Center for Embedded Computer Systems
University of California, Irvine

Assessment of Productivity Gains Achieved through Automated Source Re-Coding

Pramod Chandraiah and Rainer Dömer

Technical Report CECS-08-02
February 15, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
pramodc@cecs.uci.edu, doemer@cecs.uci.edu
http://www.cecs.uci.edu/

Assessment of Productivity Gains Achieved through Automated Source Re-Coding

Pramod Chandraiah and Rainer Dömer

pramodc@cecs.uci.edu, doemer@cecs.uci.edu

Technical Report CECS-08-02

February 15, 2008

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697, USA

Abstract

The input SoC specification plays a vital role in determining the quality of end implementation. Creating a SoC specification acceptable to the synthesis and refinement tools is immensely time-consuming and often this task dominates the time taken by the overall synthesis process. To overcome this bottleneck in the synthesis design flow, we have proposed a source re-coder. Our Source re-coder integrates manual specification programming with interactive automation. By replacing textual re-coding with automatic code transformations, our source re-coder makes it possible to create a SoC specification in significant shorter time.

In this report, we assess the productivity gains that can be achieved using our source re-coder. We have conducted an experiment on a class of students. The students were asked to provide the times needed to manually implement some important code transformations, and also the automatic times needed to implement the same transformations using our source re-coder. Based on the data collected from the students, we analyze and assess the productivity gains that can be achieved.

This technical report documents our experiments, analyzes the results, and provides some insights on potential productivity gains achievable through our source recoder approach. We conclude that our source re-coder is very effective and time efficient in re-coding SoC models and that productivity gain of multiple orders of magnitude are possible by use of automated recoding. We also extract some empirical quantities, such as the number of lines coded per designer hour, which can serve as reference to estimate manual and automatic coding times for future experiments.

Contents

List of Figures

Assessment of Productivity Gains Achieved through Automated Source Re-Coding

Pramod Chandraiah and Rainer Dömer

pramodc@cecs.uci.edu, doemer@cecs.uci.edu

Center for Embedded Computer Systems

University of California Irvine

Abstract

The input SoC specification plays a vital role in determining the quality of end implementation. Creating a SoC specification acceptable to the synthesis and refinement tools is immensely time-consuming and often this task dominates the time taken by the overall synthesis process. To overcome this bottleneck in the synthesis design flow, we have proposed a source re-coder. Our Source re-coder integrates manual specification programming with interactive automation. By replacing textual re-coding with automatic code transformations, our source re-coder makes it possible to create a SoC specification in significant shorter time.

In this report, we assess the productivity gains that can be achieved using our source re-coder. We have conducted an experiment on a class of students. The students were asked to provide the times needed to manually implement some important code transformations, and also the automatic times needed to implement the same transformations using our source re-coder. Based on the data collected from the students, we analyze and assess the productivity gains that can be achieved.

This technical report documents our experiments, analyzes the results, and provides some insights on potential productivity gains achievable through our source recoder approach. We conclude that our source re-coder is very effective and time efficient in re-coding SoC models and that productivity gain of multiple orders of magnitude are possible by use of automated recoding. We also extract some empirical quantities, such as the number of lines coded per designer hour, which can serve as reference to estimate manual and automatic coding times for future experiments.

1 Introduction

Motivated by the need to meet the time to market and aggressive design goals like low power, high performance and low cost, researchers have proposed various methodologies for effective design development, including top-down and bottom-up approaches. All these technological advances have significantly reduced the development time of embedded systems. However, design time is still a bottleneck in the production of systems, and further reduction through automation is necessary.

One critical aspect neglected in optimization efforts so far is the design specification phase, where the intended design is captured and modeled for use in the design flow.

Design flows today assume the availability of a high-quality specification, requiring the designer to manually create this specification. Today's design flows do not take advantage of the availability of reference models of application which can used to create a suitable quality specification in a System Level Design Languages (SLDL).

In our research, we address this problem of creating the SoC specification. By combining the manual coding with controlled automation, our re-coding approach aids in faster creation of a quality SoC specification.

To aid the designer in coding and re-coding, we have proposed a source re-coder. Our source re-coder is a controlled, interactive approach to implement analysis and code transformation tasks. Some of the transformations supported by source re-coder have been discussed in [1, 2, 4, 5]. The details of the source re-coder itself are presented in [3]. One of the main advantages of the source re-coder are the gains in the designer productivity due to the effective automation (compared to manual programming).

In our previous articles, the gains reported were based on the experiments conducted by a single experienced designer. In this report, we present the experiments and results conducted by a class of 15 students using source re-coder. These results not only corroborate our previous claim of significant productivity gains, but also show the need for automatic programming tools like our source re-coder.

1.1 Source Re-Coder

Our source re-coder is a controlled, interactive approach to implement analysis and transformation tasks. In other words, it is an intelligent union of editor, compiler, and powerful transformation and analysis tools. The conceptual organization of the source re-coder is shown in [3]. Unlike other program transformation tools, our approach provides complete control to generate and modify a specification model suitable for the design flow. By making the re-coding process interactive, we rely on the designer to concur, augment or overrule the analysis results of the tool, and use the combined intelligence of the re-coder and the designer for the modeling task. Our re-coder supports re-modeling of SLDL models at all levels of abstraction.

It consists of 5 main components:

- A textual editor (based on QT and Scintilla) maintaining the textual document object
- An Abstract Syntax Tree (AST) of the design model
- Preprocessor and Parser to convert the document object into AST
- Transformation and analysis tool set
- Code generator to apply changes in the AST to the document object

The parser and the code generator support C and SpecC source code. The analysis results of each transformation are remembered in the abstract syntax tree and get carried to the subsequent transformations automatically. The transformations are performed and presented to the designer instantly. The designer can also make changes to the code by typing and these changes are applied on-the-fly, keeping it updated all the time. More details of this interactive environment are discussed in [3].

2 Experiments

In the past, we have measured the productivity gains achieved using source re-coder by comparing the times taken by a single experienced designer to implement certain transformations manually, over times to implement the same transformations on the same examples using source re-coder. To get more diverse and realistic results, we conducted experiments on a set of students instead of a single experienced designer.

2.1 Setup

A class of 15 students enrolled in the graduate course "System-on-Chip Description and Modeling" [6] offered in the Department of Electrical Engineering and Computer Science at University of California Irvine, were given a MP3 audio decoder application in SpecC SLDL [7]. As an assignment, the students were asked to implement 3 kinds of transformations, both manually and automatically using our source recoder. We focused on creating behaviors [5], and recoding pointers [1]. These transformations are related in the sense that they are necessary in creating analyzable SoC models with definite structure which is necessary for architecture exploration.

The experiments were conducted over 4 weeks and were split into three assignments. In the first two assignments, the transformations were conducted manually, and in the third assignments, the same transformations on the same example were conducted using the source re-coder.

In the following sections, we will describe the experiments in detail and summarize the results reported by the students.

2.2 Experiment 1

In the first experiment, the students were given the source code of a MP3 audio decoder in SpecC language and were asked to convert two function calls into behaviors. For the first behavior, the designers were given detailed instructions to implement the transformation. For the second behavior, only brief instructions were provided. The detailed instructions given to the students are listed in Appendix-A1.

Since the main idea behind this assignment was to measure the manual time needed to implement the transformation, the students were asked to provide the time to correctly implement the transformations. The complete timing data provided by the students is given in Table 11 in Appendix-A2.

2.3 Experiment 2

In this part of the experiment, the students were given the source code of a MP3 audio decoder in SpecC language and were asked to implement two types of transformations.

- To wrap two sets of C statements into behaviors.

- To perform pointer recoding on four pointers

For creating the first behavior, the designers were given detailed instructions to implement the transformation. For the second behavior only brief instructions were provided. Similarly, the procedure to recode one pointer was explained in detail and brief instructions were provided to recode the three other pointers. The detailed assignment description is given in Appendix-A3.

Since the main idea behind this assignment was to measure the manual time needed to implement the transformation, the students were asked to provide the time to correctly implement these transformations. The complete timing data provided by the designers is given in Table 12 and Table 13 in Appendix-A4.

2.4 Experiment 3

After completion of the two manual assignments, the source re-coder was introduced to the students. The students were asked to implement the same transformations (previously implemented manually) using the source re-coder. At the end of the experiment, the students provided the time taken to implement these transformations using the source re-coder.

The detailed assignment description is given in Appendix-A5. The times reported for this experiment are given in Table 14 and Table 15 in Appendix-A6.

3 Comparison and Analysis

The detailed results provided by the students for each experiment are listed in the tables Table 11, Table 12, Table 13, Table 14, and Table 15 (in the appendix). In this section, we will summarize those results and compare the manual times from Experiments 1 and 2 with the automatic times obtained from the Experiment 3.

3.1 Function to Behavior (F2B) Recoding

The comparison of manual and the automatic times for two function-to-behavior transformations is reported in Table 1 for 15 students. Clearly, the manual times for implementing these transformations varied widely across designers from 3 hrs 50 mins (student 9) to 26 mins (student 14). The average manual time across 15 students was 1 hr and 17 mins. On the other hand, using the source re-coder, the students were able to implement the transformations rather quickly. The automatic times varied from 15 min (student 8) to 1 min (student 3). The average automatic time was 5 mins. The gain in productivity across different students (Figure 1) varied from 57.5 (student 9) to 3.7 (student 14) with an average gain of factor 18.9. Though it just takes a couple of clicks in the source re-coder to realize these transformations, it still took minutes for many students as they had to familiarize themselves with the tool and simultaneously read the instructions provided. We believe, as the designer gets comfortable with the editor and the tools in the re-coder, automatic transformations can be realized even faster. Comparing the average manual time (1 hr 17 mins) with the fastest automatic time (1 min), the gain that can be potentially achieved is about two orders of magnitude (factor 77).

Student	Manual h:min	Automatic h:min	Gain
1	1:02	0:11	5.6
2	1:39	0:09	11.0
3	0:49	0:01	49.0
4	1:00	0:04	15.0
5	1:21	0:04	20.3
6	0:55	0:04	13.6
7	0:58	0:05	11.6
8	1:26	0:15	5.7
9	3:50	0:04	57.5
10	1:19	0:04	19.8
11	0:32	0:02	16.0
12	1:02	0:03	20. 7
13	1:21	0:05	16.2
14	0:26	0:07	3.7
15	1:37	n/a	n/a
Average	1:17	0:05	18.9
Std.Dev	0.033	0.003	15.6
Max	3:50	0:15	57.5
Min	0:26	0:01	3.7

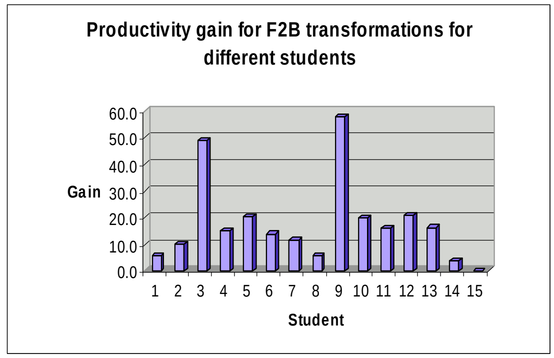


Figure 1: Plot of Gains for different students

Table 1: Comparison of manual and automatic times for re-coding 2 functions into behaviors (F2B transformation)

3.2 Analysis of F2B Transformation

Table 1 shows the time for 2 Function-to-Behavior transformations. From this table and Table 11 (Appendix-A2), the following derivations can be made. The Table 2 lists the minimum, average, and maximum values observed for different quantities. Besides the observed quantity, the potential maximum gain is obtained by comparing the average manual time observed (1:17) to the fastest automatic time (1 min), which evaluates to factor 77.

Quantities	Minimum Observed	Average Observed	Maximum Observed	Potential
Manual time for 1 F2B	0:26	1:17	3:50	--
Automatic time for 1 F2B	0:01	0:05	0:15	--
Gain	3.7	18.9	57	77

Table 2: Analysis of F2B operations

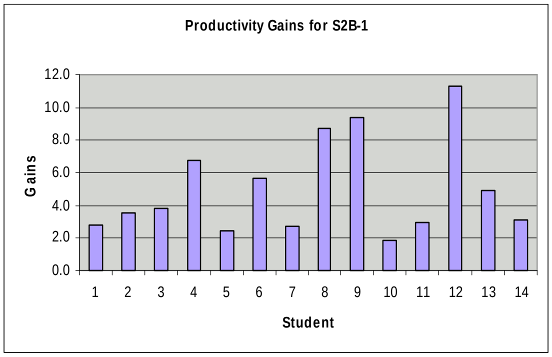
3.3 Statement to Behavior (S2B) Recoding

The comparison of manual and automatic times for 2 statement-to-behavior transformations is reported in Table 3 for 15 students. The data for each transformation is reported separately. For the first transformation, where detailed instructions were provided, the manual times varied across designers from 1 hr 18 mins (student 8) to 17 mins (student 10). The average manual time across 15 students was 41 mins. For the second transformation, where only brief instructions were provided, the manual times varied from 3hrs 30 mins to 20 mins with an average time of 1 hr and 7 mins.

On the other hand, using the re-coder, designers were able to implement the transformations quickly with times varying between 21 mins down to 3 mins for the first transformation, and 1 hr 16 mins down to 2 mins for the second transformation. The automatic times were higher than expected as the designers had to deal with some cases of tool crashes. For example, student 14, who reported a time of 1 hr and 16 mins, took the tool crash into account. The maximum productivity gain of 60 was reported by student 14. We believe, as the tool stabilizes and the designer gets comfortable with the editor and the tools in the re-coder, automatic transformations can be realized even faster. Comparing the average manual time (1 hr 7 mins) and the fastest automatic time (2 min), the gain that can potentially achieved will be 67.

	Statement to Behavior -1			Statement to Behavior -2		
Student	**Manual-1 hr:min**	**Automatic-1 hr:min**	**Gain-1**	**Manual-2 hr:min**	**Automatic-2 hr:min**	**Gain-2**
1	0:33	0:12	2.8	0:23	0:16	1.4
2	0:45	0:12	3.5	0:35	0:13	2.5
3	0:19	0:05	3.8	0:44	0:04	11.0
4	0:34	0:05	6.8	1:00	0:04	15.0
5	0:51	0:21	2.4	0:38	0:15	2.5
6	0:45	0:08	5.6	0:35	0:13	2.7
7	0:24	0:09	2.7	0:33	0:10	3.3
8	1:18	0:09	8.7	1:28	0:10	8.8
9	0:47	0:05	9.4	2:00	0:02	60.0
10	0:17	0:09	1.9	1:09	0:05	13.8
11	0:21	0:07	3.0	0:20	0:14	1.4
12	0:34	0:03	11.3	1:00	0:05	12.0
13	0:39	0:08	4.9	1:10	0:06	11.7
14	0:59	0:19	3.1	1:51	n/a	n/a
15	1:16	n/a	n/a	3:30	n/a	n/a
Average	0:41	0:09	5.0	1:07	0:09	11.2
Std.Dev.	0.013	0.004	3.0	0.034	0.003	15.5
Max	1:18	0:21	11.3	3:30	0:16	60.0
Min	0:17	0:03	1.9	0:20	0:02	1.429

Table 3: Comparison of manual and automatic times of re-coding 2 sets of statements into behaviors (S2B transformation)



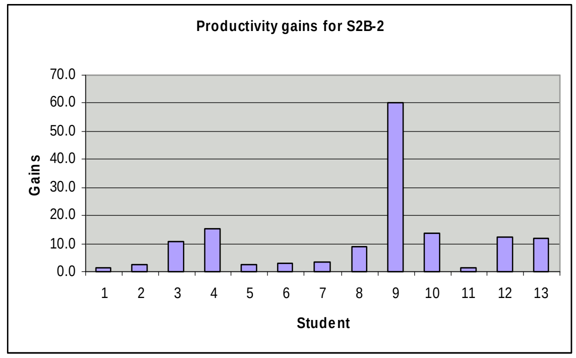


Figure 2 Plot of Gains for different students

3.4 Analysis of S2B operation

The above table shows the time for 2 Statement to Behavior transformations. From the above table and the Table 12 (Appendix-A4), the following experimental derivations can be made. Table 4 lists minimum, average and maximum values observed for different quantities.

Quantities	Minimum Observed	Average Observed	Maximum Observed
Manual time for 1 S2B	0:17	0:54	3:30
Automatic time for 1 S2B	0:02	0:09	0:21
Gain	1.4	8.1	60.0

Table 4: Analysis of S2B operations

3.5 Pointer Re-coding

The comparison of manual and automatic pointer re-coding times for 4 pointers is reported in Table 5 for 15 students. The manual times varied across designers from 1 hr 22 mins (student 4) to 23 mins (student 7). The average manual time across 15 students was 50 mins. However, the automatic pointer re-coding using source re-coder took less time, as expected. The automatic times varied from 15 mins down to 2 mins. The gain varied between 16.4 and 3.4, and the average gain was 9.79.

Student	Manual	Automatic	Gain
1	0:51	0:15	3.4
2	1:11	0:12	5.6
3	0:33	0:05	6.6
4	1:22	0:05	16.4
5	n/a	0:12	n/a
6	1:07	0:06	11.2
7	0:23	0:05	4.6
8	1:12	0:07	10.3
9	n/a	0:05	n/a

10	0:44	0:03	14.7
11	0:29	0:02	14.5
12	0:37	0:03	12.3
13	0:53	0:05	10.6
14	0:44	0:06	7.3
15	n/a	n/a	n/a
Average	0:50	0:06	9.8
Std.Dev.	0.013	0.003	4.3
Max	1:22	0:15	16.4
Min	0:23	0:02	3.4

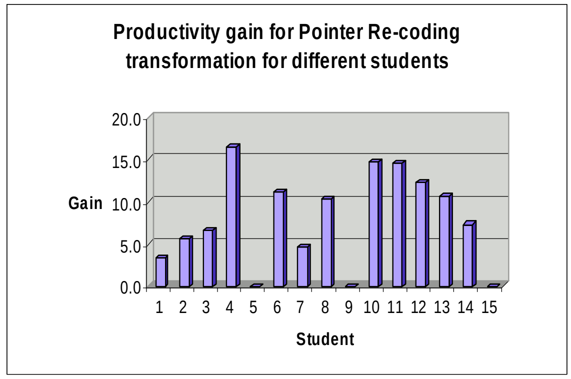


Figure 3 Plot of gains for different students

Table 5: Comparison of manual and automatic pointer re-coding times

Clearly, some of the students who could not complete the manual pointer re-coding were able to perform the recoding using source re-coder.

3.6 Analysis of Pointer Re-coding operation

The above table shows the time for 4 pointer re-coding transformations. From the above table and Table 13 (Appendix-A4), the following experimental derivations can be made. The table lists Minimum, average and maximum values observed for different quantities.

Quantities	Min. Observed	Avg. Observed	Max. Observed
Manual time for 1 PR	0:23	0:12	1:22
Automatic time for 1 PR	0:02	0:06	0:15
Gain	3.4	9.8	16.4

Table 6: Analysis of Pointer recoding operations

Comparing the average manual time of 50 mins with the fastest automatic time (2 min), the gain that can potentially be achieved will be 25. Note that the potential gain is low as in our experiment the instructions included the source for the recoded pointers, In reality, the designer will have to determine the source of pointers manually by reading the code. Thus, the manual times will be much higher, as will be the productivity gain.

3.7 Additional Feedback

Besides the timing details, the students also provided suggestions to improve the tools. The task of converting a function to a behavior and statements to a behavior first required re-scoping some local variables to the class scope so that they become available for port-mapping. The transformation to do this operation was also available in the re-coder, but required explicit invocation by the students for every variable that required re-scoping. Designers had to look at the variables and check if they are local and they re-scope them. Based on the suggestions provided by the students, this re-scoping of variables was made part of the function-to-behavior and statement-to-behavior transformations. These transformations were modified so that all the variables that are needed for port-mapping are automatically moved into the class scope.

This improvement is shown in Table 7 below. The number of interactions earlier depended on the number of variables that need to be re-scoped. These were changed to just 1 interaction based on the feedback received.

Further, some system crashes reported by the designers helped to fix a couple of implementation issues in the tool. We are confident that, after these changes to the source re-coder, the productivity gains will be much higher.

Tool in Re-coder	**Number of interactions**	
	Before this experiment	After incorporating designer suggestions
Function to Behavior	1 + (n * rescope variables)	1
Statement to Behavior	1 + (n * rescope variables)	1
Pointer Analysis	1	1
Pointer Re-coding	1	1

Table 7: Reduced number of interactions needed to invoke different tools after incorporating student feedback

4 Generalization for Future Estimation

Conducting these types of experiments is very expensive in terms of time and resources. Therefore, we attempt to generalize our observations and experimental results. We derived some more empirical results which we may use in future for estimating manual and automatic times.

Irrespective of the type of transformation, the most primitive empirical result needed to estimate manual programming time would be the number of Lines of Code (LoC) generated per hour. Based on the 3 types of transformations implemented by the students in Experiment 1 and 2, we obtained the number of lines of code that changed. Using the manual times provided by the students for those transformations, we estimated the LoC written per hour.

Using the minimum, average, and the maximum values of the manual times, we computed 3 values of LoC per hour. These results are tabulated in Table 8 below. Obviously, the variability in the manual times also reflects in the LoC written per hour. One should note that these numbers are quite optimistic as we consider only re-coding time, not the decision making time. The students were given almost line-by-line instructions to implement the code. If the students/designers have to code all by themselves, then the result will be much lower than these numbers. Moreover, these numbers do not account for errors introduced into the design, which would require tedious debugging and thereby drastically reduce the LoC written per hour further.

Task	LoC	Manual time (hr:min) Min. Manual time	Avg. Manual time	Max. Manual time	Comment
F2B-1	102	0:09	0:29	1:20	Experiment-1
F2B-2	214	0:15	0:47	2:30	Experiment-1
S2B-1	162	0:17	0:41	1:18	Experiment-2
S2B-2	158	0:20	1:07	3:30	Experiment-2
PR -1	70	0:10	0:21	1:03	Experiment-2
PR-2,3,4	112	0:13	0:28	0:58	Experiment-2
Total	818	1:24	3:56	10:39	
LoC per hour					
LoC per hour		584	208	77	Considers pure re-coding, no decision making

Table 8 Lines of Code per manual hour estimation

Similarly, the most primitive quantity to measure the automatic time using source recode is the number of interactions for each transformation. By restricting most of the transformations to just 1 user interaction, it becomes easier to estimate the automatic time. As described in Section 3.7, based on the student's feedback, we modified the transformations to restrict the number of interactions to just one. At the time of this experiment, since pointer recoding was the only transformation that had 1 interaction, we can take the minimum time for pointer recoding as an optimistic estimate for all transformations that require 1 interaction. The minimum time to recode a pointer using source re-coder is 2 mins from Table 6. Based on this argument, we can assume that the time for realizing a 1 interaction transformation using source re-coder is about 2 mins. Considering the variability, the 3 values (minimum, average, maximum) are given in Table 9 below.

Transformation type	Min. Automatic time (hr:min)	Avg. Automatic time(hr:min)	Max. Automatic time(hr:min)
One-interaction transformation	0:02	0:06	0:15

Table 9 Automatic time for 1 interaction transformation

5 Challenges in Measuring Productivity Gains

Conducting a real life experiment to measure the productivity gain achievable using a tool is a challenging task, as such experiments are limited by resource and time constraints. Some of the issues we encountered in our experiment are listed below:

1. The variations in the times provided by different designers make it hard to arrive at a common measure of manual time. Besides the variation in the manual time, the automatic times provided by the students also varied widely. This, we believe, can be attributed to some of the software crashes encountered by the users, and also the time it took to read the instructions from the assignment and get

acquainted with the recoder. These issues resulted in the variations in productivity gains. These variations are shown in Table 10 below.

2. Though the MP3 design example used for the experiment was a representative of commonly used embedded applications, the example transformations manually conducted by the students did not necessarily represent an average case of programming. Due to the time constraints, students could not be asked to conduct more manual transformations. This was one of the larger limitations of our experiment.
3. An Ideal experimental setup would be to have 2 groups of students conducting manual and automatic experiments simultaneously, and then compare the time taken by each group. However, due to the resource constraints such an experiment could not be conducted.
4. In our experiment, the students were asked to implement a fewer set of transformations on a bigger application. Another option would be to work on a smaller application, but implement more transformations to derive a complete specification model.
5. The students were asked to conduct the manual experiments first, and then used automatic recoding to implement the same transformations using the source re-coder. So the automatic part of the experiment was benefited by the knowledge of the MP3 code that was acquired during the manual experiment.
6. The learning curve that is achieved using the source re-coder would make the subsequent re-coding faster. However, this could not be accounted as students had very limited time available for this experiment.
7. The experiment was conducted with a class of graduate students and not regular designers.
8. Finally, we believe the productivity gains measured from our experiment are still conservative compared to what can be achieved in reality. This is because the errors made by the designers during manual programming are not taken into account in this experiment. In the absence of errors, the designers can direct all the effort and attention towards structuring the model instead of actually working on textual recoding. This improves the quality of the model and further increases the productivity gains.

Gains achieved by different tools	Minimum	Average	Maximum
F2B Gain	3.7	18.9	57.0
S2B Gain	1.4	8.1	60.0
PR Gain	3.4	9.8	16.4

Table 10: Variability in the gains

6 Conclusions

Tools like our source re-coder are intuitively able to help the designer in faster creation of a good SoC specification. Experiments to quantitatively measure the extent to which such a tool can be useful to designers of varying abilities were not conducted in the past.

With the help of a real class of 15 graduate students, we conducted experiments on creating a SoC specification for a real-life design example. The students were first given instructions to manually implement 3 kinds of re-coding tasks on an MP3 decoder specification, and were asked to measure the time taken to program. Following that, the same students were introduced to our automatic source re-coder, and were asked to implement the same transformations using the automatic tools available in the source re-coder.

Comparing the manual and the automatic times provided by different designers, we were able to estimate the gains that can be achieved using our interactive source re-coder. The gains achieved varied depending on the designer and the type of transformation. Some variability also resulted from the still immature source re-coder. Despite this variability, it was conclusive that our source re-coder results in significant productivity gains and effective help in reducing the overall system design time.

There were some aspects, which could not be accounted for in this experiment. For example, due to time and resource constraints, for manual implementation the designers were given line-by-line instructions to implement the manual transformation. However, in reality when designers themselves have to analyze and implement the code, it would take more time and errors before correctly realizing the transformations.

We derived certain empirical quantities such as, *Lines of code per hour* and *time for one interaction transformation*, which can serve as reference in estimating the productivity gains in future experiments.

In future, we would organize such experiments differently to even out some variables. One idea is to have one set of designers working manually, and another set of designers (of the same capability and quality) working automatically using source re-coder. If these two independent groups implement different transformations on a smaller design example and create a complete SoC model, we can compare the times taken these two groups and better estimate the productivity gains.

In summary, our Source re-coder relieves the designers from complex re-coding work and lets them think about structuring and creating parallel and analyzable models instead of worrying about implementing the transformations. Such automation will go a long way in helping designers in creating high quality specifications faster.

7 Acknowledgements

We very much thank the graduate students in class EECS-222A of Fall'07 for conducting the experiments. We also would like to thank Embedded Systems Methodology group at the Center for Embedded Systems (CECS) for fruitful discussions and providing valuable

feedback on the experiments, results and suggestions to improve such experiments in the future.

8 Reference

1. Pramod Chandraiah, Rainer Dömer: *"Pointer Re-coding for Creating Definitive MPSoC Models"*, Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, Salzburg, Austria, September 2007.
2. Pramod Chandraiah, Rainer Dömer: *"Designer-Controlled Generation of Parallel and Flexible Heterogeneous MPSoC Specification"*, Proceedings of the Design Automation Conference 2007, San Diego, California, June 2007.
3. Pramod Chandraiah, Rainer Dömer:*"An Interactive Model Re-Coder for Efficient SoC Specification"*, Proceedings of the International Embedded Systems Symposium, "Embedded System Design: Topics, Techniques and Trends" (ed. A. Rettberg, M. Zanella, R. Dömer, A. Gerstlauer, F. Rammig), Springer, Irvine, California, May 2007.
4. Pramod Chandraiah, Junyu Peng, Rainer Dömer: *"Creating Explicit Communication in SoC Models Using Interactive Re-Coding"*, Proceedings of the Asia and South Pacific Design Automation Conference 2007, Yokohama, Japan, January 2007.
5. Pramod Chandraiah, Rainer Dömer: *"Automatic Re-coding of Reference Code into Structured and Analyzable SoC Models"*, Proceedings of the Asia and South Pacific Design Automation Conference 2008, Seoul, South Korea, January 2008.
6. "System-on-Chip (SoC) Description and Modeling", Rainer Doemer, Lecture Notes for graduate-level course EECS 222A, Fall 2007. https://eee.uci.edu/07f/18430/
7. A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski: "System Design: A Practical Guide with SpecC". Kluwer Academic Publishers, 2001.

Appendix

A1. Student Instructions for Experiment 1

EECS 222A
System-on-Chip Description and Modeling
Fall 2007

Assignment 4

Posted: November 2, 2007 (week 6)
Due: November 9, 2007 (week 7)

Task: Creating Behaviors in C Code

Instructions: (by Pramod Chandraiah)

As you might know by this time, behaviors in SpecC act as a basic unit of computation. Because of their explicit syntax, compared to functions and plain C statements, the analysis and the refinement tools can easily analyze and conduct refinement tasks, such as partitioning and mapping.
In this document, we briefly describe how to encapsulate functions and C statements into behaviors, followed by precise directions to create behaviors in an MP3 decoder example.

Encapsulating Functions in Behaviors

```
behavior B ( in int p1, in int p2,  out int result)
{
   void main( )
    {
        int i1, a, b[10], s, *pa;
        a = p1+p2;
        s = p1-p2;
        pa = &s;
        .....
        result = f1(a, b[i1], pa);
        .....
    }
    int f1( int w, int x, int *p)
    { *p  = w+x+*p;
        return *p;
    }
};
```

(a) Original model (Model 1)

```
behavior  B_f1( in int w, in  int x[10], in int i, inout int s, out int c) {
     void main()
     {  c= func(w, x[i], &s);
     }
     int f1( int w, int x, int *p)
     { *p  = w+x+*p;
         return *p;
     }
 };
  behavior B (in int p1, in int p2, out int     result) {
    int a, b[10], i1, s;
   //Instantiate child behavior here
    I_B_f 1(a, b, i1, s, result);
    void main( )
    {
        int *pa;
        a = p1+p2;
        s = p1-p2;
        pa = &s;
        .....
       I_ B_f.main();
        ......
    }
 } ;
```

(b) Function encapsulated in behavior (Model 2)

The explicit port list that defines the interface of the behaviors is what makes behaviors easily analyzable. As shown in the figure above, encapsulating a function into a behavior involves creating a behavior body, creating the instance of the newly created behavior and replacing the function call with the call to the instance. Note that, creating the behavior body requires first determining the port list. Creating the instance requires first creating the port map list.

Figure 4: Page-1 of Student Instructions for Experiment 1

Initial Setup

The initial setup files are in the tar file /home/doemer/EECS222A_F07/`mp3_v1.tar.gz`.
Untar this file using the command:

```
gtar xvzf mp3_v1.tar.gz
```

A directory by name mp3_v1 will be created. Change into this directory

```
cd mp3_v1
```

The entire MP3 source code is in single file `mp3decoder.sc`. There is a `Makefile` to compile and test the decoder for a set of MP3 streams. There are two directories `reference/` and `testStream/` which you don't have to worry about at this stage.

You need to set the path for the SpecC compiler. Source the setup shell script as below:

```
source /opt/sce-20060301/bin/setup.csh
```

Now compile and test the decoder by running following commands

```
make clean
make
make test
```

The setup should compile and simulate without errors. Please just ignore any `"Can't step back"` and `"read length less than max"` messages.

Given MP3 Code

The MP3 code is a basic SpecC code. It has 4 behaviors: Stimulus [at line 5724], Monitor [at line 5692], DUT [at line 5640] and MP3Decoder [at line 2755]. Use text editor to view the source code and find these behaviors.

We want to introduce more behaviors in MP3Decoder to enable exploration. In the next section, we describe the changes you have to do to convert a function call into a behavior. Note that the line numbers we refer to are the unmodified lines in the original source file.

Task 1: Encapsulate `decodeMP3` function call in behavior `MP3Decoder`.

1. We will give a set of instructions to convert a function to behavior. We would also like to measure the time it takes to perform this conversion.
   Please make a note of the start time T0.
2. `decodeMP3()` is a function in behavior `MP3Decoder`. The global function is defined at line 2873 and is declared at 2724. There are 2 calls to this function, first call at line 2778 and second at line 2792. For this exercise, we are only interested in encapsulating the function call at line 2778.
3. First we have to create the behavior body. Create a new empty behavior (say at line 2754) with the signature:

```
behavior B_decodeMP3(
    in struct mpstr *mp,
    in char *input,
    in int isize,
    in char *output,
    in int osize,
    inout int *done,
    out int ret_port) {

};
```

   Note that this signature can be derived from the function signature. You could copy the function signature of decodeMP3 and modify it, or copy this signature from this document directly, or else just type it.

Figure 5: Page-2 of Student Instructions for Experiment 1

4. Copy the global function `decodeMP3()` into behavior B_decodeMP3 (retain the global function, don't delete it).
5. In the behavior `B_decodeMP3` create an empty main function `void main (void){}`
6. Add this function call in the empty main function
   `ret_port = decodeMP3(mp, input, isize, output, osize, done);`
7. Save the file and do: `make` and `make test` and check if the tests run fine.
8. **Note the time T1.**

9. Now we have to create the instance of this newly created behavior in the parent behavior. First, create new temporary variables in the scope of behavior MP3Decoder. We explain later the need for these variables.
```
struct mpstr *t_dummy;
char *t_dummy_0;
char *t_dummy_1;
int *t_dummy_2;
```

10. Create an instance of the new behavior in the parent behavior MP3Decoder before the main body.
    `B_decodeMP3 I_B_decodeMP3(t_dummy, t_dummy_0, len, t_dummy_1, (8192), t_dummy_2, ret);`
    The portmap needed for creating the instance is obtained from the function call. The function call is: `ret = decodeMP3( &mp, buf, len, output, 8192, &size);`
    Since the portmaps can only be variables (not expressions), temporary variables are used to store the argument expressions. So, before we make a call to the instance, we need to initialize these temporary variables with the argument expressions.
11. Replace the first call to function `decodeMP3()` in the behavior `MP3Decoder` with the call to the instance
    `I_B_decodeMP3.main()`
12. Add following assignment statements to initialize the temporary variables in the behavior `MP3Decoder` before the call to the instance.
```
t_dummy = &mp;
t_dummy_0 = buf;
t_dummy_1 = output;
t_dummy_2 = &size;
```
13. Save the file and do: `make` and `make test` and check if the tests run fine.
14. **Make a note of the time T2**

Task 2: Encapsulate `do_layer3` function call in behavior `B_decodeMP3`

1. For performing this task, unlike the previous case, we will not give the detailed directions. We will briefly outline the steps needed. You will use the source file resulting from the changes done in the previous step.
   Please make a note of the start time T3.
2. You have to encapsulate the function call `do_layer3()` which is located in the member function decodeMP3() which is a member of the behavior `B_decodeMP3`.
3. First create the behavior signature with an empty body. Use the behavior name `B_do_layer3`. To derive the signature you can use the function call signature of `do_layer3()`. To determine the direction of the ports (`IN, OUT, INOUT`) you have to analyze the program and determine the accesses. Since this is complicated you could skip this and not specify any port direction. By default these ports will be treated as `INOUT`.
4. Copy the do_layer3() function to the behavior body and add the function call to the `do_layer3()` function from the `main()`, just like the previous example.
5. Save the file and do `make` and `make test` and check if the tests run fine.
6. **Note the time T4.**

Figure 6: Page-3 of Student Instructions for Experiment 1

7. Now create the instance of the behavior `B_do_layer3` in the parent behavior `B_decodeMP3`. Use the temporary variables if you need for the port map. Replace the call to function `do_layer3` (which is located in the member function `B_decodeMP3::decodeMP3`) with the call to the instance. If you have used temporary variables, then introduce the necessary assignment statements before the instance call.
8. Save the file and do `make` and `make test` and check if the tests run fine.
9. **Make a note of the time T5**

Deliverables:

- 1-paragraph description about the two tasks above
  (i.e. how far you got, what were the problems, how did you solve it)
- Please also report the times T0, T1, T2, T3, T4 and T5 (in minutes).

Due: Week 7 (Nov 9, 2007)

--
Rainer Doemer (ET 444C, x4-9007, doemer@uci.edu)

Figure 7: Page-4 of Student Instructions for Experiment 1

A2. Times Reported by Students for Experiment 1

The results submitted by the students for the 2 tasks are given in Table 11. Time stamps T0 to T5 was the primary feedback expected from the students. However, some of the students only turned-in the durations between these stamps, which are as good. *Task 1* is the time to recode the function into behavior given the complete instructions. *Task 2* is the time to recode the 2nd function into behavior. *time (T0, T1), time(T1, T2)* constitute *Task1* and *time(T3, T4) time(T4, T5)* together constitute *Task2.*

Student	T0	T1	T2	T3	T4	T5	time(T0,T1)	time(T1,T2)	time(T3,T4)	time(T4,T5)	Task 1	Task 2	Total
1	9:30	9:40	9:55	10:13	10:30	10:50	0:10	0:15	0:17	0:20	0:25	0:37	1:02
2	1:40	1:48	2:02	2:25	3:11	3:42	0:08	0:14	0:46	0:31	0:22	1:17	1:39
3	17:35	17:40	17:44	17:46	17:52	18:26	0:05	0:04	0:06	0:34	0:09	0:40	0:49
4	8:24	8:35	8:50	9:03	9:13	9:37	0:11	0:15	0:10	0:24	0:26	0:34	1:00
5	10:14	10:21	10:50	11:07	11:24	11:52	0:07	0:29	0:17	0:28	0:36	0:45	1:21
6	8:20	8:30	8:45	8:50	9:15	9:20	0:10	0:15	0:25	0:05	0:25	0:30	0:55
7	9:20	9:30	9:43	9:45	9:53	10:20	0:10	0:13	0:08	0:27	0:23	0:35	0:58
8	n/a	n/a	n/a	n/a	n/a	n/a	0:30	0:15	0:18	0:23	0:45	0:41	1:26
9	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	1:20	2:30	3:50
10	10:23	10:43	11:06	11:07	11:28	11:43	0:20	0:23	0:21	0:15	0:43	0:36	1:19
11	n/a	n/a	n/a	n/a	n/a	n/a	0:10	0:05	0:10	0:07	0:15	0:17	0:32
12	0:00	0:17	0:28	0:00	0:18	0:34	0:17	0:11	0:18	0:16	0:28	0:34	1:02
13	2:59	3:09	3:34	8:59	9:40	9:45	0:10	0:25	0:41	0:05	0:35	0:46	1:21
14	10:36	10:45	10:47	11:01	11:06	11:16	0:09	0:02	0:05	0:10	0:11	0:15	0:26
15	8:56	9:05	9:15	9:23	10:02	10:41	0:09	0:09	0:39	0:39	0:19	1:18	1:37
Average							0:11	0:13	0:20	0:20	0:29	0:47	1:17
MAX							0:30	0:29	0:46	0:39	1:20	2:30	3:50
MIN							0:05	0:02	0:05	0:05	0:09	0:15	0:26

Table 11: Times reported by students to manually recode functions "decodeMP3" (Task1) and "do_layer3" (Task2) into behaviors

Note 1: The time stamps not provided by the students are indicated by "n/a". For example, student 8 and student 10 provided durations and not the time stamps. Student 9 only provided the times for *Task1* and *Task2.*

Note 2:

- *time(T0,T1)* is the time to create the behavior body including the portlist for 1st F2B transformation
- *time(T1,T2)* is the time to create the behavior instance including the portmap for 1st F2B transformation
- *time(T3,T4)* is the time to create the behavior body including the portlist for 1st F2B transformation

- *time(T4, T5)* is the time to create the behavior instance including the portmap for 1st F2B transformation

A3. Student Instructions for Experiment 2

EECS 222A
System-on-Chip Description and Modeling
Fall 2007

Assignment 5

Posted: November 9, 2007 (week 7)
Due: November 16, 2007 (week 8)

Tasks: Part 1: Creating Behaviors in C Code
Part 2: Pointer Elimination

Instructions: (by Pramod Chandraiah)

Part1 - Converting Statements to Behavior

In the previous assignment, we converted functions into behaviors. In this assignment, we will encapsulate a set of C statements into behaviors.

Encapsulating Statements in Behaviors

```
behavior B ( in int p1, in int p2,  out int result)
{
   void main( )
    {
       int i1, a, b[10], s, *pa;
       a = p1+p2;
       s = p1-p2;
       pa = &s;
       .....
       result = f1(a, b[i1], pa);
       .....
    }
    int f1( int w, int x, int *p)
    { *p  = w+x+*p;
       return *p;
    }
};
```

(a) Original model (Model 1)

```
behavior  B_child1( in int p1, in  int p2, out int a, out int s) {
     void main()
     {  a = p1+p2;
        s = p1-p2;
     }
};

 behavior B (in int p1, in int p2, out int  result) {
    int a, s;
   //Instantiate child behavior here
    I_B_child1(p1, p2, a, s);
    void main( )
    {
        int i1, b[10], *pa;
        I_B_child1.main();
        pa = &s;
        .....
       result = f1(a, b[i1], pa);
        .....
    }
};
```

(b) Statements encapsulated in behavior (Model 2)

The idea behind encapsulating statements into behavior is same as that of encapsulating functions, i.e. to create a new computation block with explicit interface.

As shown in the figure above, encapsulating statements into a behavior involves creating a behavior body, creating the instance of the newly created behavior, and replacing the statements with the call to the instance. Note that, creating the behavior body requires analyzing expressions in the statements, determining their access type, and determining the port list. Creating the instance requires creating the port map list.

Figure 8: Page-1 of Student Instructions for Experiment 2

Initial Setup

The initial setup files are in the tar file `mp3_v2.tar.gz`. Untar this file using the command:

```
gtar xvzf mp3_v2.tar.gz
```

A directory by name mp3_v2 will be created. Change into this directory

```
cd mp3_v2
```

You will see following files:

`mp3decoder_2.sc` – Complete source of a floating point MP3 decoder (same as the one provided for previous assignment but with two new behaviors).

`Makefile` – To compile and test the sources.

`mp3_fixpt.sc` – Complete source of fix-point MP3 decoder (This is needed for part-2 of the assignment)

`huffman.c, huffman.h` – Needed for fix point MP3 decoder. You can ignore these files as their inclusion and compilation is taken care in the `Makefile`.

There are 3 directories (`reference/, reference-fix/, testStream/`) that contain test streams and reference output. You don't have to worry about these at this stage.

You need to set the path for the SpecC compiler. Source the setup shell script as below:

```
source /opt/sce-20060301/bin/setup.csh
```

Now compile and test the decoder by running following commands

```
make clean
make
make test
```

The setup should compile and simulate without errors. (ignore `"Can't step back"` and `"read length less than max"` messages)

Given MP3 Code

The MP3 code is a basic SpecC code. It has 7 behaviors: Stimulus [at line 3179], Monitor [at line 3147], DUT [at line 3095], MP3Decoder [at line 3041], B_decodeMP3 [at line 2952], B_do_layer3 [at line 2770] and behavior Main [at line 3221]. Use text editor to view the source code and find these behaviors. We want to introduce more behaviors in B_do_layer3. In the next section, we describe the changes you have to do to encapsulate statements into behavior. Note that the line numbers we specify refer to the unmodified original source file (`mp3decoder_2.sc`).

Task 1: Encapsulate Statements in lines 2792-2823 in behavior `B_do_layer3`.

1. We will give a set of instructions to encapsulate these statements into behavior. We would also like to measure the time it takes to perform this conversion.
   Please make a note of the start time T0.
2. `do_layer3()` is a function in behavior `B_do_layer3`. This function is defined at line 2776. We will encapsulate the statements from line 2792 to line 2823 (inclusive).
   First we have to create the behavior body. Create a new empty behavior (say at line 2769) with the signature:

```
behavior Bchild1_B_do_layer3(
     inout int stereo,
     in struct frame *fr,
     inout int sfreq,
     inout int single,
     inout int stereo1,
     inout int ms_stereo,
     inout int i_stereo,
     inout int granules,
     in struct III_sideinfo sideinfo)
     {
```

Figure 9: Page-2 of Student Instructions for Experiment 2

};

Note that this signature is obtained by analyzing the lines 2792 – 2823 and determining the variables and their access types.

3. In the behavior `B_decodeMP3` create an empty main function `void main (void){}`
4. Copy the C statements (2792-2823) into the `main()` of the `Bchild1_B_do_layer3`.
5. Save the file and do: `make` and `make test` and check if the tests run fine.
6. **Note the time T1.**

7. Now we have to create the instance of this newly created behavior in the parent behavior. First, we have to move any local variables accessed by the statements into parent behavior's scope.
8. Move variables, `stereo, stereo1, single, i_stereo, ms_stereo, sfreq, sideinfo, granules` into parent behavior `B_do_layer3`. These variables are declared at the beginning of the function `do_layer3`. After moving change the name to: `R_stereo, R_stereo1, R_single, R_i_stereo, R_ms_stereo, R_sfreq, R_sideinfo, R_granules` so that there are no naming clashes.

9. Create an instance of the new behavior in the parent behavior `B_do_layer3` after the above variable declaration and before the `main` body.
```
Bchild1_B_do_layer3 I_Bchild1_B_do_layer3(R_stereo, fr, R_sfreq,
R_single, R_stereo1, R_ms_stereo, R_i_stereo, R_granules,
R_sideinfo);
```

   The portmap needed for creating the instance is obtained from the previous analysis of the statements function call.
10. Now that you have renamed some of the relocated variables, you have to change the references to the old names with the new names. There are 7 references to variable `sideinfo` in the function `do_layer3( )`, change them to use the new name `R_sideinfo`. There is 1 reference to `granules`, change it to `R_granules`. Similarly, rename the access to other variables `sfreq, stereo, stereo1, ms_stereo, i_stereo, granules`.
11. Replace the statements with the call to the instance
    `I_B_decodeMP3.main ()`
12. Save the file and do: `make` and `make test` and check if the tests run fine.
13. **Make a note of the time T2**

Task 2: Encapsulate statements in lines 2863 to 2887 in behavior `B_do_layer3`

1. For performing this task, unlike the previous case, we will not give the detailed directions. We will briefly outline the steps needed. You will use the source file resulting from the changes done in the previous step.
   Please make a note of the start time T3.
2. You have to encapsulate the statements from line 2863 to 2887 (both lines inclusive). The lines refer to the original final `mp3decoder_2.sc` and are located in the function `do_layer3 ( )` which is a member of the behavior `B_do_layer3`. These will be approximately located at lines 2896 to 2921 in the file that was saved in step-11 in previous section.
3. First create the behavior signature with an empty body. Use the behavior name `B_child2do_layer3`. To derive the signature, analyze the statements and determine the variables and their accesses. To determine the direction of the ports (`IN, OUT, INOUT`) you have to analyze the program and determine the accesses. It is recommended to determine these directions. However, if it is too complicated for you, you can skip this and not specify any port direction. By default these ports will be treated as `INOUT`.
4. Copy the statements into the `main` of the `B_child2do_layer3` behavior, just like the previous example.

Figure 10: Page-3 of Student Instructions for Experiment 2

5. Save the file and do `make` and `make test` and check if the tests run fine.
6. **Note the time T4.**
7. Now create the instance of the behavior `B_child2do_layer3` in the parent behavior `B_do_layer3`. If necessary, move the variables into the scope of the behavior. Replace the statements with the call to the newly created instance.
8. Save the file and do `make` and `make test` and check if the tests run fine.
9. **Make a note of the time T5**

At the end, please report the times **T0**, **T1**, **T2**, **T3**, **T4** and **T5**.

Part2 - Pointer Replacement

As we know, pointers in the C code create ambiguity and make the code unanalyzable and unsynthesizable by automatic tools. In this part, you will see how to replace indirect variable access through pointers with direct variable access.

Pointer re-coding example

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int *p1,*p2, *p3, *p4, (*p5)[16], p6;

4.  p1 = &x;
5.  *p1 = y+1;
6.  if(condition) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  p3 = &ab[40][10];
10. *p3 = 100;
11. p4 = a;
12. p4++;
13. *p4++ = 1;
14. p5 = &ab[5];
15. p6 = p4+v1;
```

(a) Code with pointers

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int ip3, ip4, ip5, ip6;

4.  //Nothing here
5.  x =y+1;
6.  if(condition) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  ip3 =10;
10. ab[40][ip3] = 100;
11. ip4 = 0;
12. ip4++;
13. a[ip4++] = 1;
14. ip5 = 5;
15. ip6 = ip4+v1;
```

(b) Code with p1, p3, p4, p5, p6 substituted

Pointer re-coding is a 2 step process. First, you have to determine the variable to which the pointer points to and second, replace the pointer accesses with the direct access to the target variable. The figure above shows some examples of pointer recoding. Note that pointers to the scalar variables are completely removed, and in place of pointer to the arrays, integer variables that act as indices into the array are created (e.g. line3). Expressions of the pointers that point to arrays are replaced with the array access expression formed by the actual variable the newly created index variable (e.g. line 10). Pointer `p2` is not recoded as it could point to more than 1 variable.

Initial Setup

You will use the same set up as the part-1 of this assignment, except that you will use fix-point MP3 decoder implementation which is contained in the single source file `mp3_fixpt.sc`.
As usual, you need to set the path for the SpecC compiler. Source the setup shell script as below:
`source /opt/sce-20060301/bin/setup.csh`
Now compile and test the decoder by running following commands (note that commands are different from before)

```
make clean
make all_fix
make test_fix
```

Figure 11: Page-4 of Student Instructions for Experiment 2

Given Fix-point MP3 Code

This is a fix-point MP3 implementation in SpecC. It has many behaviors, including `Mad_Stimulus` [at line 13681], `Mad_Monitor` [at line 13533], `MP3Decoder` [at line 13717] and behavior `Main` [at line 13746]. Use text editor to view the source code and find these behaviors. We will replace couple of pointers in the behavior `Calc_sample` located at line 13177. In the next section, we describe the changes you have to do to replace a pointer. Note that the line numbers we refer to are the unmodified lines in the original source file (`mp3_fixpt.sc`).

Task 3: Replace pointer `fe` in behavior `Calc_sample`

1. We will now give a set of instructions to replace the pointer expressions. We would also like to measure the time it takes to perform this conversion.
   Please make a note of the start time T0.
2. Pointer `int (*fe)[8]` is declared at line 13202 in the `main` body of `Calc_sample` behavior and this pointer points to multi-dimensional array `filter[2][2][16][8]` which is a port of the behavior. More specifically, `fe` points to the dimension `(filter)[0][phase & 1]`.
3. We have to replace all accesses to `fe` with the direct access to the array variable `filter`. As a first step, create an integer variable which acts as an index variable (`i_fe`) into the array in place of the pointer `fe`. You can remove the pointer declaration.
4. Replace the pointer initialization statement (`fe = &(filter)[0][phase & 1][0];`) at line 13214 with the initialization of the index variable `i_fe`. (`i_fe = 0`).
5. Replace the expressions involving `fe` with the direct access to `filter`. For example, replace the expression at line 13229 which is :
   ```
   ((lo) += (( *fe)[0]) * (ptr[0]));
   ((lo) += (( *fe)[1]) * (ptr[14]));
   ```
   with
   ```
   ((lo) += (((filter)[0][phase & 1][i_fe])[0]) * (ptr[0]));
   ((lo) += (((filter)[0][phase & 1][i_fe])[1]) * (ptr[14]));
   ```
   and so on…
6. Similarly, replace all the other expression involving `fe` in the `main()` body.
7. Arithmetic Expressions involving `fe` such as the one at line 13242:
   ```
   ++fe;
   ```
   is replaced with
   ```
   ++i_fe;
   ```
8. Save the file and do: `make all_fix` and `make test_fix` and check if the tests run fine.
9. **Make a note of the time T2**

Task 4: Replace pointer `fx`, `fo` and `Dptr` in behavior `Calc_sample`

1. We will note give complete instructions to recode these pointers. These pointers can be recoded on similar lines as pointer `fe`.
2. Pointer `fx` is declared at line 13204 and its initialization is at line 13215. Note that this pointer points to dimension `(filter)[0][ ~phase & 1]`.
3. Replace this pointer and make a note of the start and the end time to perform the task **(T3, T4)**
4. Pointer `fo` is declared at line 13203 and its initialization is at line 13216. Note that this pointer points to dimension `(filter)[1][ ~phase & 1]`.
5. Replace this pointer and make a note of the start and the end time to perform the task **(T5, T6)**

Figure 12: Page-5 of Student Instructions for Experiment 2

6. Pointer `Dptr` is declared at line 13205 and its initialization is at line 13217. Note that this pointer points to array `D`.
7. Replace this pointer and make a note of the start and the end time to perform the task **(T6, T7)**

At the end, please report the times **T0** trough **T7**.

Deliverables:

- 1-paragraph description about each of the tasks above
  (i.e. how far you got, what were the problems, how did you solve it)
- Please also report the times for part 1 (T0 to T5, in minutes), and for part 2 (T0 to T7, in minutes).

Due: Week 8 (Nov 16, 2007)

--
Rainer Doemer (ET 444C, x4-9007, doemer@uci.edu)

Figure 13: Page-6 of Student Instructions for Experiment 2

A4. Times Reported by Students for Experiment 2

The results submitted by the students for the part-1 are presented below in Table 12. The time stamps T0 to T5 refer to the time stamps given the description Appendix-A3. *Task 1* is the time taken for 1st Statement-to-Behavior transformation. Columns *time (T0, T1), time (T1, T2)* together constitute *Task 1*. *Task 2* (*time (T3, T4) + time (T4, T5)) is* the time taken for the 2nd Statement-to-Behavior transformation. *Total* is the time for *Task 1* and *Task 2*.

Student	T0	T1	T2	T3	T4	T5	time(T0,T1)	time(T1,T2)	time(T3,T4)	time(T4,T5)	Task 1	Task 2	Total
1	10:50	11:04	11:23	11:23	11:40	11:46	0:14	0:19	0:17	0:06	0:33	0:23	0:56
2	0:00	0:16	0:45	1:02	1:25	1:37	0:16	0:29	0:23	0:12	0:45	0:35	1:20
3	16:57	17:01	17:16	17:23	17:40	18:07	0:04	0:15	0:17	0:27	0:19	0:44	1:03
4	0:00	0:14	0:34	0:00	0:10	1:00	0:14	0:20	0:10	0:50	0:34	1:00	1:34
5	11:55	12:04	12:46	1:24	1:37	2:02	0:09	0:42	0:13	0:25	0:51	0:38	1:29
6	8:00	8:17	8:45	8:50	9:15	9:25	0:17	0:28	0:25	0:10	0:45	0:35	1:20
7	8:16	8:20	8:40	8:45	9:00	9:18	0:04	0:20	0:15	0:18	0:24	0:33	0:57
8	0:00	0:30	1:18	0:00	1:04	1:28	0:30	0:48	1:04	0:24	1:18	1:28	2:46
9	0:00	n/a	0:47	0:00	n/a	2:00	n/a	n/a	n/a	n/a	0:47	2:00	2:47
10	9:16	9:22	9:33	9:34	9:49	10:43	0:06	0:11	0:15	0:54	0:17	1:09	1:26
11	0:00	0:03	0:21	0:00	0:12	0:20	0:03	0:18	0:12	0:08	0:21	0:20	0:41
12	n/a	n/a	n/a	n/a	n/a	n/a	0:09	0:25	0:15	0:45	0:34	1:00	1:34
13	5:26	5:41	6:05	6:34	6:55	7:44	0:15	0:24	0:21	0:49	0:39	1:10	1:49
14	8:36	8:55	9:35	10:09	10:10	12:00	0:19	0:40	0:01	1:50	0:59	1:51	2:50
15	5:15	5:32	6:31	11:00	11:00	14:30	0:17	0:59	0:00	3:30	1:16	3:30	4:46
Average							0:12	0:28	0:17	0:46	0:41	1:07	1:49
MAX							0:30	0:59	1:04	3:30	1:18	3:30	4:46
MIN							0:03	0:11	0:00	0:06	0:17	0:20	0:41

Table 12: Times reported by students to manually recode the statements in lines 2792-2823 into behavior "Bchild1_B_do_layer3"(Task1) and recode the statements in lines 2863-2887 into behavior "Bchild2do_layer3"(Task2)

Note1: The time stamps not provided by the students are indicated by "n/a". For example, student 12 and student 10 provided only durations and not the time stamps.

Note 2:

- *time(T0,T1)* is the time to create the behavior body including the portlist for 1st S2B transformation
- *time(T1,T2)* is the time to create the behavior instance including the portmap for 1st S2B transformation
- *time(T3,T4)* is the time to create the behavior body including the portlist for 1st S2B transformation

- *time(T4, T5)* is the time to create the behavior instance including the portmap for 1st S2B transformation

The results submitted by the students for part-2 of the assignment are presented below in Table 13. *Task 1(=time(T0,T1))* is the time to recode the 1st pointer given the detailed instructions. *time(T2,T3), time(T4,T5) and time(T6,T7)* is the time to recode each remaining pointer. *Task 2* is the accumulated time to recode the 3 pointers. *Total* is the time to recode all the 4 pointers.

Stu.	T0	T1	T2	T3	T4	T5	T6	T7	time(T0,T1)	time(T2,T3)	time(T4,T5)	time(T6,T7)	Task 1	Task 2	Total
1	11:52	12:10	12:15	12:21	12:21	12:28	12:30	12:50	0:18	0:06	0:07	0:20	0:18	0:33	0:51
2	0:00	0:13	0:32	0:48	1:06	1:25	1:42	2:05	0:13	0:16	0:19	0:23	0:13	0:58	1:11
3	18:03	18:19	18:31	18:36	18:36	18:41	18:41	18:48	0:16	0:05	0:05	0:07	0:16	0:17	0:33
4	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	1:03	0:19	1:22
5	2:33	2:47	2:47	2:48	3:11	3:43	3:43	n/a	0:14	0:01	0:32	n/a	0:14	n/a	n/a
6	9:30	9:55	10:13	10:20	10:30	10:44	10:50	11:11	0:25	0:07	0:14	0:21	0:25	0:42	1:07
7	9:30	9:40	9:40	9:45	9:45	9:49	9:50	9:54	0:10	0:05	0:04	0:04	0:10	0:13	0:23
8	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0:27	0:45	1:12
9	0:00	0:20	0:00	0:30	n/a	n/a	n/a	n/a	0:20	0:30	n/a	n/a	0:20	n/a	n/a
10	10:54	11:13	11:14	11:20	11:21	11:29	11:30	11:41	0:19	0:06	0:08	0:11	0:19	0:25	0:44
11	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0:11	0:04	0:04	0:10	0:11	0:18	0:29
12	0:00	0:16	0:00	0:12	0:00	0:04	0:00	0:05	0:16	0:12	0:04	0:05	0:16	0:21	0:37
13	8:32	9:05	9:12	9:28	9:30	9:32	9:35	9:37	0:33	0:16	0:02	0:02	0:33	0:20	0:53
14	2:02	2:20	2:27	2:35	2:35	2:43	2:43	2:53	0:18	0:08	0:08	0:10	0:18	0:26	0:44
15	9:08	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Average									0:17	0:09	0:09	0:11	0:21	0:28	0:50
MAX									0:33	0:30	0:32	0:23	1:03	0:58	1:22
MIN									0:10	0:01	0:02	0:02	0:10	0:13	0:23

Table 13: Times reported by students to manually recode 4 pointers

Note1: Some of the students (Student 5, 9, 15) could not complete the experiment and these are indicated as "n/a".

A5. Student Instructions for Experiment 3

EECS 222A
System-on-Chip Description and Modeling
Fall 2007

Assignment 6

Posted: November 16, 2007 (week 8)
Due: November 30, 2007 (week 10)

Tasks: Part 1: Creating Behaviors in C Code using Source Re-coder
Part 2: Pointer Elimination using Source Re-coder

Instructions: (by Pramod Chandraiah)

Part1 – Create behaviors using Source Re-Coder

In the last two assignments you introduced behaviors and replaced pointers manually. As you have experienced, this manual conversion is very time consuming and error-prone. Compared to the time taken by the automatic exploration using SCE, this time is very large. You might have also noticed that some of these manual re-coding tasks can be automated to speed-up the overall process of specification development.
So we introduce source re-coder. Source re-coder is an integration of various automatic code transformations tools and an interactive editor. With source re-coder you can not only manually type in code, but also invoke automatic tools to create to automatically re-code specification.
For this assignment, you will use the source re-coder to repeat the tasks that were performed manually in the previous assignments.

Initial Setup

The initial setup files are in the tar file `mp3_v3.tar.gz`. Untar this file using the command:
`gtar xvzf mp3_v3.tar.gz`
A directory by name `mp3_v3` will be created. Change into this directory
`cd mp3_v3`
The entire floating point MP3 source code is in single file `mp3decoder.sc` and the fix-point code is in `mp3_fixpt.sc`. There is a `Makefile` to compile and test the decoder for a set of MP3 streams. There are two directories reference/ and `testStream`/ which you don't have to worry about at this stage.
You need to set the path for the SpecC compiler. Source the setup shell script as below:
`source /opt/cars/scc/bin/setup.csh`
Now compile and test the decoder by running following commands
```
make clean
make all
make test
make all_fix
```

Figure 14: Page-1 of Student Instructions for Experiment 3

make test_fix
The setup should compile and simulate without errors. (ignore "Can't step back" and "read length less than max" messages)

Given MP3 Code

The MP3 code (mp3decoder.sc, mp3_fixpt.sc) given are the same sources you received for your previous assignments.

Source re-coder

Run the source re-coder by typing: /opt/cars/cute/bin/cute & from mp3_v3 directory.
An editor comes up. There are 3 views in the editor: the *main view* which is initially blank, a *side view* which lists the directories and files and a *message panel* at the bottom with different tabs for each message type.
Open the file mp3decoder.sc from **File→Open** and selecting mp3decoder.sc. The code will appear in the *main view*. Now enable the line numbers from **View→Line numbers**. Scroll down and you will see the following 4 behaviors in the code: Stimulus [at line 5724], Monitor [at line 5692], DUT [at line 5640] and MP3Decoder [at line 2755]. Notice that bodies of behaviors are colored in blue and channels in light brown and all the global entities are uncolored.

Source re-coder has 2 tool bars. The first one consists of basic editing tools copy/paste and so on. The 2nd tool bar consists of tools to perform the code transformations. If you hover the mouse over these icons, you can see the names of each of these tools.
For this part of the assignment you only need to be aware of following tools:

- Current Position (**CP**): 2nd button from the left (Icon: Bulb)
- Build Design (**BD**): 3rd button from the left (Icon: Green recycle symbol)
- Function 2 Behavior (**F2B**): 4th button from the right (Icon: Blue rectangle with rounded edges)
- Statement 2 Behavior (**S2B**): 3rd button from the right (Icon: Lines and Blue rectangle)
- ReScope Variable to Class Scope (**RCS**): (Icon:3 red vertical triangles)

Now on, these tools will be referred as CP, BD, F2B, S2B and RCS.
Before we start, we have to compile the code and build the design. Use can you the BD button to build the main design.
The tools are invoked by placing the cursor at the desired point of interest in the code and by clicking on the appropriate button in transformation tool bar. The source re-coder invokes the tool on the object at the current cursor position.
Since the design is sparser than the source code, not every cursor position points to an object (variable/behavior/function/statement) in the design. So before you invoke F2B/S2B/RCS tool, place the cursor at the point of interest and

Figure 15: Page-2 of Student Instructions for Experiment 3

invoke **`CP`**. This will display different possible objects corresponding to this cursor position and the result will appear in the *SpecOut* tab of the *Message Panel*. Usually, this list will contain the variable and the statement at that position. For example, if you put the cursor on line 2778 next to function call `decodeMP3()` (specifically, to the left of letter 'd') and invoke `CP`, you will see 2 entries in the *message panel*. 1st entry for the function symbol `decodeMP3`, and 2nd for the whole function call statement. If you see your object of interest in this panel then you can invoke the necessary tool. If you don't see the object of interest, then the current cursor position is not valid and you have to re-position the cursor to a different position where the same variable is again used.

Now you will invoke the tools from the transformation tool bar to introduce some behaviors including those which you introduced manually in the previous 2 assignments. Note that you will re-build the design by through **`BD`** button after invoking any transformation that changes to the code.

Task set 1:

1. **Please make a note of the start time T0.**
2. Open the file `mp3decoder.sc` as explained above in the source re-coder. Build the design using **`BD`** button.
3. First, we will encapsulate one of the calls to the function `decodeMP3()` in a behavior. In the source recode navigate to the line number 2778 and place the cursor next to the function `decodeMP3( )`. Specifically, place the cursor to the left of letter `'d'`. Now invoke the **`F2B`** tool and realize the transformation instantly. The cursor will be re-positioned following the changes. You will see a behavior `B_decodeMP3` at line number 2764. Navigate around to see the changes to the code.
4. Re-build the design by clicking on **`BD`**
5. Scroll down to line 2833 to the function call `do_layer3( )`, position the cursor appropriately and invoke **`F2B`** . New behavior `B_do_layer3` will appear starting at 2770.
6. Re-build the design by clicking on **`BD`**
7. Save the file by **`File→SaveAs`** and choose the filename `mp3decoder.sc`. Note that this is the only way to save changes; the other ways through `Save/SaveAll` buttons do not work.
8. Re-build the design by clicking on **`BD`**
9. **Note the time T1.**
10. Now we would like to wrap the statements from line 2792 to 2823 into behavior. Before that, we have to move the variables to the behavior scope. This can be done using RCS tool. If you analyze these statements, you will realize that following local variables must be rescoped: `stereo`, `stereo1`, `single`, `sideinfo`, `sfreq`, `ms_stereo`, `i_stereo`, `granules`.

Figure 16: Page-3 of Student Instructions for Experiment 3

Place the cursor to the left of these variable names in the code, either at their line of declaration, or at the lines they are used and click on **RCS** button. When not sure if the cursor is pointing to the right symbol, place the cursor and click on **CP** button. **RCS** must be invoked on each of the above listed variables one at a time. You do not need to use the **BD** button in-between successive **RCS** invocation as this is taken care by **BD**.

11. Re-build the design by clicking on **BD**
12. Save the file by **File→SaveAs** and choose the filename `mp3decoder.sc`.
13. Re-build the design by clicking on **BD**

14. To wrap the statements from 2792 to 2823, select the lines (by pressing and holding the left-mouse button or using shift key and scrolling down) from **2793** to **2812** and invoke **S2B**. New behavior `B_child_B_do_layer3` will be created at line 2781 with all the lines from 2792 to 2823 selected. Note that we are not highlighting all the lines till 2823. This is because, as we mentioned before, there do not exist exact one to one correspondence between the cursor position and the objects in the main design. This is an exception and applies only to these set of lines.

15. Re-build the design by clicking on **BD**
16. Save the file by **File→SaveAs** and choose the filename `mp3decoder.sc`.
17. Re-build the design by clicking on **BD**

18. Note the time T2.

19. Rescope the variables used between lines 2896 to 2921 (`hybridIn`, `scalefacs`, `gr_info`, `gr`)
20. Now wrap the statements from line 2898 to 2925 into a behavior (lines numbers slightly changed after re-scoping) using **S2B**. After this, the statements are wrapped and replaced with instance call `I_B_child_B_do_layer4` at line 2927.

21. **BD, File→SaveAs, BD**
22. **Note the time T3.**

Task set 2:

23. In addition to the 4 behaviors, we will introduce couple of more behaviors. **Note the time T4**
24. Wrap statements from line 2941 to 2968 into behavior. Highlight from 2941-2968 and call **S2B**. `I_Bchild_B_do_layer5.main()` will be introduced at line 2985.
25. **BD, SaveAs, BD**
26. Encapsulate function call `III_antialias` at line 2992 into behavior using **F2B**. Do not forget to re-scope `gr_info` before doing this.

Figure 17: Page-4 of Student Instructions for Experiment 3

27. **BD**, **SaveAs**, **BD**
28. Repeat the same for function III_hybrid( ) which is now at line 3050. Do not forget to re-code `ch`, `hybridOut`, before doing so. I_B_III_hybrid.main() will appear at line 3139.
29. **BD**, **SaveAs**, **BD**
30. Close the file.
31. **Note time T5**
32. Test the changes by running following commands from your terminal:

```
make clean
make all
make test
```

At the end, please report the times **T0**, **T1**, **T2**, **T3**, **T4** and **T5**.

Part2 - Pointer Replacement

Just like the way behaviors can be created with a click of a button using source re-coder, pointers can also be recoded.

Given Fix-point MP3 Code

This is the same fix-point MP3 implementation you used for the previous assignment.

Source recoder

Run the source re-coder by typing: `/opt/cars/cute/bin/cute` & from `mp3_v3` directory.
Note the 2 more tools in the transformation tool bar.

- `Points-to List` **(PL)**: 9th button from right (Icon: Pointer with a question mark)
- `Recode Pointer` **(RP)**: 8th button from right(Icon: Pointer with crossing line)

Now on these tools will be referred to as **PL** and **RP**

Open the file `mp3_fixpt.sc` from **File→Open** menu. The code will appear in the *main view*. Now enable the line numbers from **View→Line numbers**. Scroll down and notice the following 4 behaviors in the code: `Mad_Stimulus` [at line 13681], `Mad_Monitor` [at line 13533], `MP3Decoder` [at line 13717] and behavior `Main` [at line 13746]. We will replace couple of pointers in the behavior `Calc_sample` located at line 13177.
Build the design using **BD**.

Replace pointers (fe, fx, fo, Dptr) in behavior Calc_sample

1. **Note the time T0**

Figure 18: Page-5 of Student Instructions for Experiment 3

2. Build the design using `BD`
3. Place the cursor next to `fe` (say at line 13202 next to letter 'f') and click on `PL`. This will display the target variable the pointer points-to in the *message panel*. If the pointer points to only one variable (as in the case of fe) then you can recode it. If the `PL` does not display any target variable, then it means that source re-coder failed to analyze the code.
4. Invoke `RP` and observe the changes to the code.
5. Re-build the design through `BD`.
6. Repeat these steps 3 to 5 for `fx`, `fo`, and `Dptr`.
7. `BD`, `SaveAs`, `BD`
8. `make clean`
9. `make all_fix`
10. make `test_fix`
11. **Note the time T1**

Deliverables:

- 1-paragraph description about each of the tasks above
  (i.e. how far you got, what were the problems, how did you solve it)
- Please also report the times for part 1 (T0 to T5, in seconds/minutes), and for part 2 (T0 to T1, in seconds/minutes).

Due: Week 10 (Nov 30, 2007)

--
Rainer Doemer (ET 444C, x4-9007, doemer@uci.edu)

Figure 19: Page-6 of Student Instructions for Experiment 3

A6. Times Reported by Students for Experiment 3

The results provided by students for the part-1 of the assignment are consolidated in the Table 14 below. The time stamps T0 to T5 refer to the time stamps given the description Appendix-A5. *Task 1* is the time taken for 2 Function-to-Behavior and 2 Statement-to-Behavior transformations. Columns *time(T0, T1), time(T1, T2),* and *time(T3,T4)* give the break-up of *Task 1*. *Task 2(=time(T4,T5))* is the time taken to implement additional transformations (not conducted manually before) 2 Function-to-Behavior and 1 Statement-to-Behavior.

Stu.	T0	T1	T2	T3	T4	T5	time(T0,T1)	time(T1,T2)	time(T3,T4)	time(T4,T5)	Task 1	Task 2	Total
1	23:21	23:32	23:44	0:00	0:40	1:09	0:11	0:12	0:16	0:29	0:39	0:29	1:08
2	0:00	0:09	0:22	0:36	0:46	0:59	0:09	0:12	0:13	0:12	0:36	0:12	0:48
3	17:23	17:24	17:29	17:33	17:38	17:41	0:01	0:05	0:04	0:03	0:10	0:03	0:13
4	n/a	n/a	n/a	n/a	n/a	n/a	0:04	0:05	0:04	0:15	0:13	0:15	0:28
5	9:35	9:39	10:00	10:15	10:15	10:31	0:04	0:21	0:15	0:16	0:40	0:16	0:56
6	9:15	9:19	9:27	9:40	n/a	n/a	0:04	0:08	0:13	n/a	0:25	n/a	n/a
7	12:40	12:45	12:54	13:04	n/a	n/a	0:05	0:09	0:10	n/a	0:24	n/a	n/a
8	n/a	n/a	n/a	n/a	n/a	n/a	0:15	0:09	0:10	0:15	0:34	0:15	0:49
9	n/a	n/a	n/a	n/a	n/a	n/a	0:04	0:05	0:02	0:10	0:11	0:10	0:21
10	10:30	10:34	10:43	10:48	10:58	11:05	0:04	0:09	0:05	0:10	0:18	0:10	0:28
11	n/a	n/a	n/a	n/a	n/a	n/a	0:02	0:07	0:14	0:09	0:23	0:09	0:32
12	n/a	n/a	n/a	n/a	n/a	n/a	0:03	0:03	0:05	0:10	0:11	0:10	0:21
13	10:26	10:31	10:39	10:45	10:49	10:56	0:05	0:08	0:06	0:04	0:19	0:04	0:23
14	2:18	2:25	2:44	4:00	4:04	4:08	0:07	0:19	1:16	0:04	1:42	0:04	1:46
15	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Average							0:05:37	0:09:29	0:13:51	0:11:27	0:28:57	0:11:27	0:41:08
MAX							0:15	0:21	1:16	0:29	1:42	0:29	1:46
MIN							0:01	0:03	0:02	0:03	0:10	0:03	0:13

Table 14: Times reported by students for conducting 4 F2B operations and 3 S2B operations automatically using source re-coder

Note 1:

- Some of the time stamps *T0 – T5* were not provided by the students and these are indicated as "n/a". Instead, these students provided only the durations
- Student 15 did not conduct the experiment
- Student 6 and 7 did not provide the *time(T4, T5),* which is a measure of the time to conduct the additional transformation (2 F2B + 1 S2B)

Note 2:

- *time(T0, T1)* is the time to automatically recode 2 functions into behaviors
- *time(T1, T2)* is the time to automatically recode 1st set of statements
- *time(T3, T4)* is the time to automatically recode 2nd set of statements

- *time(T4, T5)* is the time to do additional tasks (2 F2B and 1 S2B)

Note 3:

- The time *time(T3, T4)* reported by student 14 is unusually high, as the student encountered software crash and took even that time into account. So, we decided not to consider this data for comparison of automatic and manual times.

The results provided by students for the part-2 (Pointer Recoding) of the assignment are consolidated in the Table 15 below. In the table, *Task 1* is same as *time(T0,T1)*.

Student	T0	T1	time(T0,T1)	Task 1	Total
1	1:15	1:30	0:15	0:15	0:15
2	0:00	0:12	0:12	0:12	0:12
3	17:43	17:48	0:05	0:05	0:05
4	n/a	n/a	0:05	0:05	0:05
5	10:23	10:35	0:12	0:12	0:12
6	10:13	10:19	0:06	0:06	0:06
7	1:21	1:26	0:05	0:05	0:05
8	n/a	n/a	0:07	0:07	0:07
9	n/a	n/a	0:05	0:05	0:05
10	12:00	12:03	0:03	0:03	0:03
11	n/a	n/a	0:02	0:02	0:02
12	n/a	n/a	0:03	0:03	0:03
13	5:30	5:35	0:05	0:05	0:05
14	9:39	9:45	0:06	0:06	0:06
15	n/a	n/a	n/a	n/a	n/a
Average			0:06:33	0:06:33	0:06:33
MAX			0:15	0:15	0:15
MIN			0:02	0:02	0:02

Table 15: Times reported by students for conducting 4 pointer re-coder operations using source re-coder