C to SpecC Conversion Style

Kiran Ramineni
Daniel Gajski

CECS Technical Report 03-13
April 14, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{kiran,gajski}@cecs.uci.edu

C to SpecC Conversion Style

Kiran Ramineni
Daniel Gajski

CECS Technical Report 03-13
April 4, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{kiran,gajski}@cecs.uci.edu

Abstract

We present a methodology and style guidelines for automatic translation from a given design specification in raw C code to SpecC code. Traditional conversion from C to SpecC relies on manual refinement which is painfully time consuming and error prone. The automation of the refinement process provides a useful tool to reduce the time spent in conversion of C to SpecC so that the system designer can spend more time with design decisions based on exploration with SpecC code.

Contents

List of Figures

List of Tables

C to SpecC Conversion Style

Kiran Ramineni and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
{kiran, gajski}@ics.uci.edu

Abstract

We present a methodology and style guidelines for automatic translation from a given design specification in raw C code to SpecC code. Traditional conversion from C to SpecC relies on manual refinement which is painfully time consuming and error prone. The automation of the refinement process provides a useful tool to reduce the time spent in conversion of C to SpecC so that the system designer can spend more time with design decisions based on exploration with SpecC code.

1 Introduction

In the recent years, the dramatic increase of behavioral and structural complexity of SoC designs has raised the abstraction level of system specification. Along with the higher levels of abstraction comes the need for efficient system level synthesis of functional specification to target architectures. The wide variety of available target architectures makes the job of making the optimal choice all the more complicated. This calls for a methodology to efficiently explore design spaces and fast tools for refinement of functional system specification to an architecture model, so that more architectures may be explored and evaluated. Our System level design methodology [1] is aiming at refining an initial, functional system specification into a detailed implementation description ready for manufacturing.

System level methodology consists of a set of models and transformations (Figure 1). The executable models represent the same system at different levels of abstraction at different phases of the design process. The transformations are a series of well-defined steps through which higher level models are gradually refined into lower level models. Our methodology starts with the capture of the intended functionality in the form of *specification model* which describes the functionality as well as the performance, power, cost and other constraints of the intended design. *Architecture exploration*, which synthesizes the specification into an *architecture model*, includes the design tasks of allocation, partitioning of behaviors, channels, and variables, and scheduling. *Communication synthesis* synthesizes the abstract communications between behaviors in the architecture model into an implementation. In the resulting *communication model*, communication is described in terms of actual wires and timing is described with bus protocols.

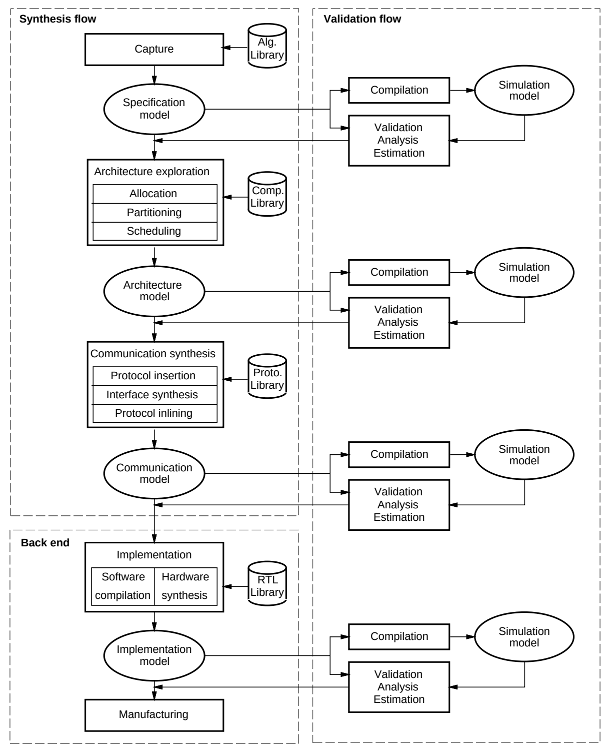

Figure 1. System Level Methodology.

In this paper we will focus on refinement from an unclean specification model into a clean specification model. We will identify a set of building blocks and describe a methodology that implements the refinement. The input to our refinement tool is the unclean specification of an abstract functional model and the output is the clean specification model that will be used by System level methodology for analysis and exploration.

The rest of the paper is organized as follows. Section 2 talks about SpecC language in general and different behaviors inside SpecC language. Section 3 will focus on the basic constructs of a specification model. We will illustrate different possible interleaving of basic constructs to get a complicated nested code in section 4. The implementation of the methodology for a complex example which has multiple nestings is shown in section 5 and some experimental results are shown in section 6. Section 7 summarizes the paper with conclusions. Appendix contains a sample of code converted from C to SpecC with some hierarchial tree diagrams and code before and after the transformation.

2 Code Refinement

2.1 Sequential vs Parallel programming model

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

In general, hardware language consists of a number of components executing in parallel, which can be called the parallel programming model[4].

One step in behavior exploration is converting a sequential programming model to a parallel programming model.

2.2 Clean model and non-clean model

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

Clean computation: Behaviors are defined hierarchically; each behavior can also contain a number of behavior instantiations of other behaviors. For example, in the C language, behaviors are represented by functions, and the behavior instances are represented by function calls. In a clean computation behavior, only two types of behaviors, leaf behavior and non-leaf behavior are allowed. Leaf behavior contains a sequence of statements without any behavior instances. Non-leaf behavior contains only behavior instances without any statement execution. Figure 2(function A) is a leaf behavior, Figure 2(function B) is a non-leaf behavior. Figure 2(function C) is a non-clean computation behavior.

Clean communication: In a clean communication model, parameters are passed by value among behavior instances. Figure 2(function D) is a non-clean communication model; Figure 2(function E) is a clean communication model.

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

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

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

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

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

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

(a) C code

Figure 2. Examples of clean computation and communication in C language

The *behavior exploration* process can be divided into two steps, namely, *semantic(functionality) refinement* and *syntactic refinement*, which can be further divided into substeps.

2.3 Semantic refinement

Neither the concept of pipelining nor parallelism exists within the C language. However, to efficiently perform behavior modeling, system level design language must support these two concepts.

Behavior-parallel: Two behaviors are defined as behavior-parallel if the execution sequence of the two behaviors does not influence the simulation result. Otherwise, the two behaviors are defined as behavior-sequential.

Behavior-pipeline: If, within a sequential programming model, a number of behaviors are executed one after another in a loop body, and behavior communicates only with the next behavior, then the execution relation between these behaviors can be termed as: behavior-pipeline.

Architecture-parallel: If, during behavior to architecture mapping, behavior A and behavior B are mapped to different architecture components, then the implementation relation between A and B is called architecture-parallel. Otherwise it is called architecture-sequential.

Architecture-sequential: If, during behavior to architecture mapping, behavior A and behavior B are mapped to one architecture component, then the implementation relation between A and B is called architecture-sequential. Typically, you would be doing this mapping if there is a lot of communication between behavior A and B and mapping these two behaviors on two different components makes the system bus throttled and then the communication becomes the bottleneck.

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

2.4 Syntactic refinement

The syntactic refinement step modifies already granualarized C code to SpecC syntax so that tools that use SpecC specification can analyze well to assist the system designer. This is very important and time consuming part since a good refinement ensures not only a good starting point but also some correct design decisions later.

SpecC is a true super set of ANSI-C. In other words, every C program is also a SpecC program. In addition to the ANSI-C constructs, the SpecC language includes extensions for hardware design.

2.5 Hello World Example

A SpecC program is a collection of classes. There are three types of classes, namely behaviors, channels, and interfaces. These directly reflect the structure of the SpecC model. Syntactically, a SpecC behavior is specified by use of a behavior definition, such as the behavior Main in the figure. In general, a behavior definition consists of a set of ports, a set of local variables, instantiations and methods, and a mandatory main method.

```
/* HelloWorld.c */

#include <stdio.h>

void main(void)
{
 printf("HelloWorld!\n");
};
```

(a) C code

```
// HelloWorld.sc

#include <stdio.h>

behavior Main
{
  void main(void)
     {
        printf("HelloWorld!\n");
     }
};
```

(b) SpecC code

Figure 3. A Hello World example

A SpecC program starts with the execution of the main method of the root behavior, which is identified by its name Main. Please note that main and Main are names which are recognized by automated tools, but these names are not keywords. In the SpecC version of the Hello World example, the main method is identical to the main function of the ANSI-C version. The only difference is that it is encapsulated in the Main behavior. In general, it is always the main method that is executed when an instantiated behavior is called. Also, the completion of the main method determines the termination of the execution of the behavior.

2.6 SpecC Structural Hierarchy

In SpecC, structural hierarchy is supported in the style of standard block diagrams. More specifically, structure is represented as a hierarchical network of behaviors and channels.

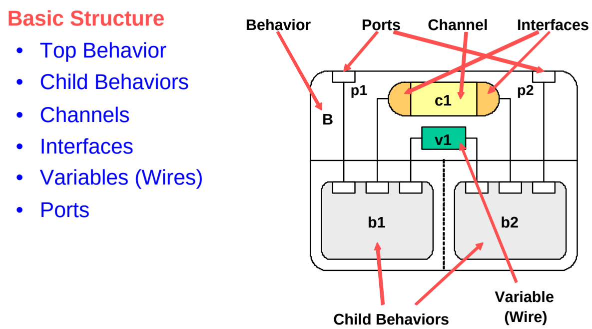

Figure 4. SpecC Basic Structure

The example on the right shows a behavior B with two ports, p1 and p2, through which it can communicate with its environment. Internally, these ports are connected to two child behaviors, b1 and b2. These child behaviors can communicate in two ways. First, both are connected to a shared variable v1, which is written by b1 and then read by b2. Second, b1 and b2 can communicate through the channel c1. For example, the behavior b1 calls a function Write provided by the left interface of channel c1. Similar, behavior b2 calls a Read function provided by the right interface. Please note that the figure only shows one level of the structural hierarchy. The child behaviors b1 and b2 can again consist of a network of behaviors and channels.

2.7 SpecC Behaviors

Behavioral hierarchy is the composition of child behaviors in time. In SpecC, child behaviors can either be executed sequentially or concurrently. Sequential execution, as shown on the left hand side, can be specified by standard sequential statements, or as a finite state machine (FSM) model with explicit state transitions. On the right hand side, concurrent execution is either parallel or pipelined.

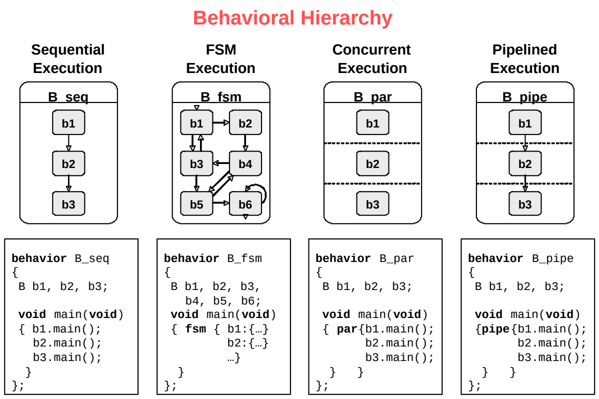

Figure 5. SpecC Behavior Hierarchy

Concurrent execution is shown in the third figure, where b1, b2 and b3 run in parallel. They all start simultaneously when B_par starts. Once all of them have completed their execution, B_par will also finish. Syntactically, parallel execution is specified by use of the par construct. Very similar to the par construct, the pipe construct allows execution in pipelined fashion.

3 Basic Constructs

In this section, we specify guidelines for C to SpecC translation for different control statements by recognizing some basic blocks in a typical input C program.

3.1 If (no else) Statement

An If statement Figure 6 (part a) is clean, if the code inside the braces of if condition (*If Clean Code Segment*) is a sequence of just data statements and there are no calls to other behaviors. For this type of statement, we can get valid SpecC code just by wrapping the whole block of the If statement as is into a leaf behavior.

An If statement Figure 6 (part b) is unclean, if the code inside the braces of if condition (*If Unclean Code Segment*) is a composite of data statements as well as calls to other behaviors. *Start* and *End* states are fictitious dummy states which correspond to entrance and exit, respectively inside *If_fsm_block*. First task for converting this type of code is transforming *If Unclean Code Segment* into a composite behavior which is clean in SpecC. *If condition check* can be transformed to a *Yes/No FSM* (Finite State Machine) as it resembles decision making as it resembles decision making. If the condition is satisfied then the behavior representing *If Unclean Code Segment* will be called.

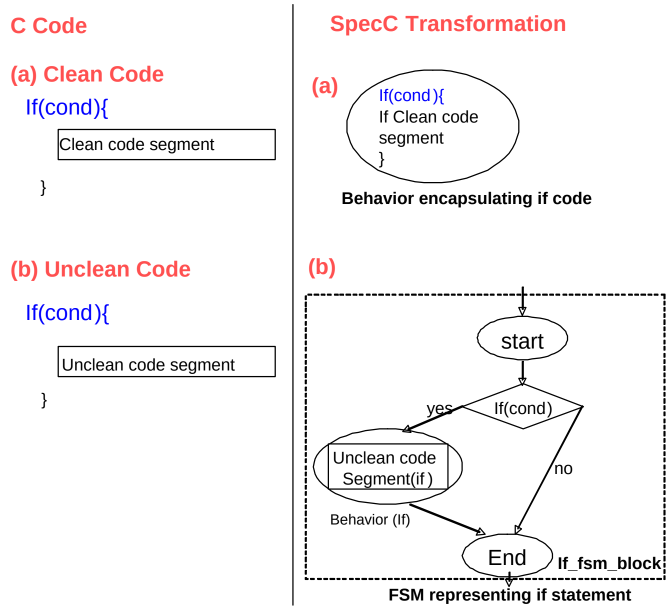

Figure 6. If statement

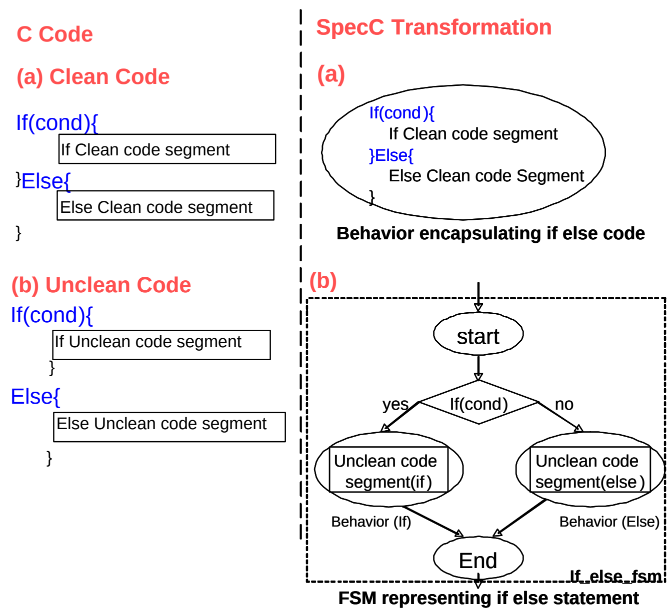

Figure 7. If Else statement

3.2 If Else Statement

An If Else statement Figure 7 differs from an If statement Figure 6 on including an *Else* part. If Statement is a subset of If Else statement since If statement does not have Else part. An If Else statement Figure 7 (part a) is clean if the code inside the braces of if condition (*If Clean Code Segment*) and else condition (*Else Clean Code Segment*) are sequences of just data statements and there are no calls to other behaviors. For this type of statement, we can get valid SpecC code just by wrapping the whole blocks of "If Clean Code Segment" and "Else Clean Code Segment" with the condition check into a leaf behavior.

An If Else statement Figure 7 (part b) is unclean if it satisfies one of the conditions below:

- If part Code Segment is Unclean
- Else part Code Segment is Unclean
- Both If and Else Code Segments are Unclean

To derive a valid SpecC code, first we need to make one composite behavior for each of If and Else Unclean code segments. Then, we can introduce *Yes/No FSM* for the condition check. If the condition is satisfied, we make a call to the If composite behavior. Otherwise, we call the Else composite behavior.

3.3 While Statement

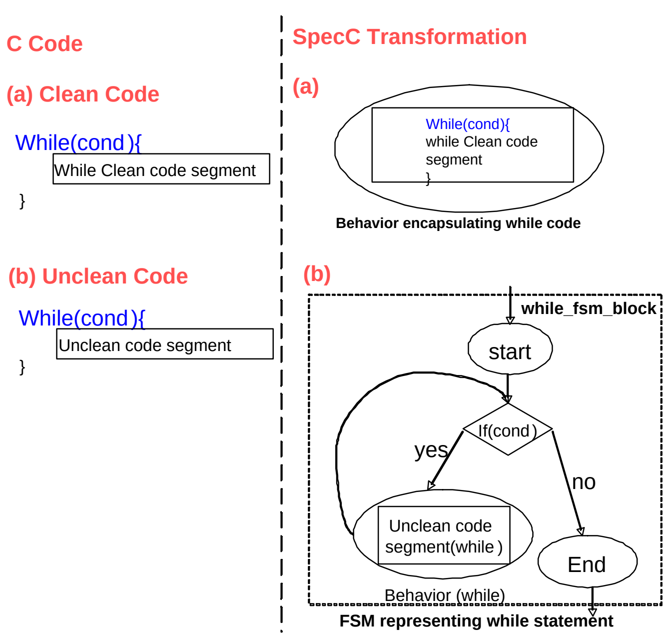

Figure 8. While statement

A While statement Figure 8 (part a) is clean if *While Clean Code Segment* is a sequence of just data statements and there are no calls to other behaviors. For this type of statement, we can get valid SpecC code just by wrapping the whole block of the While statement as is into a leaf behavior.

A While statement Figure 8 (part b) is unclean if *While Unclean Code Segment* is a composite of data statements as well as calls to other behaviors. First step in converting this type of code is transforming *While Unclean Code Segment* into a composite behavior which is clean in SpecC. *If condition check* can be transformed to a *Yes/No FSM* as it resembles decision making. If the condition is satisfied then the behavior representing *While Unclean Code Segment* will be called and then the control loops back to the condition checking. This will repeat until the condition becomes false. Then, Exit state (End) will be called which signifies end of the while statement.

3.4 Do While Statement

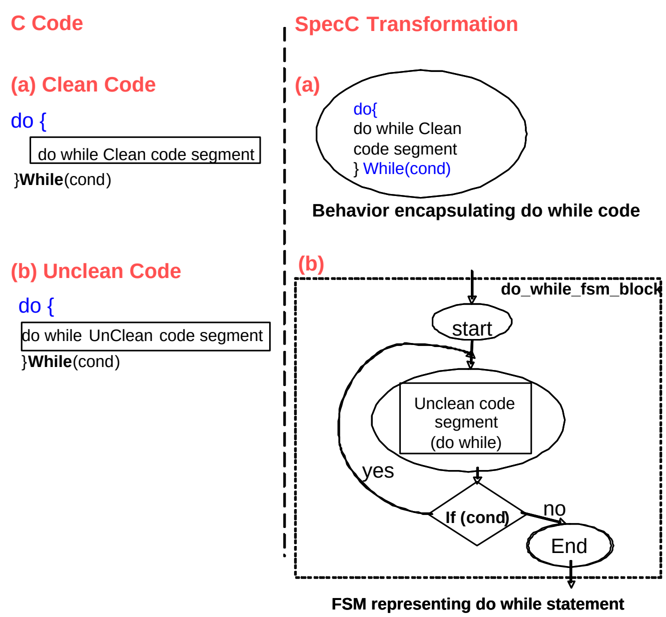

Figure 9. Do While statement

A Do While statement Figure 9 is just a small modification to the While statement Figure 8. In the While statement, the condition is checked first before executing *While (Un)clean Code Segment*. On the contrary, in the Do While statement, first the *Do While (Un)clean Code Segment* is executed and then the condition is checked.

A Do While statement Figure 9 (part a) is clean if *Do While Clean Code Segment* is a sequence of just data statements and there are no calls to other behaviors. For this type of statement, we can get valid SpecC code just by wrapping the whole block of the Do While statement as is into a leaf behavior.

A Do While statement Figure 9 (part b) is unclean if *Do While Unclean Code Segment* is a composite of data statements as well as calls to other behaviors. First step in converting this type of code is transforming *Do While Unclean Code Segment* into a composite behavior which is clean in SpecC. *If condition check* can be transformed to a *Yes/No FSM* as it resembles decision making. When executed, first the behavior representing *Do While Unclean Code Segment* is called then the If condition checked with *Yes/No FSM*. If the condition satisfies, then the behavior representing *Do While Unclean Code Segment* will be called and then the control loops back to *Yes/No FSM*. This will repeat until the condition becomes false. Then, Exit state (End) will be called which signifies end of the do while statement.

3.5 For Statement

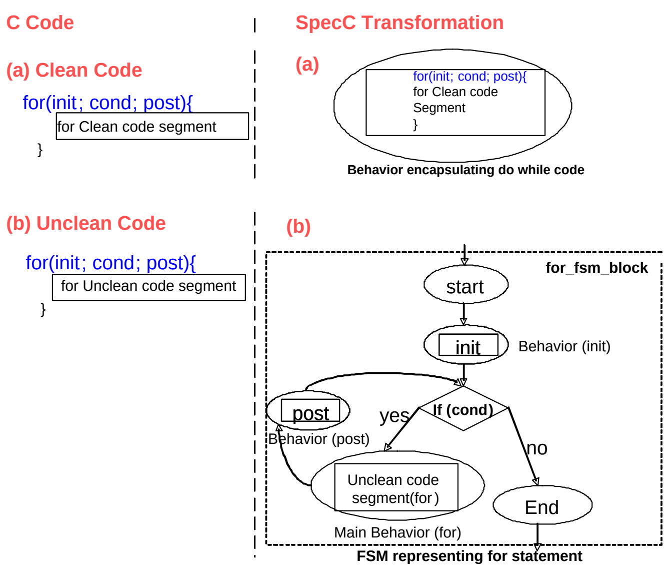

Figure 10. For statement

A For statement Figure 10 is just a small modification to the While statement Figure 8. In the While statement, there is only one block of code (*While (Un)clean Code Segment*) where as in the For statement, along with *For (Un)clean Code Segment* there are two more blocks of code. One block is *init* statements and the other is *post* statements. *Init* statements are executed once at the start of the For statement block. *Post* statements are executed everytime the *For (Un)clean Code Segment* is executed.

A For statement Figure 10 (part a) is clean if *For Clean Code Segment* is a sequence of just data statements and there are no calls to other behaviors. For this type of statement, we can get valid SpecC code just by wrapping the whole block of the For statement as is into a leaf behavior.

A For statement Figure 10 (part b) is unclean if *For Unclean Code Segment* is a composite of data statements as well as calls to other behaviors. First step in converting this type of code is transforming *For Unclean Code Segment* into a composite behavior which is clean in SpecC. Then transform init and post statements to appropriate clean SpecC behaviors. Generally, init and post statements contain some variable initialization, increment and decrement operations. So, they can be easily translated to leaf behaviors if they contain just the data statements and no calls to other behaviors. Otherwise, they are transformed to composite behaviors. *Condition check* can be transformed to a *Yes/No FSM* as it resembles decision making.

When executed, first the behavior representing *Init* statement is called once and then the *Yes/No FSM* is called. If the condition is satisfied then the behavior representing *For Unclean Code Segment* will be called followed by *Post* behavior and then the control loops back to *Yes/No FSM*. This loop will repeat until the condition becomes false. Then, Exit state (End) will be called which signifies end of the for statement.

4 Combination Of Constructs

This section deals with translating C code with various combinations of basic constructs into SpecC code.

4.1 While and If Statements (Clean)

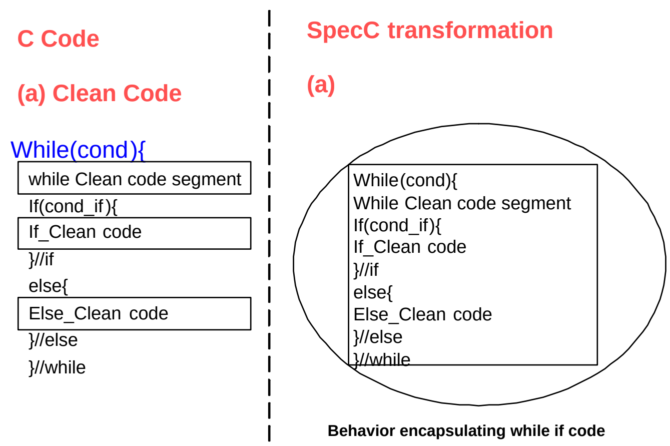

Figure 11. Combination of clean While and If statements

A While statement is combined with an If statement as the Figure 11 depicts. But both statements are clean. So, the translation is simple as we wrap both these statements into a simple leaf behavior.

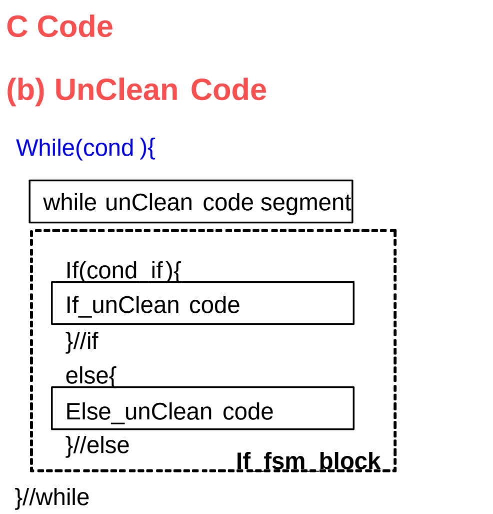

Figure 12. Combination of Unclean While and If Elsestatements

4.2 While and If Else Statements (Unclean)

In Figure 13, A While statement which is unclean is combined with an if else statement. The unclean while statement translation is done according to Figure 8 and the If Else statement translation is done according to Figure 7. Since the If Else statement is a sequence to *While Unclean Code Segment*, a new finite state (if_fsm_block) is introduced just after the behavior representing *While Unclean Code Segment*. So, the final translation is nothing but plugging the right FSMs which represent the basic building blocks at the right places.

4.3 While and If Else Statements 2(Unclean)

This combination (Figure 14) differs from the previous combination (Figure 12) in the sequence of execution of *If Else* and *While Unclean Code Segment*. As the Figure 15 illustrates, the *While Unclean Code Segment* is the sequence to the If Else statement. Appropriate changes (flipping the basic blocks) are made to the sequence of execution in Figure 15 which differs from Figure 13.

5 Example

Figure 16 depicts a complex nesting of one for loop, two Do while loops, one If Else statement and one While loop. Here, only While block has calls to other behaviors. While

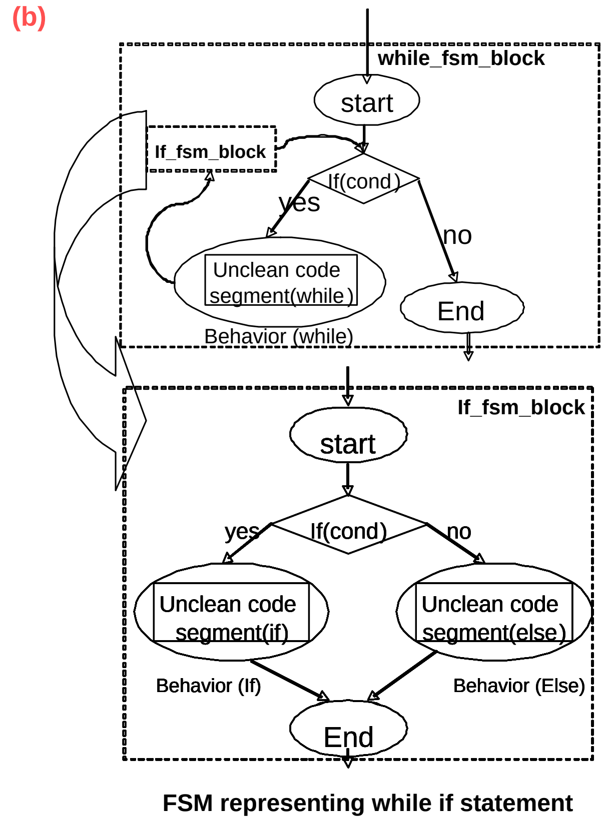

Figure 13. FSM for combination of Unclean While and If Elsestatements

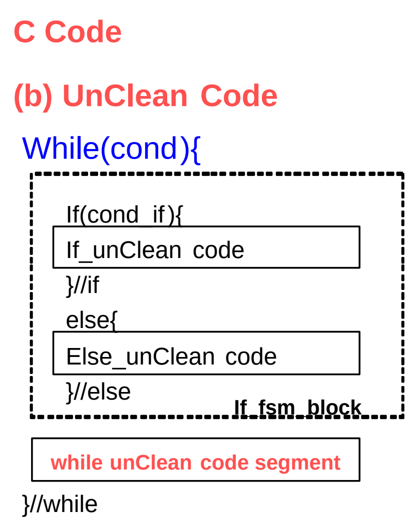

Figure 14. Second Combination of Unclean While and If Elsestatements

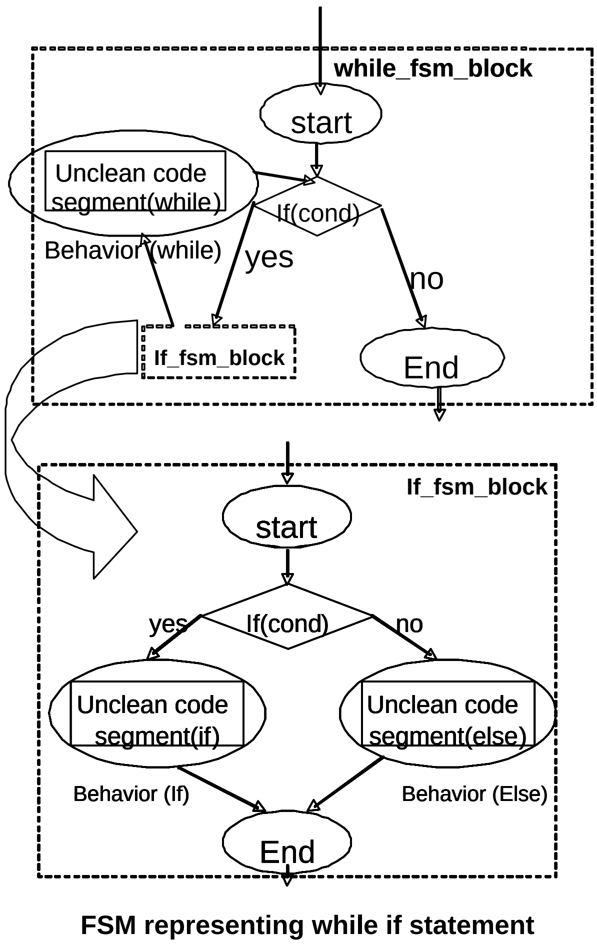

Figure 15. FSM for second Combination of Unclean While and If Elsestatements

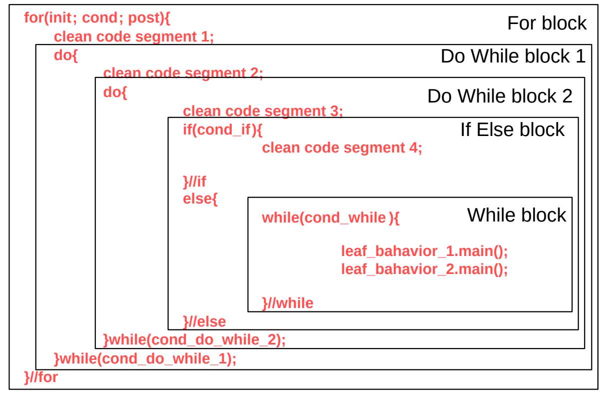

Figure 16. Complex Example of nesting

block is a Composite behavior having two sequential leaf behaviors named *leaf_behavior_1* and *leaf_behavior_2*. The code segments in all other blocks are clean as they only have data statements. But, since While block is the inner most block inside the nesting, it is propagating unclean behavior to the If Else block. The If Else block, in turn makes the Do While block 2 unclean. The Do While block 2 makes the Do While block 1 unclean which in turn makes the For block unclean. So, this is like a *ripple effect* where unclean behavior in the deepest child behavior makes the top most parent unclean.

We can adopt two approaches to get a valid SpecC code out of this huge complex nesting of different basic blocks. *Top Down* approach, where you start from the outer most block *(parent)* and then work on inner block just below the current block until you reach the innermost block.

When we apply the *Top Down* on Figure 16, ordering follows this pattern:

1. For Block
2. Do While Block 1
3. Do While Block 2
4. If Else Block
5. While Block

If we follow *Bottom Up* approach, the order is reversed. First we work on the inner most block, make it clean, then work on the its immediate parent block and so on, till we reach the top most block. We have followed the *Top Down* approach for making this complex example clean.

5.1 Translation : Step 1

While working on the top most block, we abstract the next level block and we include it as a child behavior.

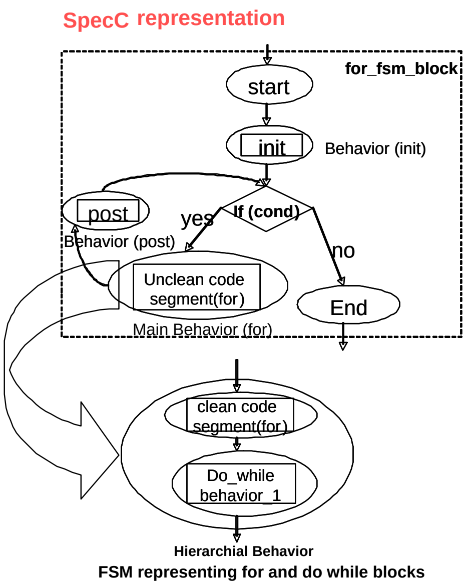

Figure 17. For Block

The For block Figure 17has a small modification from Figure 10. Figure 10 contains a behavior encapsulating a for unclean code segment but Figure 17 has a composite behavior with two sequential behaviors. One of the two sequential behaviors is the leaf behavior encapsulating *clean_code_segment_1* and the other one is *abstracted Do_While_behavior_1*.

5.2 Translation : Step 2

Do_while_Block_1 (Figure 18) is a parent to Do_while_Block_2. We can abstract the latter as a simple behavior following the execution of *clean_code_segment_2*. So, the simplest translation possible is embedding *clean_code_segment_2* into a leaf behavior and abstracting Do_while_Block_2 as a simple behavior. Finally, by modifying Figure 9 so that the hierarchial behavior reflects two sequential behaviors (clean_code_segment_2 and Do_While_block_2) in substitution of the *unclean_code_segment*, we get a valid SpecC translation.

5.3 Translation : Step 3

Step 3 (Figure 19) is similar to step 2 (Figure 18) except that *Do_while_Block_2* has *If_Else_block* as the child block.

5.4 Translation : Step 4

If_Else_block (Figure 20) has *While_block* as the child block and Figure 20 depicts the difference from Figure 7 in

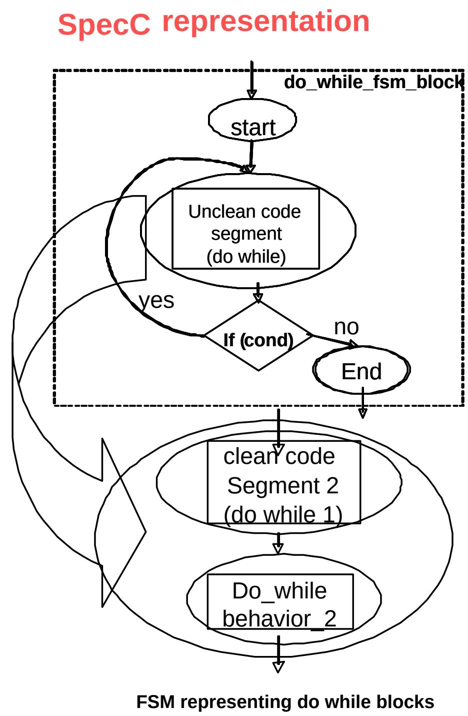

Figure 18. Do While Block 1

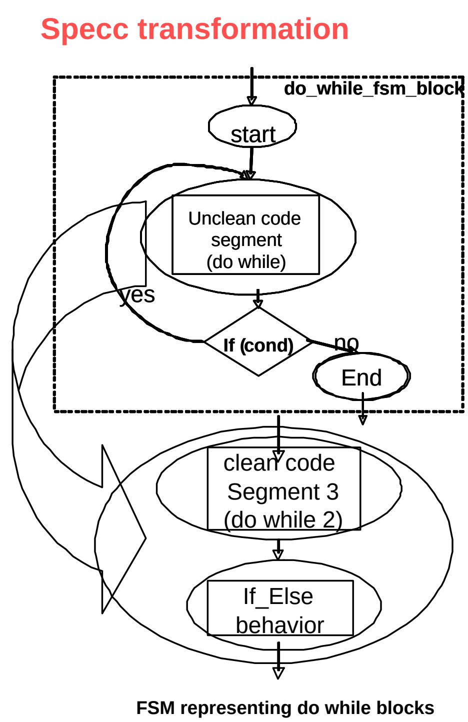

Figure 19. Do While Block 2

that *else block* is a composite sequential behavior consisting of a clean leaf behavior for *clean_code_segment_3* and an *abstracted while* (child) behavior.

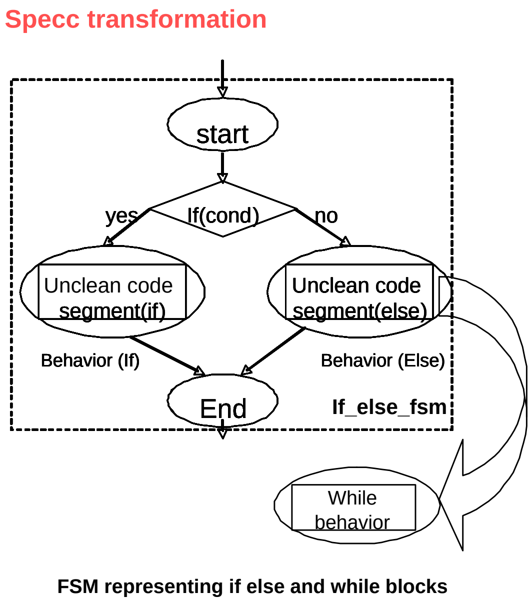

Figure 20. If Else Block

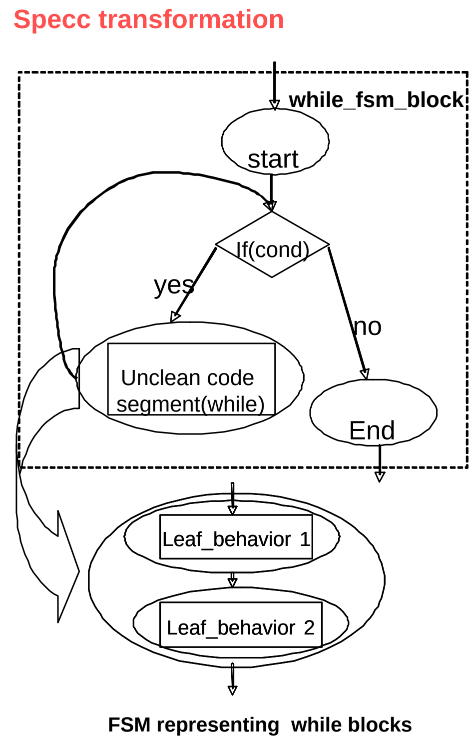

Figure 21. While Block

5.5 Translation : Step 5

Step 5 (Figure 21) depicts the inner most while block. This while block has a sequence of two behaviors named *leaf_behavior_1* and *leaf_behavior_2*. Figure 21 depicts the difference from Figure 8 in that there is a composite sequential behavior containing *leaf_behavior_1* and *leaf_behavior_2*.

6 Experimental Results

Based on the refinement rules defined in previous section, we cleaned raw JBIG [3] C code to pure SpecC code. Table 1 shows the results of the refinement from Unclean SpecC code to Clean SpecC Code.

Our input is raw C code which was about 3900 lines in total. Number of behaviors is not applicable to the raw C code. So, the immediate translation of the C code into SpecC code gave 3969 lines. But the resulting SpecC code is not clean since this it was encapsulating the raw C functions into behaviors based on some granularity decisions. Before the refinement of the unclean SpecC code, there were 29 behaviors in total, out of which 17 were *leaf* behaviors and the rest (12) were *other* behaviors. The problem with the *other* behaviors is that SpecC methodology (Figure 1) used for exploration can not analyze them properly. So, it is important to eliminate the *other* behaviors by refining them to either of *leaf or sequential, FSM and pipe* behaviors.

After the refinement, there were 85 behaviors in total out of which there were 64 *leaf*, 5 *sequential*, 16 *fsm* and 0 *other* behavior. All the *other* type of behaviors were converted to either of *leaf*, *sequential* and *fsm* behaviors. Before refinement there were no *fsm* or *sequential* type behaviors. After refinement, there were 23.5% of total behaviors were *fsm* and *sequential* behaviors.

We tested JBIG code on a small image file of size 150 X 179 pixel. The results were same for both clean and unclean codes. But there was a significant change in simulation timings. The simulation time for the raw C code was just 0.3 sec where as the unclean SpecC code took 0.7 seconds and the clean SpecC code took 1.1 seconds for the clean code. This was the result of introducing more behaviors in the process of making the code cleaner.

An interesting observation was number of lines of code and size of the code has increased after each refinement. This was an effect of defining more leaf, fsm and sequential behaviors in the process of cleaning the code. *Appendix* contains a sample of code converted from C to SpecC with some hierarchial tree diagrams and code before and after the transformation.

Description	**Raw C Code**	**Unclean SpecC code**	**Clean SpecC code**
Formatted lines of code	3900	3969	5238
Formatted code size	77,448 bytes	155,756 bytes	184,297 bytes
Number of behaviors	-	29	85
... classified as 'leaf'	-	17 (58.6%)	64 (75.3%)
... classified as 'sequential'	-	0 (0%)	5 (5.9%)
... classified as 'fsm'	-	0 (0%)	16 (18.8%)
... classified as 'other'	-	12 (41.4%)	0 (0%)
Simulation time	0.3 sec	0.6 sec	1.0 sec
Simulation results	same	same	same
Ready for Analysis	no	partly	completely

Table 1. JBIG experimental results.

The original raw C code was not analyzable where as part of the unclean SpecC code could be used for analysis. But the clean SpecC code is completely analyzable.

7 Conclusion and future work

In this paper, We presented the refinement rules and algorithms for transforming an unclean specification model into a clean specification model in our design methodology. We suggested a set of rules for conversion that facilitates an efficient approach to derive a clean specification from an unclean specification model. We tested our set of conversion guidelines on JBIG specification code which was impossible for analysis using SpecC methodology. Experiments were performed to support our methodology. The methodology might increase of productivity of the designers by relieving them from tedious and error-prone task of rewriting models. For the future, we aim at automating conversion from unclean specification model to clean specification using our design methodology.

References

[1] Daniel D. Gajski et al., *System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers, 2001.

[2] A.Gerstlauer, *SpecC Modeling Guidelines*, University of California, Irvine, Technical Report ICS-TR-00-xx, September, 1998.

[3] Junyu Peng, Lukai Cai, Anand Selka, Daniel D. Gajski, *Design of a JBIG Encoder using SpecC Methodology*, University of California, Irvine, Technical Report ICS-TR-00-13, June 2000.

[4] Lukai Cai, Daniel D. Gajski, *C/C++ Based System Design Flow Using SpecC, VCC and SystemC*, University of California, Irvine, Technical Report CECS-02-30 June, 2002.

A. UnClean SpecC code for a sample file

A.1 Statistics

```
Statistics
----------
% sir_stats sde_diff_encode_line.sir

Design name .......................... : sde_diff_encode_line

  Formatted lines of code ............ :     2278

  Formatted code size ................ :   116829 bytes

Number of behaviors .................. :        5

  ... classified as 'leaf' ........... :        4 (80.0%)

  ... classified as 'other' .......... :        1 (20.0%)

Number of channels ................... :        0

Number of interfaces ................. :        0
```

A.2 Tree

```
Hierarchy Tree
--------- ----
% sir_tree -bclt sde_diff_encode_line_unclean.sir

  class type
          is one of [BC], indicating behavior (B) or channel (C).

     storage class
          is intern or extern (one of [ix]), indicating  internal
          class  with  known  body  (i),  or  external class with
          unknown body (x).

     classification
          is one of [cfhlopstwx], indicating for behaviors:  con-
          current  (c),  FSM (f), leaf (l), pipeline (p), sequen-
          tial (s), exception (t), external (x),  or  other  (o);
          for  channels: leaf (l), hierarchical (h), wrapper (w),
          external (x), or other (o).


class type  storage class  classification  description

B                i                l           behavior adaptive_template

B                i                l           behavior arith_encode

B                i                l           behavior deterministic_prediction

B                i                l           behavior model_templates

B                i                o           behavior sde_diff_encode_line
```

A.3 The UnClean code

```

#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "deterministic_prediction";
import "adaptive_template";
import "model_templates";

behavior sde_diff_encode_line(in struct local_data *ld,
                     in struct jbg_enc_state *s,
                 in unsigned long stripe,
                     in int layer,
                 in int plane)
{  int int1, int2, flag, options, at_determined, count, cx, tx;
   unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
   unsigned *t, mx;
 struct jbg_arenc_state *par1;



deterministic_prediction deterministic_prediction_exec(options, y, j, l1,l2,l3,
            h1, h2, h3, flag);
adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx,
 count,flag);

arith_encode arith_encode_exec(par1, int1, int2);
model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

void main(void){
      ld->line_h1 = ld->line_h2 = ld->line_h3 = ld->line_l1 = ld->line_l2 = ld->line_l3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) {
    ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;
    ld->line_l3 = (long)*(ld->lp2 - ld->lbpl) << 8;
      }
      ld->line_l2 = (long)*ld->lp2 << 8;
      ld->line_l1 = (long)*ld->lp1 << 8;

      /* encode line */
      for (ld->j = 0; ld->j < ld->hx; ld->lp1++, ld->lp2++) {
    if ((ld->j >> 1) < ld->lbpl * 8 - 8) {
      if (ld->y > 1)
        ld->line_l3 |= *(ld->lp2 - ld->lbpl + 1);
      ld->line_l2 |= *(ld->lp2 + 1);
      ld->line_l1 |= *(ld->lp1 + 1);
    }
    do {
```

```
/*
      assert(ld->hp - (s->lhp[s->highres[plane]][plane] +
               (stripe * ld->hl + ld->i) * ld->hbpl)
         == (ptrdiff_t) ld->j >> 3);

      assert(ld->lp2 - (s->lhp[1-s->highres[plane]][plane] +
            (stripe * ld->ll + (ld->i>>1)) * ld->lbpl)
         == (ptrdiff_t) ld->j >> 4);
*/
      ld->line_h1 |= *(ld->hp++);
      if (ld->j < ld->hbpl * 8 - 8) {
        if (ld->y > 0) {
          ld->line_h2 |= *(ld->hp - ld->hbpl);
          if (ld->y > 1)
        ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl);
        }
      }
      do {
        ld->line_l1 <<= 1;  ld->line_l2 <<= 1;  ld->line_l3 <<= 1;
        if (ld->ltp && s->tp[ld->j >> 1] < 2) {
          /* pixel are typical and have not to be encoded */
          ld->line_h1 <<= 2;  ld->line_h2 <<= 2;  ld->line_h3 <<= 2;

          /*#ifdef DEBUG
          do {
        ++tp_pixels;
          } while (++(ld->j) & 1 && (ld->j) < hx);
          #else */
          (ld->j) += 2;
          /*          #endif */

        } else
          do {

        ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;

            options=s->options;
        y=ld->y;
        j=ld->j;
        l1=ld->line_l1;
        l2=ld->line_l2;
        l3=ld->line_l3;
        h1=ld->line_h1;
        h2=ld->line_h2;
        h3=ld->line_h3;

        deterministic_prediction_exec.main();

        if (flag==1){
          continue;
        }
        else{
```

```
                  y=ld->y;
          j=ld->j;
          l1=ld->line_l1;
          l2=ld->line_l2;
          l3=ld->line_l3;
          h1=ld->line_h1;
          h2=ld->line_h2;
          h3=ld->line_h3;
          tx=s->tx[plane];
                  flag=5;

          model_templates_exec.main();
                  ld->cx=cx;

                  par1=ld->se;
          int1=ld->cx;
                  int2=(ld->line_h1 >> 8) & 1;

          arith_encode_exec.main();
        /*#ifdef DEBUG
        encoded_pixels++;
        #endif*/

        /*diff_adaptive_template(ld, s);*/
          at_determined=ld->at_determined;
          j=ld->j;
          h1=ld->line_h1;
          h2=ld->line_h2;
          t=&(ld->t);
          c=ld->c;
                  c_all=&(ld->c_all);
                  mx=s->mx;
          hx=ld->lx;
                  count=3;
          flag=1;
          adaptive_template_exec.main();

        }
            } while (++(ld->j) & 1 && (ld->j) < ld->hx);
      } while ((ld->j) & 7 && (ld->j) < ld->hx);
    } while ((ld->j) & 15 && (ld->j) < ld->hx);
      } /* for (j = ...) */

      /* low resolution pixels are used twice */
      if (((ld->i) & 1) == 0) {
    ld->lp1 -= ld->lbpl;
    ld->lp2 -= ld->lbpl;
      }
}
};
```

B. Clean SpecC code for a sample file

B.1 Statistics

```
Statistics
----------
% sir_stats sde_diff_encode_line_clean.sir
Design name ........................... : sde_diff_encode_line
  Formatted lines of code ............. :     2677
  Formatted code size ................. :   124765 bytes
Number of behaviors ................... :       30
  ... classified as 'leaf' ............ :       29 (96.7%)
  ... classified as 'fsm' ............. :        1 ( 3.3%)
Number of channels .................... :        0 Number of
interfaces .................            :        0
```

B.2 Tree

```
Hierarchy Tree
--------- -----

% sir_tree -bclt sde_diff_encode_line_clean.sir

  class type
          is one of [BC], indicating behavior (B) or channel (C).

     storage class
          is intern or extern (one of [ix]), indicating  internal
          class  with  known  body  (i),  or  external class with
          unknown body (x).

     classification
          is one of [cfhlopstwx], indicating for behaviors:  con-
          current  (c),  FSM (f), leaf (l), pipeline (p), sequen-
          tial (s), exception (t), external (x),  or  other  (o);
          for  channels: leaf (l), hierarchical (h), wrapper (w),
          external (x), or other (o).

{\bf class type, storage class, classification, description}

B i l   behavior assign_ld

B i l   behavior increment_ld_hp

B i l   behavior increment_ld_i

B i l   behavior increment_ld_j

B i l   behavior increment_ld_y

B i l   behavior increment_long_plus_plus

B i l   behavior increment_plus_plus

B i l   behavior init_ld_i

B i l   behavior init_long_to_zero

B i l   behavior init_to_zero

B i f   behavior sde_diff_encode_line
B i l   |------ sde_diff_encode_line_init init
B i l   |------ init_ld_j init_j
B i l   |------ dummy for_loop_repeat
B i l   |------ sde_diff_encode_line_leaf_1 leaf1
B i l   |------ dummy do_while_loop_1
B i l   |------ dummy do_while_loop_1_repeat
B i l   |------ increment_ld_lp1 increment_lp1
```

```
B i l   |------ increment_ld_lp2 increment_lp2
B i l   |------ sde_diff_encode_line_leaf_2 leaf2
B i l   |------ dummy do_while_loop_2
B i l   |------ dummy do_while_loop_2_repeat
B i l   |------ sde_diff_encode_line_leaf_3 leaf3
B i l   |------ sde_diff_encode_line_leaf_5 leaf5
B i l   |------ dummy do_while_loop_3_repeat
B i l   |------ dummy do_while_loop_3
B i l   |------ deterministic_prediction_init deterministic_prediction_init_exec
B i l   |------ deterministic_prediction deterministic_prediction_exec
B i l   |------ model_templates_init model_templates_init_exec
B i l   |------ model_templates model_templates_exec
B i l   |------ sde_diff_encode_line_arith_encode_init sde_diff_encode_line_arith_encode_init_
B i l   |------ arith_encode arith_encode_exec
B i l   |------ adaptive_template_init adaptive_template_init_exec
B i l   |------ adaptive_template adaptive_template_exec
B i l   \------ sde_diff_encode_line_leaf_4 leaf4

B i l   behavior sde_encode_diff_enable_flag
```

B.3 The Clean code

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "deterministic_prediction";
import "adaptive_template";
import "model_templates";
import "definitions";

behavior  sde_diff_encode_line_init(in struct local_data *ld){
    void main(){

      ld->line_h1 = ld->line_h2 = ld->line_h3 = ld->line_l1 = ld->line_l2 = ld->line_l3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) {
    ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;
    ld->line_l3 = (long)*(ld->lp2 - ld->lbpl) << 8;
      }
      ld->line_l2 = (long)*ld->lp2 << 8;
      ld->line_l1 = (long)*ld->lp1 << 8;
      }
};

behavior sde_diff_encode_line_leaf_1(in struct local_data *ld){
    void main(){

    if ((ld->j >> 1) < ld->lbpl * 8 - 8) {
      if (ld->y > 1)
        ld->line_l3 |= *(ld->lp2 - ld->lbpl + 1);
      ld->line_l2 |= *(ld->lp2 + 1);
      ld->line_l1 |= *(ld->lp1 + 1);
    }
    }
};

behavior sde_diff_encode_line_leaf_2(in struct local_data *ld){
    void main(){

      ld->line_h1 |= *(ld->hp++);
      if (ld->j < ld->hbpl * 8 - 8) {
        if (ld->y > 0) {
          ld->line_h2 |= *(ld->hp - ld->hbpl);
          if (ld->y > 1)
        ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl);
        }
      }
    }
};
```

```
behavior sde_diff_encode_line_leaf_3(in struct local_data *ld, in struct jbg_enc_state *s){
    void main(){
        ld->line_l1 <<= 1;  ld->line_l2 <<= 1;  ld->line_l3 <<= 1;
    }
};

behavior sde_diff_encode_line_leaf_5(in struct local_data *ld, in struct jbg_enc_state *s){
    void main(){
          /* pixel are typical and have not to be encoded */
          ld->line_h1 <<= 2;  ld->line_h2 <<= 2;  ld->line_h3 <<= 2;
          (ld->j) += 2;
    }
};

behavior sde_diff_encode_line_leaf_4(in struct local_data *ld){
    void main(){
      /* low resolution pixels are used twice */
      if (((ld->i) & 1) == 0) {
    ld->lp1 -= ld->lbpl;
    ld->lp2 -= ld->lbpl;
      }

    }
};

behavior deterministic_prediction_init(in struct local_data *ld,
                     in struct jbg_enc_state *s,
                 out int options,
                 out unsigned long  y,
                 out unsigned long  j,
                 out unsigned long  l1,
                 out unsigned long  l2,
                 out unsigned long  l3,
                 out unsigned long  h1,
                 out unsigned long  h2,
                 out unsigned long  h3,
                 in int flag){
    void main(){
        ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;

            options=s->options;
        y=ld->y;
        j=ld->j;
        l1=ld->line_l1;
        l2=ld->line_l2;
        l3=ld->line_l3;
        h1=ld->line_h1;
```

```
        h2=ld->line_h2;
        h3=ld->line_h3;
    }
};

behavior model_templates_init(   in struct local_data *ld,
                 in struct jbg_enc_state *s,
                 in int plane,
                 out unsigned long  y,
                 out unsigned long  j,
                 out unsigned long  l1,
                 out unsigned long  l2,
                 out unsigned long  l3,
                 out unsigned long  h1,
                 out unsigned long  h2,
                 out unsigned long  h3,
                 out int tx,
                 out int flag,
                 in int cx){
    void main(){
                  y=ld->y;
          j=ld->j;
          l1=ld->line_l1;
          l2=ld->line_l2;
          l3=ld->line_l3;
          h1=ld->line_h1;
          h2=ld->line_h2;
          h3=ld->line_h3;
          tx=s->tx[plane];
                  flag=5;
    }
};

behavior sde_diff_encode_line_arith_encode_init(in struct local_data *ld,
               out struct jbg_arenc_state *par1,
               out int int1,
               out int int2,
               in int cx){
    void main(){

                  ld->cx=cx;
                  par1=ld->se;
          int1=ld->cx;
                  int2=(ld->line_h1 >> 8) & 1;
    }
};

behavior  adaptive_template_init(in struct local_data *ld,
                 in struct jbg_enc_state *s,
                 out int at_determined,
                 out unsigned long  j,
                 unsigned long  h1,
                 out unsigned long  h2,
```

```
                 out unsigned *t,
                 out unsigned long  *c,
                 unsigned long  *c_all,
                 out unsigned mx,
                 out unsigned long  hx,
                 out int count,
                 out int flag
                ){
    void main(){

          at_determined=ld->at_determined;
          j=ld->j;
          h1=ld->line_h1;
          h2=ld->line_h2;
          t=&(ld->t);
          c=ld->c;
                  c_all=&(ld->c_all);
                  mx=s->mx;
          hx=ld->lx;
                  count=3;
          flag=1;
    }
};

behavior sde_diff_encode_line(in struct local_data *ld,
                     in struct jbg_enc_state *s,
                 in unsigned long stripe,
                     in int layer,
                 in int plane)
{  int int1, int2, flag, options, at_determined, count, cx, tx;
   unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
   unsigned *t, mx;
 struct jbg_arenc_state *par1;


sde_diff_encode_line_init init(ld);
sde_diff_encode_line_leaf_1 leaf1(ld);
sde_diff_encode_line_leaf_2 leaf2(ld);
sde_diff_encode_line_leaf_3 leaf3(ld, s);
sde_diff_encode_line_leaf_4 leaf4(ld);
sde_diff_encode_line_leaf_5 leaf5(ld, s);

init_ld_j init_j(ld);
increment_ld_lp1 increment_lp1(ld);
increment_ld_lp2 increment_lp2(ld);

deterministic_prediction_init deterministic_prediction_init_exec(ld, s, options, y, j, l1, l2,
deterministic_prediction deterministic_prediction_exec(options, y, j, l1,l2,l3, h1, h2, h3, fl
adaptive_template_init adaptive_template_init_exec(ld, s, at_determined, j, h1, h2, t, c, c_al
adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx, count,

sde_diff_encode_line_arith_encode_init sde_diff_encode_line_arith_encode_init_exec(ld, par1, i
arith_encode arith_encode_exec(par1, int1, int2);
```

```
model_templates_init model_templates_init_exec(ld, s, plane, y,j, l1, l2, l3, h1, h2, h3, tx,
model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

dummy for_loop_repeat, do_while_loop_1, do_while_loop_1_repeat,
do_while_loop_2, do_while_loop_2_repeat;
dummy do_while_loop_3, do_while_loop_3_repeat;

void main(void){

  fsm{
    init : goto init_j;
    init_j : goto for_loop_repeat;
    for_loop_repeat : if(ld->j < ld->hx) goto leaf1; goto leaf4;
    leaf1 : goto do_while_loop_1;
    do_while_loop_1 : goto leaf2;
    do_while_loop_1_repeat : if ((ld->j) & 15 && (ld->j) < ld->hx) goto
    do_while_loop_1; goto increment_lp1;
    increment_lp1 : goto increment_lp2;
    increment_lp2 : goto for_loop_repeat;
    leaf2 : goto do_while_loop_2;
    do_while_loop_2 : goto leaf3;
    do_while_loop_2_repeat : if ((ld->j) & 7 && (ld->j) < ld->hx) goto
    do_while_loop_2; goto do_while_loop_1_repeat;
    leaf3 : if ((ld->ltp && s->tp[ld->j >> 1] < 2)) goto leaf5; goto
    do_while_loop_3;
    leaf5 : goto do_while_loop_2_repeat;
    do_while_loop_3_repeat : if(++(ld->j) & 1 && (ld->j) < ld->hx) goto
    do_while_loop_3; goto do_while_loop_2_repeat;
    do_while_loop_3 : goto deterministic_prediction_init_exec;
    deterministic_prediction_init_exec : goto deterministic_prediction_exec;
    deterministic_prediction_exec : if(flag != 1) goto
    model_templates_init_exec; goto  do_while_loop_3_repeat;
    model_templates_init_exec : goto model_templates_exec;
    model_templates_exec : goto sde_diff_encode_line_arith_encode_init_exec;
    sde_diff_encode_line_arith_encode_init_exec : goto arith_encode_exec;
    arith_encode_exec : goto adaptive_template_init_exec;
    adaptive_template_init_exec : goto adaptive_template_exec;
    adaptive_template_exec : goto do_while_loop_3_repeat;

    leaf4 : break;
  }
}
};
```