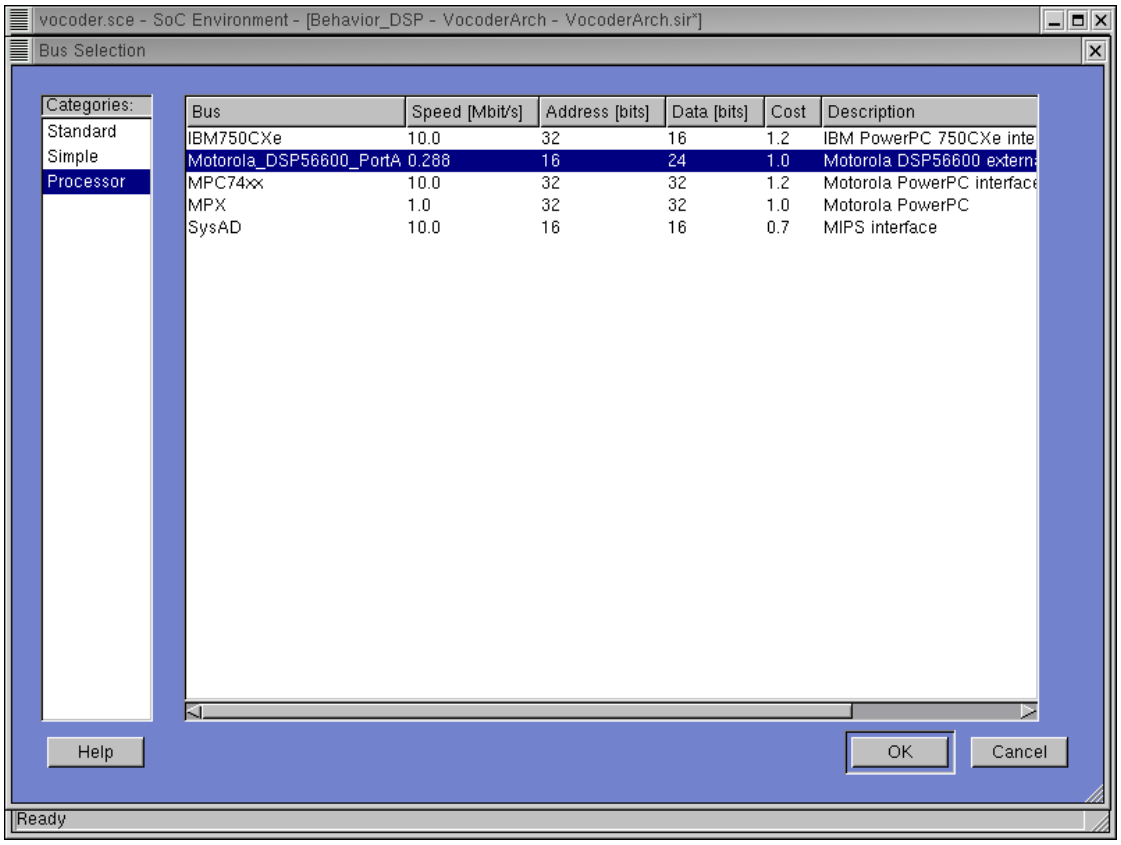
As explained earlier, we begin by selecting a suitable bus for our system. Note that in the presence of only two components, one bus would suffice. However, in general the user may select multiple buses if the need arises. Bus allocation is done by selecting Synthesis$\longrightarrow$Busses from the menu bar.

4.1.1. Select bus protocols (cont'd)



A new window pops up showing the bus allocation table. Since there are no busses selected at the time, this table is empty. We now click on Add to add bus(es) from the protocol database.

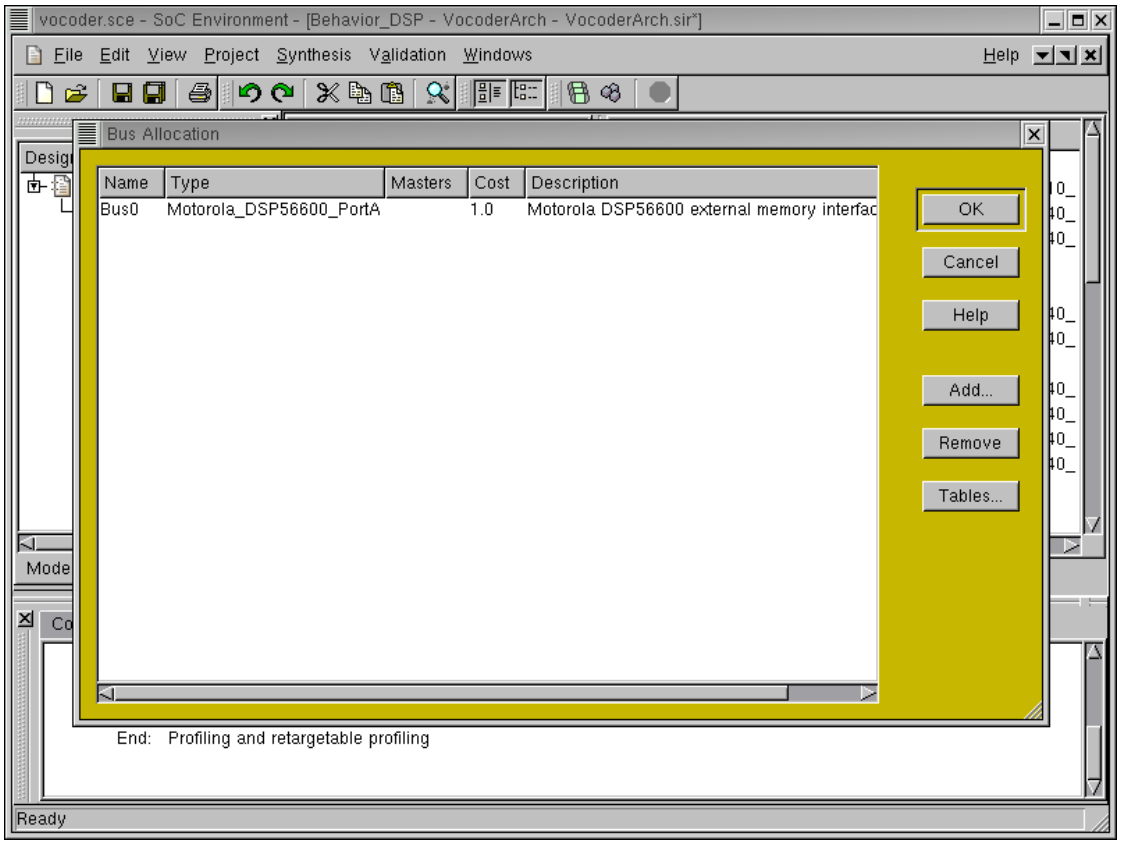
4.1.2. Select bus protocols (cont'd)



A new window pops up showing the contents of the protocol database. The column on the left shows the three categories of protocols. During component selection for architecture exploration, we had a classification of components. Likewise, the classification here shows us the available types of busses. On selecting a particular category by Left click, the busses under that category are displayed. For our demo purposes, we select the Processor bus "Motorola_DSP56600_PortA" and click OK.

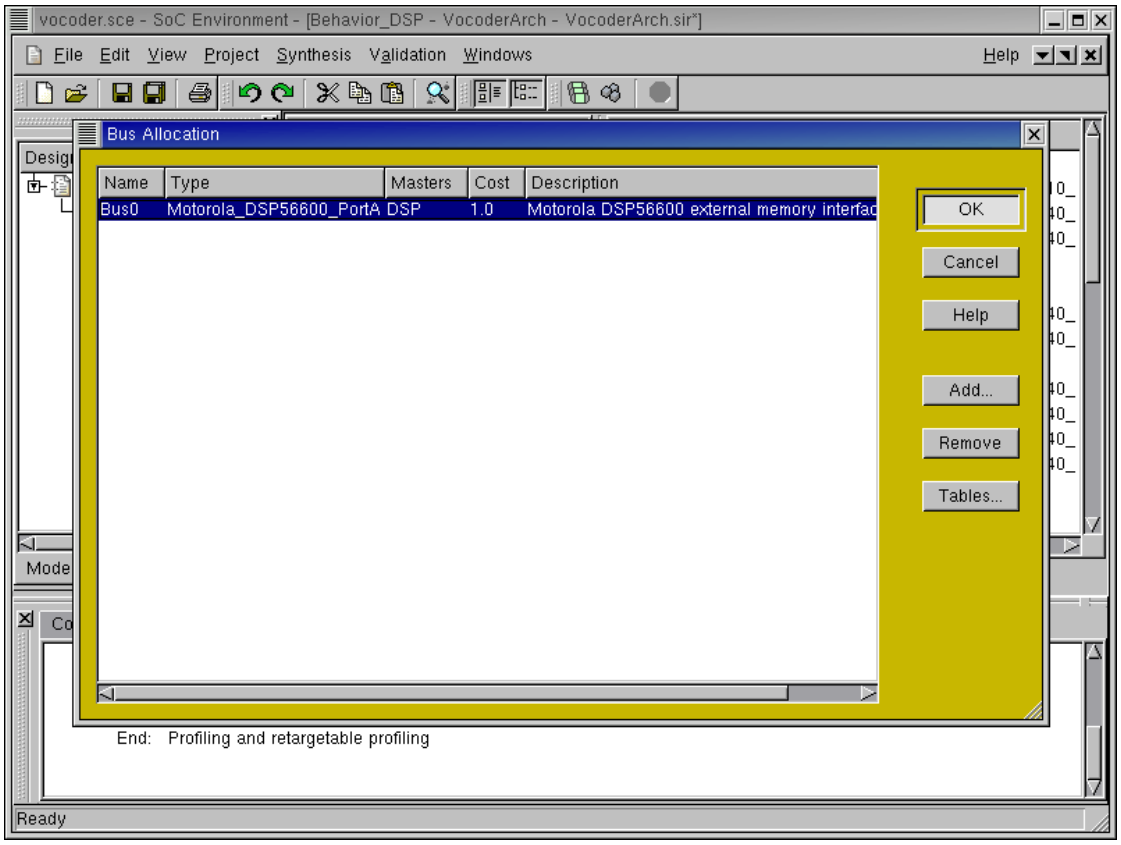
Note that the architecture chosen for the design has an impact on the selection of busses. More often that not, the primary component in the design dictates the bus selection process. In this case, we have a DSP with an associated bus. It makes sense for the designer to select that bus to avoid going through the overhead of generating a custom bus adapter.

4.1.3. Select bus protocols (cont'd)



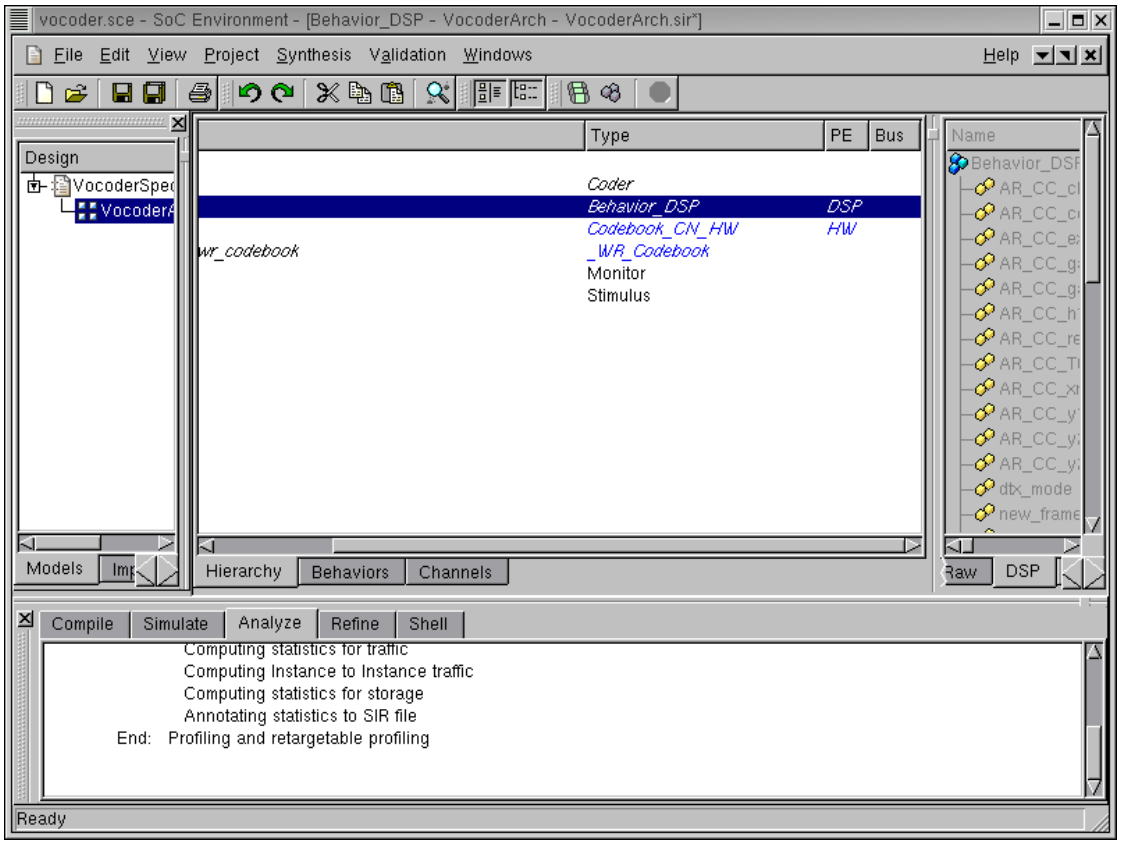
The selection is now displayed in the bus allocation table as shown in the screen shot. A default name of "Bus0" is given to identify this system bus. In order to include this bus in the design, we need to specify which component is going to be the master on the bus. This is done by Left click under the "Masters" column. Since this bus is for the Motorola 56600 processor that we have chosen, the master is the processor. Recall that the name given to the processor component was "DSP." We thus enter the name "DSP" under the "Masters" column and press RETURN.

4.1.4. Select bus protocols (cont'd)



The bus selection is now complete and we can finish off with the allocation phase by Left clicking on OK.

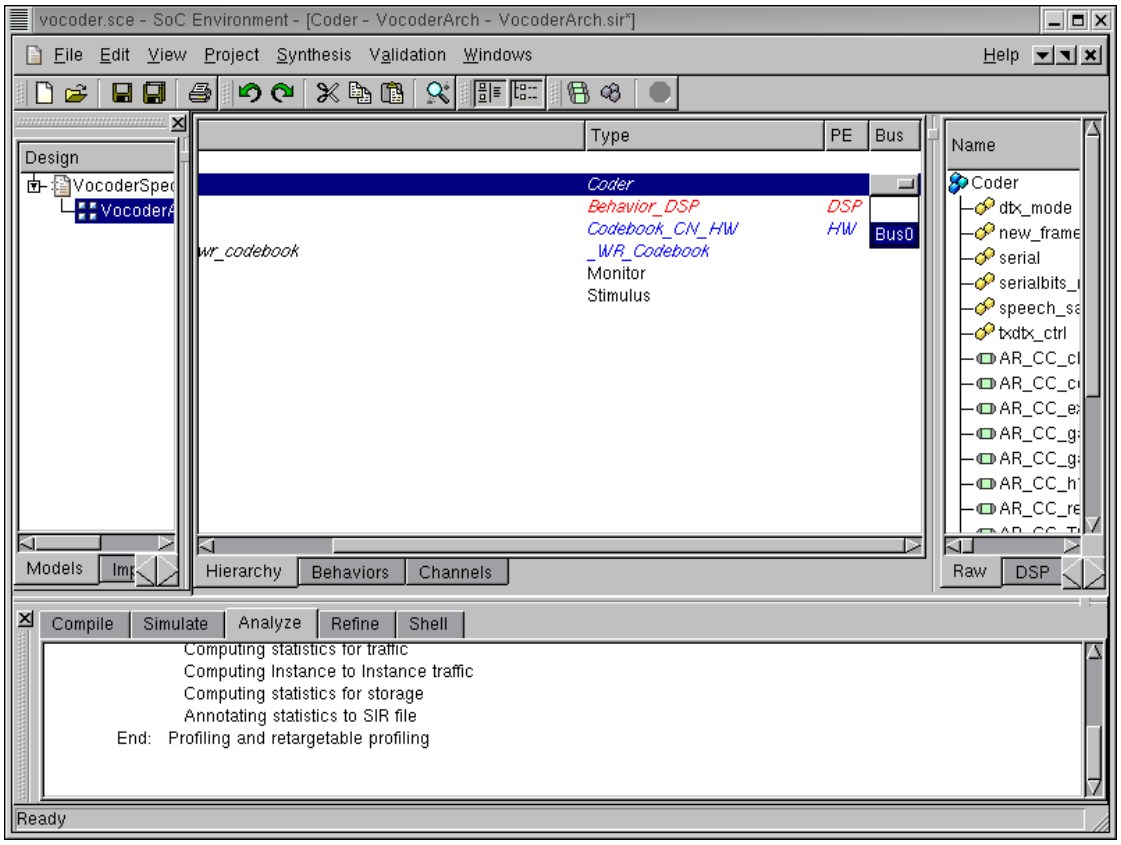
4.2. Map channels to buses



Once the bus allocation has been done, we need to group the channels of the architecture model and assign them to the system buses. Recall that in the architecture model, we had communication between components with abstract variable channels. We now have to assign those variable channels to the system bus.

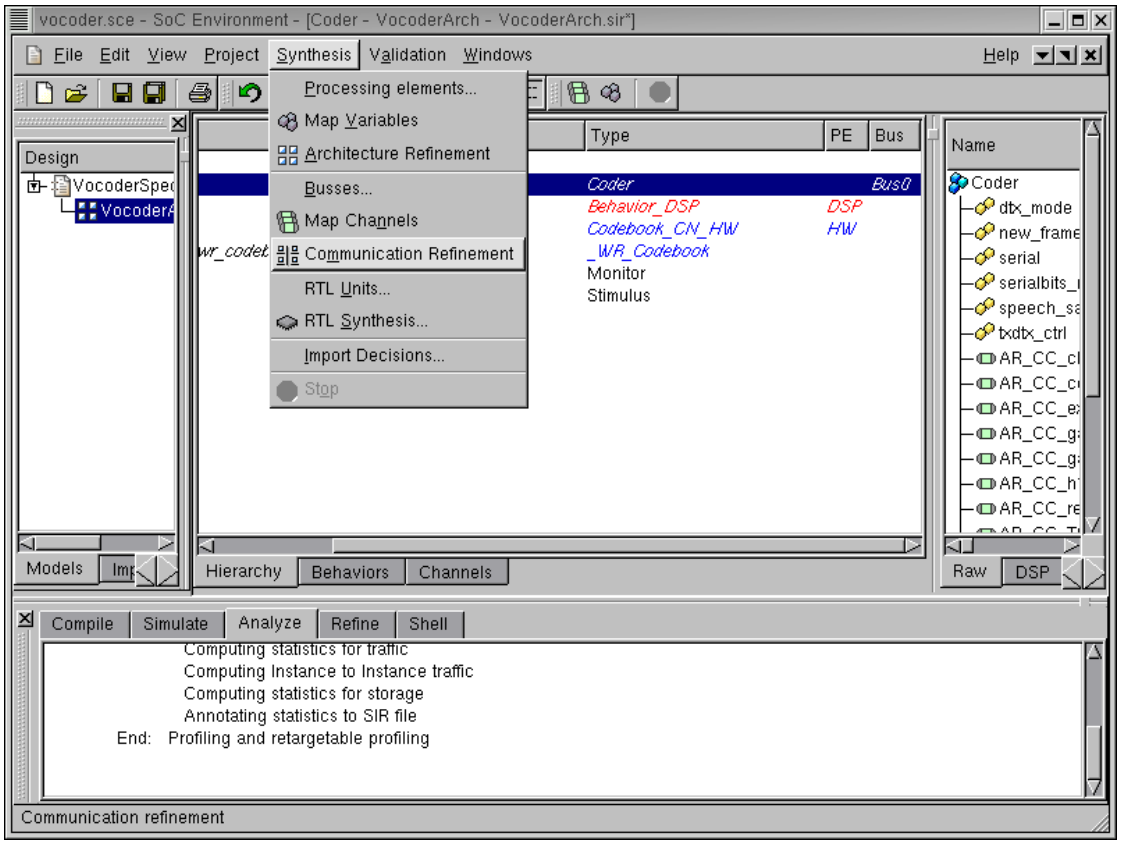
Expand the design hierarchy window and scroll to the right to find a new column entry for Bus.

4.2.1. Map channels to buses (cont'd)



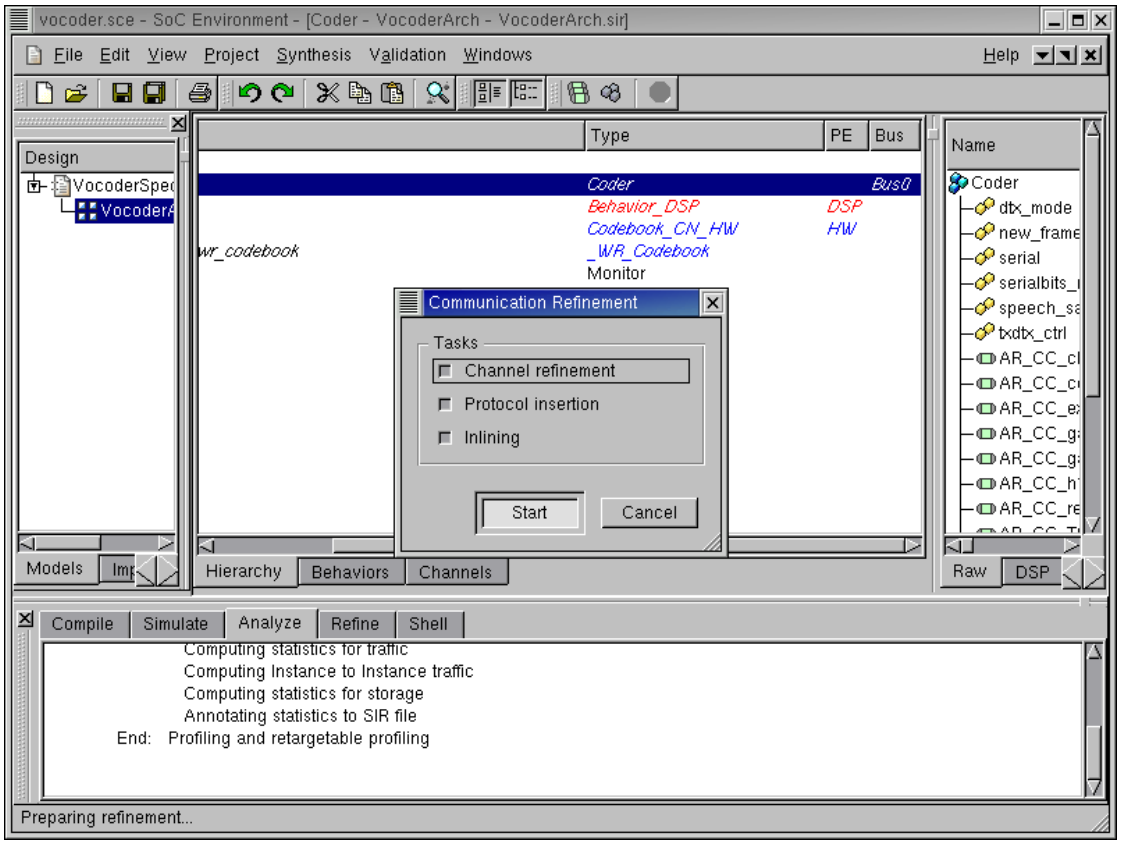
Like component mapping, bus mapping may be done by assigning variable channels to buses. However, to speed things, we may assign the top level component to our system bus. Since we have only one system bus, all the channels will be mapped to it. This is done by Left clicking in the row for the "Coder" behavior under the bus column. Select the defualt "Bus0" and press RETURN.

4.3. Communication refinement



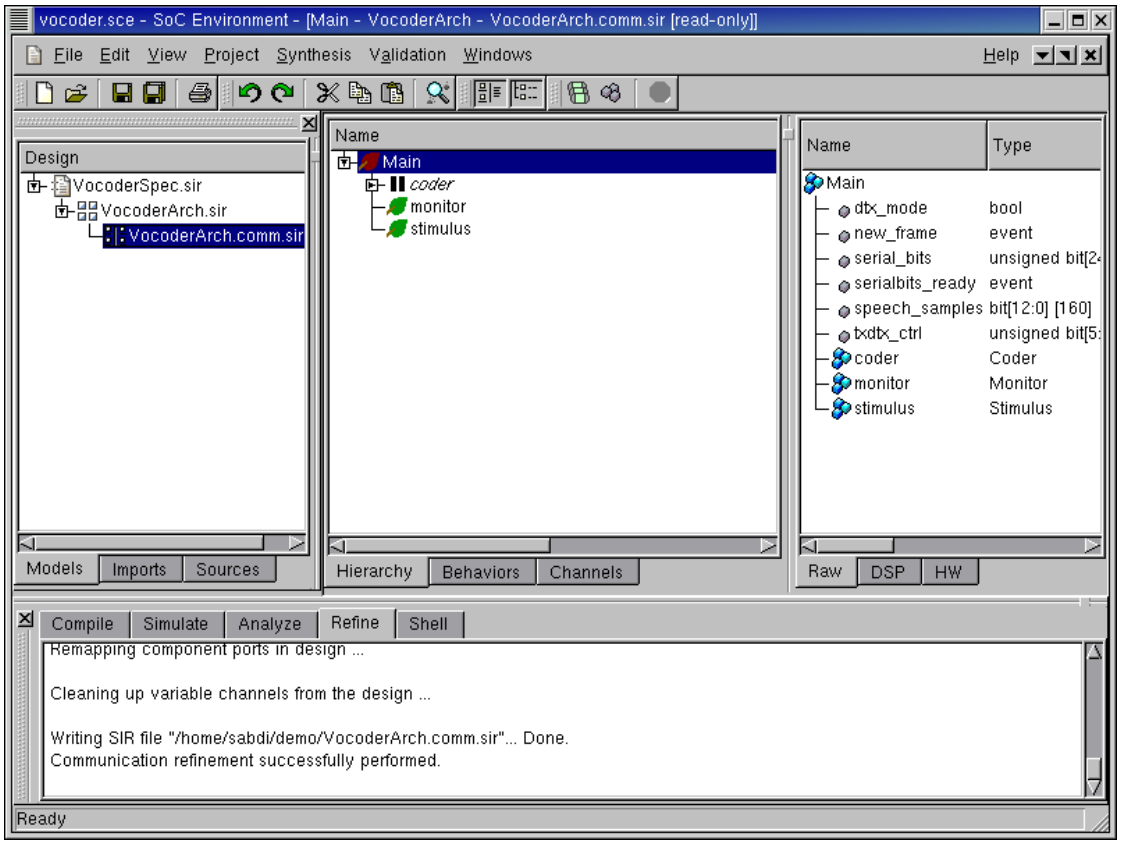
Now that we have completed bus allocation and mapping, we may proceed with communication refinement. Like architecture refinement, this process automatically generates a new model that reflets our desired bus architecture. To invoke the communication refinement tool, select Synthesis⟶Communication Refinement from the menu bar.

4.3.1. Communication refinement (cont'd)



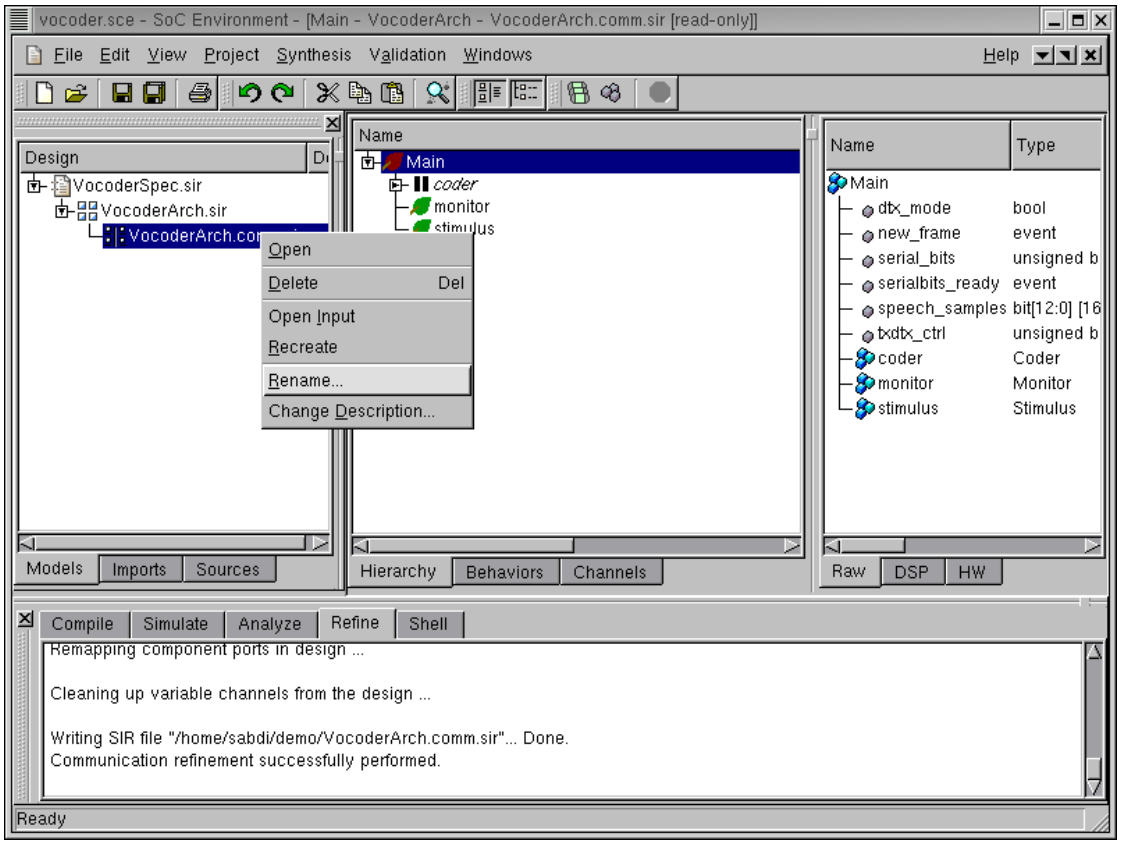
A new window pops up giving the user the option to perform various stages of the refinement. The user may choose to partially refine the model without actually inserting the bus, and only selecting the channel refinement phase. This way, he can play around with different channel partitions. Likewise, the user might want to play around with different bus protocols while avoiding "Inlining" them into components. This way he can plug and play with different protocols before generating the final inlined model. By default all the stages are performed to produce the final communication refinement. Since we have only one bus, and hence a default mapping, we opt for all three stages and Left click on Start to proceed.

4.3.2. Communication refinement (cont'd)



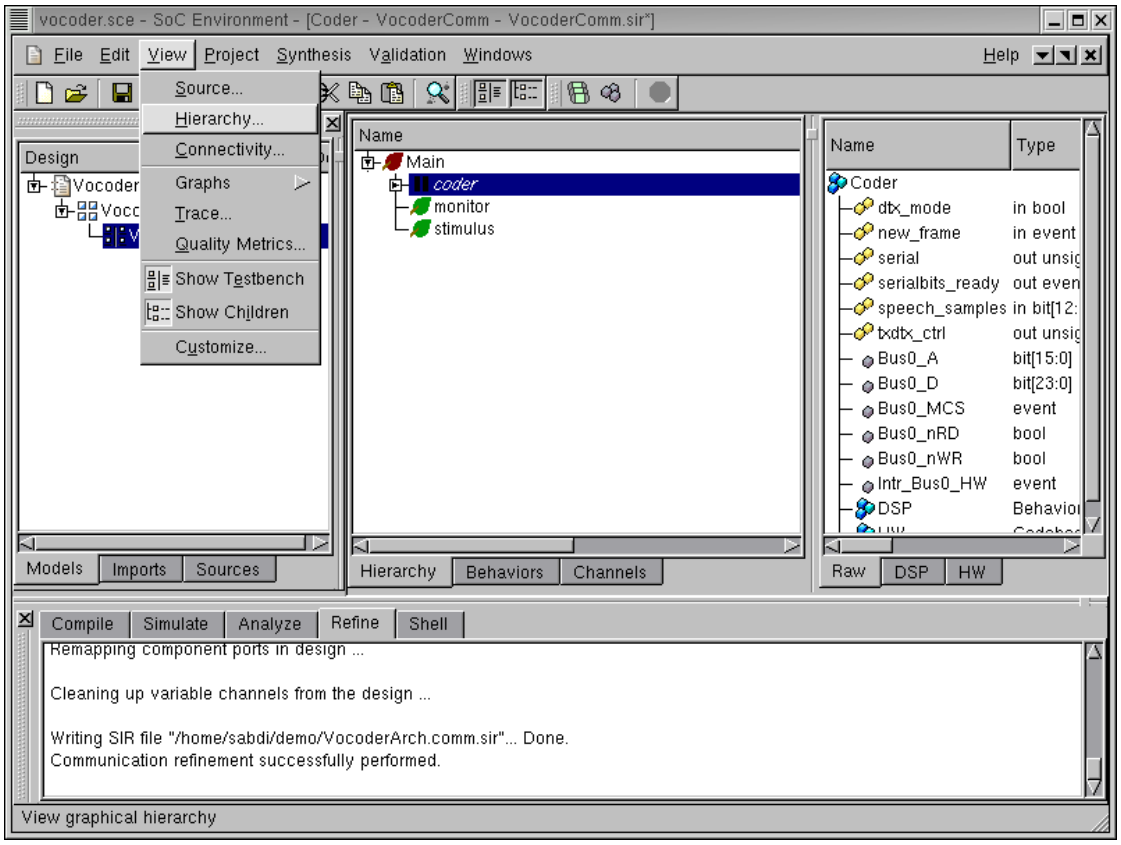
During communication refinement, note the various tasks being performed by the tool in the logging window. The tool reads in channel partitions, groups them together, imports selected busses and their protocols, implements variable channel communication on busses and finally inlines the bus drivers into respective components. Once communication refinement has finished, a new model is added in the project manager window. It is named VocoderArch.comm.sir. Also note that we have a new design management window on the right side in the GUI.

4.3.3. Communication refinement (cont'd)



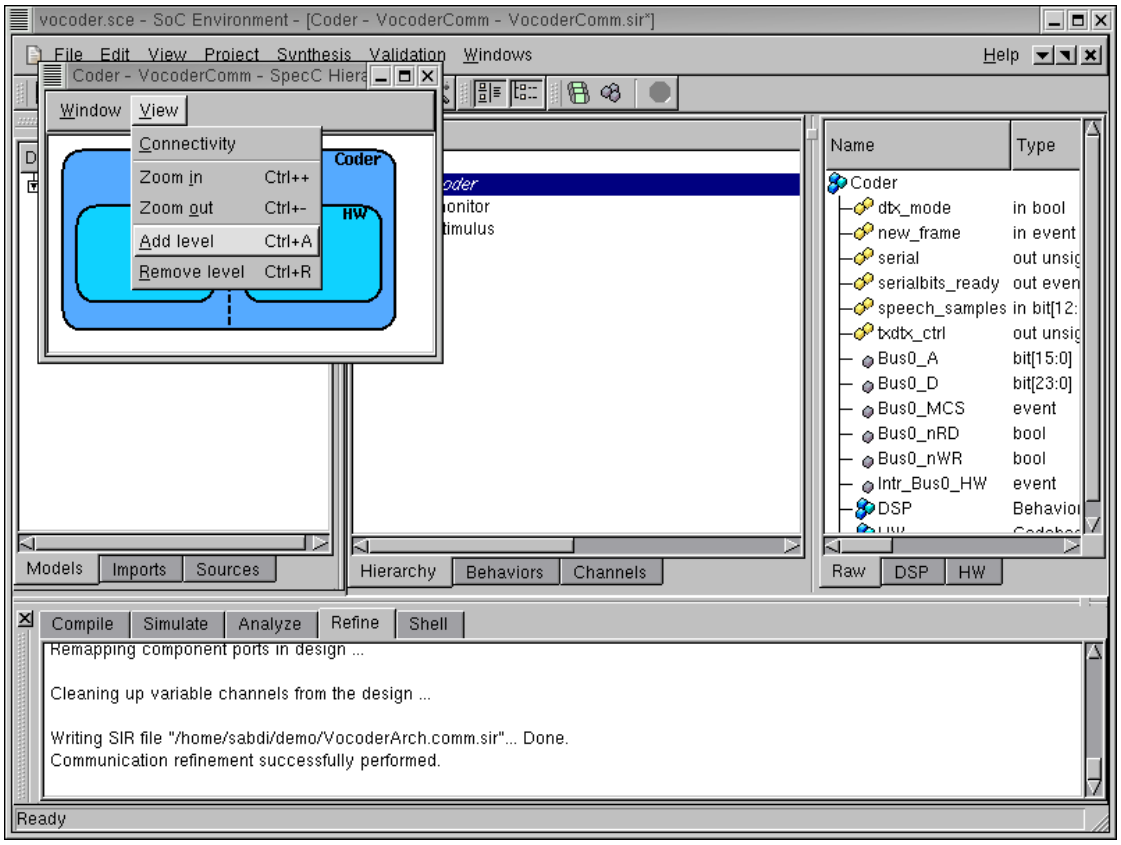
We now need to give our newly created communication model a reasonable name. To do this Right click on "VocoderArch.comm.sir" in the project manager window and select Rename from the poped up menu. Now rename the model to VocoderComm.sir.

4.4. Browse communication model



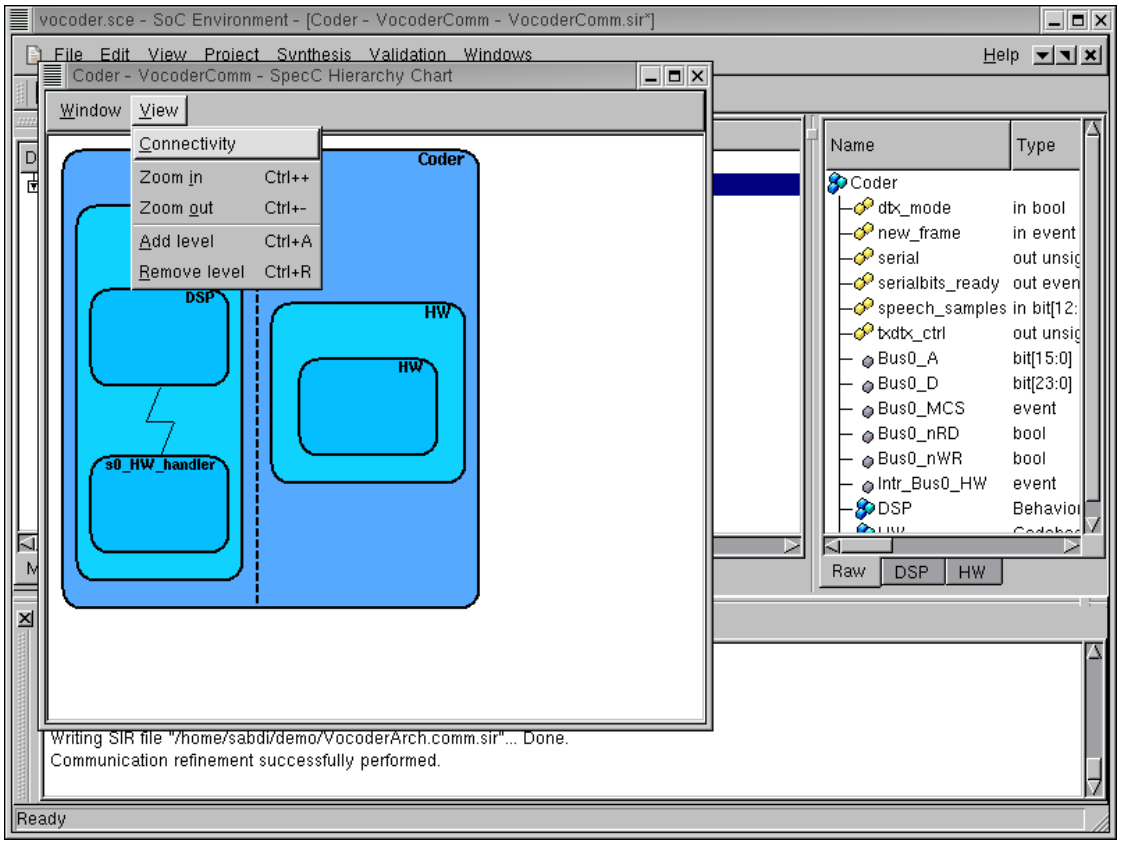
Like we did after architecture refinement, we browse through the communication model generated by the refinement tool. We have to first check whether it is semantically and structurally representing a model as described in our SoC methodology. To observe the model transformations produced by communication refinement, we need a graphical view of the model. This is done by Left clicking to choose the "Coder" behavior in the design hierarchy window and selecting View⟶Hierarchy from the menu bar.

4.4.1. Browse communication model (cont'd)



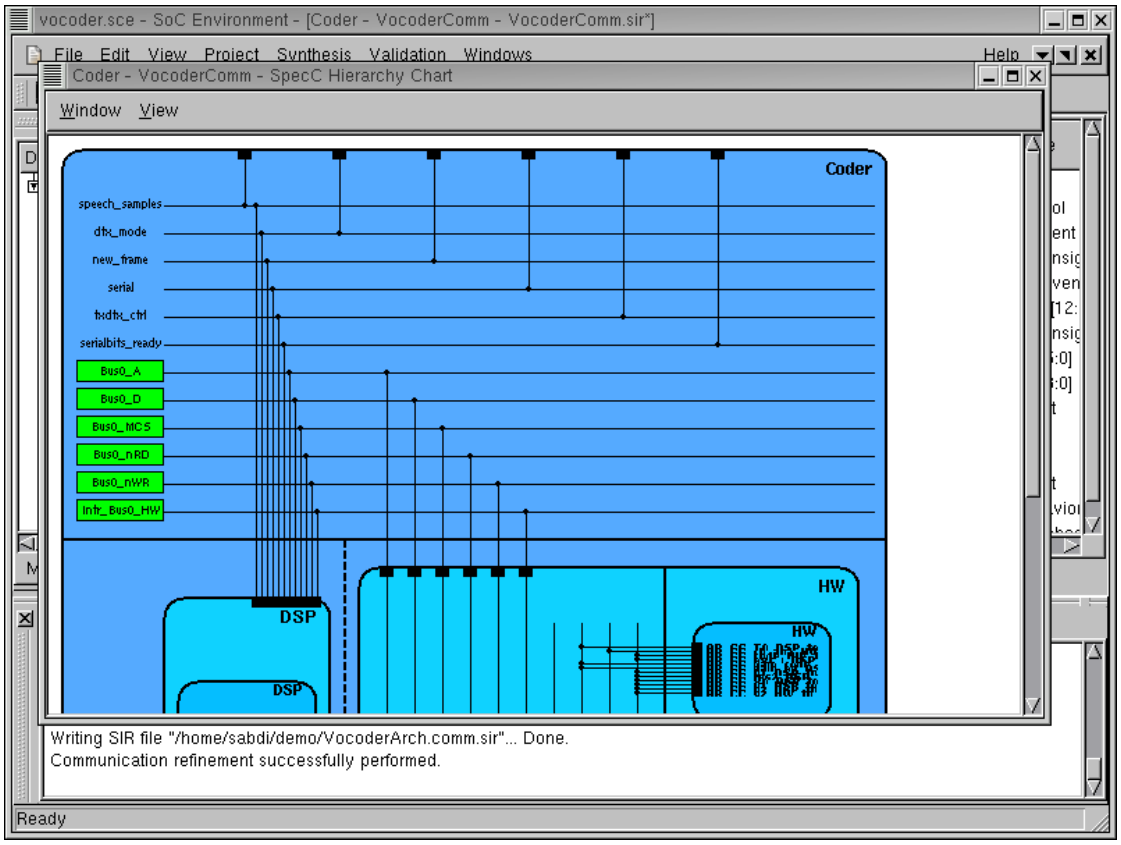
A new window pops up showing the model with DSP and HW components. We have to observe the bus controllers generated during refinement and the added details to the model. Hence, we select View⟶Add level from the menu bar to view the model with greater detail.

4.4.2. Browse communication model (cont'd)



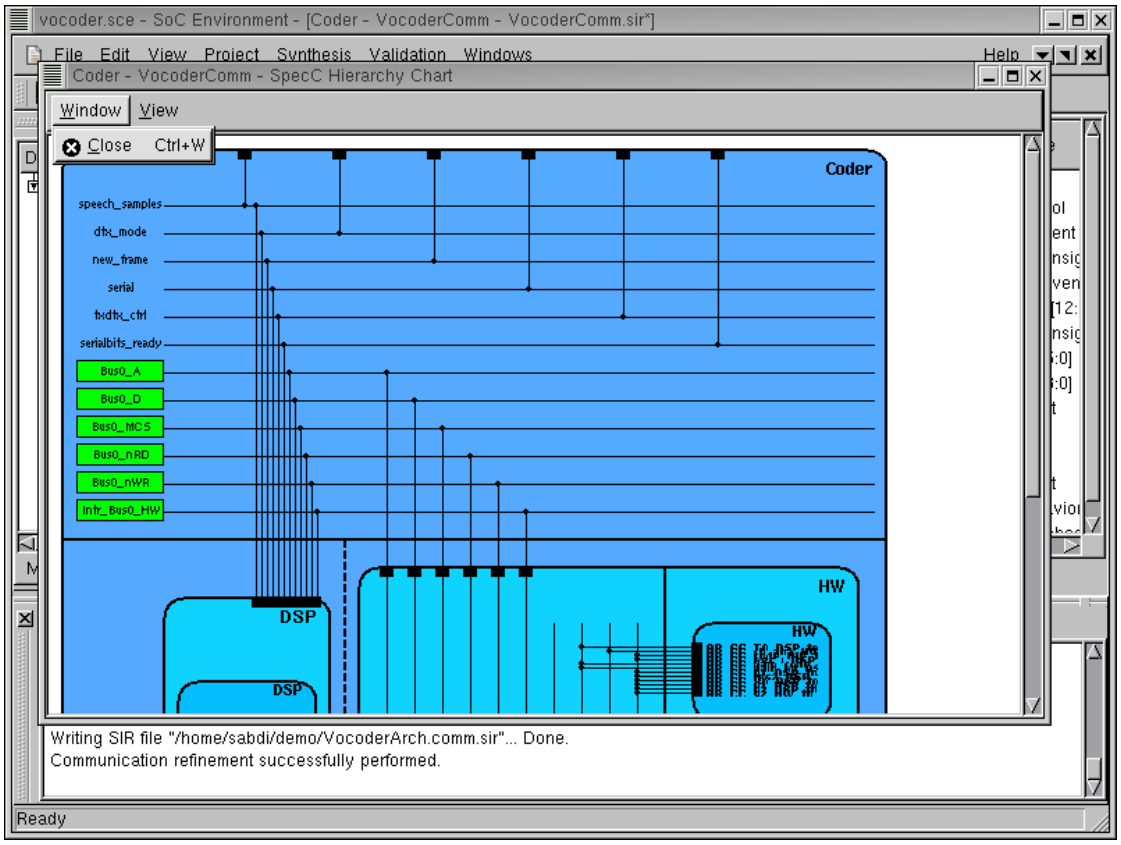
In the next level of detail, we can now see the interrupt handler "s0_HW_handler" behavior added in the master to serve interrupts from the HW slave. To view the actual wire connections of the system bus, enlarge window and select View—→Connectivity from the menu bar.

4.4.3. Browse communication model (cont'd)



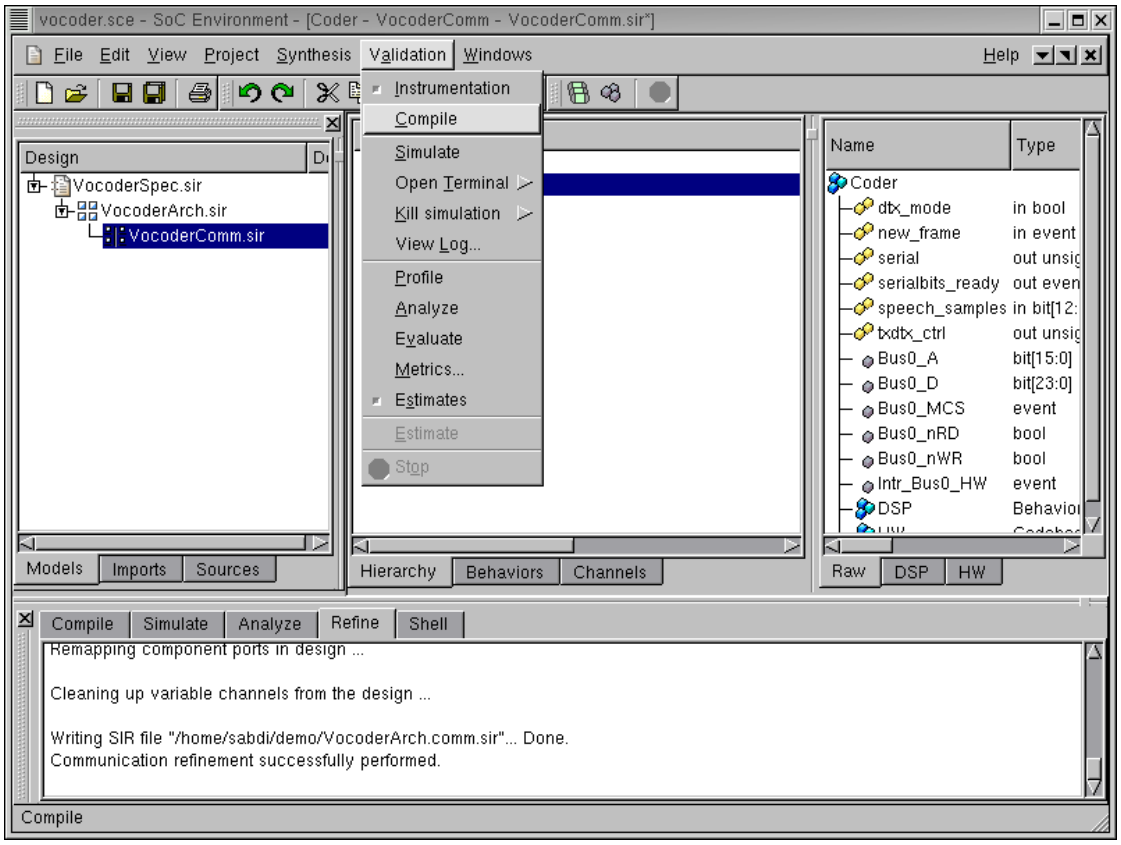
The wire level detail of the connection between components can now be seen in the window. Note that the system bus wires are distinguished by green boxes. Hence we see that the bus is introduced in the design and the individual components are connected with the bus instead of the abstract variable channels. On observing the hierarchical view further, we can see the drivers in each components. These drivers take the original variables and implement the high-level send/receive methods using the bus protocol.

4.4.4. Browse communication model (cont'd)



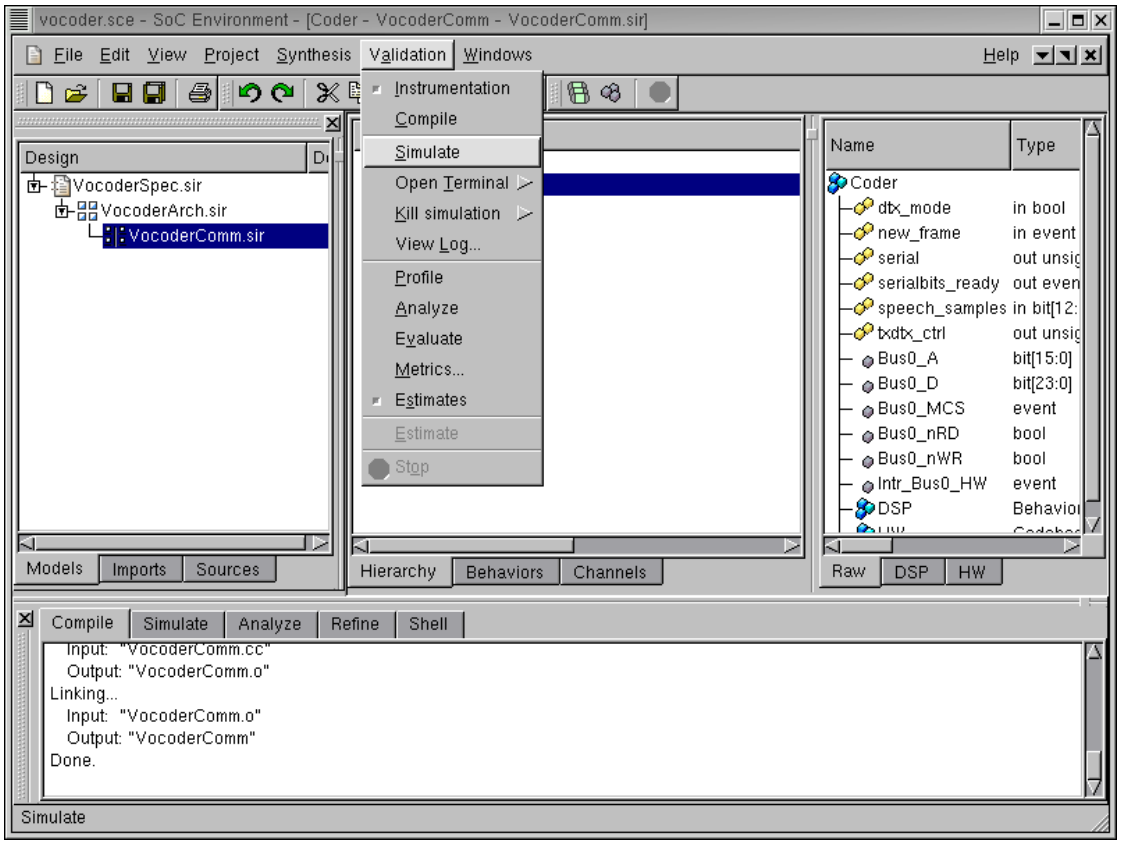
We have thus seen that the structure of communication model follows the semantics of the model explained in our methodology. We may complete the browsing session by selecting Window⟶Close from the menu bar.

4.5. Validate communication model



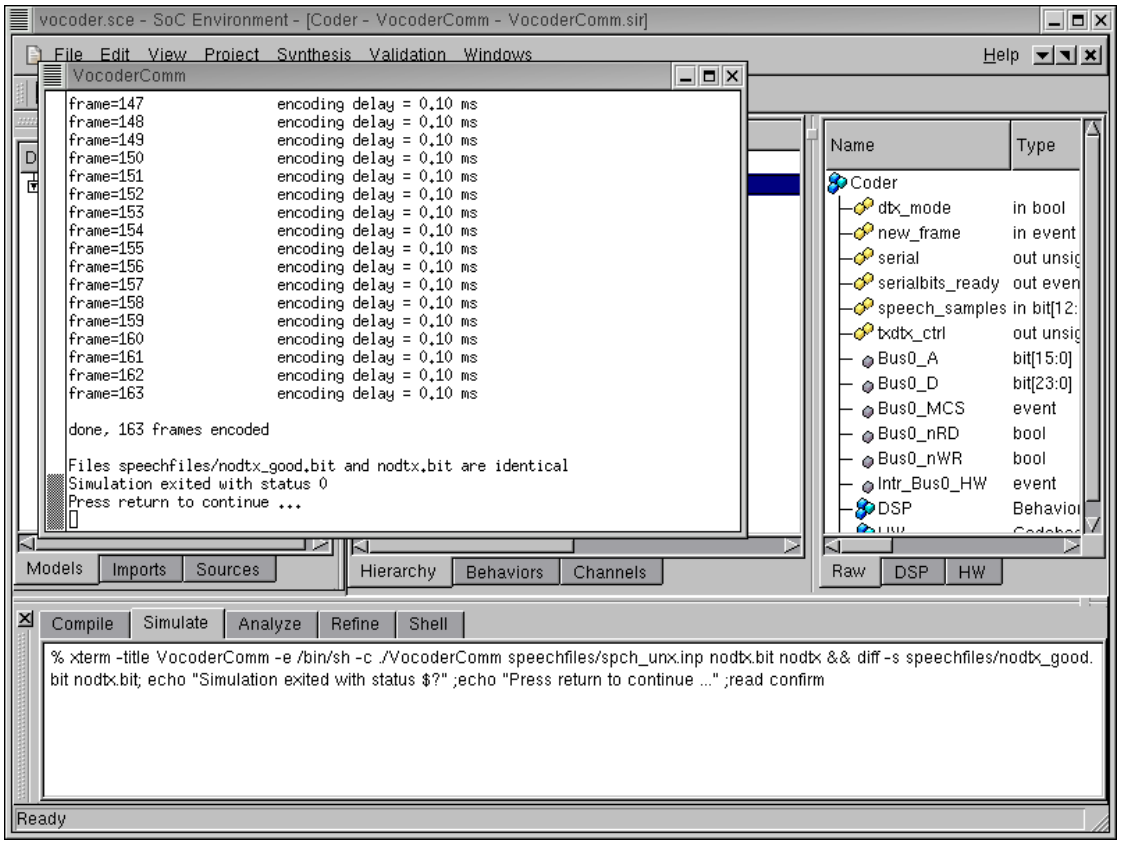
As a direct analogy to the validation of the architecture model, we have a step for validating the communication model. The newly generated model has already been verified to adhere to our notion of a typical communication model. We must now verify that the communication model generated after the refinement process is functionally correct or not. Towards this end, the model is first compiled. This is done by selecting Validation—→Compile from the menu bar.

4.5.1. Validate communication model (cont'd)



The model should compile without errors and this may be observed in the logging window. Once the model has successfully compiled, we must proceed to simulate it. This is done by selecting Validation$\longrightarrow$Simulate from the menu bar.

4.5.2. Validate communication model (cont'd)



An xterm now pops up showing the simulation in progress. Note that simulation is considerably slower for the communication model than for the architecture and communication model. This is because of the greater detail and structure added during the refinement process. Also, it may be noted that there is non-zero encoding delay. This is because the new model is timed and takes into account the delay caused during computation and communication.

With the completion of correct model simulation, we are done with the phase of communication synthesis. Our new model now has two components connected by a system bus. The model is now ready for implementation synthesis.

Chapter 5. Implementation Synthesis

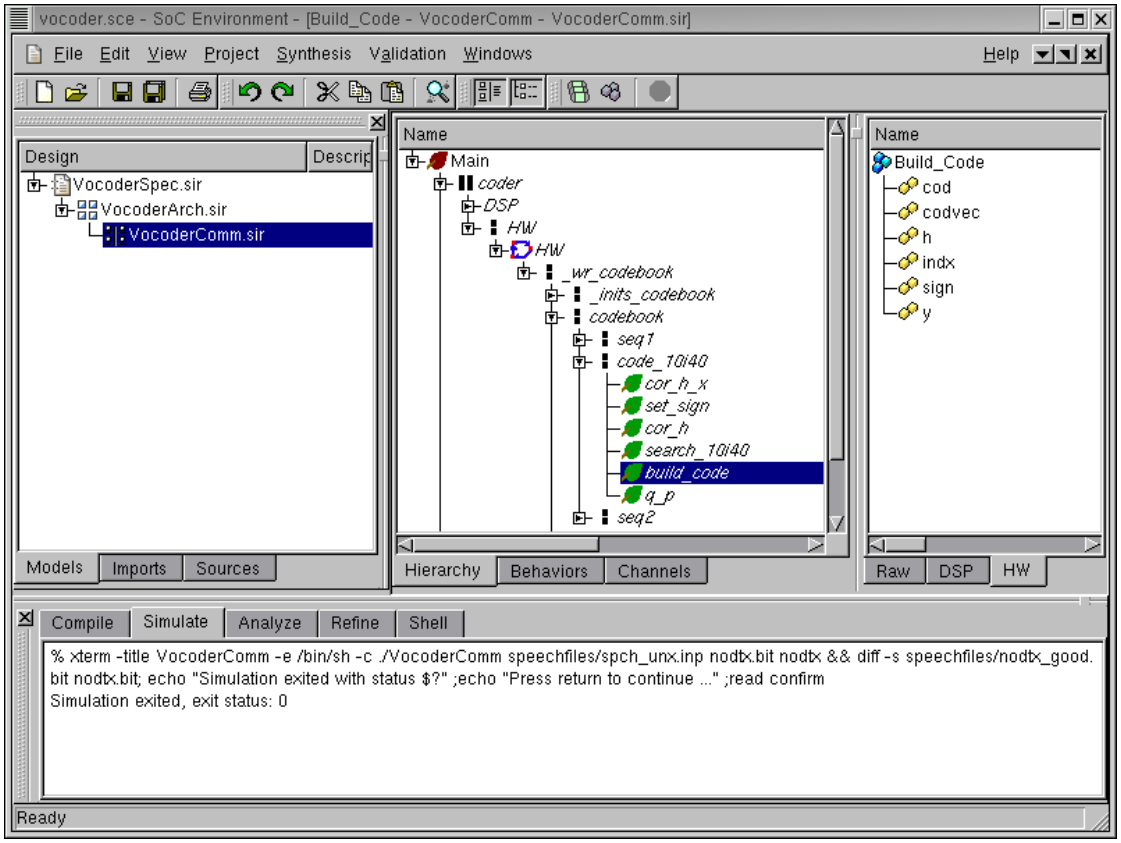
Once we are done with communication synthesis and have obtained a communication model, we have to generate a cycle-accurate implementation from it. The functionality of each component is described for the given set of RTL components or processor instruction-set. In the RTL refinement process, the timing is refined to individual clock cycles based on each component's clock period. The major steps of this refinement step encompass the three parallel tasks for different parts of the communication model. They are enumerated as follows.

Firstly, we have to generate cycle accurate model for custom hardware components, which must eventually be synthesized into a netlist of RTL units.

Secondly, the behaviors mapped to a programmable processor must be compiled into the processor's instruction set and linked against its RTOS. The linking is required only if the processor allows for dynamic scheduling.

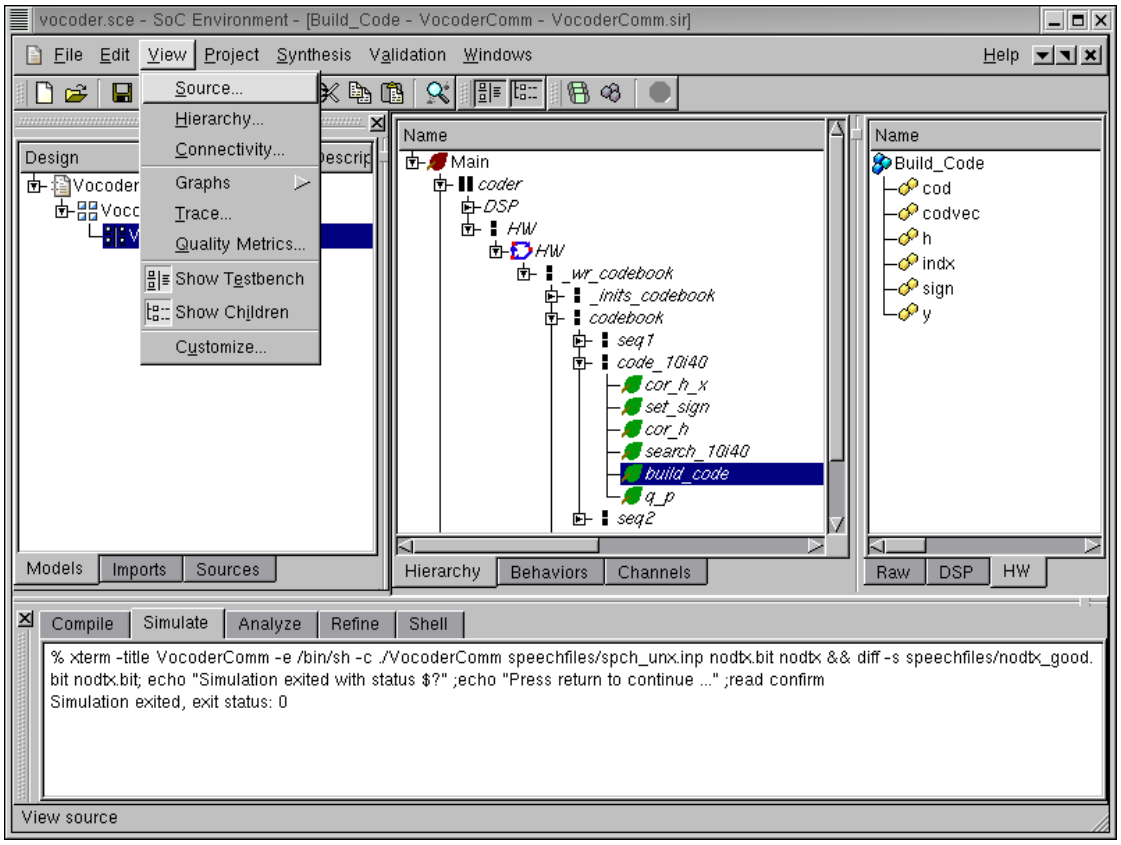
Finally, the bus interfaces and drivers must be refined. The protocol functionality is converted into a cycle true implementation of bus protocols on each component. This requires generation of bus interfaces on the hardware components and assembly code for bus drivers on the software side. The result of all these steps is a cycle accurate model as desired in our SoC methodology. This model can be used as an input to the standard EDA tools.

5.1. Select RTL components



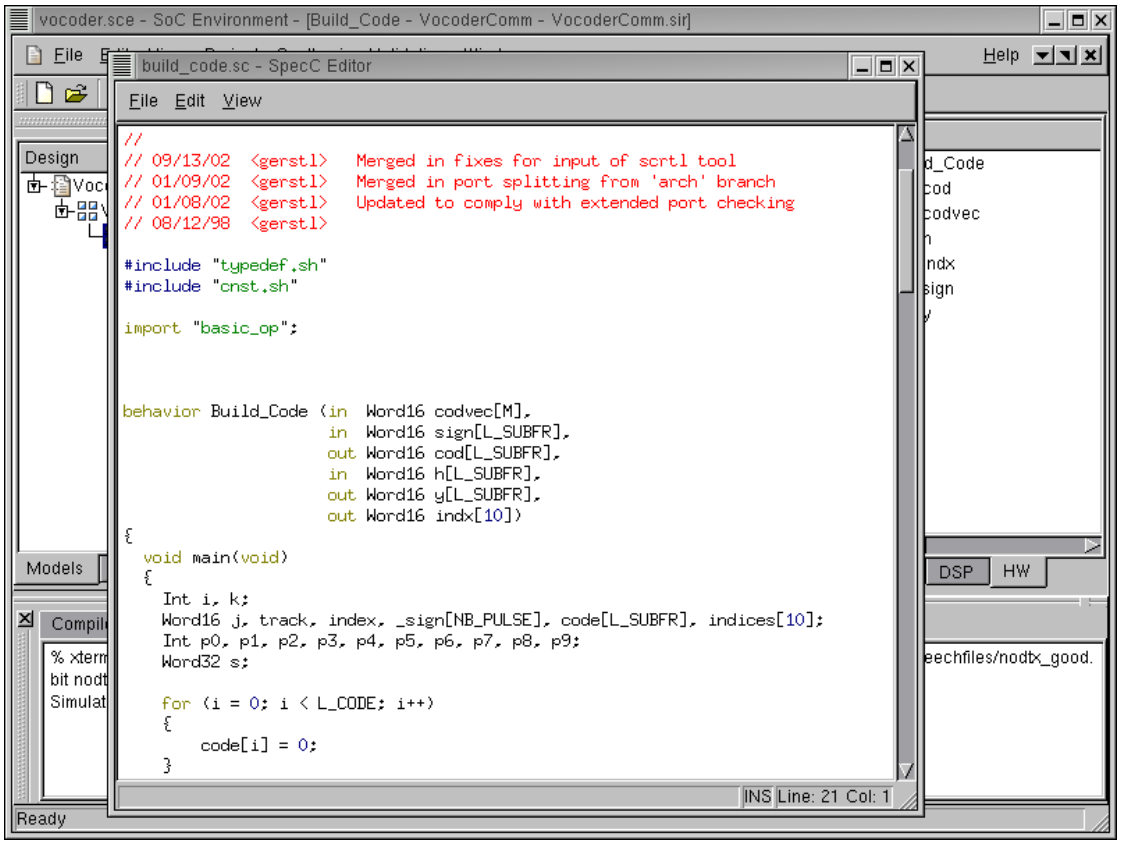
After we produce a valid communication model, the next step is to generate the clock cycle accurate model. Essentially, we have to perform RTL refinement for the behaviors mapped to hardware and software compilation for the behaviors mapped to software. We first look at the RTL refinement part. To demostrate the concept of RTL refinement for the hardware part, we choose a suitable behavior. Browse the hierarchy in the design hierarchy window and select by Left clicking the "build_code" behavior. We shall be performing RTL refinement on this behavior.

5.1.1. Select RTL components (cont'd)



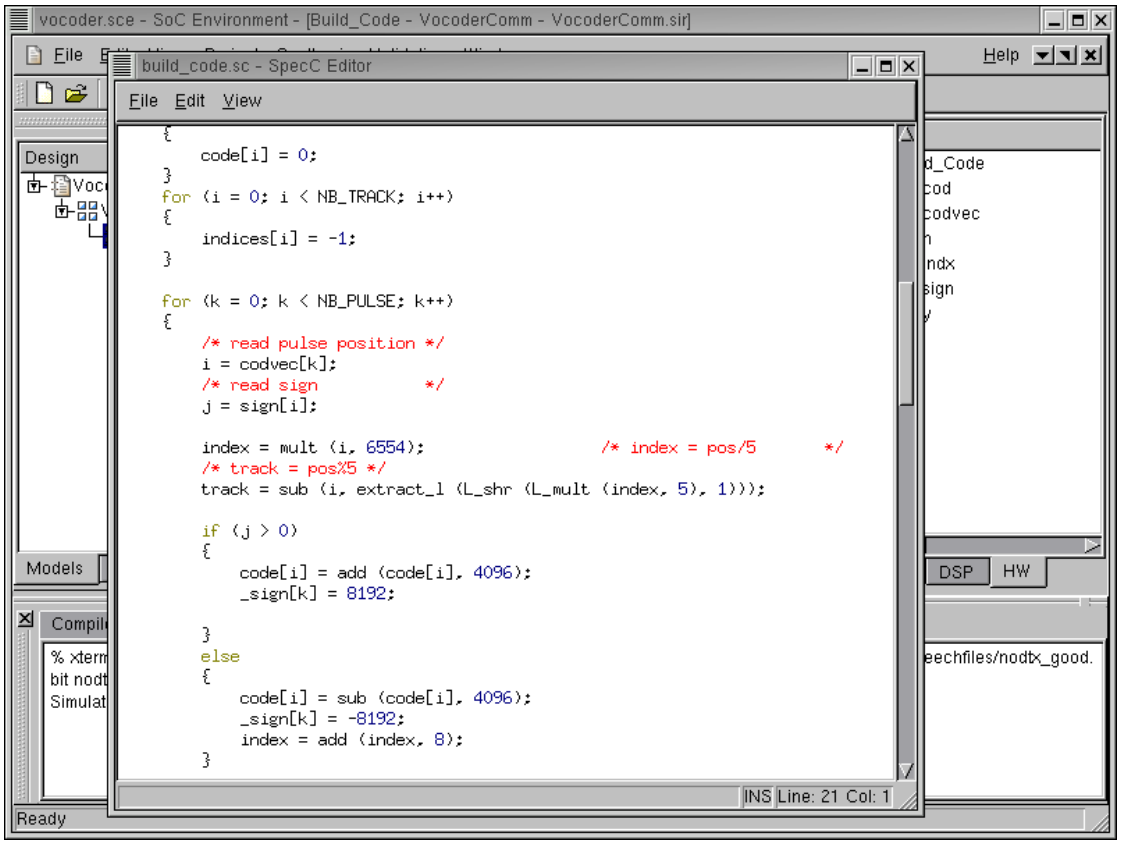
We begin by looking at the source code of "build_code" behavior to see what kind of constructs it posseses. This is done by selecting View⟶Source from the menu bar.

5.1.2. Select RTL components (cont'd)



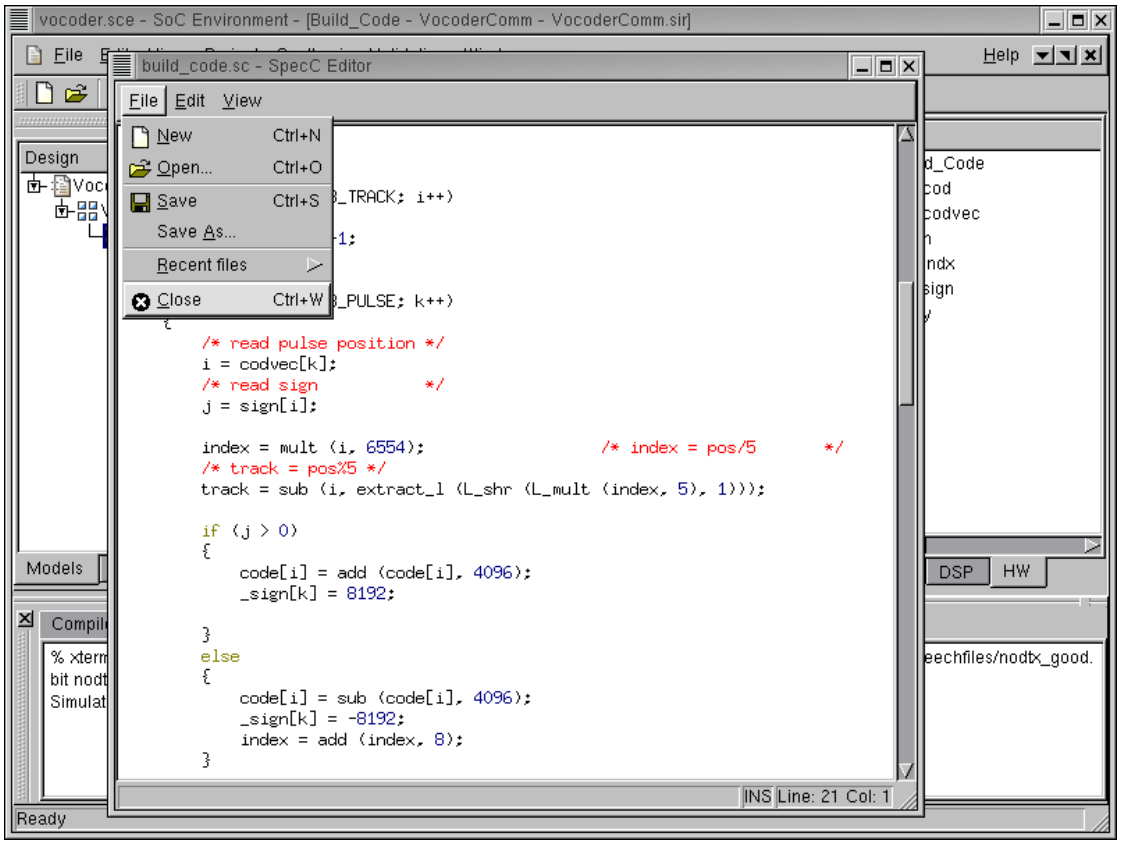
The source code editor window pops up showing the source code for behavior "Build_Code"

5.1.3. Select RTL components (cont'd)



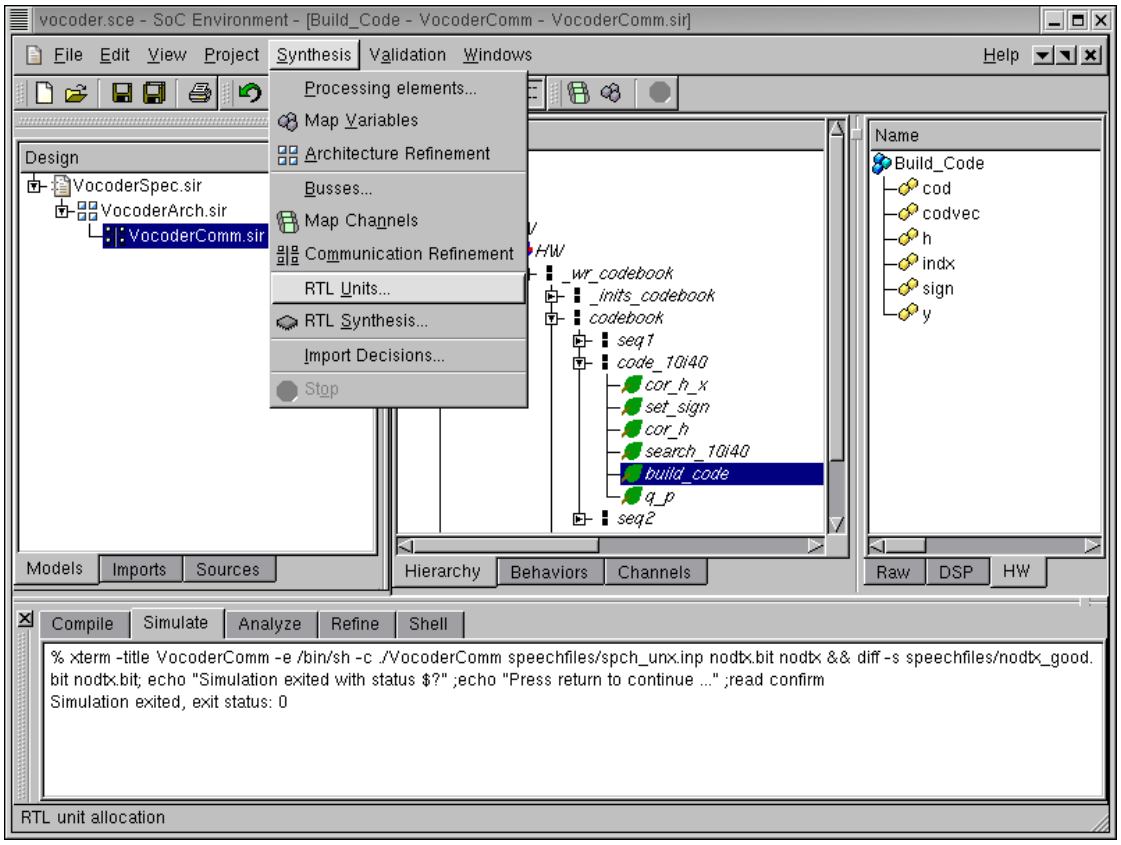
Scroll down the window to find that the behavior code has loops and conditional constructs. So, RTL refinement for this behavior will show that the tool can handle these constructs.

5.1.4. Select RTL components (cont'd)



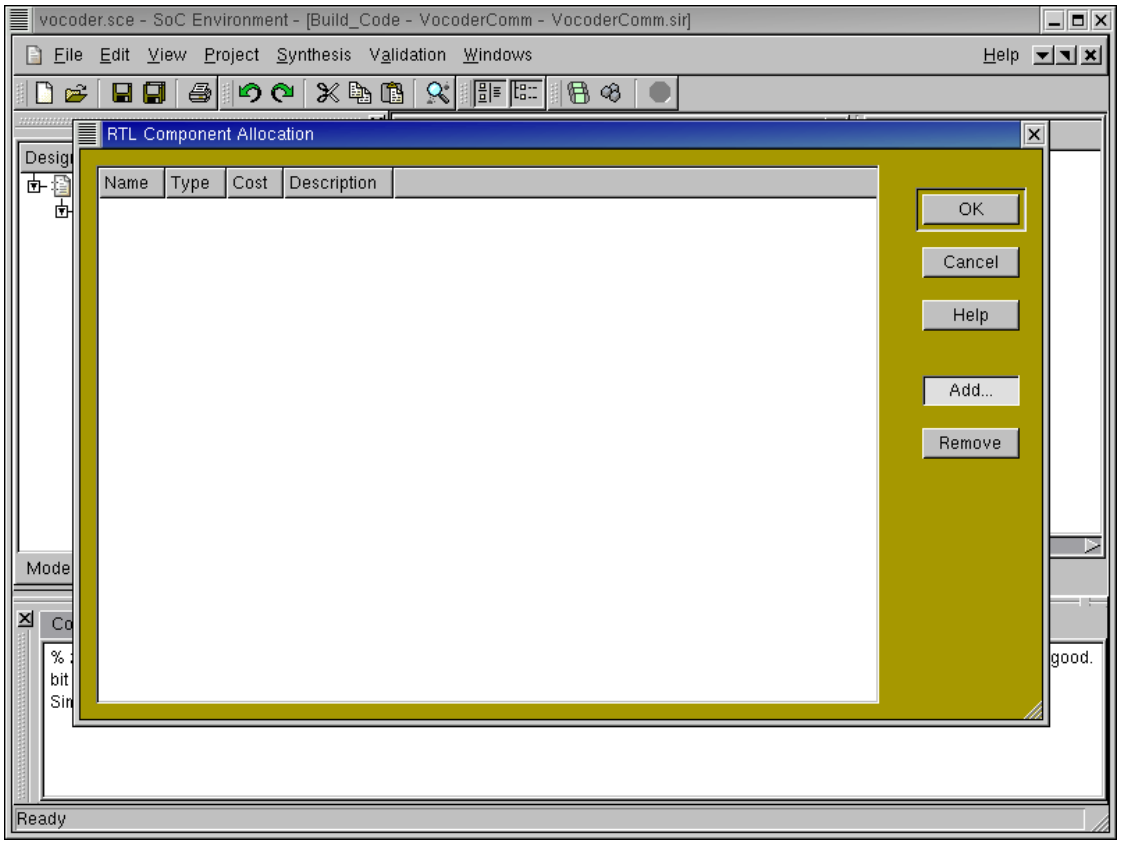
Close the source code editor window by selecting File⟶Close

5.1.5. Select RTL components (cont'd)



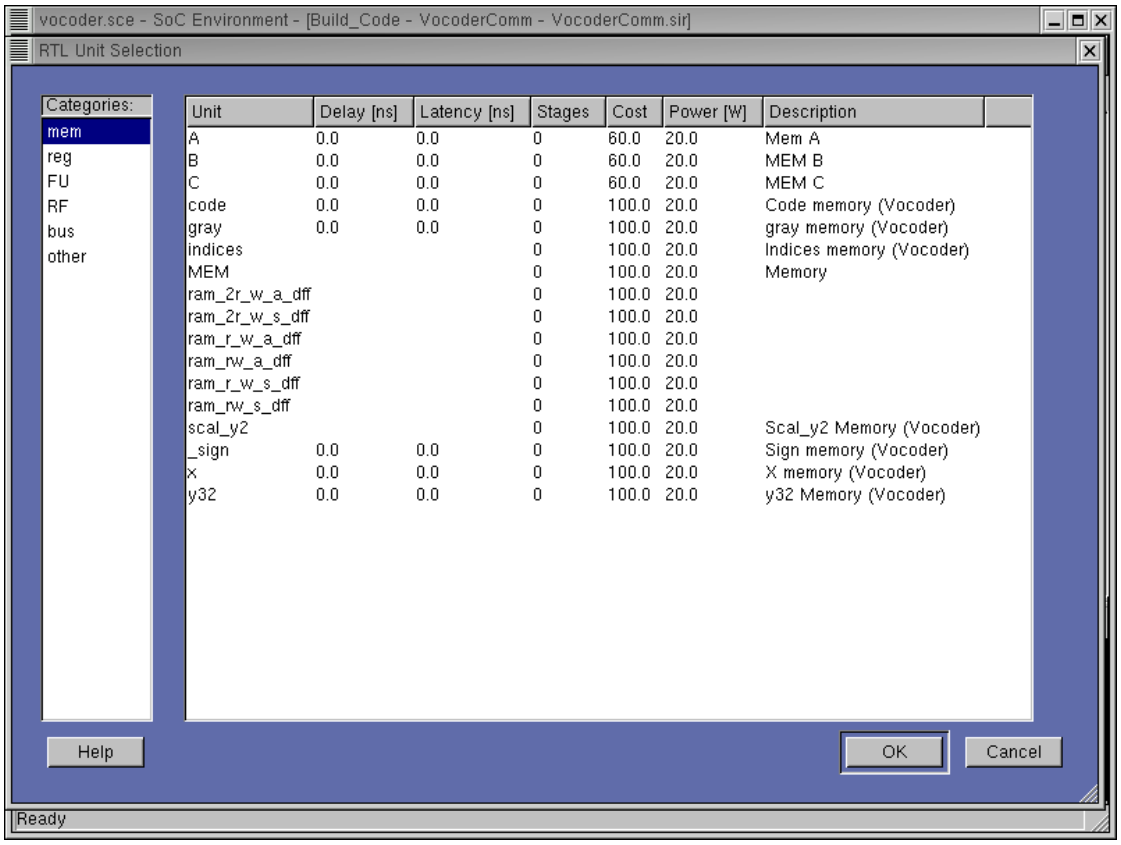
In order to perform refienement, we need to have RTL units that will be used to generate the structural RTL model. The next step is thus to select RTL units for allocation. To perform the allocation select Synthesis—→RTL Units from the menu bar.

5.1.6. Select RTL components (cont'd)



An RTL allocation window pops up just like for components and busses. Left click on Add to see the available units on the database.

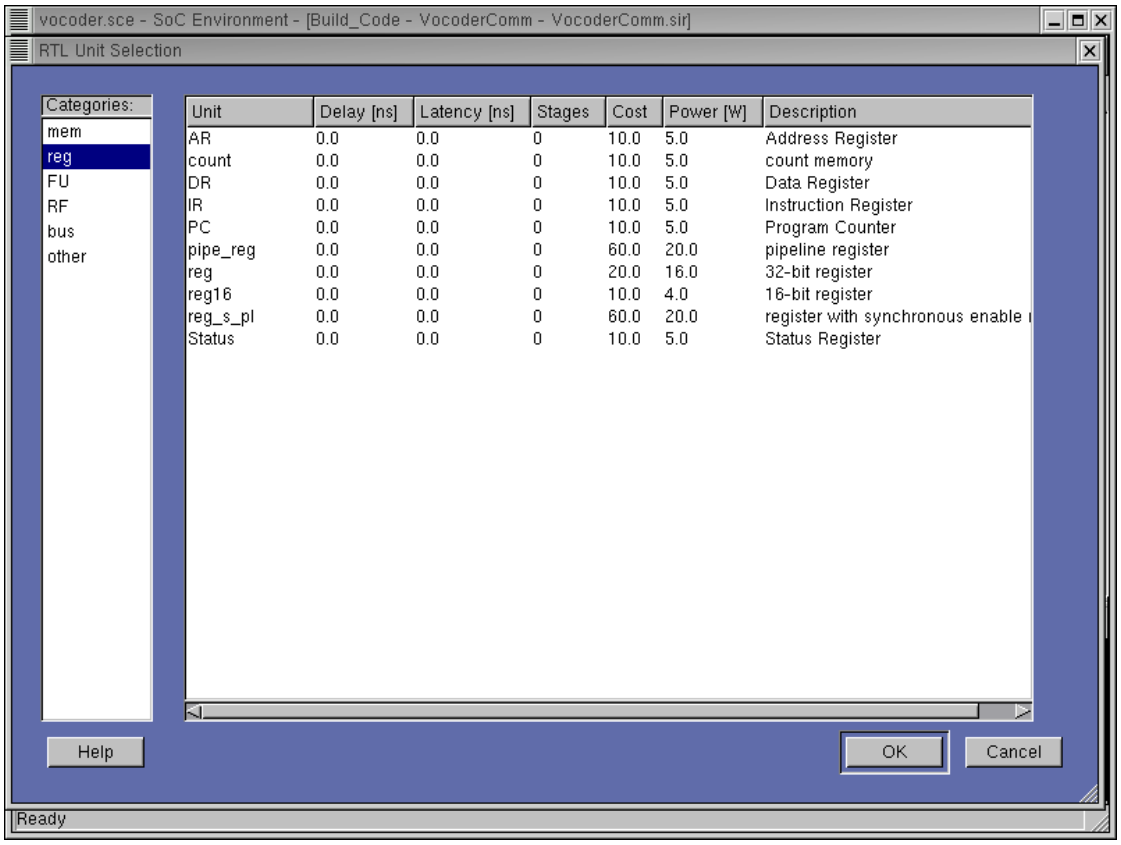
5.1.7. Select RTL components (cont'd)



A new window pops up for RTL unit selection. There are various categories for the RTL units listed on the left-most colunm. Left click on "mem" to see the memory units and their properties.

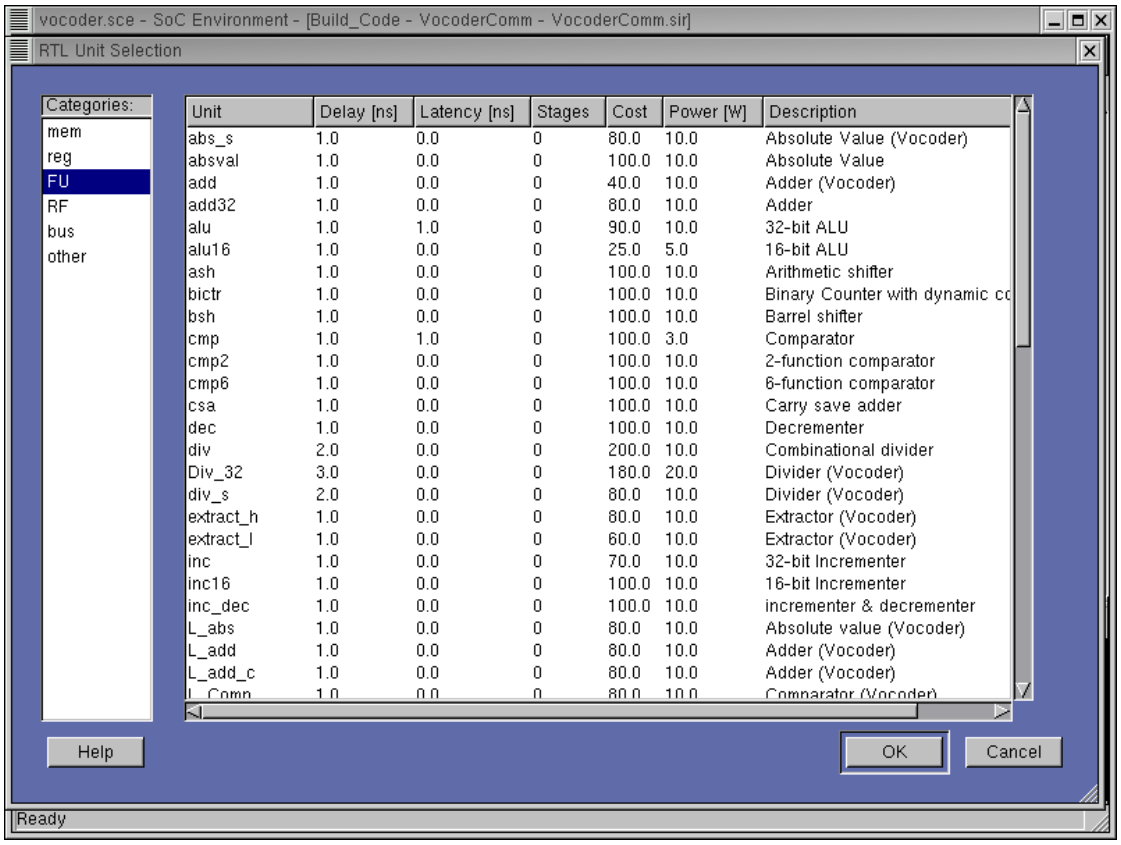
Note the various types of available memories. The units are parameterized and come with several metrics that may be used to make the optimal allocation choice.

5.1.8. Select RTL components (cont'd)



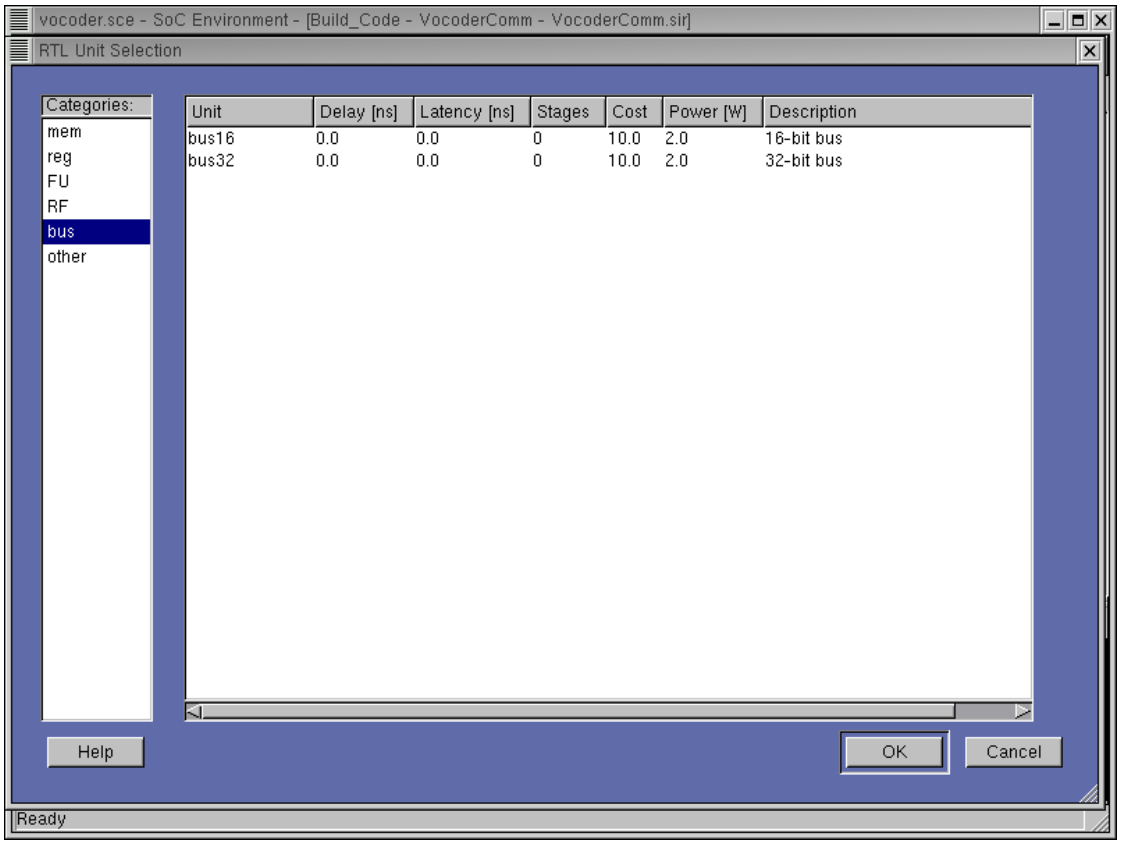
Left click on "reg" to see the registers and their properties.

5.1.9. Select RTL components (cont'd)



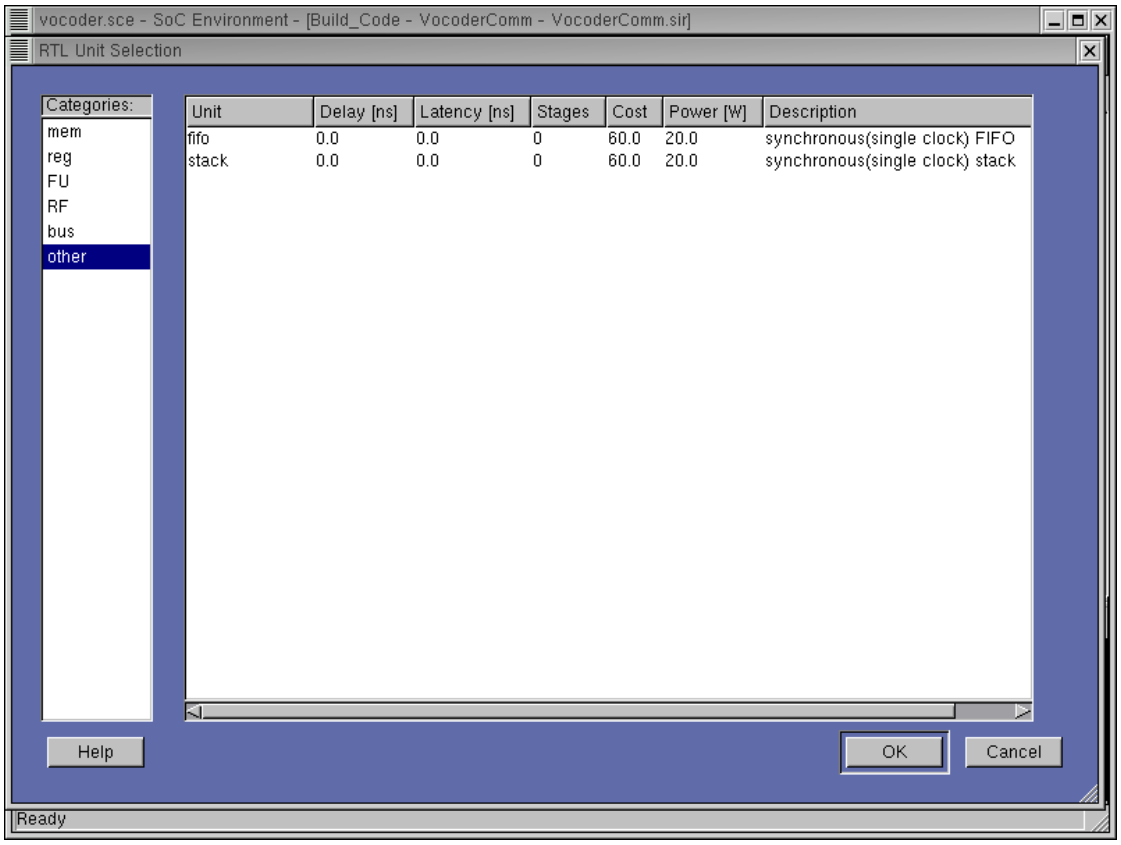
Likewise we have a set of functional units. It may be seen that the functional unit allocation must ideally comply with the profiling data. As seen during profiling, we have integer arithmetic operations in plenty. Therefore, it makes sense to allocate integer ALUs. Left click on "FU" to see the various functional units and their properties.

5.1.10. Select RTL components (cont'd)



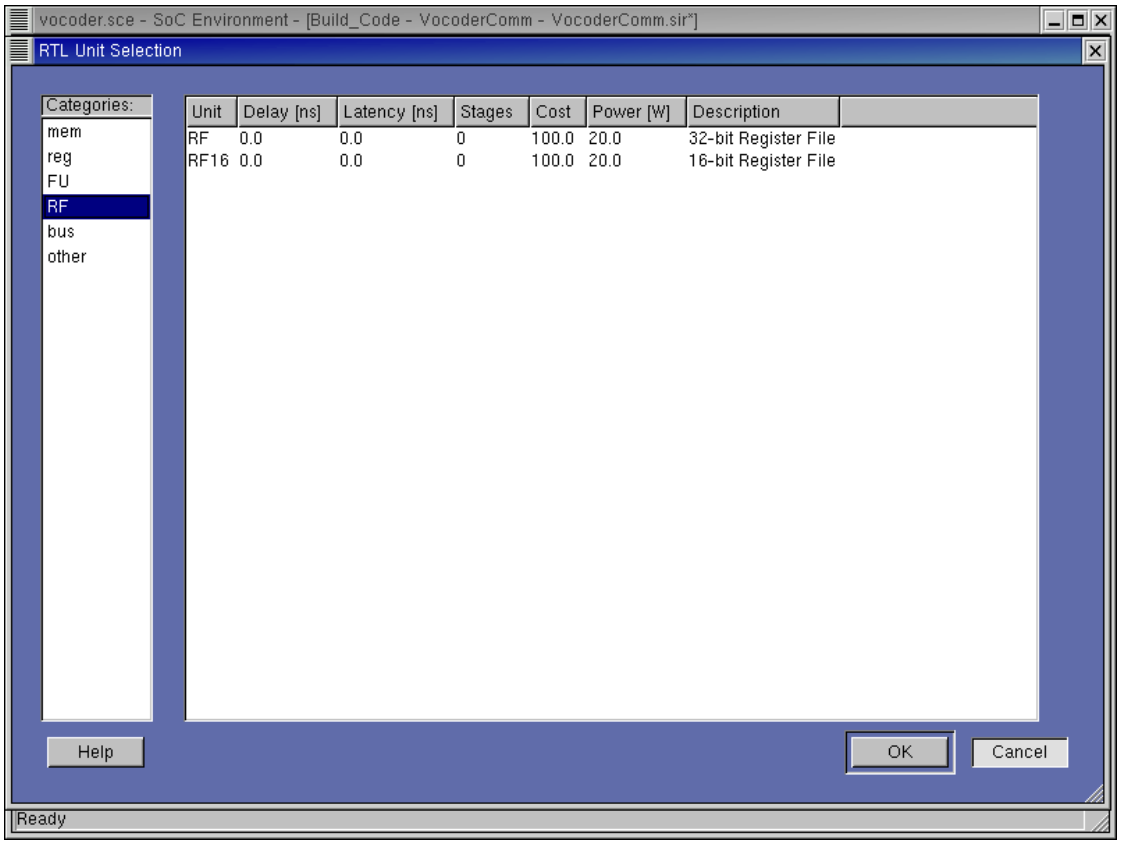
Left click on "bus" to see the available busses and their properties. These busses will be used for internally connecting the RTL units within the HW component.

5.1.11. Select RTL components (cont'd)



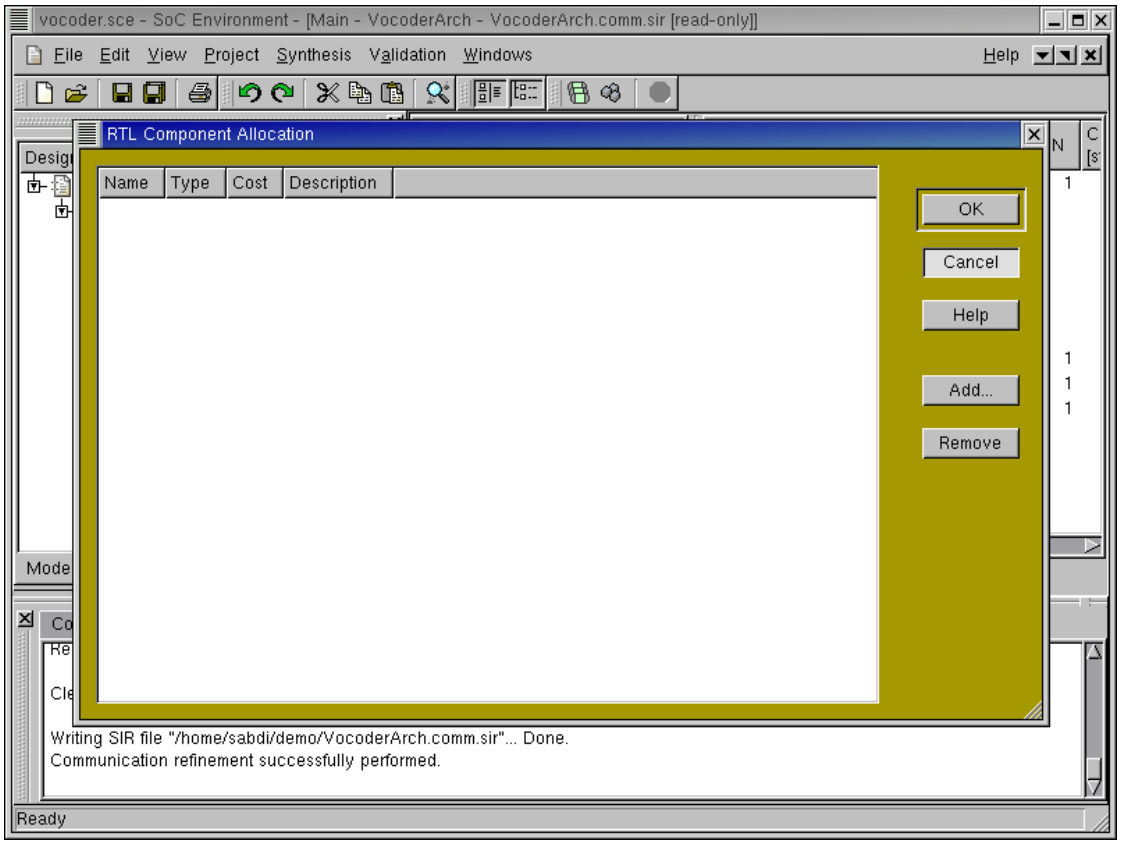
Left click on "other" to see units that do not fit in above categories.

5.1.12. Select RTL components (cont'd)



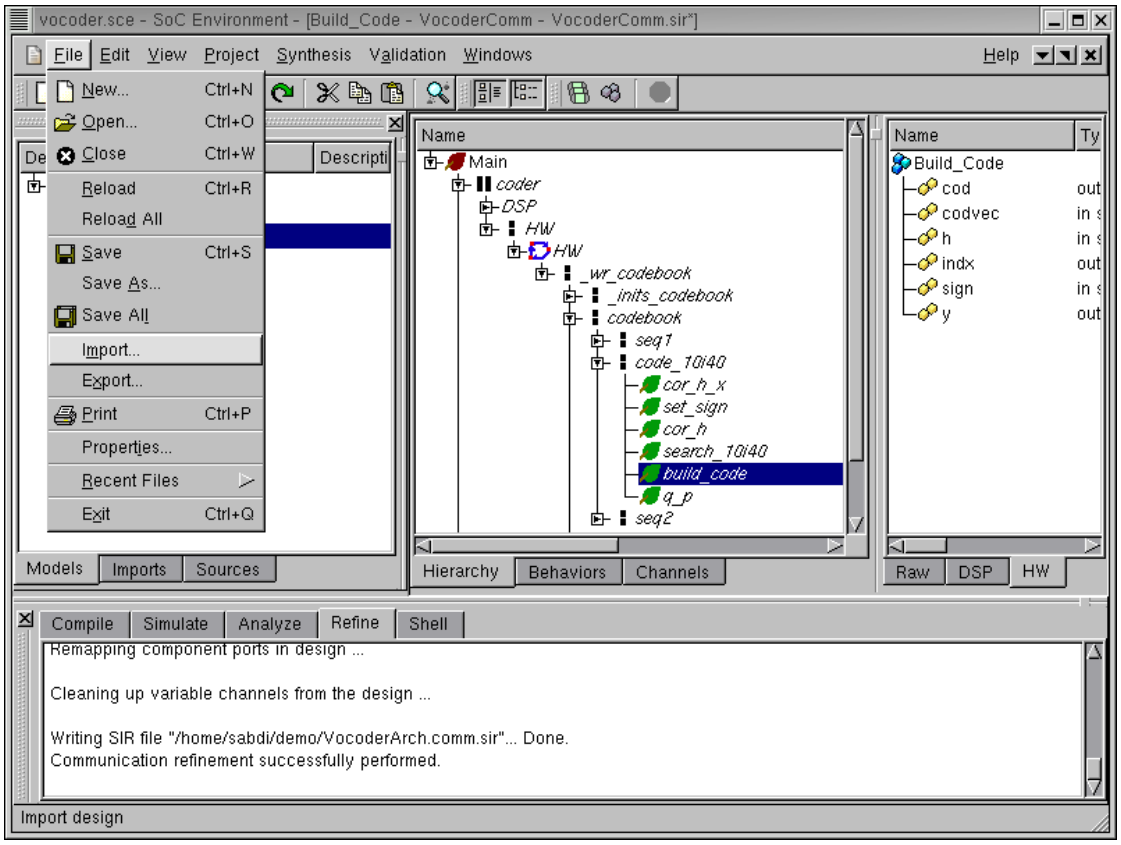
Left click on "RF" to see the various register files and their properties. Since selecting a large number of units for our example takes a lot of time, a preallocated set of RTL units may be directly imported into the design. Exit the RTL Unit Selection window by Left clicking on Cancel.

5.1.13. Select RTL components (cont'd)



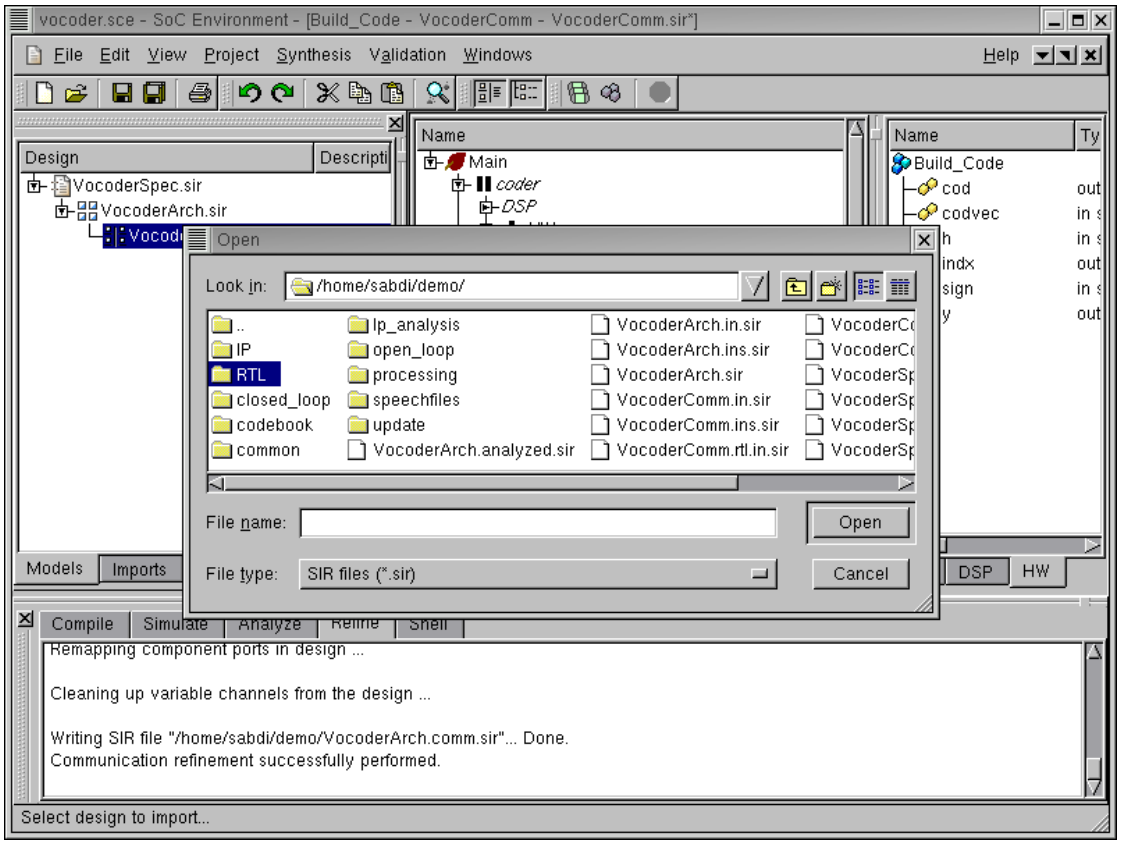
Exit the RTL component allocation window as well by Left clicking on Cancel.

5.1.14. Select RTL components (cont'd)



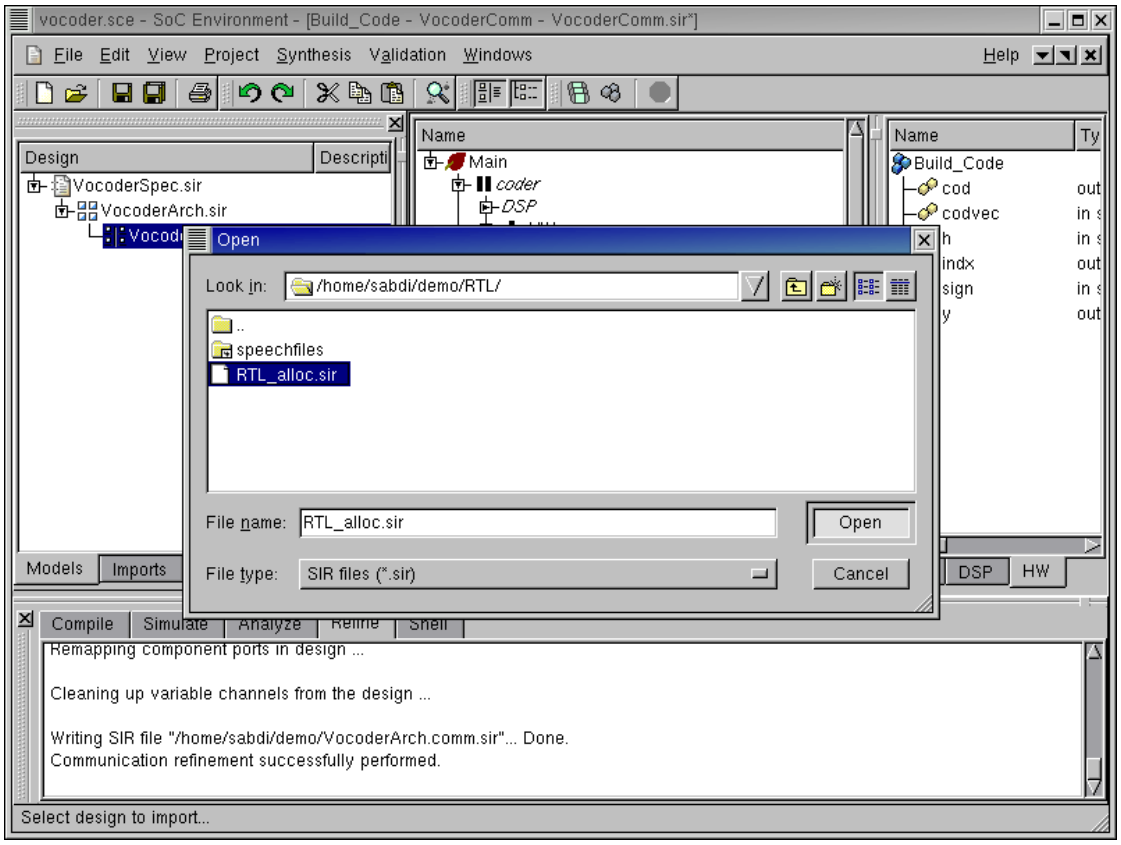
We now begin the RTL refinement process by importing the preallocated set of RTL units by selecting File—→Import from the menu bar.

5.1.15. Select RTL components (cont'd)



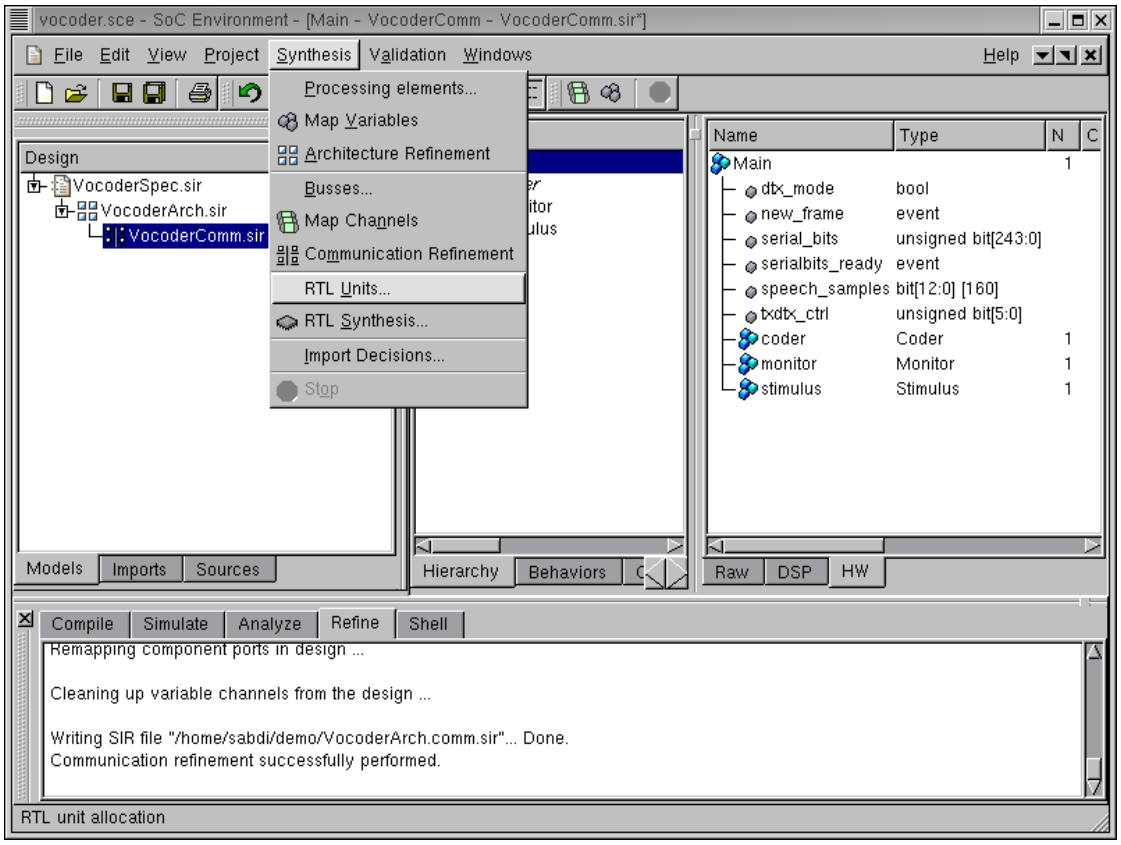
The library consisting on the RTL unit allocation for this design is kept under the "RTL" directory in the demo directory. In the file selection window that pops up, double Left click on "RTL" to enter the directory.

5.1.16. Select RTL components (cont'd)



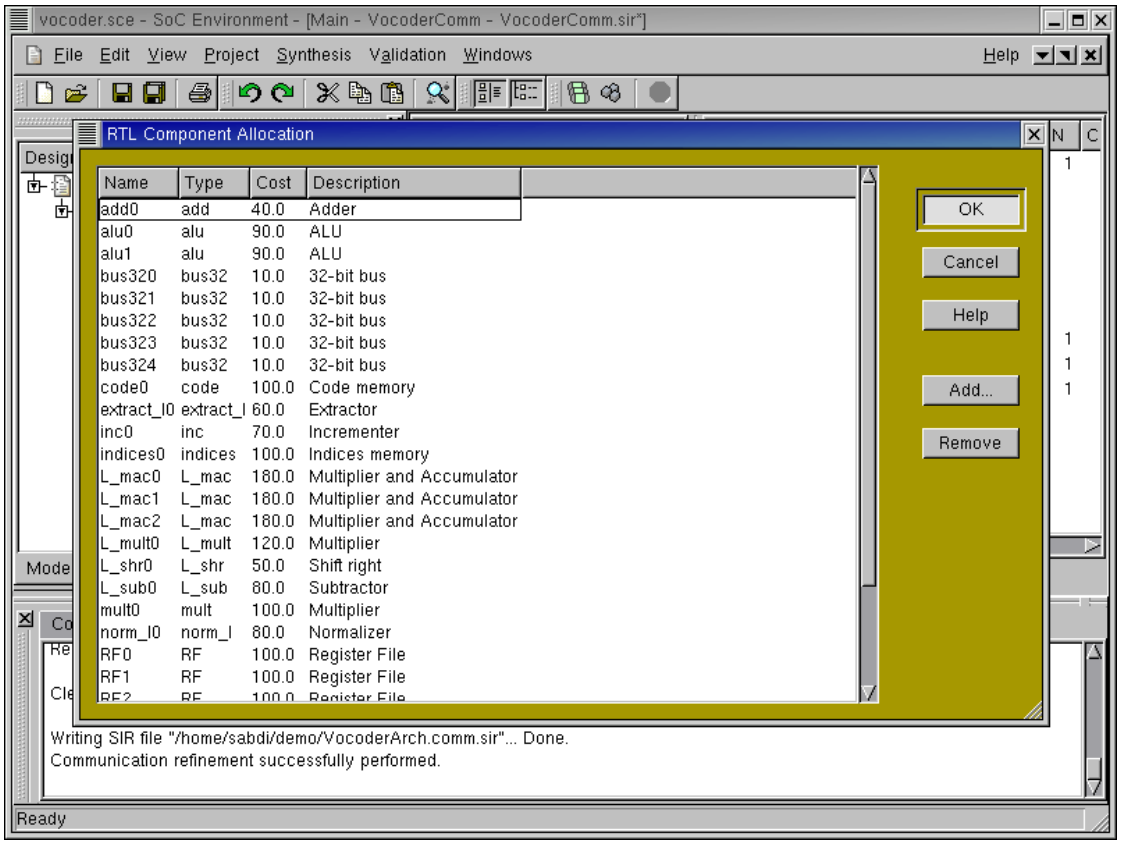
Once inside the "RTL" directory, select the file "RTL_alloc.sir" and Left click on Open.

5.2. RTL refinement



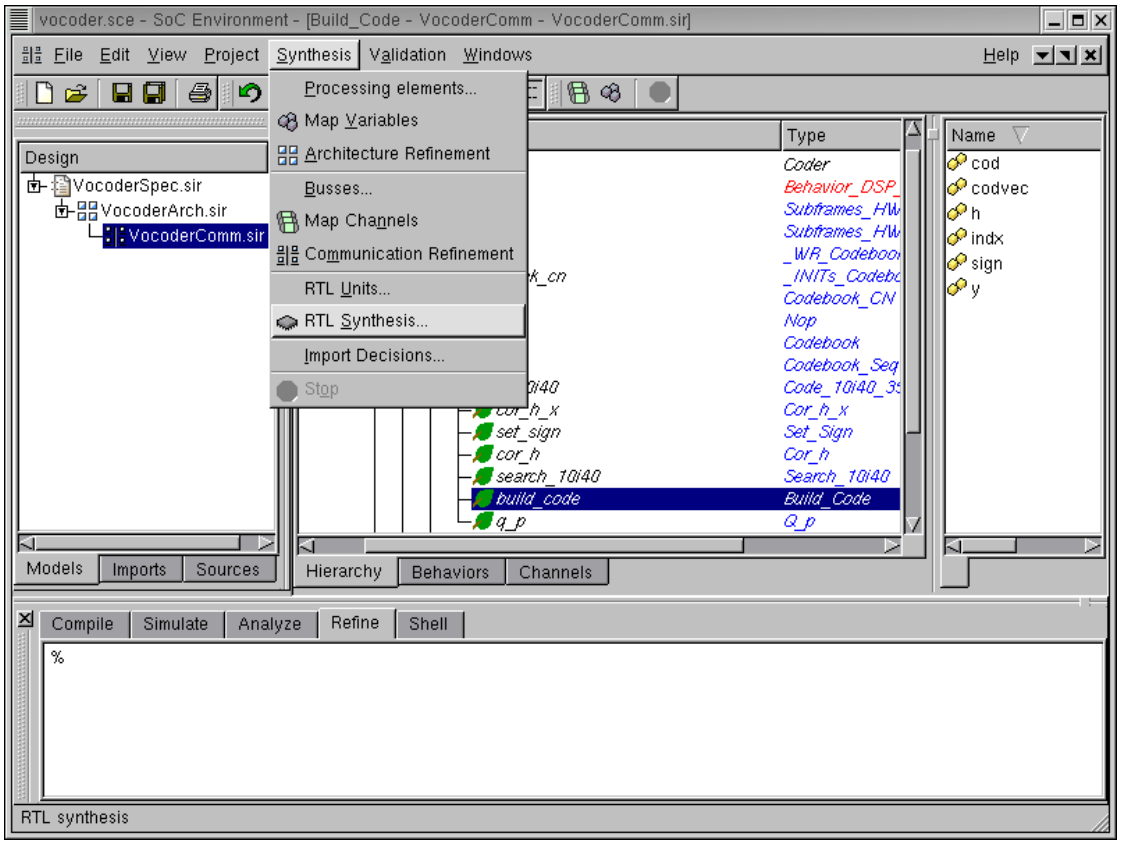
We begin RTL refinement by first verifying that the imported file has correctly allocated the RTL units. To do this, select Synthesis—→RTL Units from the menu bar.

5.2.1. RTL refinement (cont'd)



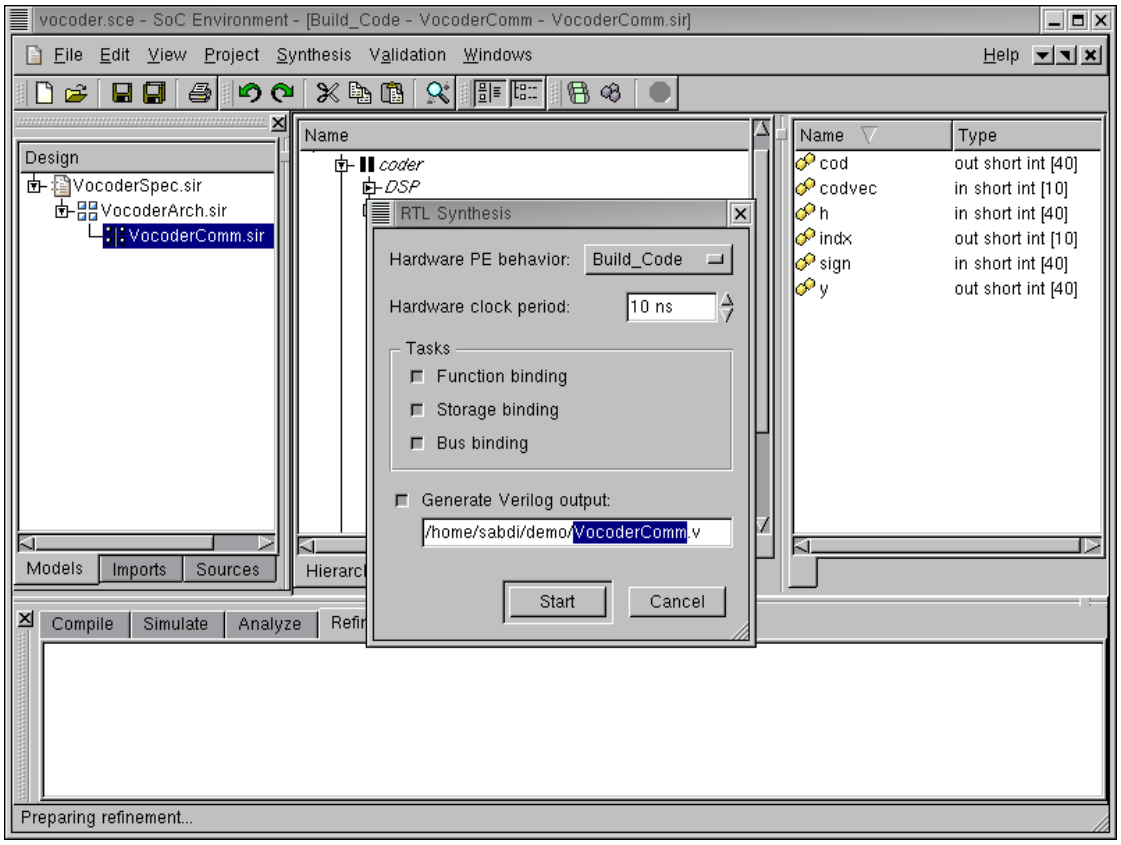
The RTL Component Allocation table pops up showing us all the allocated units. Note the presence of units for Integer arithmetic as discussed earlier. We can also see internal busses and register files in the allocation.Left click on OK to confirm the allocation.

5.2.2. RTL refinement (cont'd)



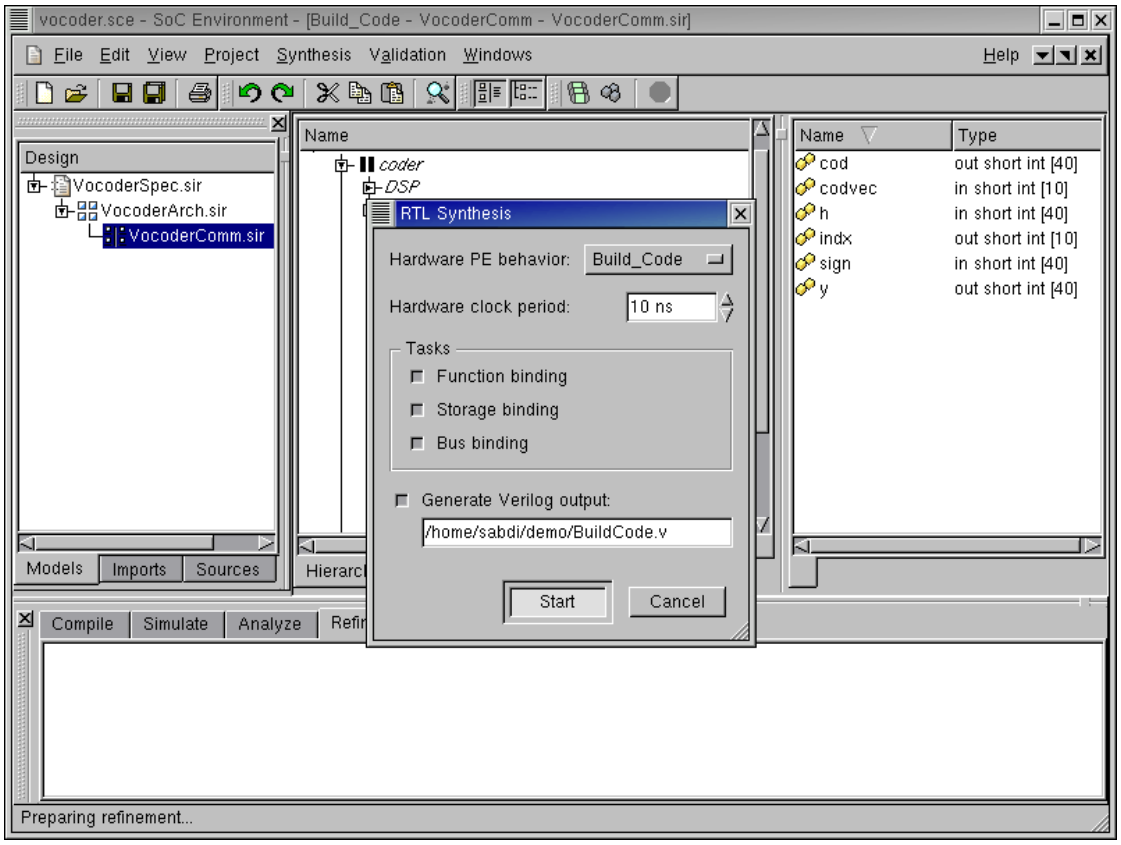
Launch the RTL refinement tool on behavior "Build Code" by selecting Synthesis⟶RTL Synthesis from the menu bar.

5.2.3. RTL refinement (cont'd)



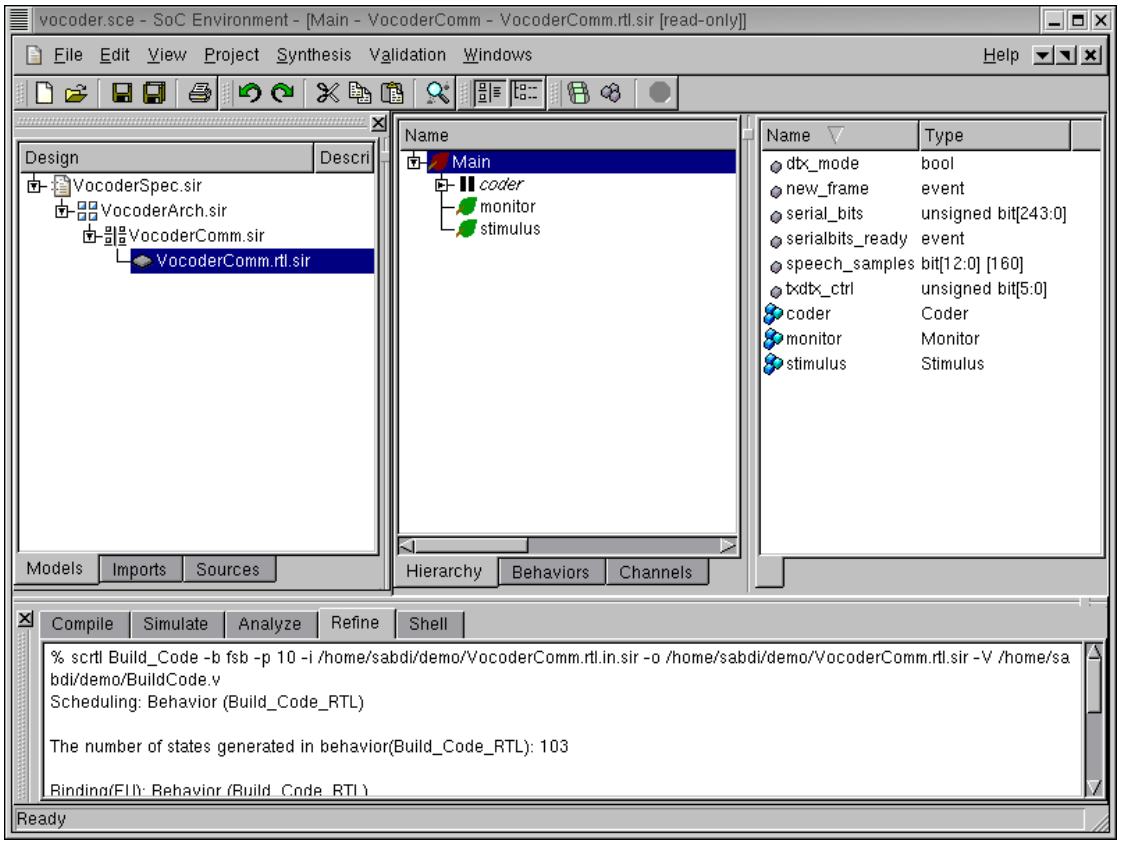
In the newly popped up window, change the name of the generated verilog file to "Build-Code.v" so that we can clearly identify that this refinement is for the "BuildCode" behavior.

5.2.4. RTL refinement (cont'd)



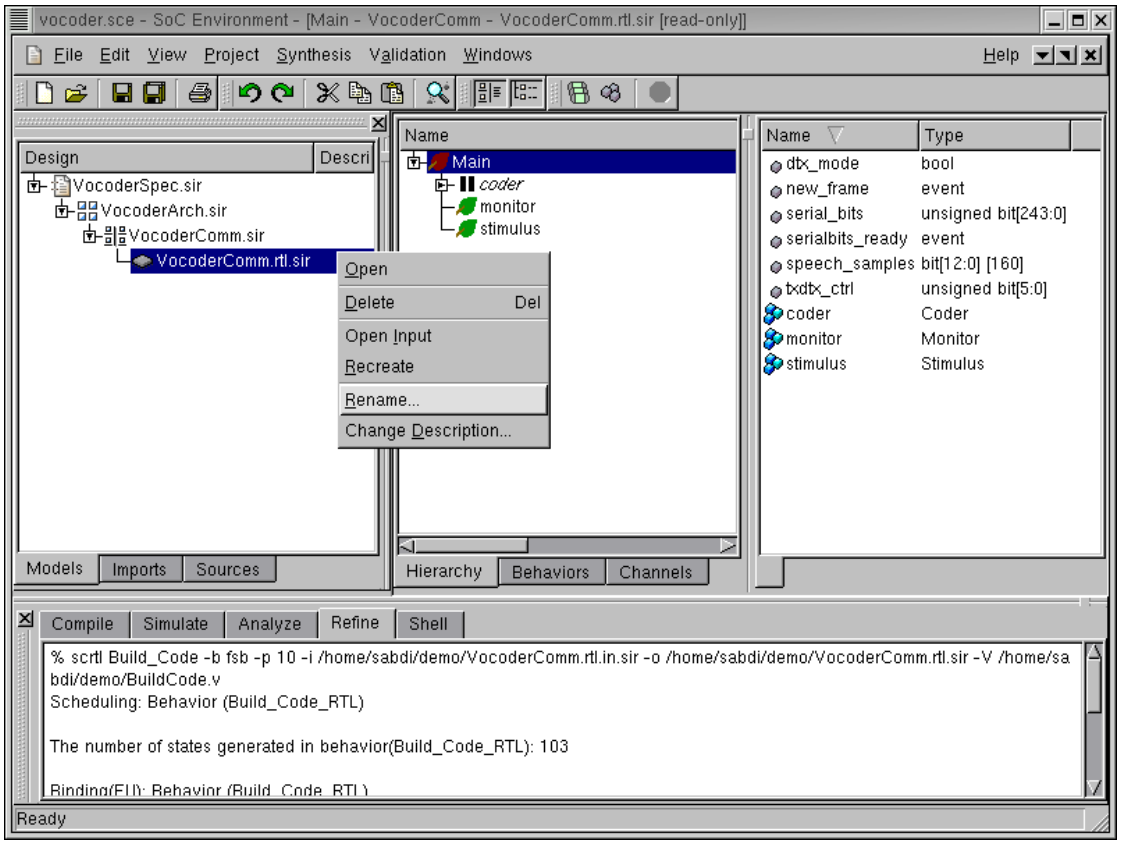
Left click on Start to begin RTL refinement. Notice that like in the earlier refinement phases, we have options for partial refinement steps. The user might avoid some binding steps if he wants to look at intermediate models. Also note that we have selected a clock period of 10 ns, corresponding to the speed of our custom hardware unit. It may be recalled that while selecting the hardware component, we specified a hardware component with clock speed of 100 Mhz, which imposes a clock period of 10 ns.

5.2.5. RTL refinement (cont'd)



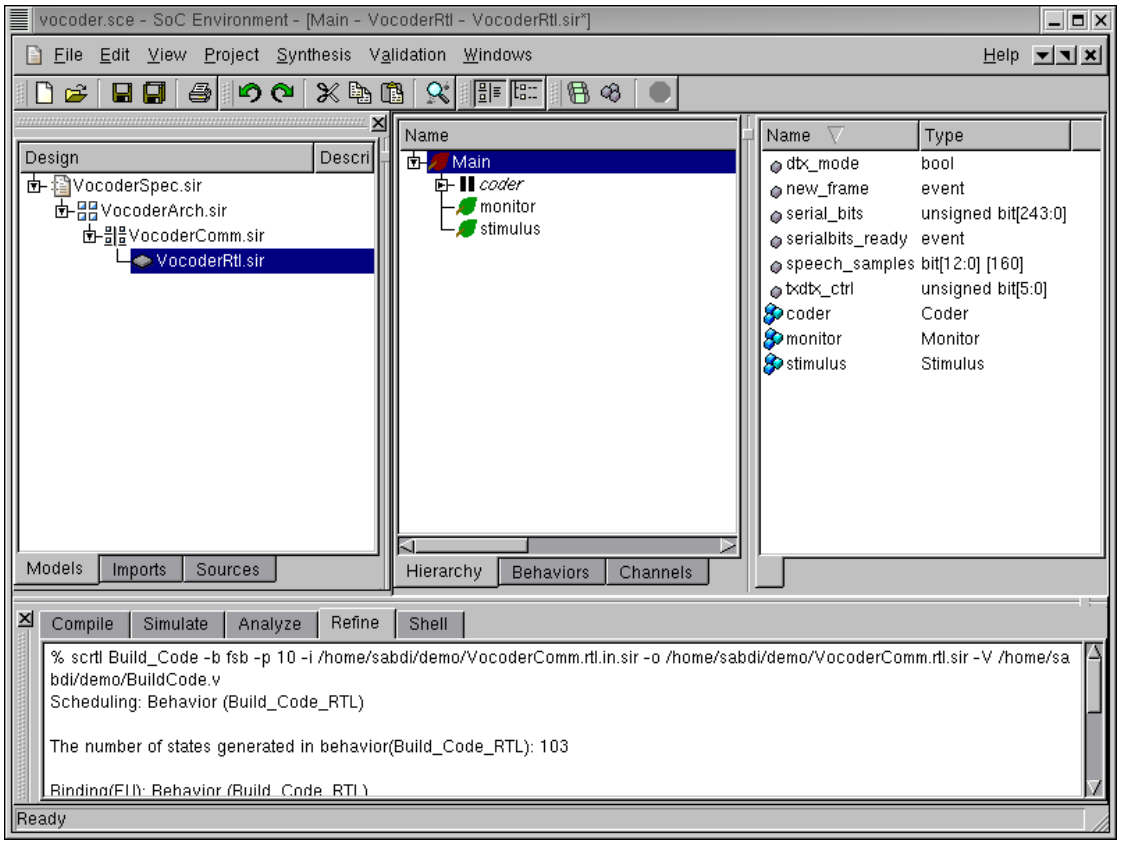
Note that RTL synthesis generates 103 states as seen on the logging window as the tool generated the RTL model for "BuildCode". Also note that a new model "VocoderComm.rtl.sir" is added in the Project manager window.

5.2.6. RTL refinement (cont'd)



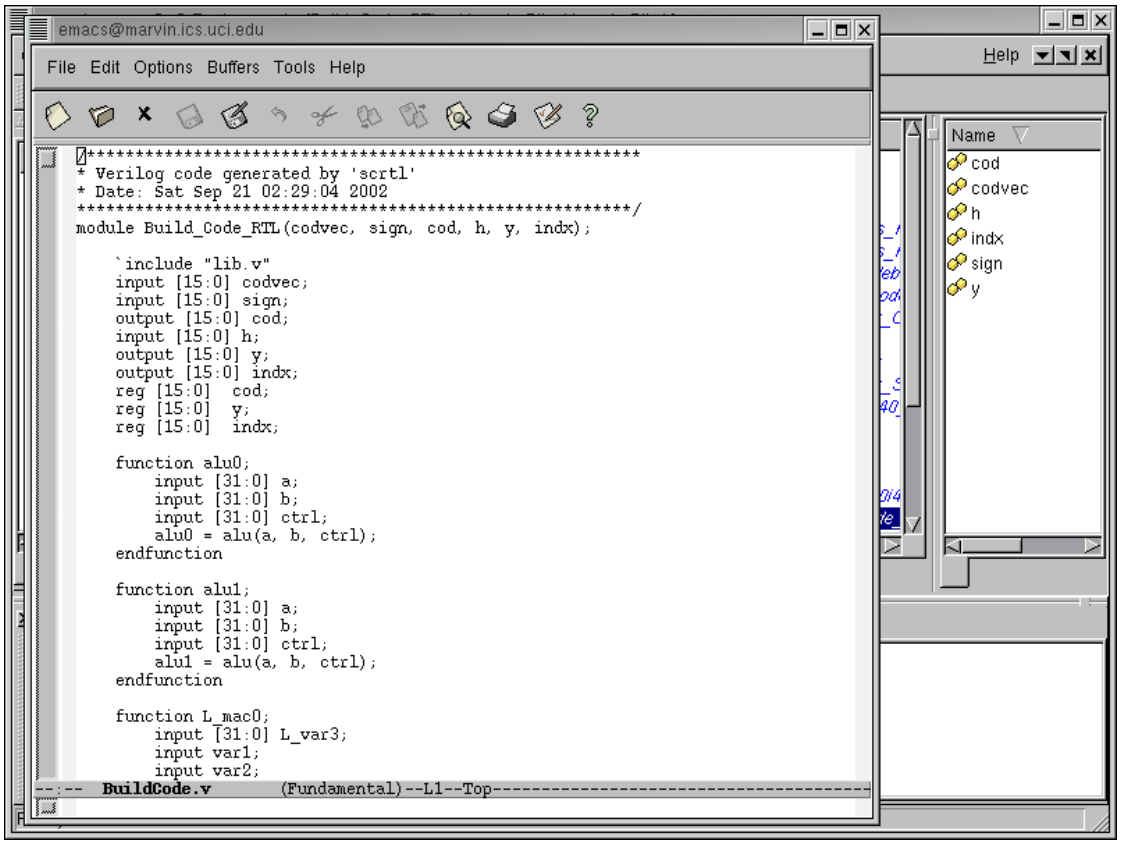
Like before, we must give our new model a suitable name. We can do this by Right clicking on "VocoderComm.rtl.sir" and selecting Rename from the pop up menu.

5.2.7. RTL refinement (cont'd)



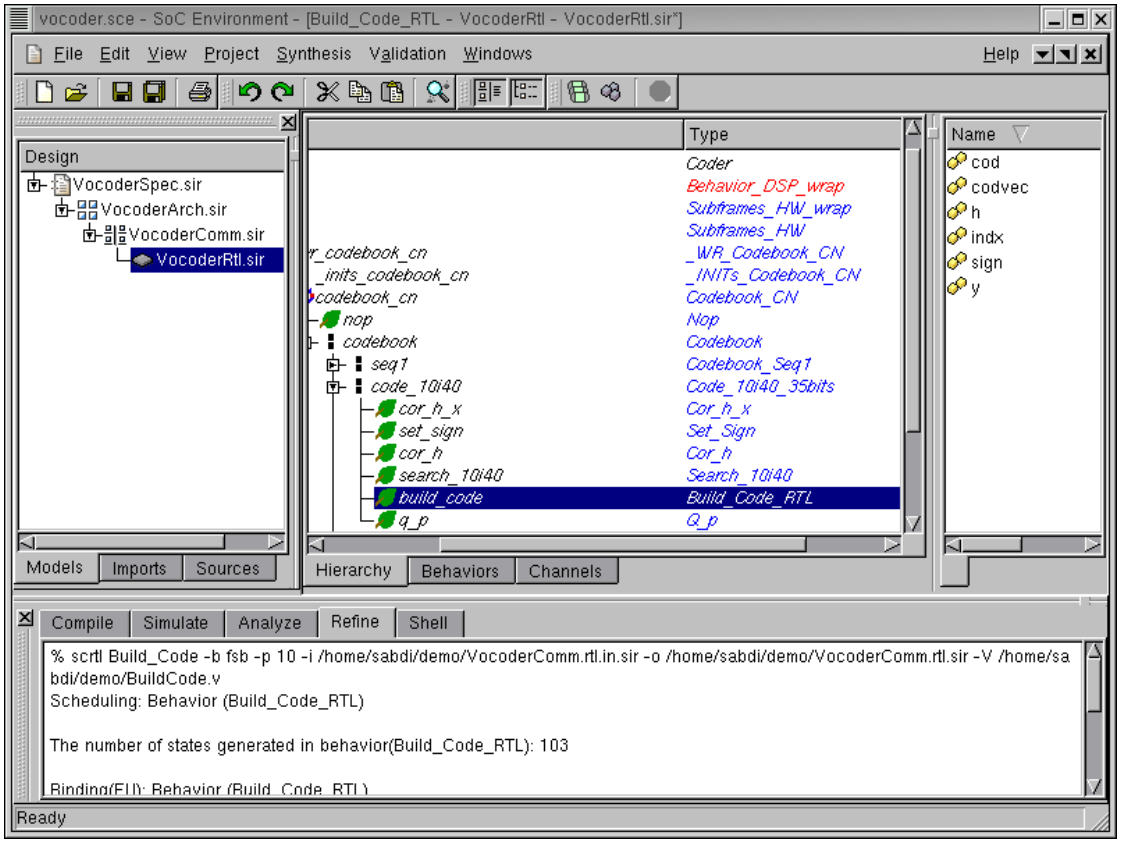
Rename the model to "VocoderRTL.sir."

5.2.8. RTL refinement (cont'd)



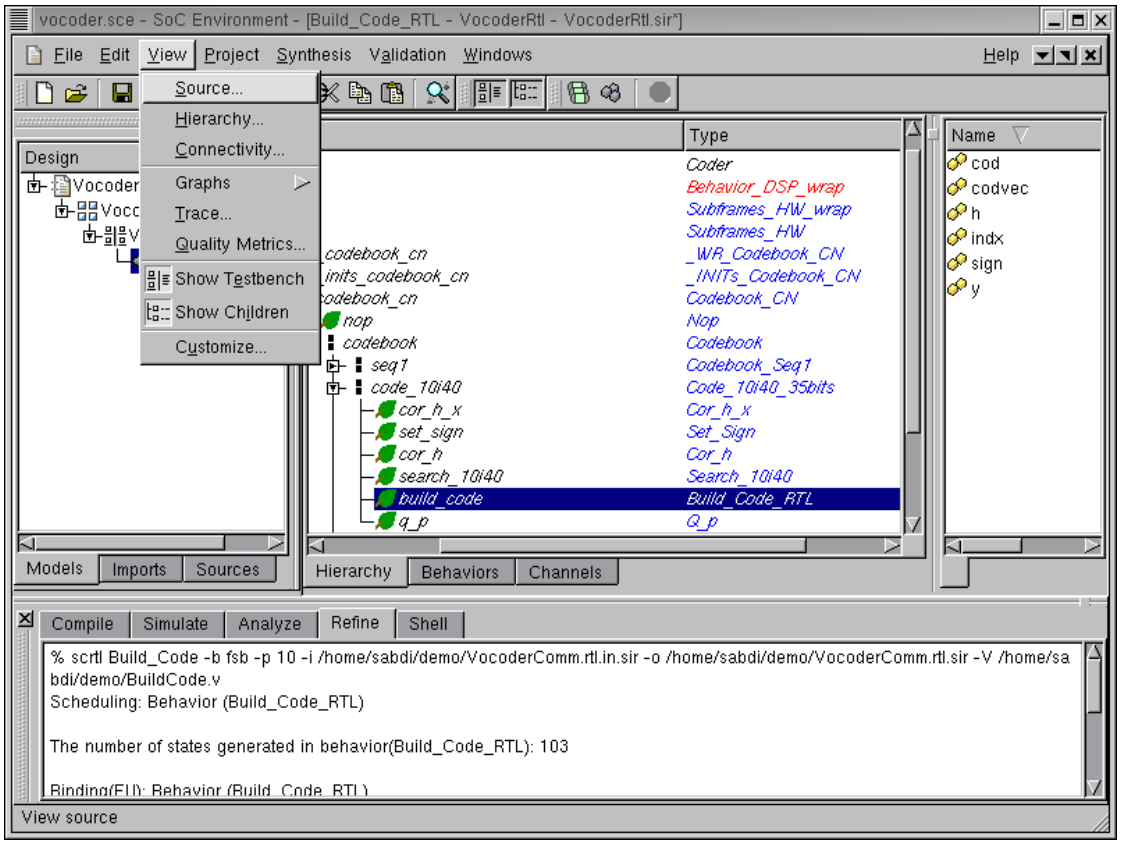
Check out the verilog code generated in the file BuildCode.v. This code is generated by the RTL refinemet tool. The designer may go the shell and launch his favorite editor to browse through the generated verilog code. Note that the verilog code has a single module named "Build_Code_RTL". This represents the sequential leaf behavior from our original design. Also note the functions representing each of the allocated RTL units.

5.3. Browse RTL model



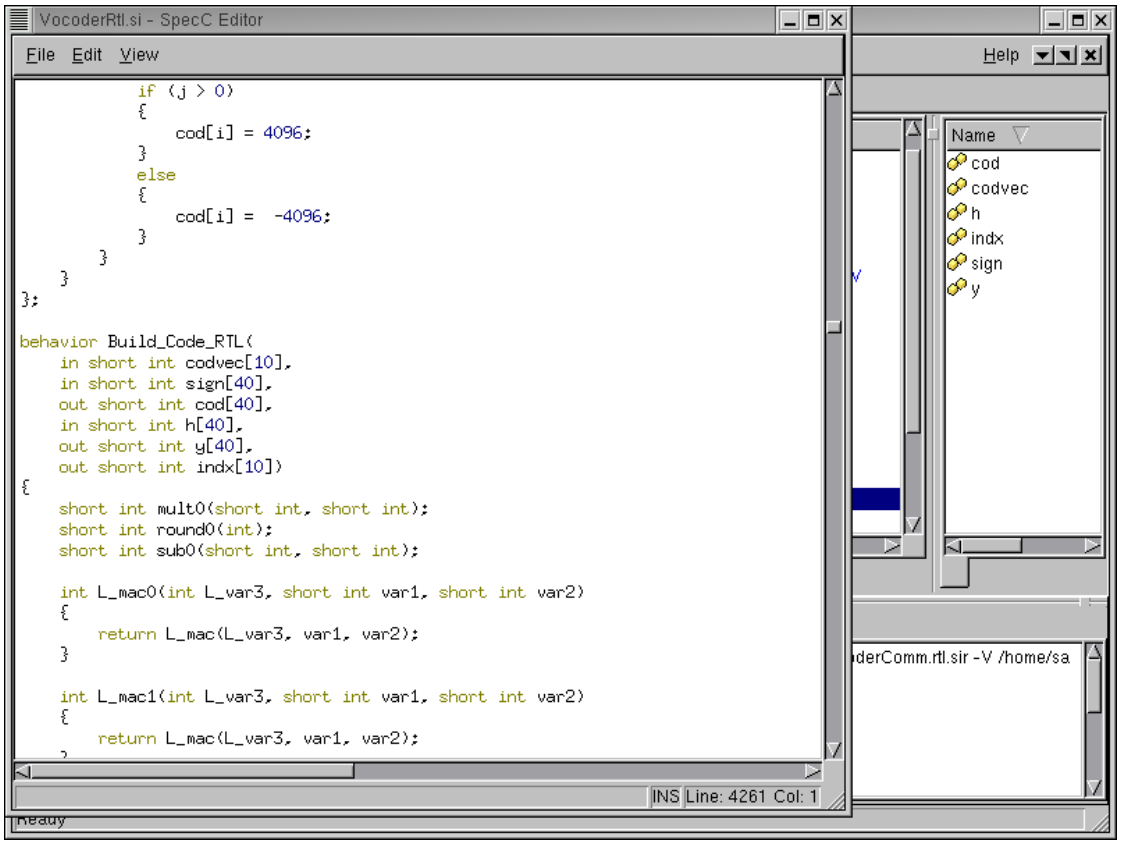
We now browse through the newly created model in the Design hierarchy window. Note that the type of the "Build_Code" behavior has now changed to "Build_Code_RTL" after the synthesis.

5.3.1. Browse RTL model (cont'd)



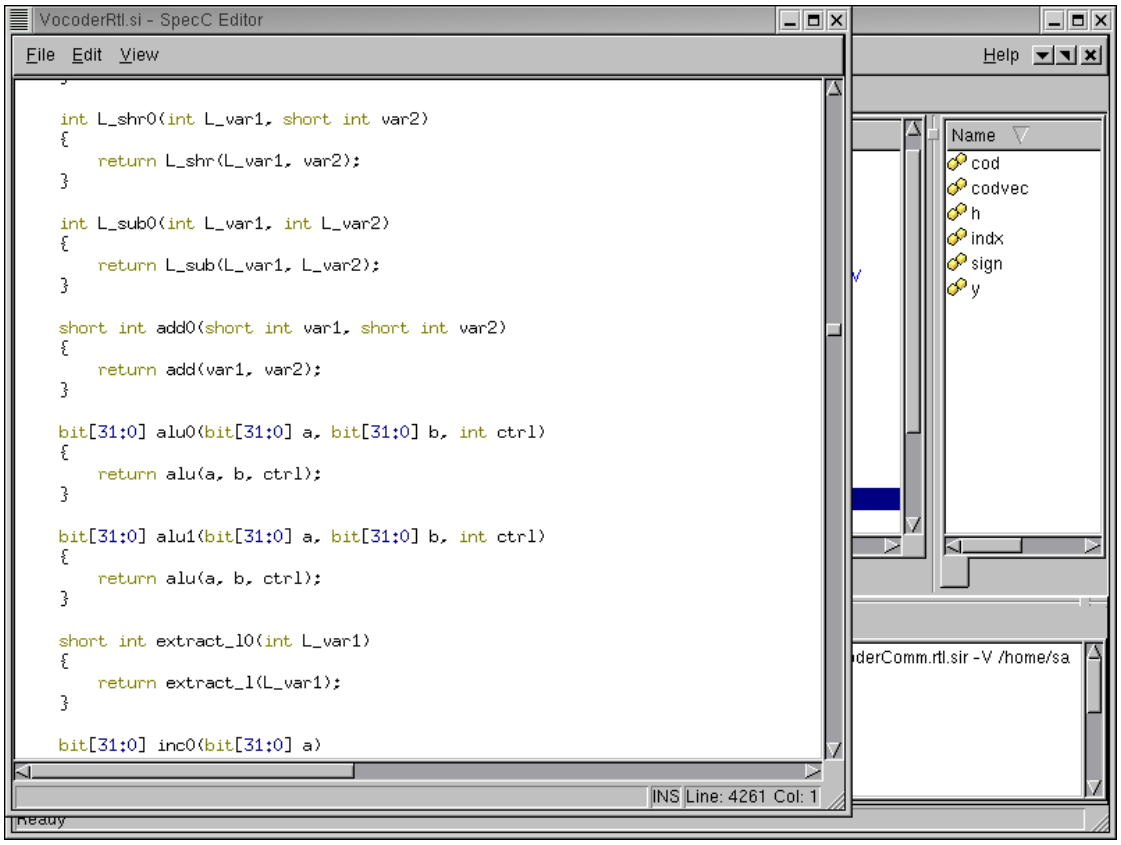
Select the behavior "Build_Code_RTL" by Left clicking on it. We now take a look at the synthesized source code to see if the RTL refinement tool has correctly generated the RTL model. Do this by selecting View⟶Source from the menu bar.

5.3.2. Browse RTL model (cont'd)



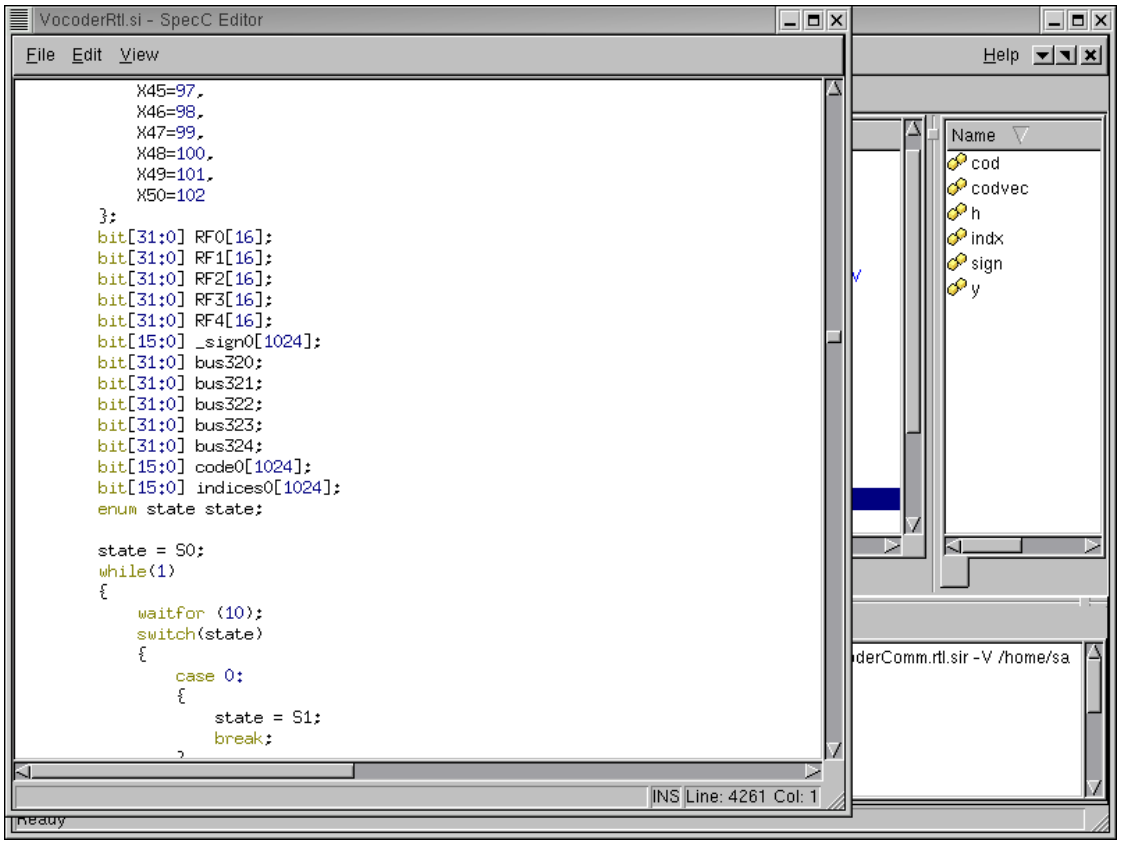
The source code editor pops up showing the RTL code for behavior "Build_Code_RTL."

5.3.3. Browse RTL model (cont'd)



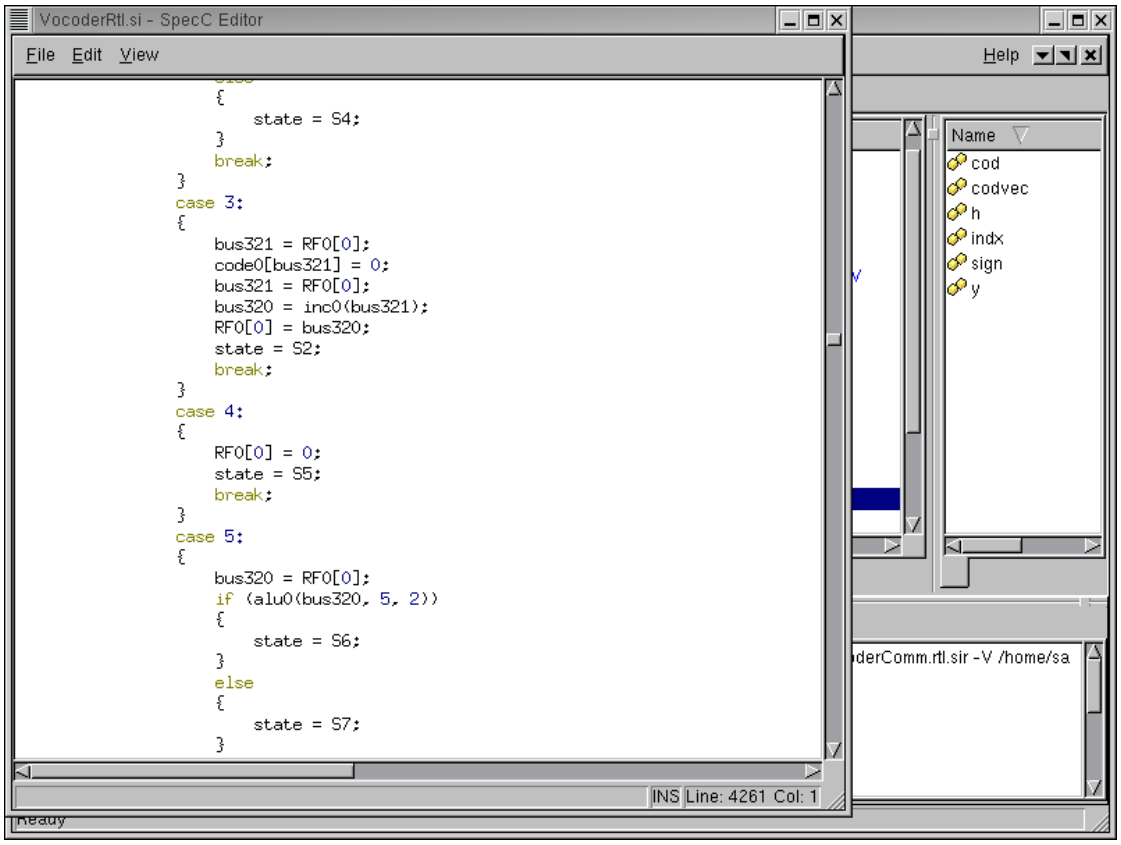
Scrolling down the editor window shows several function declarations in theis behavior. It is to be noted that these declarations correspond to the functions implemented for the allocated RTL units.

5.3.4. Browse RTL model (cont'd)



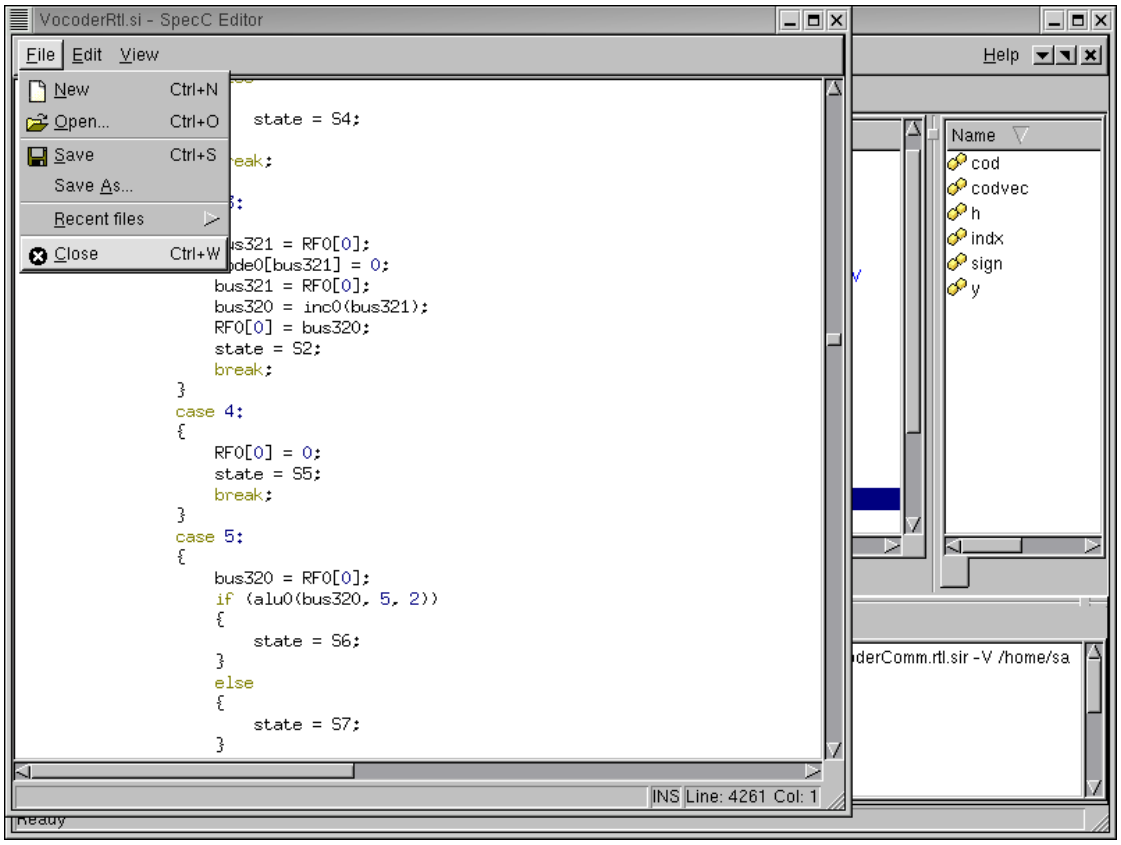
Scrolling down further shows the assignments for the state variables. Recall that the RTL synthesis produced 103 states. These states are enumerated here from 0 through 102. Note the final assignment (X50 = 102). Also, we can observe a while loop with a waitfor (10) statement in it. The waitfor statement signifies the clock delay indicating that state transitions are made at clock edges.

5.3.5. Browse RTL model (cont'd)



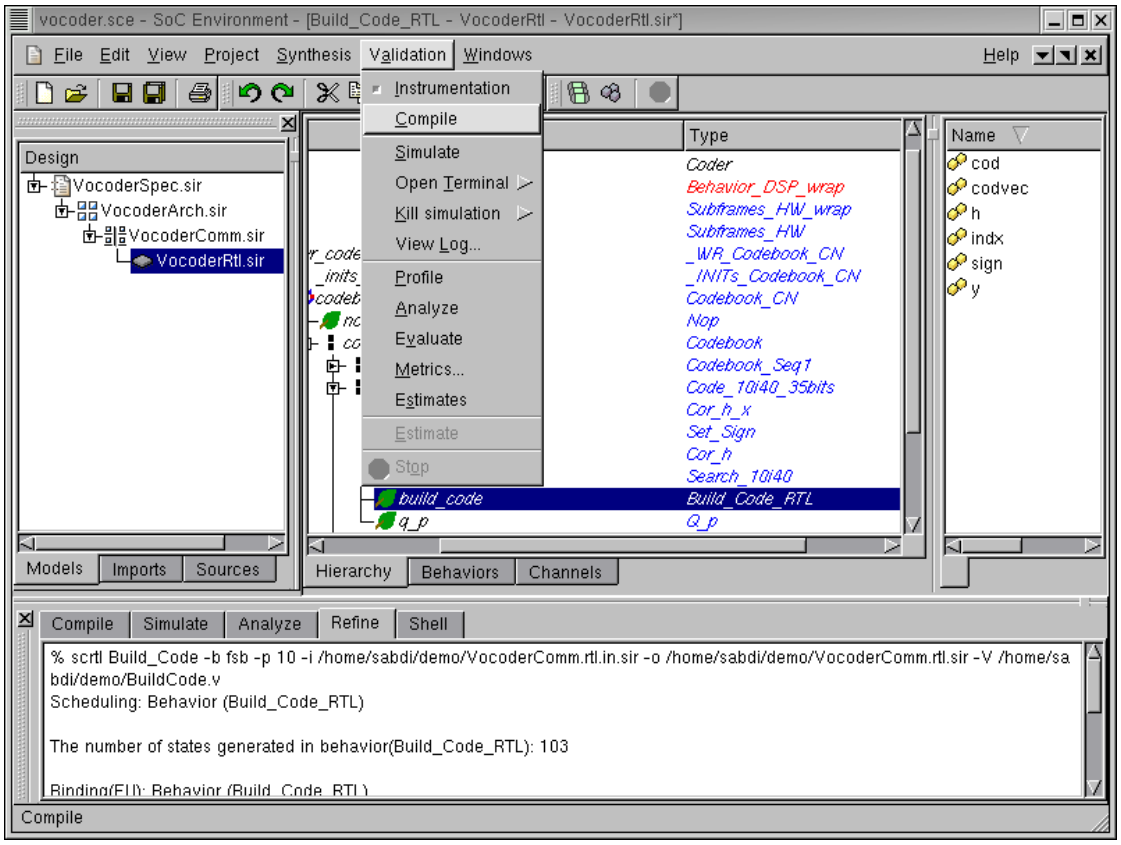
Further observations of the generated code show read/write operations on the register files. For instance RF0 is the register file written in the statement bus321 = RF0[0]; as shown in the code.

5.3.6. Browse RTL model (cont'd)



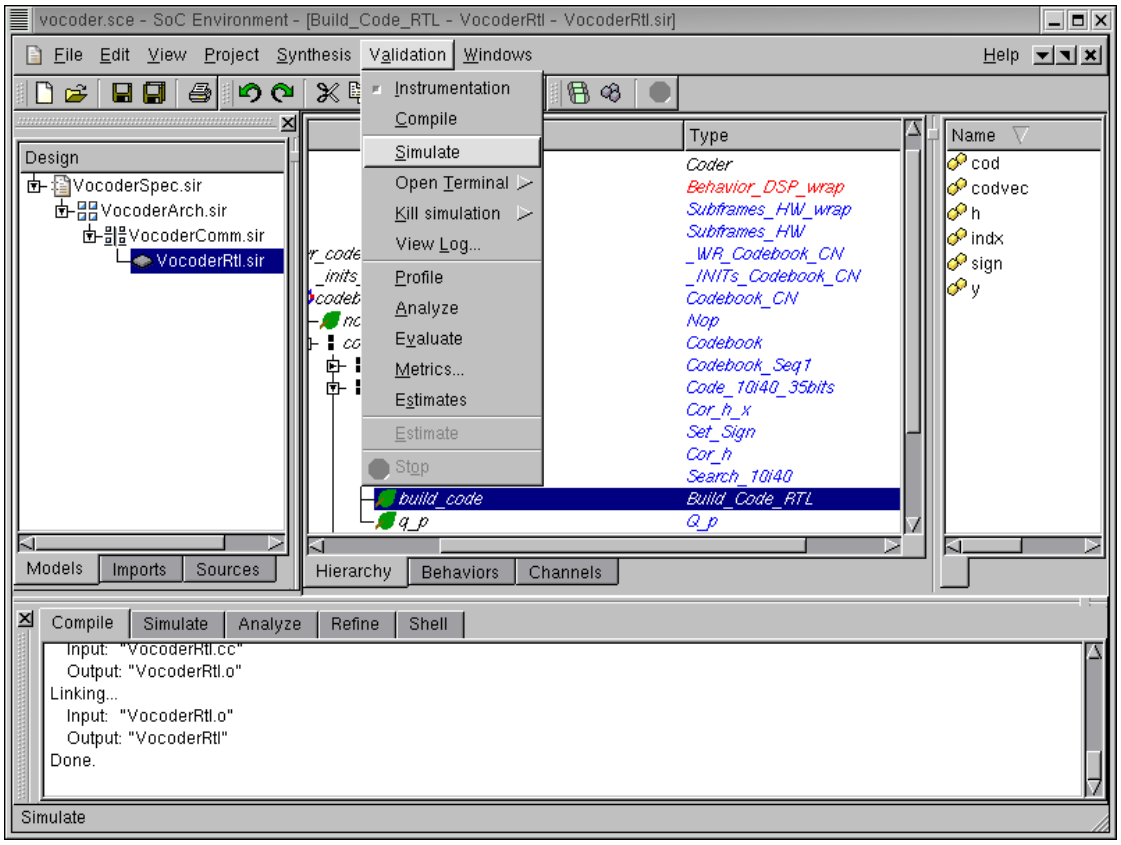
We conclude the browsing session by closing the editor using File⟶Close from the menu bar.

5.4. Validate RTL model



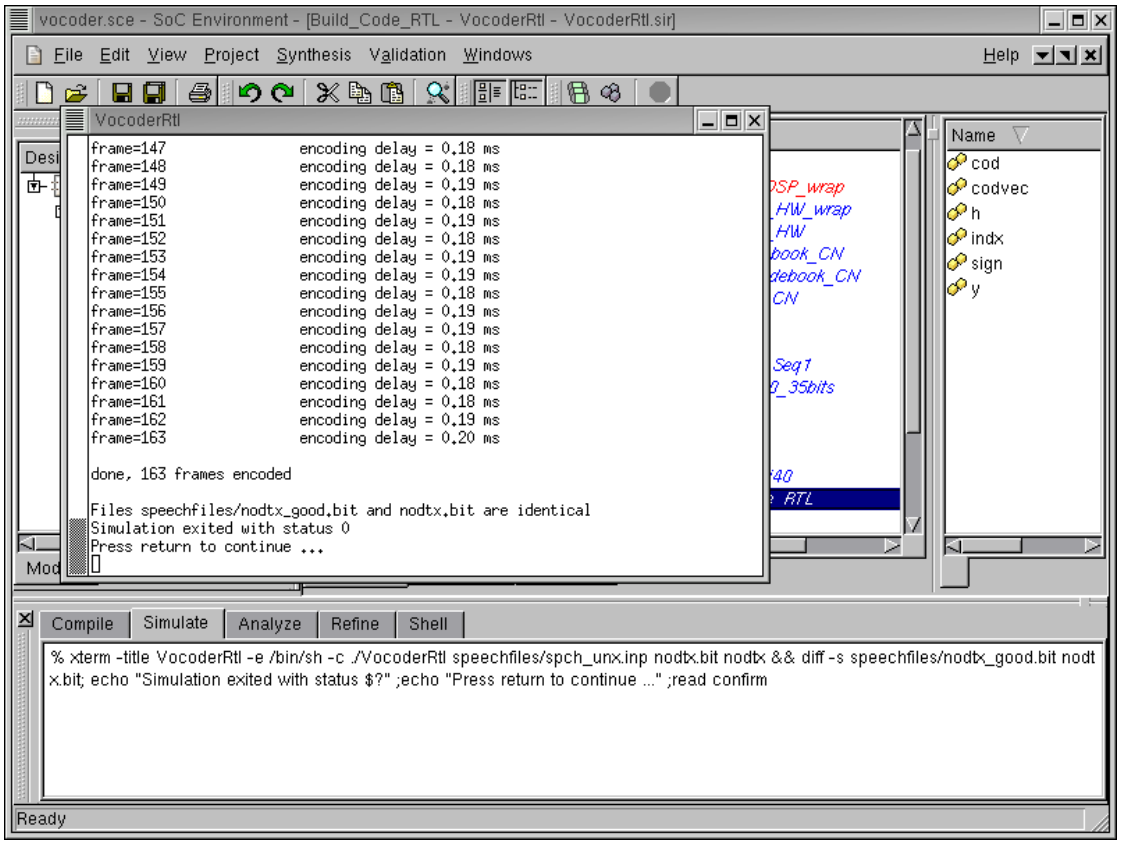
For demo purposes, we shall not perform RTL refinement on remaining behaviors assigned to HW. Proceed instead to validate the generated RTL model by selecting Validation⟶Compile from the menu bar.

5.4.1. Validate RTL model (cont'd)



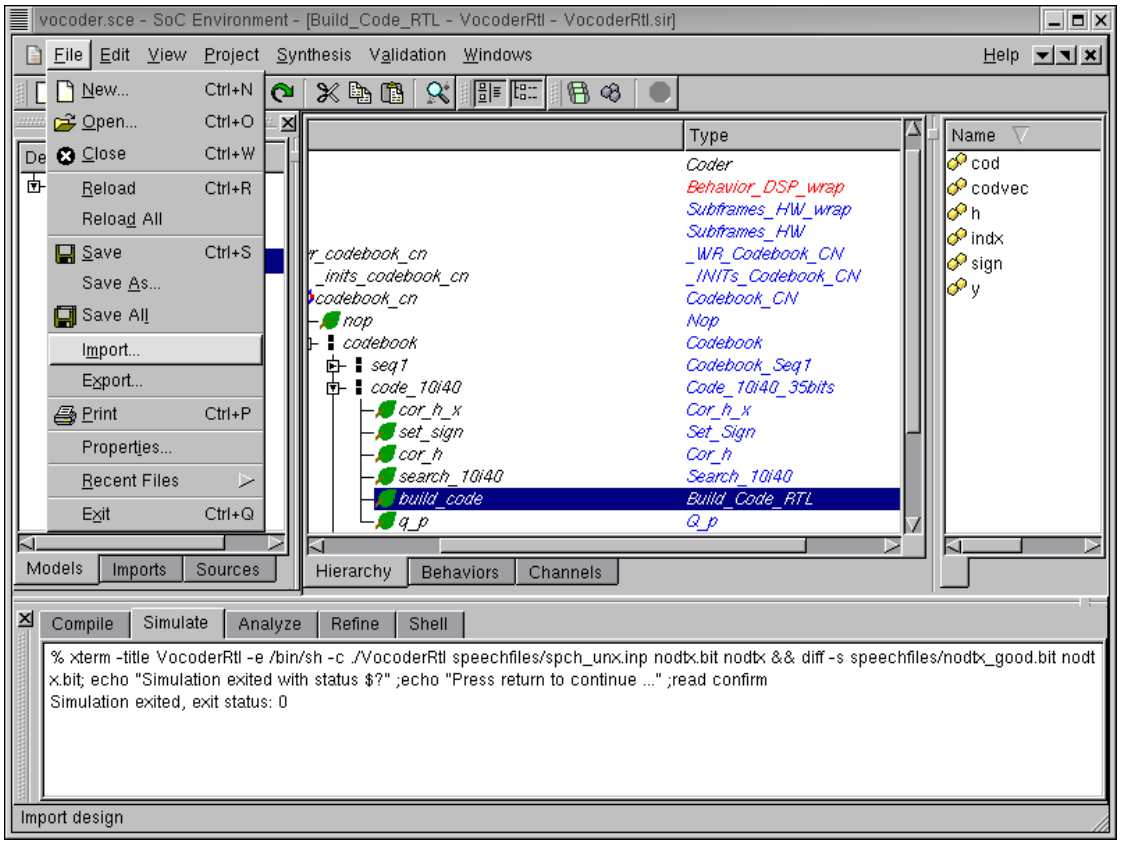
Note that the RTL model compiles correctly generating the executable VocoderRtl as seen in the logging window. We now proceed to simulate the model by selecting Validation$\longrightarrow$Simulate from the menu bar.

5.4.2. Validate RTL model (cont'd)



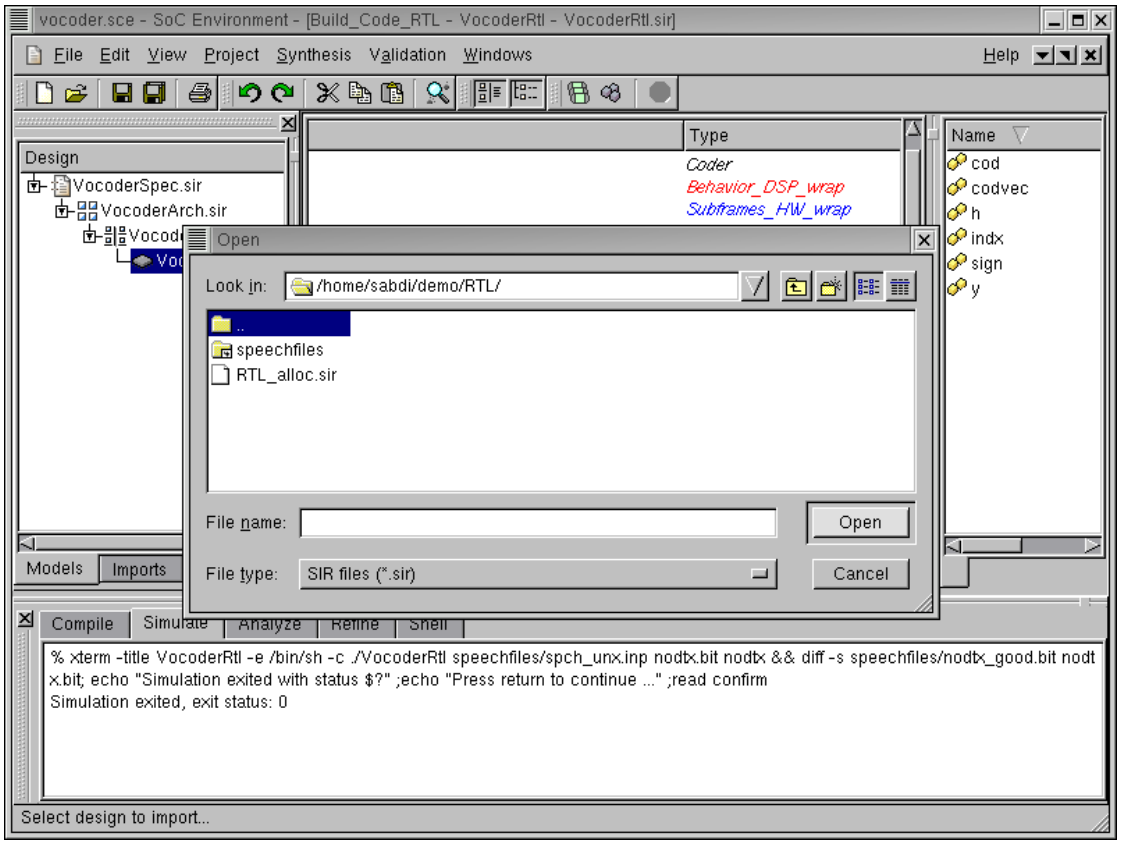
The simulation window pops up showing the progress and successful completion of simulation. We are thus ensure that the RTL refinement has taken place correctly. Also note that we can perform the refinement on any behavior of our choice. This indicates that the user has complete freedom of delving into one behavior at a time and testing it thoroughly. Since the other behaviors are at higher level of abstraction, the simulation speed is much faster than the situation when the entire model is synthesized. This is a big advantage with our methodology and it enables partial simulation of the design. The designer does not have to refine the entire design to simulate just one behavior in RTL.

5.5. SW code generation



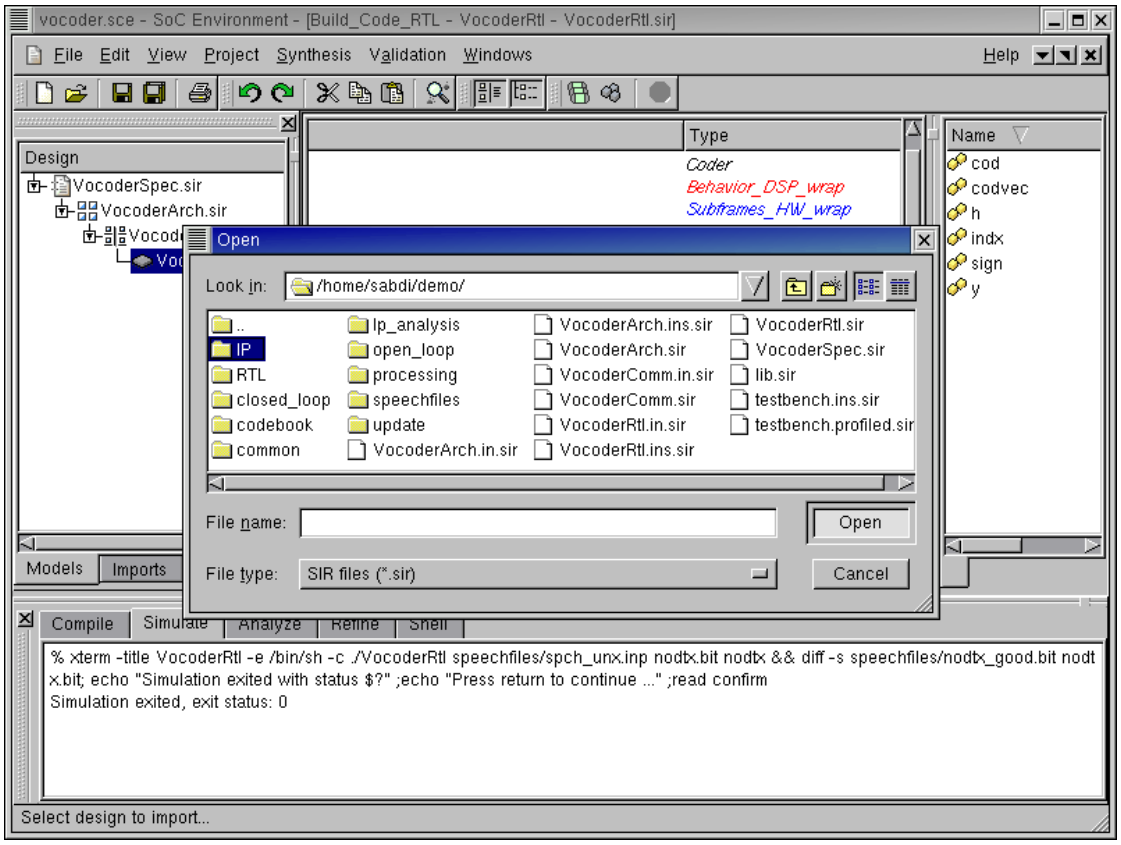
So far we have only generated the RTL model for the hardware part of our design. The behaviors that are mapped to the processor must be compiled for the DSP. We must now import the instruction set simulator (ISS) for the DSP Motorola 56600. Select File⟶Import from the menu bar.

5.5.1. SW code generation (cont'd)



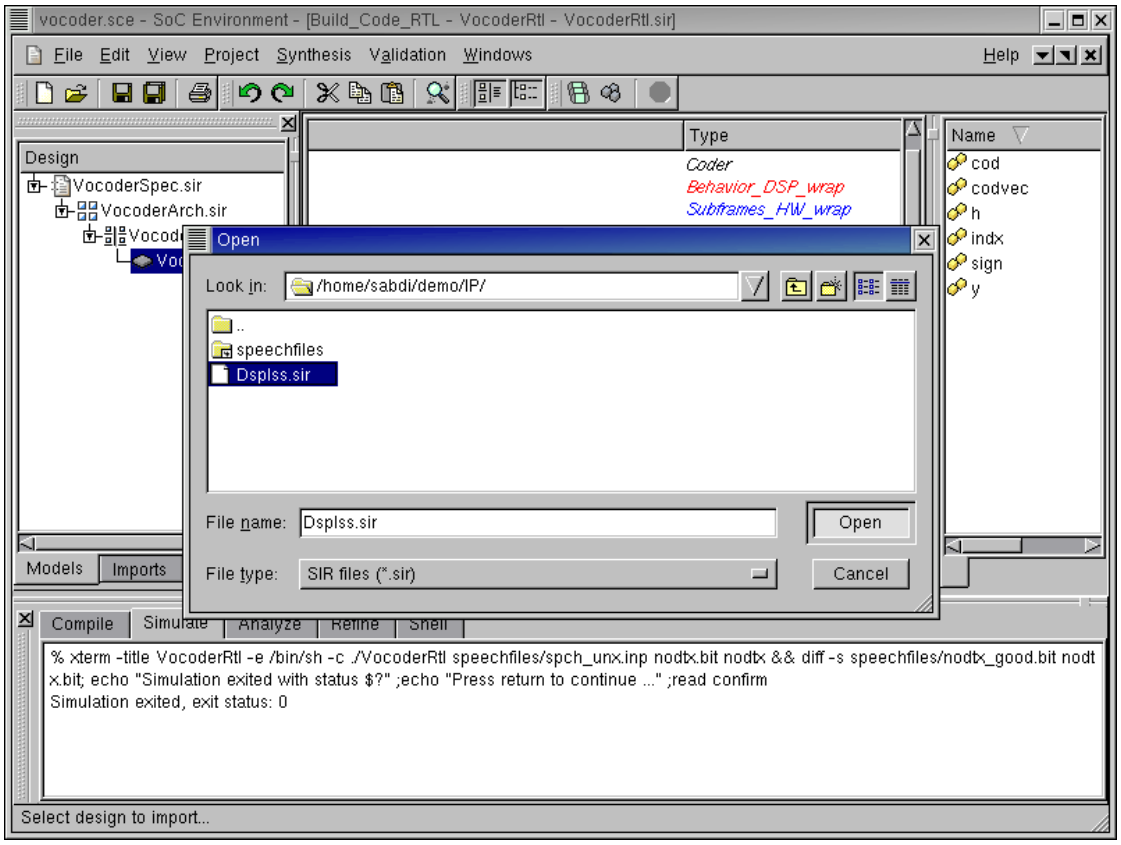
Go one level up in the directory tree by double Left click on ".."

5.5.2. SW code generation (cont'd)



Select directory "IP" from the file selection menu by double Left click.

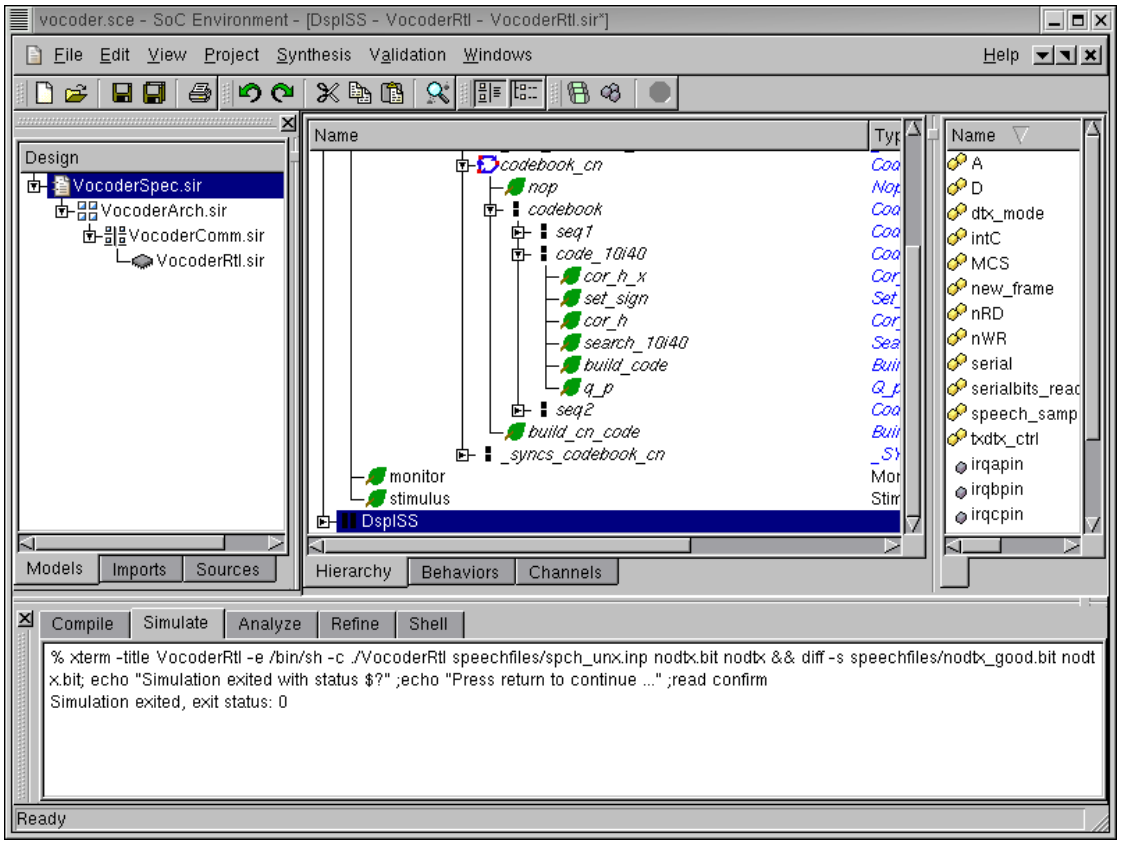
5.5.3. SW code generation (cont'd)



Inside deirectory IP, select "DspIss.sir" and Left click on Open.

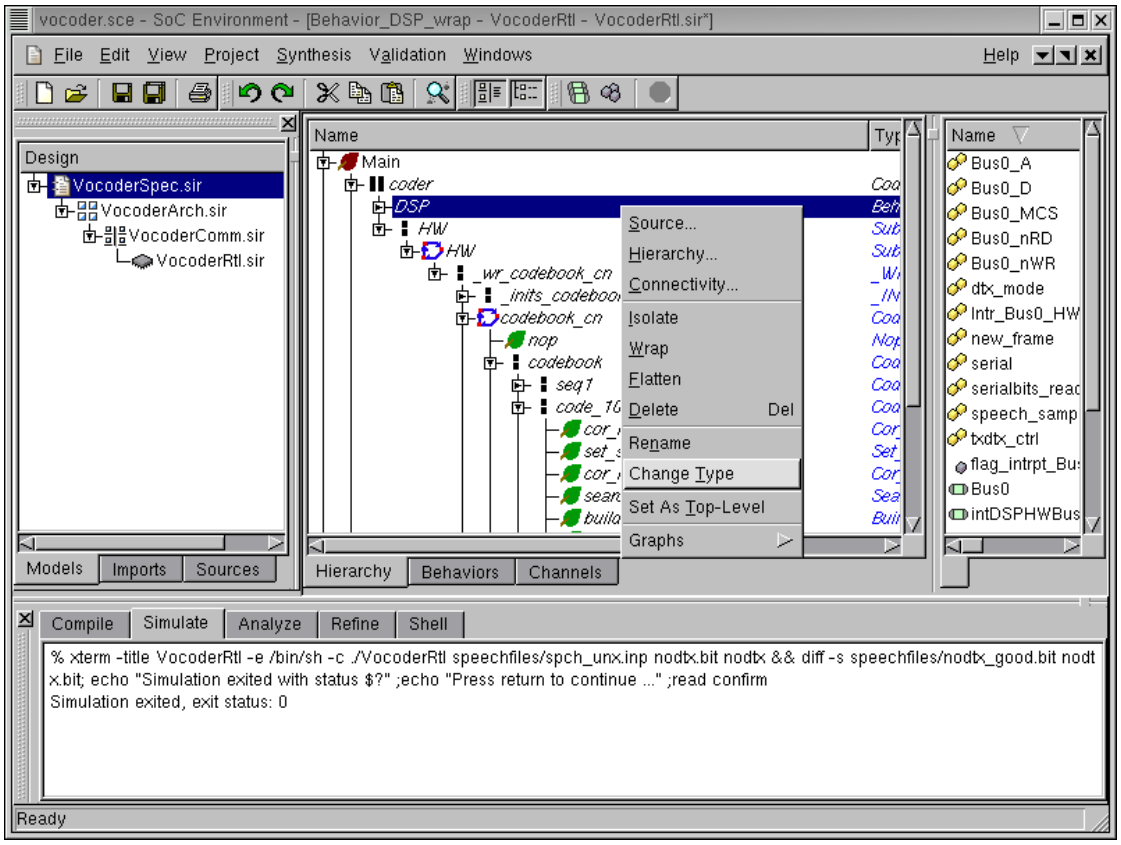
The SIR file contains the instruction set simulator for our chosen DSP. The behavior loads the compiled object code for the tasks that were mapped to DSP and executes it on the instruction set simulator.

5.5.4. SW code generation (cont'd)



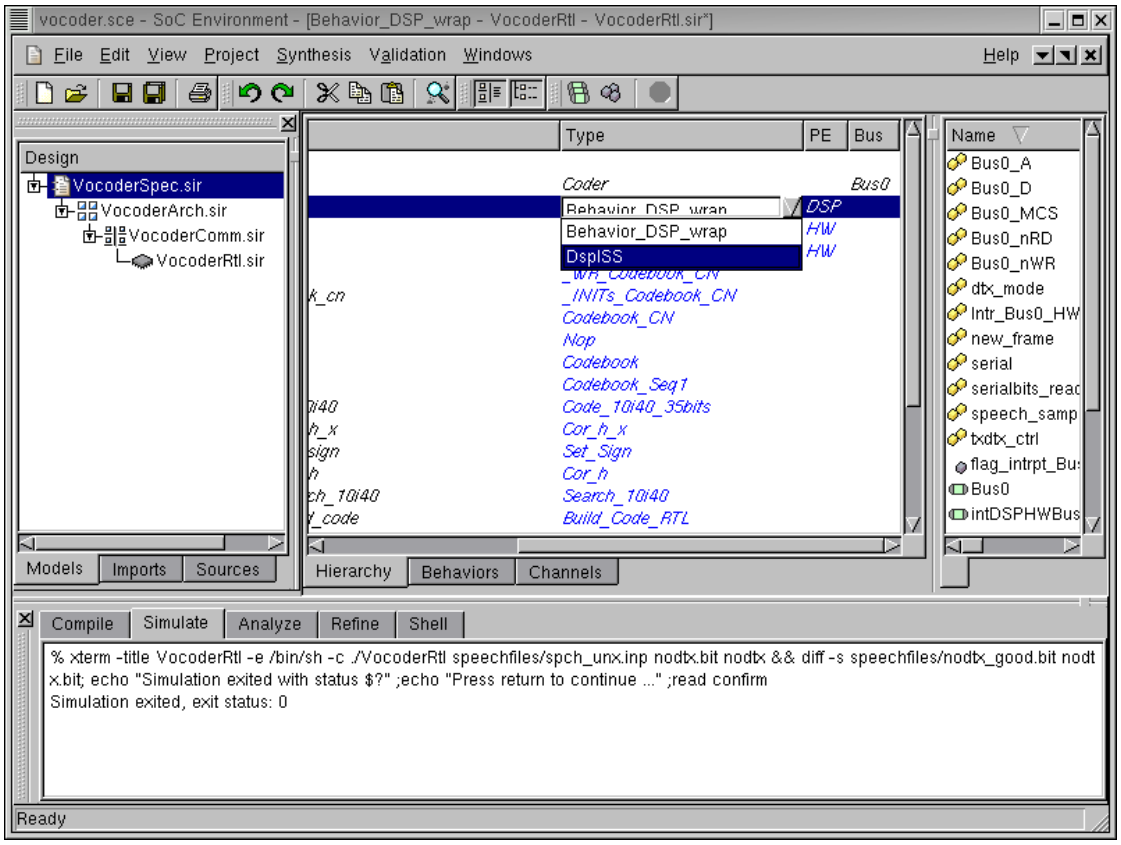
Once "DspIss.sir" is imported, we can notice DspISS as a new root behavior in the design hierarchy tree. This is because the DspISS behavior has not been instantiated yet.

5.5.5. SW code generation (cont'd)



In the design hierarchy tree, select behavior DSP. Right click and select Change Type.

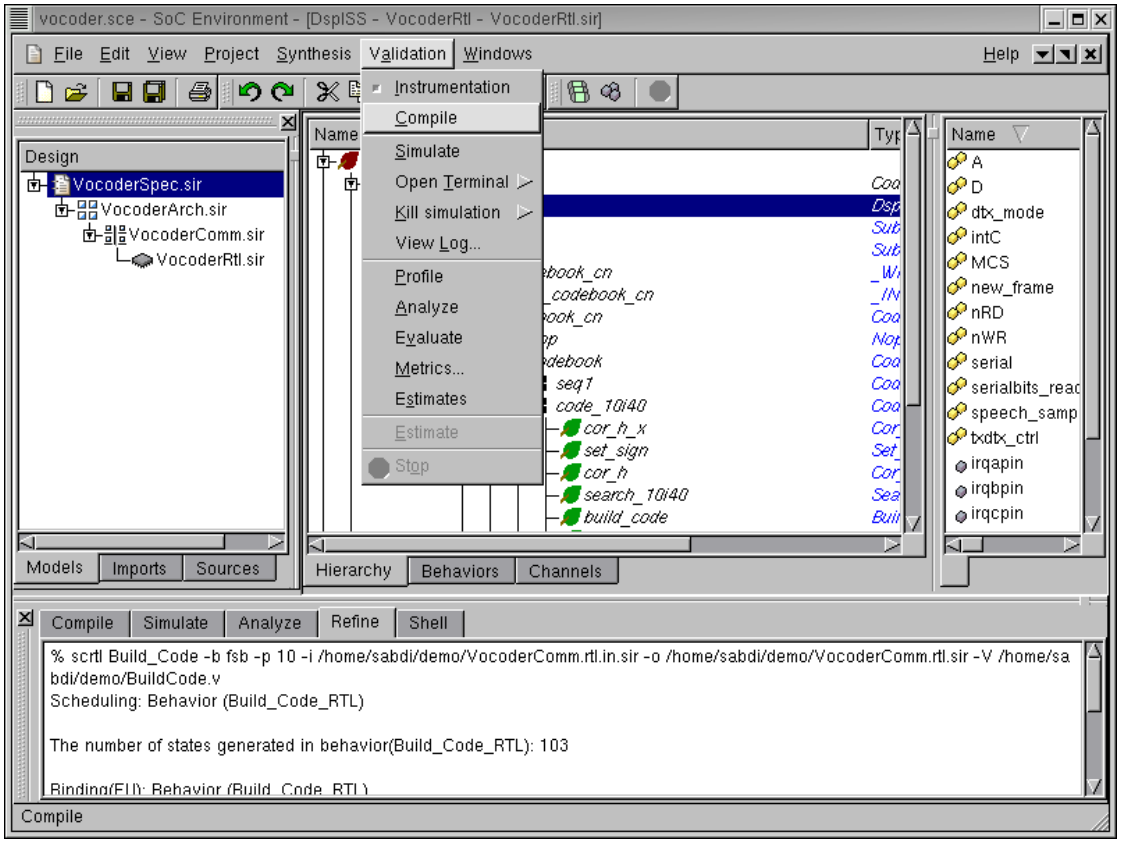
5.5.6. SW code generation (cont'd)



The type of "DSP" behavior may now be changed by selecting DspISS.

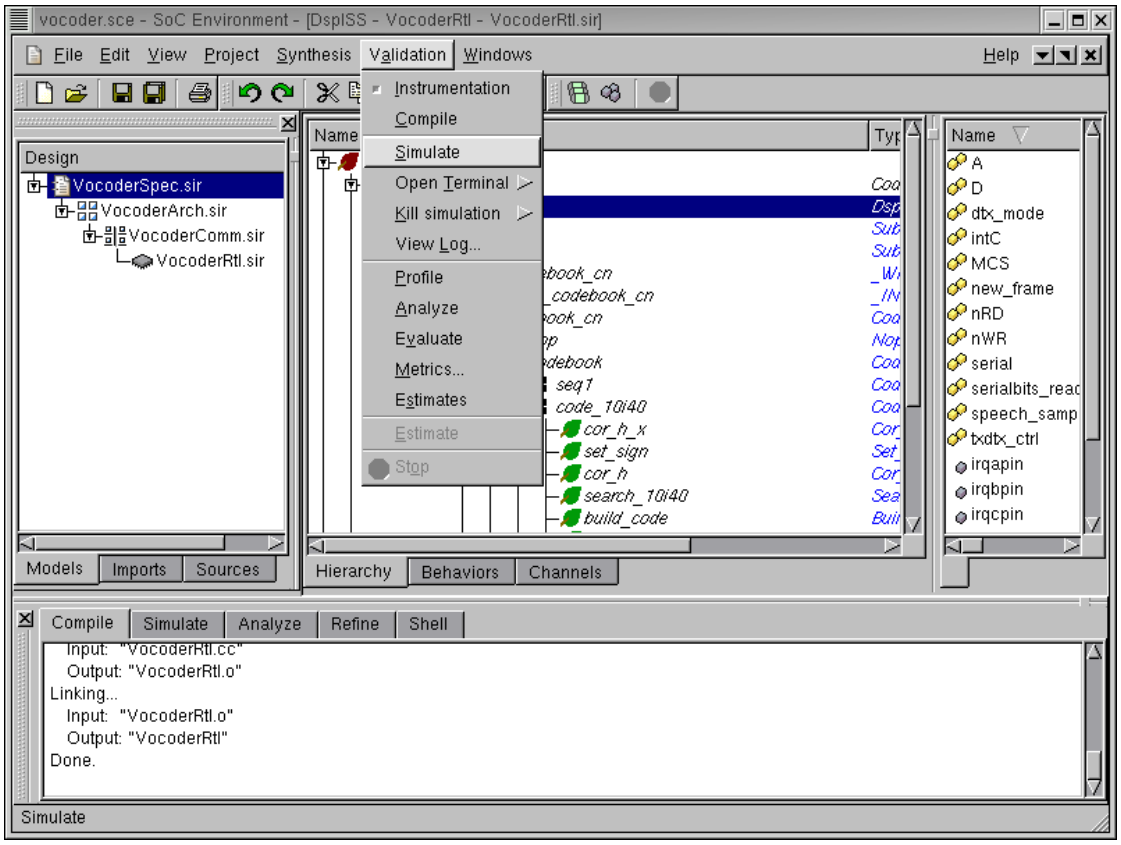
By doing this, we have now refined the software part of our design to be implemented with the DSP56600 processor's instruction set. Recall that the software part mapped to DSP has already been compiled for the DSP56600 processor and the object file is ready. As mentioned earlier, the new behavior will load this object file and execute it on the DSP's instruction set simulator. Thus the model becomes clock cycle accurate.

5.6. Validate implementation model



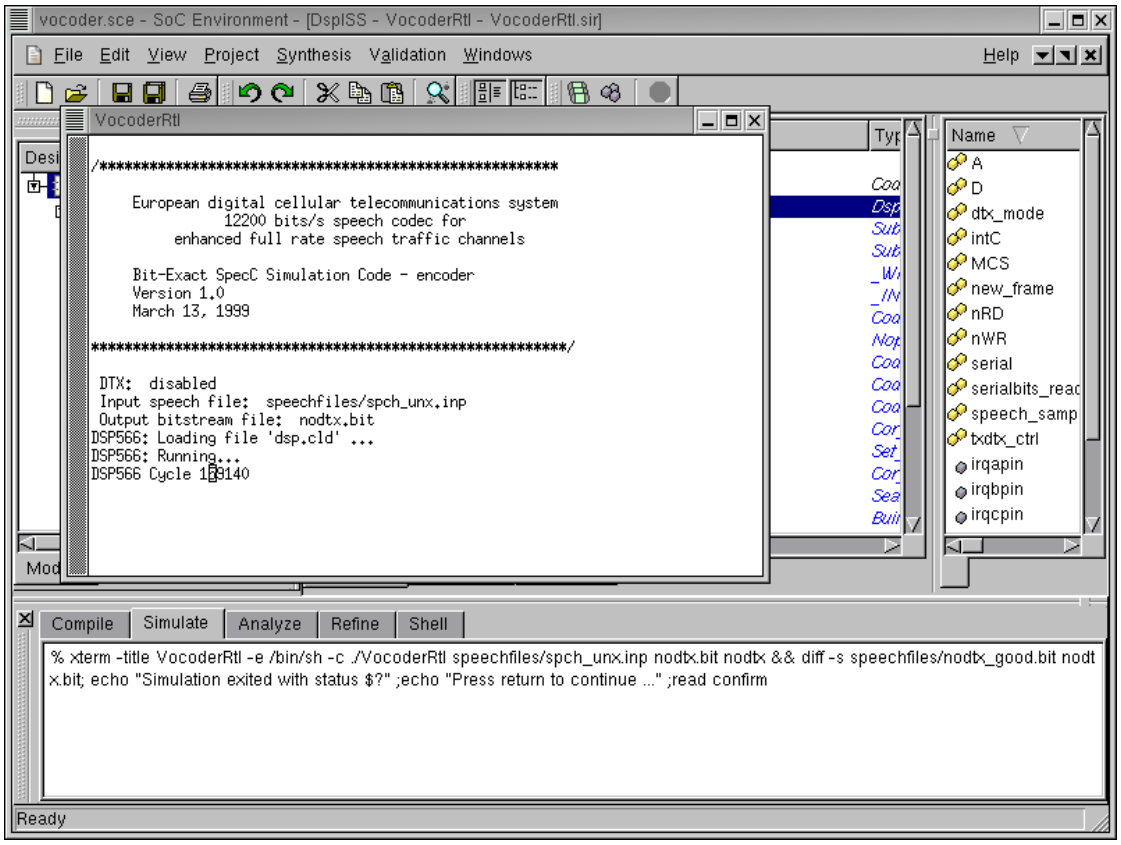
We now have the clock cycle accurate model ready for validation. We begin as usual with compiling the model by selecting Validation—→Compile from the menu bar.

5.6.1. Validate implementation model (cont'd)



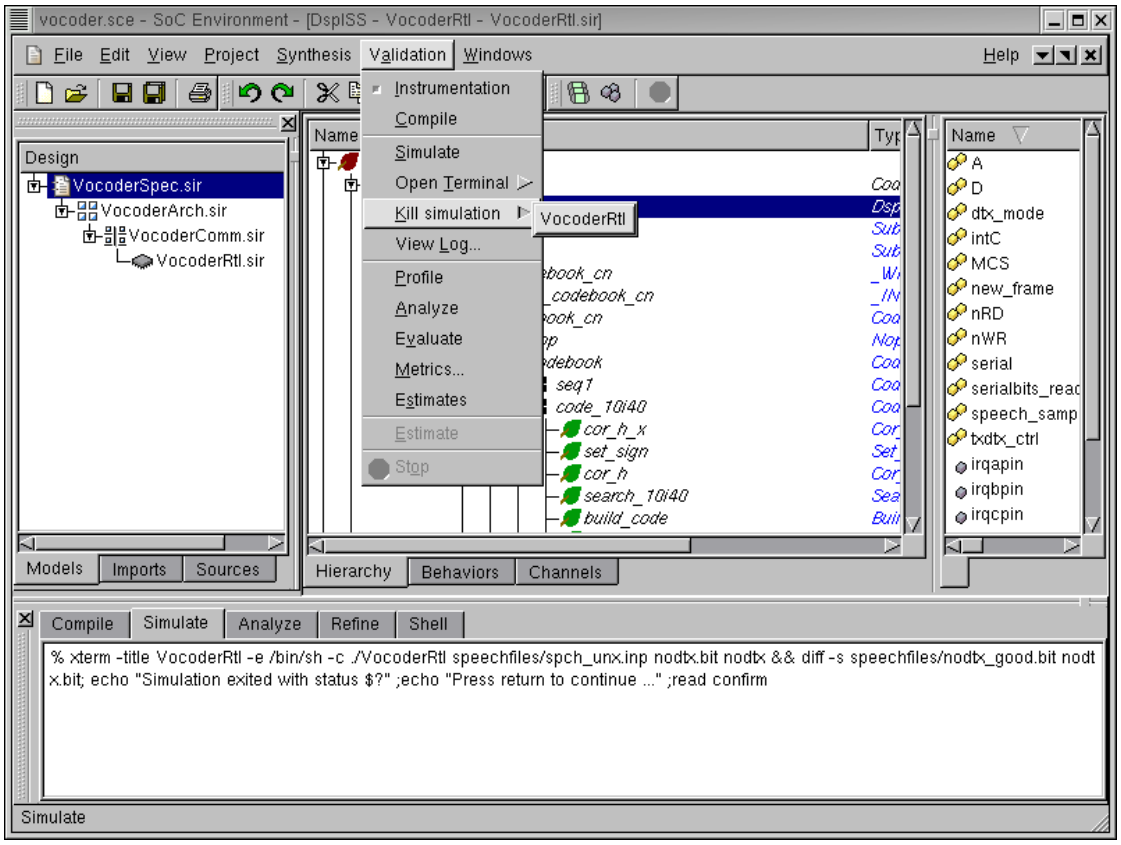
The model compiles correctly as shown in the logging window. We now proceed to simulate the model by selecting Validation$\longrightarrow$Simulate from the menu bar.

5.6.2. Validate implementation model (cont'd)



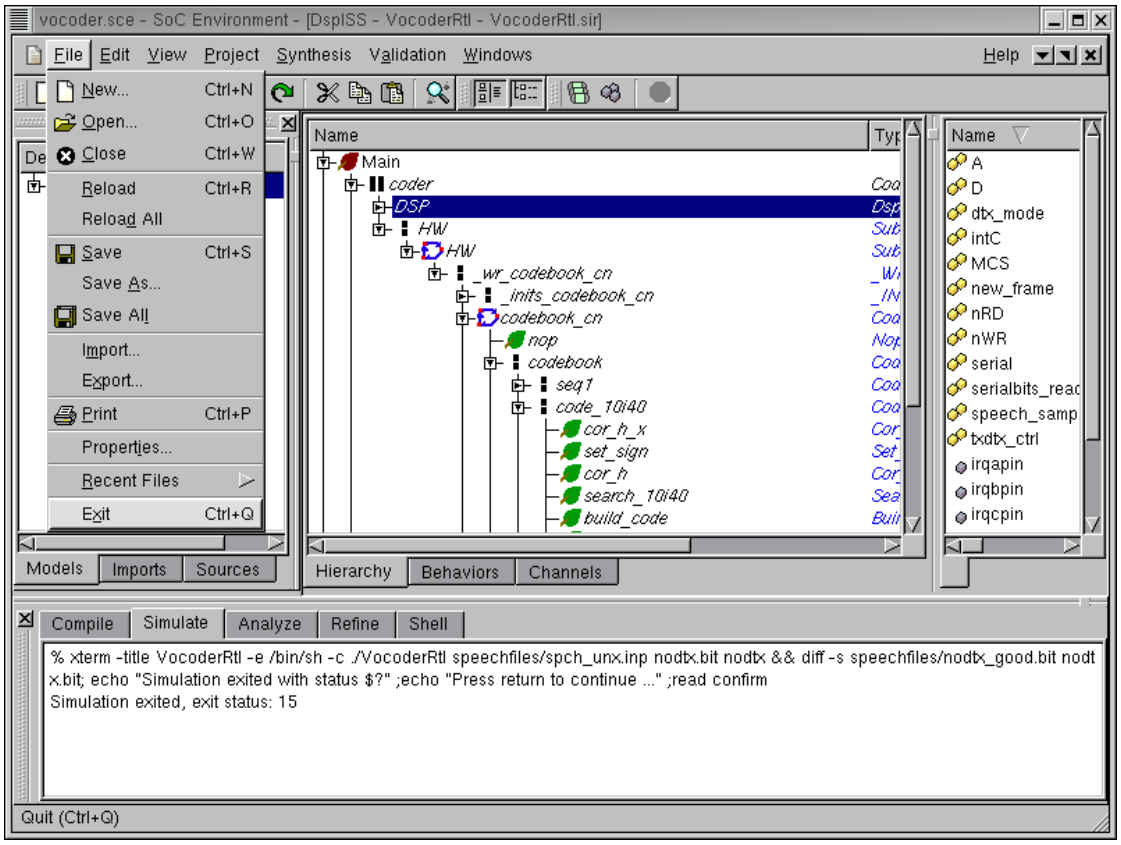
Like in the earlier cases, a simulation window pops up. The DSP Instruction set simulator can be seen to slow down the simulation speed considerably. This is because the simulation is being done one instruction at a time in contrast to the high level simulation we had earlier.

5.6.3. Validate implementation model (cont'd)



It may take hours for the simulation to complete. The simulation may be killed by selecting Validation—→Kill simulation from the menu bar.

5.6.4. Validate implementation model (cont'd)



The demo has now concluded. To exit the SoC environement, select Project—→Exit from the menu bar.

Chapter 6. Conclusion

In this tutorial we presented the System on Chip design methodology. The SoC methodology defines the 4 models and 3 transformations that bring an initial system specification down to an RTL-implementation. In addition to validation through simulation, the well-defined nature of the models enables automatic model refinement, and application of formal methods, for example in verification.

The complete design flow was demostrated on an industrial strength example of the Vocoder Speech encoder.We have shown how SCE can take a specification model and allow the user to interactively provide synthesis decisions. In going from specification to RTL/Instruction-set model for the GSM Vocoder, we noted that compared to traditional manual refinement, the automatic refinement process gives us more than a 1000X productivity gain in modeling, since designers do not need to rewrite models.

Table 6-1. Vocoder Refinement Effort

Refinement Step	Modified Lines	Manual Refinement	Automated Refinement
Spec -> Arch	3,275	3~4 months	~1 min.
Arch -> Comm	914	1~2 months	~0.5 min.
Comm -> RTL/IS	6,146	5~6 months	~2 min.
Total.	**10,355.**	**9~12 months.**	**~4 mins.**

To draw the conclusion, SCE enables the designer to use the following powerful advantages that have never been available before.

1. **Automatic model generation.**

   New models are generated by *Automatic Refinement* of abstract models. This means that the designer may start with a specification and simply use design decisions to automatically generate models reflecting those decisions.

2. **Eliminates SLDL learning.**

   SCE *eliminates the need for system-level design languages* to be learnt by the designer. Only the knowledge of C for creating specification is required.

3. **Eliminates SLDL learning.**

   This also *enables non-experts to design* systems. There is no need for the designer to worry about design details like protocol timing diagrams, low level interfaces etc. Consequently, *software developers can design hardware* and *hardware designers can develop software*.

4. **Supports platforms.**

   SCE is great for *platform based design* . By limiting the choice of components and busses, designers may select their favorite architecture and then play around with different partitioning schema.

5. **Customized methodology.**

   SCE can also be *customized to any methodology* as per the designer's choice of components, system architecture, models and levels of abstraction.

6. **Enables IP trading.**

   SCE simplifies *IP trading* to a great extent by allowing interoperability at system level. With well defined wrappers, the designer can plug and play with suitable IPs in the design process. If an IP meets the design requirements, the designer may choose to plug that IP component in the design and not worry about synthesizing or validating that part of the design.

References

A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers Inc. June, 2001

D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers Inc. March, 2000

D. Gajski, F. Vahid, S. Narayan, and J. Gong *Specification and Design of Embedded Systems*. Prentice Hall June, 1994

D. Gajski, F. Vahid, S. Narayan, and J. Gong “SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design”. IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 84-100 1998, Awarded the IEEE VLSI Transactions Best Paper Award, June 2000

D. Gajski, L. Ramachandran, F. Vahid, S. Narayan, and P. Fung “100 hour design cycle : A test case”. Proc. Europ. Design Automation Conf. EURO-DAC 1994