# ICS

## TECHNICAL REPORT

# SpecC Profiler: Specification-level Exploration Tool

Sinisa Srbljic
Mario Stefanec
Ivan Benc

Technical Report #00-29
September 12, 2000

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA



Information and Computer Science
University of California, Irvine

# SpecC Profiler: Specification-level Exploration Tool

Sinisa Srbljic
Mario Stefanec
Ivan Benc

Technical Report #00-29
September 2000
(Version as on September 12, 2000)

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

sinisa@ics.uci.edu
mario@ics.uci.edu
ibenc@ics.uci.edu

## Abstract

*SpecC methodology of the system design consists of four major hierarchical levels: specification, architecture, communication, and implementation. The SpecC Profiler is a high-level process within the SpecC methodology, which analyzes system design at the specification level. To achieve fast profiling with satisfactory accuracy, the SpecC Profiler relies on simulation and front-end compiler tools. Each subpart of the specification-level design is associated with the profiling information, targeting computational complexity, storage requirements, and communication complexity of the specification design. In addition, SpecC Profiler predicts the performance, such as number of instructions or execution time.*

*This report describes the profiler architecture and implementation. The accuracy of the profiler is asserted by comparing the performance predicted by the profiler with the results of simulated execution of different applications, like JPEG and Vocoder.*

# Contents

## List of Figures

## List of Tables

# 1 Introduction

During system-level design, designers choose different models in different phases of the design process in order to implement desired functionality using a set of physical components. The SpecC design process or methodology is a set of design tasks that analyze and explore a set of models at different hierarchical levels, i.e. at different stages of the design process [1]. It starts with specification model, transforming it into architecture model, followed by the communication and implementation model.

Profiler is a part of the unified SpecC methodology. Its main role within the SpecC methodology is to improve the specification model. The specification model defines the functionality of the system, and does not reflect any architecture, communication or implementation information. The profiler helps designer to analyze and explore the specification model, targeting computation complexity, storage requirements, and communication demands of the given specification model and all of its parts. Efficiency of the given specification model can be improved by introducing parallelism, or by using different calculation algorithms. Refinement of the design's specification model is an iterative process that stops when designer is satisfied with the predictions provided by the profiler. Refined specification model is then transformed into the architecture model [1]. At the architecture level, SpecC methodology uses estimator, another SpecC exploration tool, which gives more precise, clock cycle accurate, information about performance of the design on the different architectures. Next stage in SpecC methodology is the communication model [1]. It introduces actual wires and timing relationships according to the used communication protocols. Finally, communication model is transformed into the implementation model [1], which represents a clock-cycle accurate description of the system.

Since the system-level design process is time-consuming, due to searching through the large design space, the main goal of the profiler is to improve the productivity and quality of the design process. The productivity is achieved by shortening the time needed to design the system. Since the profiler reduces the time needed to design the system, it enables the search of even larger design space, improving the quality of design by examining more possible solutions. Moreover, the profiler is also used as a simple performance tool that predicts the metrics such as execution time, number of executed instructions, etc.

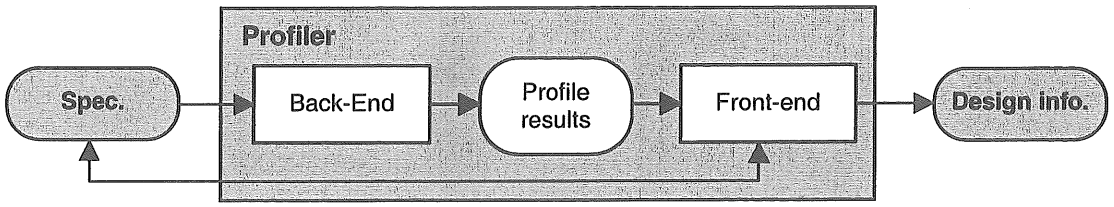



**Figure 1:** General profiler architecture

Section 2 describes the SpecC profiler architecture. The details of the profile results extraction and calculation are given in Section 3. Performance prediction process is presented in Section 4. In Section 5 we discuss the accuracy of performance prediction. The implementation details of the profiler and our future work are described in Section 6.

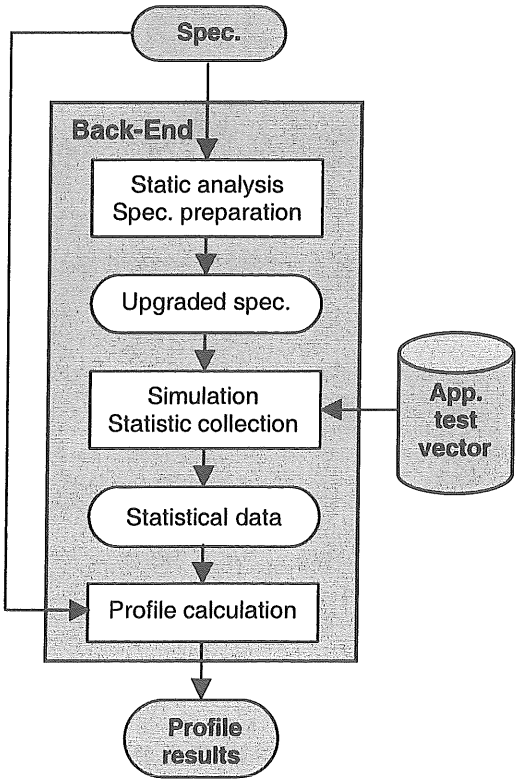



**Figure 2:** Basic back-end functions

## 2 Profiler architecture

Figure 1 shows general profiler architecture consisting of two major parts: back-end and front-end. The profiler back-end takes the specification model of the design as input, analyzes it, performs profiling, and generates profile results as output. Profile results are input for the profiler front-end, which predicts performance, enables modification of the specification model, and displays all of these results in graphical form. While Section 2.1 describes profiler back-end in more details, Section 2.2 describes the front-end.

### 2.1 Profiler back-end architecture

Figure 2 shows basic back-end functions: static analysis, specification preparation, simulation, statistic results collection, and profile results calculation.

Specification model is written in the SpecC language [1-3]. The basic SpecC language syntax and semantic structure is a *behavior* [1-

3]. The *Behavior hierarchy* is used to decompose a system's specification into the sequential and parallel subbehaviors, i.e. it enables the decomposition of specification into the hierarchy of components. Each behavior is a class consisted of a set of *ports* for I/O communication with other behaviors, a set of *subbehavior instantiations*, a set of *private variables* and *functions*, and a public main *function* [1].

*Static analysis* performs control flow analysis, which builds the basic blocks structure for all parts of the specification model, i.e. for all behaviors in a given hierarchy, as well as for all functions within each behavior.

Once basic blocks information is known, *specification preparation* upgrades the specification model and prepares it for the simulation and statistical data collection. For example, the specification model is instrumented by inserting counters in each basic block in order to keep track of the number of basic block executions.

A functionally accurate *simulation* of the *upgraded specification model* is run for each of a set of test vectors from the *application-test-vectors database*. Upgraded specification model is compiled and executed on the host machine. If the simulation is run with different test vectors, then profiling results represent average value for a given set of vectors.

During the simulation, *statistic results collection* updates counters containing statistical information about total number of the basic blocks executions. At the end of the functional simulation, it produces *statistical data* for all basic blocks.

Statistical data, combined with information obtained during the static analysis, is used to calculate the *profile results*. *Profile results calculation* is computationally intensive process, and since it is a core function of the profiler back-end, it is separately presented in details in Section 3.

### 2.2 Profiler front-end architecture

Figure 3 shows basic front-end functions: display, editor, and performance prediction.

*Display* presents, in graphical form, various data associated with the specification model: SpecC source code, behavior hierarchy, behavior-to-behavior communication, profile results, and performance prediction results. Figure 4 shows the example of the output of the display.

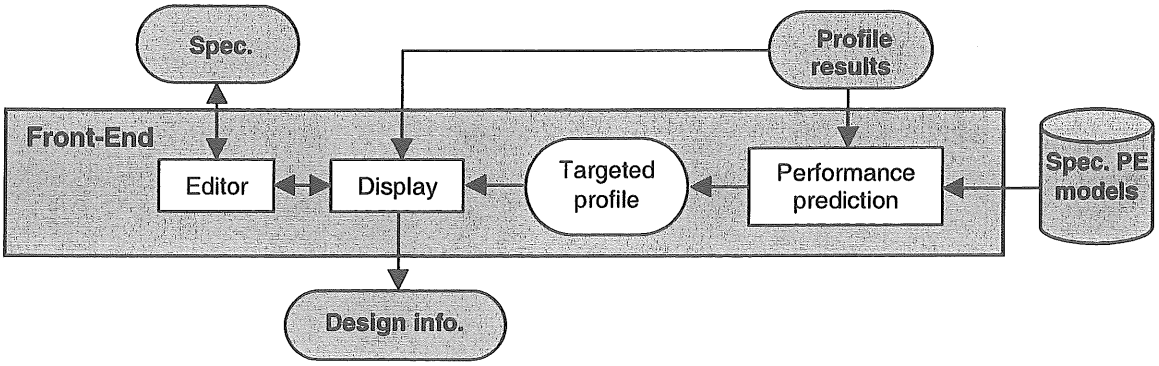



**Figure 3:** Basic front-end functions

The profile and performance prediction results are displayed both graphically and numerically. The presentation of the results is organized hierarchically due to the amount of data to be displayed. For example, at the top of the results hierarchy, designer sees graph with only three bars: average number of operations executed by a chosen behavior, behavior I/O traffic, and behavior storage requirements. By a double-click on the appropriate bar, the designer sees more details. For example, by double-click on the operation bar, the graph with the following average information is presented: number of arithmetic operations, number of memory accesses, number of function calls, number of logic and bit operations, and number of data type conversion operations. Designer can chose either to see even more details for each of the given bars, or can return back to the higher level of the results hierarchy.

*Editor* cooperates with the display, enabling rearranging and changing of the SpecC source code. Since source code can be saved and profiled again, it is possible to see the impact of the design modification to the specification model parameters.

Profile results, combined with the information stored in the specification PE models (where a PE can be a standard processor or a custom hardware), is used to predict the performance. *Performance prediction* is significant function of the profiler front-end, and is separately presented in details in Section 4. In addition, in Section 4 we describe how to built specification PE models.

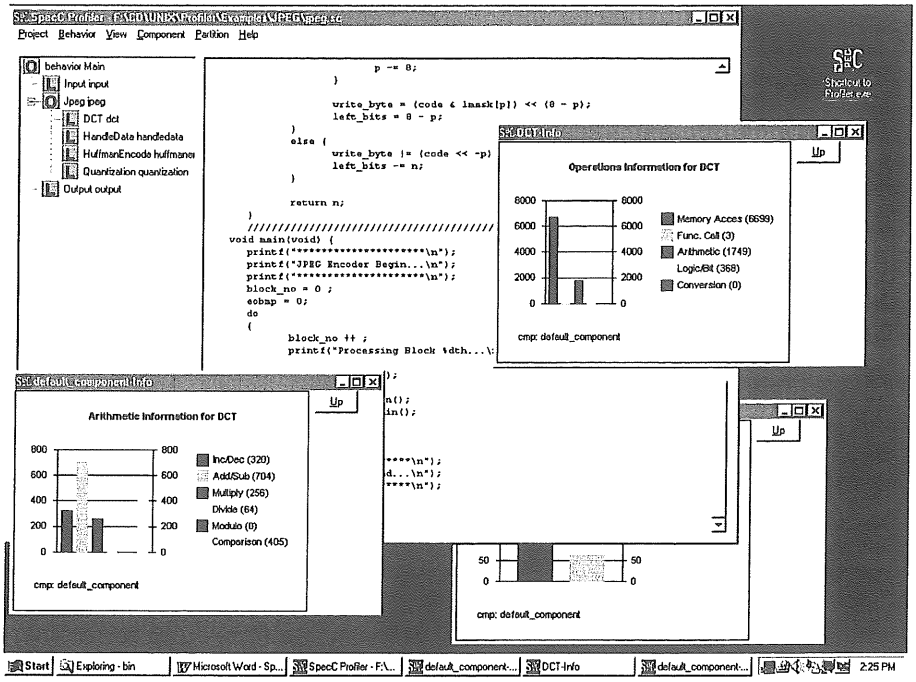



**Figure 4:** Profiler working environment

## 3 Profile results extraction and calculation

The profile results, which are output of the profiler back-end, are organized as a two-dimensional table. These tables store profiling information, such as: average number of operations, I/O traffic, and storage requirements. For each behavior, profiler back-end generates separate table. Table 1 shows the example of the table with profile results, and since these tables are large (56 rows and 29 columns), only small part of the table is shown.

The calculation of *the average number of operations* is based on the following information: statistical data collected during functional simulation and results of static analysis of the SpecC specification model. Static analysis performs control flow analysis of the high-level intermediate SpecC code that is stored as syntax tree [1].

In order to calculate the average number of operations executed by the behavior, we take into the account all functions executed by the behavior, as well as all of its subbehaviors. Since the average number of operations of a *compound behavior* depends on the average

number of operations of all its subbehaviors, we start the process of calculation of the average number of operations with *leaf behaviors*. The leaf behavior is not compound, which means that it does not contain any subbehavior. The process of calculation continues with those compound behaviors for which we already calculated the average number of operations for all subbehaviors.

Since the functionality of a behavior is defined by its main function, the average number of operations executed by the behavior is defined to be equal to the average number of operations of its main function, including operations in all other called functions and all instantiationed subbehaviors. Therefore, to calculate the average number of operations executed by the behavior, first we must calculate the average number of operations executed by each function. However, functions can be recursive, so the average number of operations of the functions is calculated by solving the following system of liner equations:

$$A = C \times A + O \qquad (1)$$

$A$ is $n$-dimensional vector, where $a_i$ is the average number of operations executed by the function $i$ and $n$ is the number of functions in the behavior, including the main function. $C$ is a square matrix, where $c_{i,j}$ denotes how many times function $i$ called function $j$. $O$ is $n$-dimensional vector. For the function $j$ in the leaf behavior, the element $o_j$ is calculated as:

$$o_j = ( \Sigma NoBl * ExBl ) / ExFn \qquad (2)$$

while for the function $j$ in the compound behaviors is calculated as:

$$o_j = ( \Sigma NoBl * ExBl ) / ExFn + ( \Sigma NoSB * ExSB ) / ExFn, \qquad (3)$$

where *NoBl* is the number of operations in a basic block, *ExBl* is the number of basic block executions, *NoSB* is the average number of operations in a subbehavior, *ExSB* is the number of subbehavior executions, and *ExFn* is the number of function executions. The summation for a given function $j$ include all basic block of the function, as well as all calls to subbehaviors.

**Table 1:** Profile results example

| Operations | Data types: int | long | float |
|---|---|---|---|
| IDN | 2 | 0 | 0 |
| CON | 1 | 0 | 0 |
| ADD | 1 | 0 | 0 |
| ASN | 1 | 0 | 0 |

The only unknown in the system of liner equations (1) is the matrix $A$. The values of all other parameters are available either from simulation (matrix $C$, *ExBl*, *ExFn*, and *ExSB*), from static analysis of the specification model (*NoBl*), or from previous calculation steps (*NoSB*). Therefore, by solving the system of liner equations, we calculate the average number of operations for all functions. The average number of operations executed by the behavior is then set to be equal to average number of operations of the main function.

*I/O traffic* of the behavior is calculated by solving the linear system of equations similar to (1). However, in this case matrix $A$ represents average I/O traffic of the behavior, *NoBl* is the number of accesses to the port variable in a basic block and *NoSB* is the average I/O traffic in a subbehavior.

*Storage requirement* consists of the memory required for storing behavior code, static memory, dynamic memory, stack requirements and port buffers. Storage required for storing static memory and port buffers is obtained by scanning of the source code, while behavior code requirements is calculated by simulating the target code generation process. Stack requirements are calculated by using the information collected during the simulation and by building the procedure call tree. Dynamic memory requirements are also calculated by using the information collected during the simulation and by regenerating the information of run-time dynamic allocation process.

**Table 2:** Spec. PE model

| Operations | Data types: int | long | float |
|---|---|---|---|
| IDN | 1 | 2 | 4 |
| CON | 1 | 2 | 4 |
| ADD | 1 | 2 | 4 |
| ASN | 0 | 0 | 0 |

We give the profile results for storage requirements in term of data types, like integer, long, double, etc. When designer choose one of the specification PE models in order to see the performance prediction results, profiler front-end converts these types into the number of bytes according to the given architecture. The weights that convert data types into the number of bytes are part of the specification PE models, which is in detail described in next section.

## 4 Performance prediction

Performance prediction is one of the basic profiler front-end functions, as it is explained in Section 2.2. As a performance measure, designer can chose either execution time, number of executed instructions, or some other target PE architecture parameter. Performance is predicted based on profiling results generated by

the profiler back-end (see Table 1) and the specification PE models.

Specification PE models are built as the tables, where elements of the table correspond to elements of Table 1, which stores the profiling results. Table 2 presents the example of the specification PE model. Two similar methods are used to build the table, one used for standard processors, and the other one used for a custom hardware. For the standard processors, table is generated by the process similar to the process of target code generation. For custom hardware, the table is generated by the process similar to the process of synthesis of the control unit, or to the process of the synthesis of the register-transfer level (RTL) instructions.

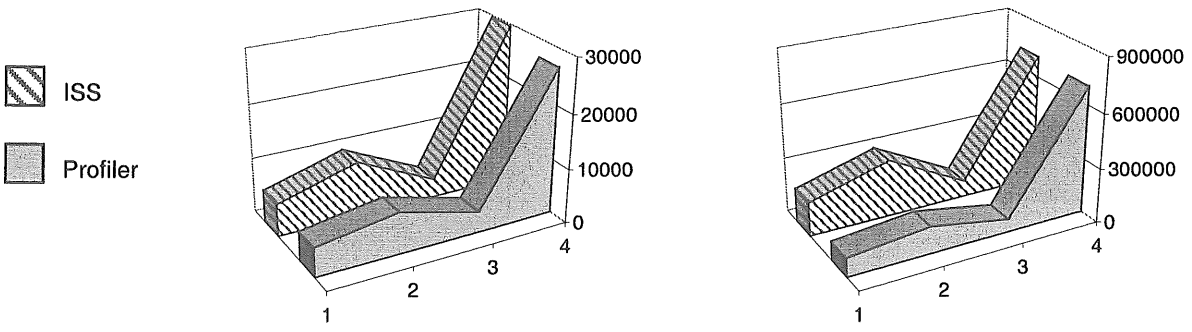
For example, for the SpecC source code: {int c; int a; c=a+12345;} profiler generates the profiling results presented in Table 1, while the compiler generates the following target machine code {MOVE a, R1; MOVE #12345, R1; ADD R1, R2; MOVE R2, c}. From these results, we conclude that each integer identifier and constant contribute with one MOVE instruction into the target code, each integer addition contributes with one ADD instruction, and there is no contribution from assignment. According to this discussion, we put the appropriate weights into the Table 2 that reflect the compilation process from the high-level profiling information into the target machine code. Weights are determined either to represent processor cycles, execution time, number of target instructions, etc. As an example, in Table 2 we present weights for the number of processor cycles, assuming that operations with integers can be done in one cycle, with long integer in two cycles, and floating point numbers in four cycles.

By multiplying the appropriate elements of the Table 1 and Table 2, and then summing up all the calculated amounts, we calculate the performance according to the chosen weights.

## 5 Profiler evaluation

The profiler is evaluated by running design examples, and comparing the profiler's performance prediction with the clock accurate instruction set simulation [4-6]. In this paper, we present the comparison of the profiler's prediction for the JPEG example running on the MC56600 processor, with the prediction of the instruction set simulator for the same processor [4, 5].

JPEG is an image compression standard [4, 5], consisted of four major building blocks. Handle data fragmentizes picture into 8x8 pixels blocks. Discrete cosine transformation (DCT) transfers blocks into frequency domain. The DCT output coefficients are then quantified in a




(*a*) Average number of instructions (*b*) Average execution time [ns]

**Figure 5:** Comparison of profiler prediction and instruction set simulator (ISS) results for JPEG example using MC56600 processor

(JPEG behaviors: 1 – Huffman encode, 2 – Handle data, 3 – Quantization, 4 - DCT)

quantization block, and finally encoded in Huffman Encode block. Each previously mentioned JPEG building block is mapped to one SpecC behavior, and comparisons for those four behaviors are made in Figure 5.

For each behavior two comparisons are made, one comparing the prediction of the number of instructions, and second comparing the execution time prediction. As can be seen in the Figure 5, profiler accurately predicts the relatively performance for both the number of instructions and execution time for all four behaviors. It accurately determines which one of the given two behaviors is more computationally intensive. For absolute performance, it can be noticed that for all behaviors the prediction is within 10% of the actual result, except for the execution time prediction for the Handle Data behavior. The Handle Data behavior performs large number of system calls (for example print calls) that are not currently included in the profile results.

## 6 Conclusion

We implemented the profiler as heterogeneous distributed system running on both UNIX and Windows operating systems. Currently, profiler is used and tested by the Center for Embedded Computer Systems, University of California, Irvine, USA, and School of EE and Computing, University of Zagreb, Croatia. Profiler back-end is written in C++, and we prepared executable codes by using gcc compiler for both UNIX and Windows environment. Profiler front-end is written in Visual Basic (display, editor, and performance prediction), while run-time support

is written in Visual C++. Run-time support enables communication and synchronization of the back-end and the front-end when they are running on Windows operating system. We generated the executable code for front-end only for Windows system.

Our first experiance during the testing of the SpecC profiler shows significant improvement in the productivity of system-level design process by using profiling results and confirms accuracy of the performance prediction results. By increasing the database of the available target specification PE models (where a PE can be a standard processor or custom hardware), we also expect that the quality of design will be significantly improved.

Currently we are implementing additional functions in the profiler back-end like the prediction of the point to point communication, size of dynamically allocated memory, stack size, and size of the memory for source code. We are enhancing the profiler front-end functionality by adding more features like performance prediction for heterogeneous target systems, parallel/sequential exploration feature, the worst case analysis feature, and target system synchronization exploration feature.

## 7 Acknowledgments

We would like to thank Professor Daniel D. Gajski for inviting as to work on the SpecC profiler project. We would also like to thank Andreas Gerstlauer, as well all all other members of the Center for Embedded Computer Systems, University of California, Irvine, for provided help and valuable advises.

## 8 References

[1] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, SpecC: Specification Language and Methodology, Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7822-9, March 2000

[2] Jianwen Zhu, Rainer Doemer and Daniel D. Gajski, "Syntax and Semantics of the SpecC Language," Proceedings of the Synthesis and System Integration of Mixed Technologies 1997, Osaka, Japan, December 1997.

[3] Rainer Dömer, Jianwen Zhu, Daniel D. Gajski, "The SpecC Language Reference Manual," UC Irvine, Technical Report ICS-TR-98-13, March 1998.

[4] Hanyu Yin, Haito Du, Tzu-Chia Lee, Daniel D. Gajski, "Design of a JPEG Encoder using SpecC Methodology," UC Irvine, Technical Report ICS-TR-00-23, July, 2000.

[5] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao and D. Gajski, "Design of a JPEG Encoding System," UC Irvine, Technical Report ICS-TR-99-54, November 1999.

[6] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski and Arkady M. Horak, "Design of a GSM Vocoder using SpecC Methodology," UC Irvine, Technical Report ICS-TR-99-11, March 1999.

# A Installation instructions

## A.1 Installation on WindowsNT

1) Double click on the Setup.exe

2) Follow instructions displayed by the installation. Remember name of the folder in which you installed the profiler.

3) Add to the PATH environment variable subdirectory "`bin`" which is located in the directory that contains installed profiler. For example, if you installed profiler into "`c:\SpecC\profiler`" directory, you should add "`c:\SpecC\profiler\bin`" to your `PATH` environment variable.

IMPORTANT: Do not use folders which names contain " " (space), the profiler won't work properly in such cases.

## A.2 Installation on UNIX

1) Unpack the binary distribution archive into an empty directory called `SpecCProf` (or select a similar name)

```
=> mkdir SpecCProf
=> cd SpecCProf
=> gtar xvzf Profiler_V101_BIN.tar.gz
```

2) Put the directory containing profile executable in the `PATH`

# B Front-end user reference manual

## B.1 Introduction

Graphical user interface (profiler client) for the SpecC profiler has been developed in order to simplify the process of the spec profiling. Client as input uses a text file with the description of the spec profile. This file is created by the profiler when -g switch is enabled:

```
=>profiler -p -g example
```

Beside the spec profile file, client uses files with descriptions of different processing elements.

## B.2 Menus

### *B.2.1 Project*

Open
: Opens an existing spec profile file. Spec profile files end with `"_gui.txt"` suffix.

Compile/Profile
: If you have SpecC system installed on you Windows machine, you can use this option to recompile the project and get new profiling results.

Arguments
: If you are using Compile/Profile option, use this to specify the arguments that will be passed as input arguments for the simulation.

Telnet
: Starts telnet application. Telnet application is used as a command window for the UNIX based systems.

Exit
: Exit profiler GUI.

### *B.2.2 Behavior*

Name
: Shows names of the behaviors in the behaviors view.

Operations
: Shows number of operations for each behavior in the behaviors view.

Traffic
: Shows traffic for each behavior in the behaviors view.

Storage

Shows storage required for each behavior in the behaviors view.

Partitions

Shows partitioning of the behaviors in the behaviors view.

### *B.2.3 View*

SpecC source

Enables you to see and modify the source code of the project. If you want to use this feature `_gui.txt` file must be located in the directory which contains source files of your project.

Hierarchy (disabled)

Will enable you to see the behavior hierarchy picture.

Communication (disabled)

Will enable you to see picture of the communication paths.

Save source

Saves all modifications made on the source code.

### *B.2.4 Component*

Open

Opens file with the PE component description. Component description files end with `"_cmp.txt"` suffix.

Select

Opens dialog for selecting the active PE component.

### *B.2.5 Partition*

Save

Saves partition information to the file.

### *B.2.6 Help*

About

Displays about box.

## B.3 How to use the profiler front-end

### *B.3.1 Profiling on WindowsNT*

We have prepared two examples (`HelloWorld, TLC`) which will enable you to start using profiler much faster and easier:

1) Start the profiler client by DOUBLE CLICK on the PROFILER CLIENT ICON.

2) To start one of the prepared examples, choose OPEN from the PROJECT menu.

3) Open dialog box will appear and you have to choose the EXAMPLES directory. Within EXAMPLES directory choose either HELLOWORLD or TLC directory.

4) Within chosen directory open appropriate `_gui.txt` file (`HelloWorld_gui.txt` or `TLC_gui.txt`).

5) After opening `_gui.txt` file, left part of the window, behavior view, shows Main behavior, while right part of window, source code view, shows the source code of the Main behavior.

6) CLICK to the + LEFT of the Main behavior to see all SUBBEHAVIORS of the Main behavior.

7) CLICK on the NAME of the behavior to see the SOURCE CODE of the chosen behavior in the source code view.

8) To see names, number of operations, traffic, storage or partitions for the behaviors, choose one of the options from the BEHAVIOR menu NAME, OPERATIONS, TRAFFIC, STORAGE or PARTITIONS, respectively.

9) On the RIGHT CLICK to any behavior, a pop-up menu with the following options appears: GENERAL, OPERATIONS, TRAFFIC, STORAGE and PARTITIONS.

10) By choosing GENERAL option a new window with bar chart is displayed. Chart shows number of operations, traffic and storage groups information of the chosen behavior.

11) Double click on any group displays chart with the informations for that group.
Groups are ordered hierarchically:

| Operations | Traffic | Storage |
|---|---|---|
| Memory Access | Input | Static |
| Constant | Bit | Bit |
| Identifier | Event | Event |
| Array Access | Int | Int |
| Other | Char/Bool | Char/Bool |

Function Call
Arithmetic
  Inc/Dec
  Sub/Add
  Multiply
  Divide
  Modulo
  Comparison
Logical/Bit
  Bitslice
  Pos/Neg
  Not
  Or
  And
  Eor
  Shift
Conversion
  Short
  Int/Long
  LongLong
  Enum
  Pointer
  Float
  Single
  Double
  LongFloat
Output
  *same as Input
  Short
  Int/Long
  LongLong
  Enum
  Pointer
  Float
  Single
  Double
  LongFloat
Dynamic
  *same as Static
Stack
  *same as Static
Ports
  *same as Static

UP button shows upper level in the hierarchy.

12) To load a component choose OPEN from the COMPONENT menu.

13) Open dialog box will appear and you have to choose the COMPONENTS directory. Within. COMPONENTS directory double click on one of the component files:

```
Motorola_No_Instructions_cmp.txt
Motorola_Time_cmp.txt
CustomHW_time_cmp.txt
```

14) After opening the component select the active component by choosing SELECT option from the COMPONENT menu. After selecting desired component click on OK button. Selected component becomes active and profiling results for that component are shown for all behaviors.

15) Changing of the source code of the behaviors is possible in the source view. Choose SAVE SOURCE from the VIEW menu to save modifications.

16) Choose COMPILE/PROFILE from the PROJECT menu to obtain profiling results for the modified source code. Progress dialog appears on the screen showing progress of the compilation and profiling process. When profiling is complete, behavior view is updated with new profiling results.

17) Use actions described in 6 - 16 to analyze new profiling results.

*B.3.2 Profiling on UNIX*

We have prepared four examples for UNIX platform: `HelloWorld`, `TLC`, `Vocoder` and `JPEG`. Before profiling on the UNIX map UNIX drive with examples to your local NT machine. Follow steps described in 3.1. Following steps are different:

3) Open dialog box will appear and you have to choose the EXAMPLES directory from the mapped UNIX drive. Within EXAMPLES directory choose HELLOWORLD, TLC, VOCODER or JPEG directory.

4) Within chosen directory open appropriate `_gui.txt` file (`HelloWorld_gui.txt`, `TLC_gui.txt`, `TestBench_gui.txt` or `Tb_gui.txt`).

16) To obtain results for the modified source code choose TELNET from the PROJECT menu. This brings up the telnet window.

17) Log on to the UNIX machine with installed SpecC system.

18) Go to the EXAMPLES directory on the mapped UNIX drive and enter the directory that contains modified files.

19) Start make:

```
=>make
```

This will recompile and simulate the project, and also calculate new profiling results.

20) To load `_gui.txt` files with new profiling results choose OPEN from the PROJECT menu and repeat steps 3 and 4.

*B.3.3 Profiling of your own projects*

If you want to use profiler GUI in profiling your own projects put them into existing EXAPMLES directory:

1) Create new directory in the EXAMPLES directory

```
=>mkdir My_project
```

2) Create SIR file for your project.

3) To prepare SIR file for profiling execute the command:

```
=>profiler -i My_project_SIR_FileName
```

4) Compile instrumented SIR file into an executable file:

```
=>scc My_project_SIR_FileName_ins -sir2out
```

5) Execute instrumented executable with any number of test vectors.

6) To calculate profile for the project and create `_gui.txt` file execute the command:

```
=>profile -p -g My_project_SIR_FileName
```

If you are using WindowsNT operating system follow instructions described in 3.1. In this case, if you want to use Compile/Profile feature you have to provide a makefile in the directory containing the source files and name it make_My_project_SIR_FileName.

If you are using UNIX operating system follow instructions described in 3.2.