SCE Specification Model Reference Manual

Andreas Gerstlauer
Rainer Dömer

System-On-Chip Environment (SCE)
Version 2.2.0 beta
April 12, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

http://www.cecs.uci.edu

SCE Specification Model Reference Manual

Andreas Gerstlauer
Rainer Dömer

System-On-Chip Environment (SCE)
Version 2.2.0 beta
April 12, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

gerstl@cecs.uci.edu
doemer@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

This manual describes and defines the specification model that forms the input to system design flow in the System-On-Chip Environment (SCE). The specification model is captured by the designer and describes the desired system behavior in a purely functional way. As the starting point for the system-level design process, it will later be gradually refined through a series of interactive and automated steps down to an actual implementation. Therefore, having a clear and unambigious input model as defined by this manual is crucial for effective design space exploration.

Following some general guidelines for developing efficient specification models, detailed rules that define the specification modeling style are given. In case of SCE, specification models are written in version 2.0 of the SpecC language. On top of the basic SpecC syntax and semantics, the modeling rules impose additional restrictions for a proper and valid specification model. Finally, steps required to convert an existing C description into a specification model are outlined and some conversion examples are given.

Contents

1 Introduction **1**

1.1 SCE Methodology 1
1.2 SpecC Language 3
1.3 Specification Model 4

2 Modeling Guidelines **5**

2.1 Computation 5
2.1.1 Granularity 6
2.1.2 Hierarchy 6
2.1.3 Encapsulation 6
2.1.4 Concurrency 7
2.1.5 Time 8
2.2 Communication 8
2.2.1 Semantics 8
2.2.2 Dependencies 9

3 Modeling Style **10**

3.1 Computation 12
3.1.1 Leaf Behaviors 12
3.1.2 Hierarchical Behaviors 13
3.2 Communication 14
3.2.1 Behavior Interfaces (Ports) 14
3.2.2 Connectivity (Variables and Channels) 15

4 C Code Conversion 16
4.1 Code Refinement . . . 16
4.1.1 Syntactic Refinement . . . 16
4.1.2 Semantic Refinement . . . 17
4.2 Basic Constructs . . . 17
4.2.1 If (no else) Statement . . . 18
4.2.2 If Else Statement . . . 18
4.2.3 While Statement . . . 20
4.2.4 Do While Statement . . . 20
4.2.5 For Statement . . . 22
4.3 Composite Constructs . . . 23
4.3.1 Clean While and If Statements . . . 23
4.3.2 Unclean While and If Else Statements . . . 23
4.4 Example . . . 25
4.4.1 Translation: Step 1 . . . 27
4.4.2 Translation: Step 2 . . . 28
4.4.3 Translation: Step 3 . . . 28
4.4.4 Translation: Step 4 . . . 30
4.4.5 Translation: Step 5 . . . 30

Acknowledgments 32

References 33

List of Figures

1.1 SCE design flow. 2

3.1 Specification model top-level structure. 10
3.2 Specification model top-level code. 11

4.1 If statement. 18
4.2 If Else statement. 19
4.3 While statement. 20
4.4 Do While statement. 21
4.5 For statement. 22
4.6 Combination of clean While and If statements. 23
4.7 Combination of unclean While and If Else statements. 24
4.8 FSM for combination of unclean While and If Else statements. 24
4.9 Second combination of unclean While and If Else statements. 25
4.10 FSM for second combination of unclean While and If Else statements. . . . 26
4.11 Complex example of nesting. 27
4.12 Step 1 - For Block. 28
4.13 Step 2 - Do While Block 1. 29
4.14 Step 3 - Do While Block 2. 29
4.15 Step 4 - If Else Block. 30
4.16 While Block. 31

Chapter 1

Introduction

The System-On-Chip design environment (SCE) is an example of an implementation of a state-of-the-art system-level design methodology [1]. SCE is a framework that combines a set of tools under a common graphical user interface (GUI). Using this framework, the designer can take an initial specification down to an actual implementation through a series of interactive and automated steps. Starting from a purely functional description of the desired system behavior, an implementation of the design on a heterogoeneous system architecture with multiple processing elements (PEs) connected through system busses is produced at the end of the design flow.

1.1 SCE Methodology

The SCE system-level design methodology is shown in Figure 1.1. The SCE methodology is a set of four models and three transformation steps that take a system specification down to an RTL implementation [1].

The system design flow consists of two main parts: (a) system synthesis, and (b) a backend for hardware and software synthesis. In the SCE methodology, system synthesis is further subdivided into two orthogonal tasks, architecture exploration and communication synthesis. Architecture exploration implements the computation behavior of the specification on a set of processing elements that form the system architecture. Communication synthesis, on the other hand, implements the communication functionality of the specifica-

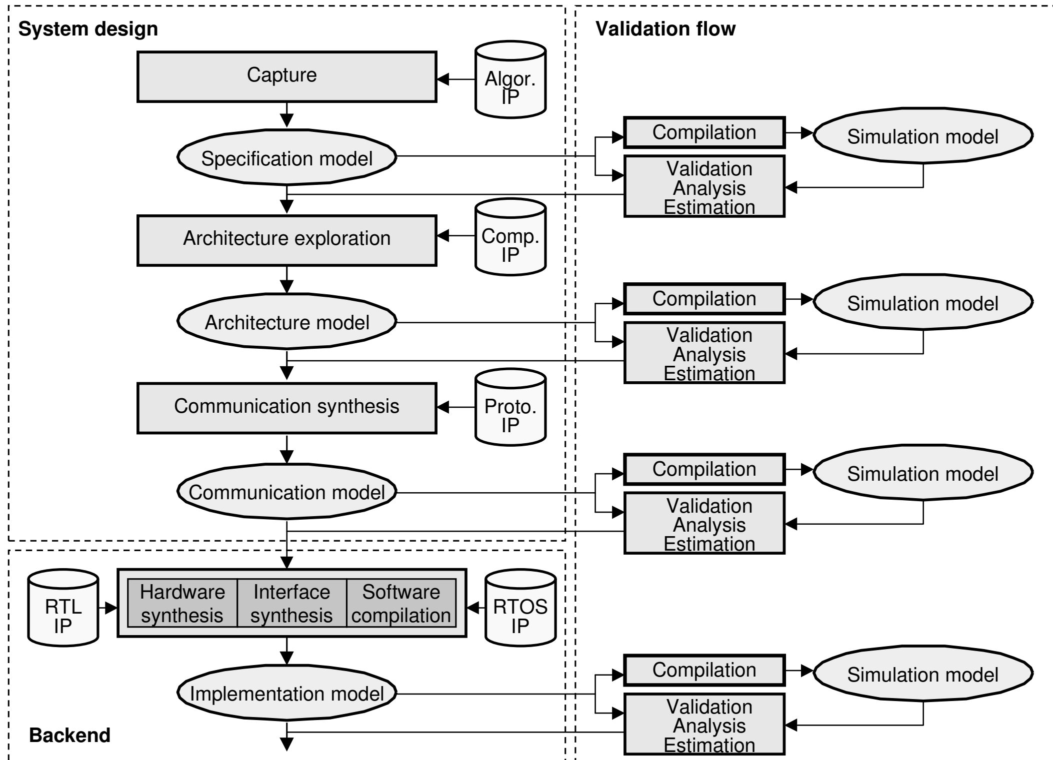

Figure 1.1: SCE design flow.

tion over the system busses.

Each system synthesis and backend task refines the model of the design at the current stage of the design process into a new model representing the details of the implementation added during the synthesis step. At the output of each task, the model of the design reflects the implementation decisions made in the previous step. At the same time, each model forms the input to the next task.

The system-level design process starts off with a specification of the desired system behavior. This *specification model* is written by the user and forms the input to the design process.

In the SCE methodology, the first task of system synthesis is architecture exploration. Architecture exploration selects a set of processing elements and maps the computation behavior of the specification onto the PEs. Architecture exploration refines the specification

model into the intermediate architecture model. The architecture model describes the PE structure of the system architecture and the mapping of computation behaviors onto the PEs, including estimated execution times for the behavior of each PE.

Architecture exploration is followed by communication synthesis to complete the system synthesis process. Communication synthesis selects a set of system busses and protocols, and maps the communication functionality of the specification onto the system busses. Communication synthesis creates the communication model which reflects the bus architecture of the system and the mapping of communication onto the busses.

The communication model is the result of the system synthesis process. It describes the structure of the system architecture consisting of PEs and busses, and the implementation of the system functionality on this architecture. It is timed in both computation and communication, i.e. simulation detail is increased by events for estimated execution and communication delays.

The communication model is a structural view at the system level. At the same time, the specification of the functionality of each PE of the system in the form of a behavioral view at the register-transfer level forms the input to the RTL synthesis of those components in the backend. In a hierarchical fashion, each PE is synthesized separately in the backend and the behavioral view of the PE is replaced with a structural view of its RTL or instruction-set (IS) microarchitecture. The result of this backend process is the implementation model.

The implementation model is a cycle-accurate, structural description of the RTL/IS architecture of the whole system. In a hierarchical fashion, the implementation model describes the system structure and the RTL structure of each PE in the system. Simulation detail is increased down to the clock level, i.e. the timing resolution is in terms of clock events for each local PE clock.

1.2 SpecC Language

The SCE methodology is supported by the SpecC system-level design language [2]. The SpecC language as an example of a modern system-level design language (SLDL) was developed under support and control of the SpecC Technology Open Consortium (STOC) [3] to satisfy all the requirements for an efficient formal description of the models in the SCE methodology.

In the SpecC methodology, all four models of the design process starting with the specification model and down to the implementation model are described in the SpecC language. One common language removes the need for tedious translation. Furthermore, all the models in SpecC are executable which allows for validation through simulation, reusing one single testbench throughout the whole design flow. In addition, the formal nature of the models enables application of formal methods, e.g. for verification or equivalence checking.

1.3 Specification Model

As outlined previosuly, the specification model is the input to the SCE design flow. It is captured graphically or textually by the designer in the form of SpecC code to specify the system functionality to implement. All other models of the SCE design flow will be generated automatically from the specification model through a sequence of interactive, GUI-assisted refinement steps. As such, the specification model needs to precisely and unambiously describe the desired system behavior on top of the SpecC language semantics. Furthermore, the specification model defines the possible design space for exploration and quality of implementation results therefore depends to a large extent on the characteristics of the specification model. For example, any premature references to implementation detail will prevent exploration of solutions outside of the scope imposed by such restrictions.

In this manual, we define how to describe a proper specification model in SpecC. First, a set of general guidelines for writing good specification models will be given in Chapter 2. Then, in Chapter 3, specific and detailed rules and restrictions imposed on the specification model style are defined. Finally, Chapter 4 describes how straight-line C code can be efficiently converted into a SpecC specification model.

Chapter 2

Modeling Guidelines

The specification model is the input of the design flow. It is captured by the user to specify the desired system functionality. The specification is a behavioral view of the system, i.e. it describes the desired functionality in an abstract manner. The specification model is a purely functional model, free of any implementation details. Therefore, objects at the specification level are abstract entities that do not correspond to real physical components.

A key aspect of the specification model is to separate computation from communication. On the one hand, this is a requirement for composability of a system out of components including the reuse of pre-existing IP components. On the other hand, this separation of concerns allows to implement computation and communication in two separate steps of the design flow.

2.1 Computation

In terms of computation, the specification is hierarchically composed of SpecC behaviors. Behaviors are arranged sequentially, concurrently, or in a mix of both, i.e. in a pipelined fashion. Behaviors at the leaves of the hierarchy contain basic algorithms in the form of straight-line C code that perform arithmetic and logical operations on data. In addition to temporary data, leaf behaviors will encapsulate any permanent storage required by the algorithm.

2.1.1 Granularity

The basic, indivisible units of granularity for design space exploration are SpecC behaviors. That is, during the design process the specification will be partitioned along behavior boundaries but behaviors at the leaves of the hierarchy form the smallest, indivisible units for exploration. Therefore, leaf behaviors contain basic algorithms in the form of C code, reading from their inputs, processing a data set, and producing outputs.

Algorithms of the specification model are split into leaf behaviors along the boundaries defined between reading and writing of data structures. On the other hand, all the code needed to process a complete, consistent data set is kept together in one leaf behavior.

Also, the ratio of communication to computation should be minimized yet the size of the leaf behaviors be kept small and manageable with well-defined, sensible interfaces and possible reuse in mind. As a rule of thumb, what would be a traditional C function will become a leaf behavior with typically half a page to maximally two pages of code.

2.1.2 Hierarchy

At each level of hierarchy, the system is composed of self-contained blocks with well-defined interfaces enabling easy composition, rearrangement, and reuse. Closely related functionality is grouped through hierarchy. Higher-level behaviors encapsulate tightly coupled groups of subbehaviors such that the ratio of external to internal communication is minimized. On the other hand, the number of subbehaviors per parent should be kept small and manageable. As a guideline, behaviors typically have 2-5 children on average.

At each level, the behavior hierarchy has to be clean. Different behavioral concepts must not be mixed in the same level. A behavior is either a hierarchical composition of subbehaviors or a leaf behavior with sequential code. Similarly, a hierarchical behavior is either a sequential, parallel, pipelined or FSM composition of subbehaviors but does not contain arbitrary C code.

2.1.3 Encapsulation

Information in the specification model has to be localized as much as possible. This includes code (functions, methods), storage (variables), and communication (port variables,

channels). Each hierarchical unit (behavior) encapsulates and abstracts as many local details as possible, hiding them from the higher levels. Hierarchical behaviors encapsulate dependencies and communication of a group of subbehaviors, providing only an interface to their combined functionality.

At the leaves, behaviors encapsulates all the code and storage needed by the algorithm. As mentioned above, global, static variables become member variables of the leaf behavior. Furthermore, global functions that are called out of leaf behaviors have to be avoided unless they represent basic operations equivalent to built-in operators. Instead, depending on size and number of callers, functions are converted into separate leaf behaviors that get instantiated as subbehaviors of the caller, or global functions are moved into the calling behavior where they become local methods. As mentioned above, an exception are small helper functions with a few lines of code that are used ubiquitously and can be considered basic operations (on the same level as additions or multiplications).

2.1.4 Concurrency

Any concurrency available between independent behaviors in the specification has to be exposed through their parallel or pipelined composition. That is, all behaviors that do not have any control or data dependencies (or data dependencies only across iterations) are arranged to execute in a concurrent fashion. Furthermore, the behavior hierarchy is constructed in such a way as to maximize the number of independent behaviors and hence the available parallelism.

Dependent behaviors, on the other hand, are not arranged in a concurrent fashion. Instead, their dependencies are captured explicitly through transitions. An exception are rare (control) dependencies between otherwise highly independent top-level tasks, for example. In those cases, communication and synchronization are modeled using channels between the tasks.

Concurrent behaviors in the specification model reflect the available parallelism in the specification. Therefore, they should be as independent as possible. Data or control dependencies between behaviors at the specification level are explicitly captured through the behavior hierarchy. Instead of concurrent behaviors that communicate or synchronize through variables or events, the behaviors are split into independent parts that can run in

parallel and dependent parts that have to be executed sequentially.

2.1.5 Time

The specification model is untimed and all behaviors execute in zero logical time. Therefore, the only events in the system are events for synchronization in order to specify causality. The ordering of events in the system is based on causal relationships only and there is no notion of time. The system is partially ordered based on causality as determined by the explicit or implicit dependencies between behaviors. As the design flow progresses, timing information that will be added to the system will successively introduce additional order based on delays.

Apart from the untimed behavior, however, the specification model can contain constraints for execution times of parts of the specification. During the design process, it then has to be assured that any time introduced into the model does not violate any of the constraints.

2.2 Communication

In terms of communication, exchange of data between behaviors in the specification model is encapusulated into SpecC channels that connect behaviors through ports. Channels describe how data and synchronization messages are transfered between two communication partners in an abstract way.

2.2.1 Semantics

Behaviors at the specification level communicate via message-passing channels. Behaviors exchange data by sending and receiving messages over communication channels with appropriate semantics. In the case of a sequential composition, message-passing degenerates to simple variables. Data is exchanged by reading and writing from/to the variable. In the general case of data communication between concurrent behaviors, however, a message-passing channel is instantiated.

The specification model instantiates channels out of a SpecC channel library with predefined, known semantics. By using the predefined channels out of the library, com-

monly needed communication functionality is available for integration into the specification model.

Note that the specification models of channels do not imply any specific implementation of their abstract semantics. The code inside the channel is for simulation of the correct semantics during execution only. It is the task of communication synthesis to refine those abstract channels into an actual implementation of the desired semantics using the available system bus protocols and PE interfaces.

2.2.2 Dependencies

Data dependencies in the specification are reflected explicitly in the behavioral hierarchy as transitions between behaviors, either through a sequential composition or conditionally using the `fsm` statement. In this case, channels degenerate to simple variables connecting behaviors, and the need for implicit synchronization through message-passing is eliminated.

All dependencies are explicitly captured through the connectivity between behaviors and no hidden side effects exist. Global variables have to be avoided completely. Static variables accessed from a single leaf behavior become member variables of that behavior. Global variables used for communication have to be turned into explicit dependencies in the form of connectivity as behaviors are only allowed to exchange data through their ports.

Chapter 3

Modeling Style

In general, the specification input model is written in SpecC and as such has to adhere to the syntax and semantics of the SpecC language [2]. However, to form a valid specification model that can be input into the SpecC design flow, additional rules and restrictions on top of the SpecC base have to be adhered to as defined in this section. Note that, apart from that, unless otherwise noted here, any valid SpecC code is an acceptable specification model.

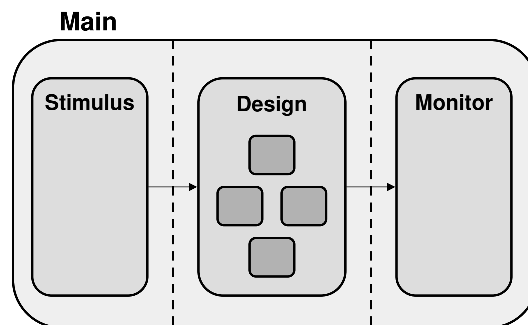

Figure 3.1: Specification model top-level structure.

Figure 3.1 and Figure 3.2 show an example template for a valid specification model. A specification model has to be an executable SpecC model, i.e. it has to define a `Main` behavior. A specification model consists of a testbench that surrounds the actual design to be implemented. A testbench consists of stimulating (`Stimulus`) and monitoring (`Monitor`) behaviors that are executing concurrently to the actual design (`Design`) in the top-most

```
import "c_double_handshake";

behavior Stimulus(i_sender input) {                       // Stimuli creator
  void main(void) {
    // while(...) { ... ; input.send(...) ; ... }
  }
};

behavior Monitor(i_receiver output) {                     // Output monitor
  void main(void) {
    // while(...) { ... ; output.receive(...) ; ... }
  }
};

behavior Design(i_receiver input, i_sender output) {      // System design
  // ...

  void main(void) {
    // fsm { ... }
  }
};

behavior Main() {                                         // Top level
  c_double_handshake input, output;

  Stimulus  stimulus(input);
  Design    design(input, output);
  Monitor   monitor(output);

  int main(void) {
    par {
      stimulus.main();
      design.main();
      monitor.main();
    }
  }
};
```

Figure 3.2: Specification model top-level code.

`Main` behavior, and that drive the design under test and check the generated output against known good values.

The design to be implemented is defined by a single SpecC behavior (`Design`) which in turn can be hierarchically composed out of a tree of subbehaviors. For a valid specification model, all the behaviors that are part of this tree have to comply with the rules and restrictions for describing computation and communication that will be defined in the following sections. Note, however, that these restrictions do not apply to the testbench part. Therefore, the testbench can be freely described using any valid SpecC code. For example, while the code of the design to be implemented has to be available completely in SpecC source form, the testbench can link against external translation units (libraries) for additional functionality.

3.1 Computation

The computational part of the specification is described through the execution semantics of the hierarchy of SpecC behaviors that form the design to be implemented. For a valid specification model, this behavior hierarchy has to be clean. A clean hierarchy is defined as a tree of behaviors in which every behavior is either a leaf behavior or a hierarchical composition of subbehaviors as defined in the following sections.

3.1.1 Leaf Behaviors

In each leaf behavior, the behavior `main()` method contains a piece of straight-line, plain ANSI-C code. Specifically, the following rules define the restrictions that apply to leaf behaviors.

Rule 3.1 *A leaf behavior must not contain any channel or behavior instances. It can, however, contain instances of variables.*

Rule 3.2 *A leaf behavior has exactly one method, the* `main()` *method. The* `main()` *method can contain valid ANSI-C statements only. Specifically, the following restrictions apply:*

(a) no SpecC-specific statements (like `par`*),*

(b) no `piped` variables, and

(c) only valid ANSI-C data types are allowed inside expressions and for all variable definitions.

Rule 3.3 *For leaf behaviors that should be implementable in hardware, the following additional restrictions apply:*

(a) no pointers (no variables or expressions of pointer type),

(b) no multi-dimensional arrays, and

(c) no composite variables of `struct`, `union`, or `enum` type.

If any of these restrictions are violated, the corresponding leaf behavior will be limited to a software implementation. In order to allow the greatest possible flexibility for exploration, these restrictions have to be followed as much as possible for all leaf behaviors.

Rule 3.4 *Leaf behaviors can only make calls to ubiquitous global functions that have a native implementation on all PEs in the processor library.*

3.1.2 Hierarchical Behaviors

A hierarchical behavior is a composition of several subbehavior instances in a sequential, parallel, pipelined or FSM fashion. More specifically, the following rules must be followed when composing hierarchical behaviors.

Rule 3.5 *A hierarchical behavior has exactly one method, the `main()` method, and the `main()` method contains exactly one statement that is either*

- *a `seq`,*
- *a `par`,*
- *a `pipe`, or*
- *a `fsm` statement.*

Rule 3.6 *Each subbehavior instance can be called at most once inside the composition.*

Rule 3.7 *For the expressions in the arguments of a* `pipe()` *statement and in the* `if()` *statements of* `fsm` *transitions the same rules and restrictions as for the C code in leaf behaviors (Section 3.1.1) apply.*

3.2 Communication

All communication in the specification model, both inside the design to be implemented and between the testbench and the actual design, is described through variables and channels that connect ports of behaviors.

3.2.1 Behavior Interfaces (Ports)

The list of ports of a behavior defines the interface between the behavior and its environment. Behaviors are only allowed to communicate with other behaviors through their ports.

Rule 3.8 *Behaviors can have ports of standard (variable with direction) type or of interface type. For standard ports, the following restrictions apply:*

(a) only valid ANSI-C data types, and

(b) no ports of pointer type are allowed.

For ports of interface type, only interfaces that are part of the standard SpecC channel library are allowed.

Rule 3.9 *Behaviors are not allowed to export any methods, i.e. they cannot implement any interfaces.*

Rule 3.10 *Behaviors are not allowed to (directly or indirectly, e.g. through a call to a global function) access variables and channels that are outside of their local scope. Code inside behaviors can only reference variables or call methods of interfaces that are defined inside the behavior as ports or local instances. Accesses of global variables or channels are forbidden.*

3.2.2 Connectivity (Variables and Channels)

Inside hierarchical behaviors, the connectivity of subbehaviors instances is defined by mapping ports of the hierarchical behavior or instances of variables and channels onto the ports of the subbehaviors.

Rule 3.11 *Ports of subbehavior instances inside a behavior can only be connected to the ports of the parent behavior or to variables or channels instantiated inside the parent. It is not allowed to map other subbehavior instances onto a subbehavior port.*

Rule 3.12 *Given the restrictions on standard port types (see Section 3.2.1), variables used for connections (i.e. mapped to ports) can only be of the following data types:*

(a) valid ANSI-C data types, but

(b) no variables of pointer type.

Rule 3.13 *Variables with storage class* `piped` *are only allowed inside hierarchical behaviors with a* `pipe` *composition (see Section 3.1.2) to connect subbehaviors that act as pipeline stages.*

Rule 3.14 *Only the following standard channel types are allowed to be instantiated and mapped to ports:*

- `c_double_handshake`
- `c_typed_double_handshake`

Chapter 4

C Code Conversion

In many cases, system design projects can leverage existing C code that describes all or part of the desired system functionality. In order to feed into the SCE design flow, such C models need to be converted into corresponding SpecC system models following the guidelines and rules described in Chapter 2 and Chapter 3, respectively.

The rest of this chapter will outline this process by showing how a straight-line C model can be converted into a valid SpecC specification model. For more information and details, please refer to [6].

4.1 Code Refinement

The code refinement process can be divided into two steps, namely, *syntactic refinement* and *semantic refinement*.

4.1.1 Syntactic Refinement

As a first step of C code conversion, syntactic refinement converts a C program into a semantically equivalent SpecC program through purely syntactical conversions.

Basically, syntactic refinement converts the C functional call hierarchy into an equivalent SpecC behavioral hierarchy. In general, each C function is converted into a SpecC behavior and behaviors are composed hierarchically according to the hierarchy of function

calls in the C program. If necessary, adjustments to the behavior granularity can be made during this step.

An important aspect of this refinement process is to make data dependencies in the C code explicit by exposing and converting them into corresponding behavior dependencies through ports and connections. On the one hand, function parameters and arguments can be directly converted into behavior ports. On the other hand, however, global variables need to be localized and any accesses to formely global variables have to be routed through ports of the hierarchy.

4.1.2 Semantic Refinement

On top of syntactic refinement, semantic refinement is concerned with removing artifical restrictions imposed on the description by the limitations of the semantics of the C language. Based on the extended semantics of the system-level design language, the specification is refined to model the desired system behavior in a more natural way.

The main aspect of semantic refinement is to expose all available parallelism in the specification using parallel and pipelined compositions of the SpecC language wherever possible. Neither the concept of pipelining nor parallelism exists within the C language. However, to efficiently perform exploration, a system level design model must explicitly provide these two concepts.

Parallel Two behaviors are combined in a parallel composition (`par` construct) if the execution sequence of the two behaviors does not influence the simulation result. Otherwise, the two behaviors are defined as behavior-sequential.

Pipelined If, within a sequential programming model, a number of behaviors are executed one after another in a loop body, and one behavior communicates only with the next behavior, then the behaviors should be composed in a pipelined fashion (`pipe` construct).

4.2 Basic Constructs

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

4.2.1 If (no else) Statement

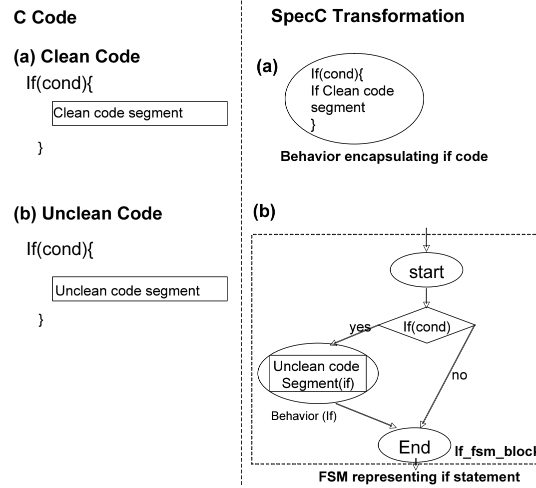

Figure 4.1: If statement.

An If statement (Figure 4.1(a)) is clean, if the code inside the braces of the if condition (*If Clean Code Segment*) is a sequence of data statements only and if 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 4.1(b)) is unclean, if the code inside the braces of the 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. If the condition is satisfied then the behavior representing *If Unclean Code Segment* will be called.

4.2.2 If Else Statement

An If Else statement (Figure 4.2) differs from an If statement (Figure 4.1) on including an *Else* part. An If statement is a subset of an If Else statement since an If statement does not

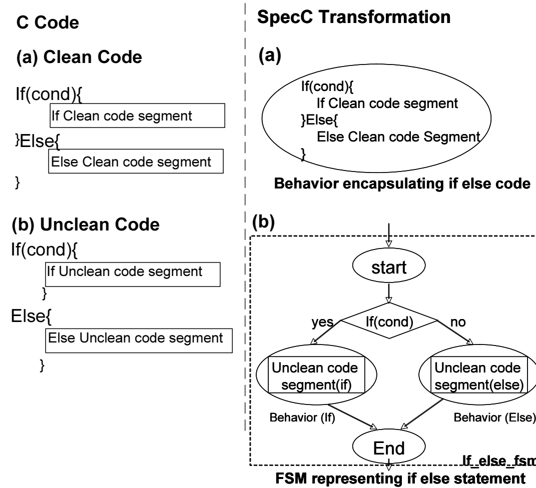

Figure 4.2: If Else statement.

have an Else part. An If Else statement (Figure 4.2(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 4.2(b)) is unclean if it satisfies one of the conditions below:

(a) If part Code Segment is Unclean

(b) Else part Code Segment is Unclean

(c) 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.

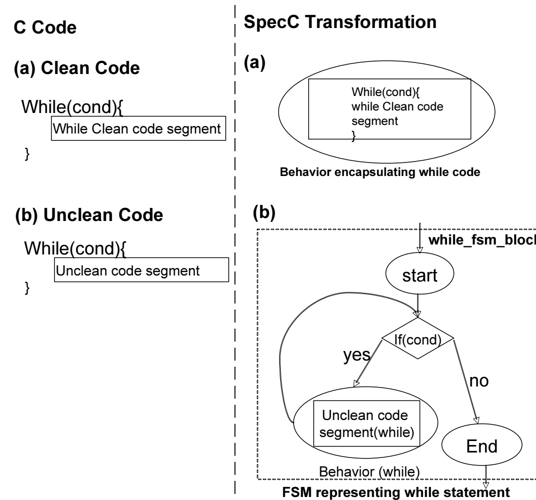

Figure 4.3: While statement.

4.2.3 While Statement

A While statement (Figure 4.3(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 4.3(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.

4.2.4 Do While Statement

A Do While statement (Figure 4.4) is just a small modification to the While statement (Figure 4.3). In the While statement, the condition is checked first before executing *While*

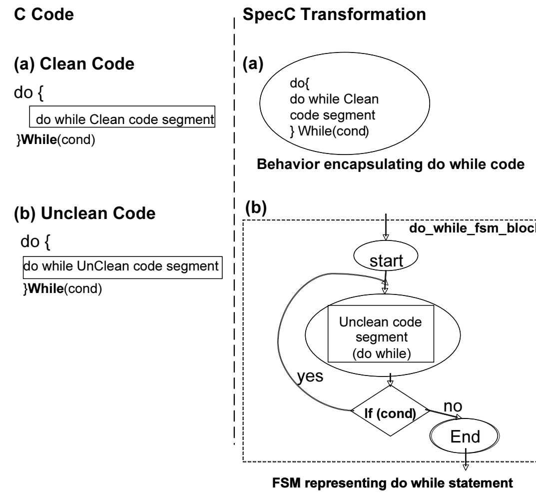

Figure 4.4: Do While statement.

(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 4.4(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 4.4(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.

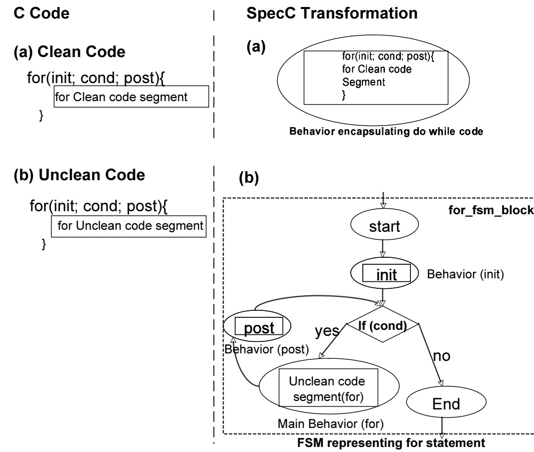

Figure 4.5: For statement.

4.2.5 For Statement

A For statement (Figure 4.5) is just a small modification to the While statement (Figure 4.3). 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 4.5(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 4.5(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.

4.3 Composite Constructs

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

4.3.1 Clean While and If Statements

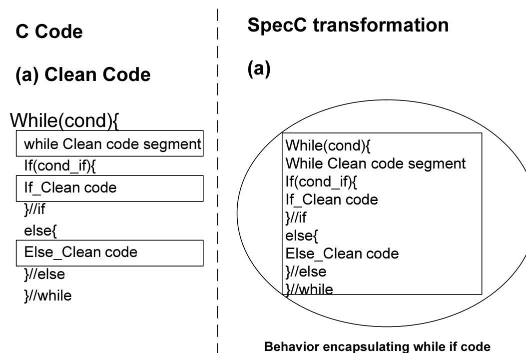

Figure 4.6: Combination of clean While and If statements.

A While statement is combined with an If statement as Figure 4.6 depicts. But both statements are clean. So, the translation is simple as we wrap both these statements into a simple leaf behavior.

4.3.2 Unclean While and If Else Statements

In Figure 4.7, a While statement which is unclean is combined with an If Else statement. The unclean While statement translation is done according to Figure 4.3 and the If Else

C Code

(b) UnClean Code

```
While(cond){
  while unClean code segment
  If(cond_if){
    If_unClean code
  }//if
  else{
    Else_unClean code
  }//else
  If_fsm_block
}//while
```

Figure 4.7: Combination of unclean While and If Else statements.

SpecC transformation

(b)

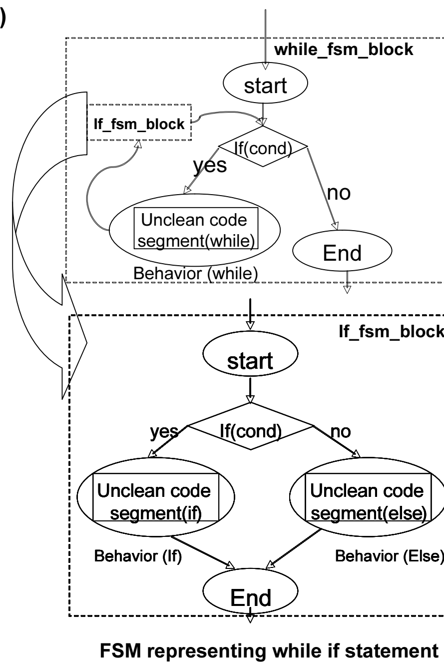

Figure 4.8: FSM for combination of unclean While and If Else statements.

statement translation is done according to Figure 4.2. 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 (Figure 4.8).

C Code

(b) UnClean Code

```
While(cond){
  If(cond_if){
    If_unClean code
  }//if
  else{
    Else_unClean code
  }//else
  [If_fsm_block]
  while unClean code segment
}//while
```

Figure 4.9: Second combination of unclean While and If Else statements.

Figure 4.9 shows another example that differs from the previous combination (Figure 4.7) in the sequence of execution of *If Else* and *While Unclean Code Segment*. As the Figure 4.10 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 4.10 which differs from Figure 4.8.

4.4 Example

Figure 4.11 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 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

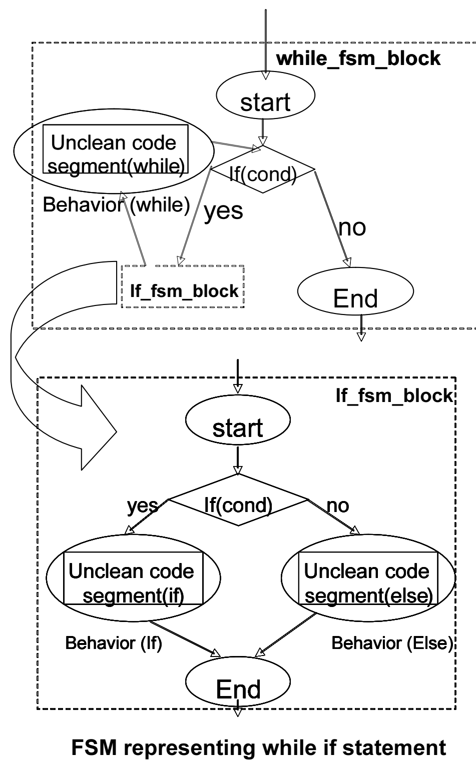

Figure 4.10: FSM for second combination of unclean While and If Else statements.

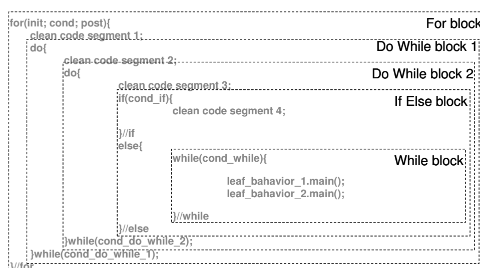

Figure 4.11: Complex example of nesting.

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: a top-down or a bottom-up approach.

When we apply a top-down flow on Figure 4.11, 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 a 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.

For this example, we will show a top-down approach in making this complex example clean.

4.4.1 Translation: Step 1

While working on the top most block, we abstract the next level block and we include it as a child behavior. The For block (Figure 4.12) has a small modification from Figure 4.5.

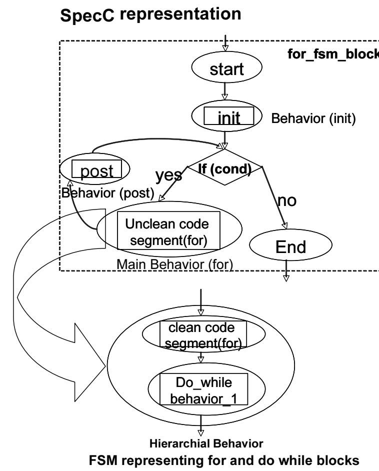

Figure 4.12: Step 1 - For Block.

Figure 4.5 contains a behavior encapsulating a for unclean code segment but Figure 4.12 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*.

4.4.2 Translation: Step 2

Do_while_Block_1 (Figure 4.13) 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 4.4 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.

4.4.3 Translation: Step 3

Step 3 (Figure 4.14) is similar to step 2 (Figure 4.13) except that *Do_while_Block_2* has *If_Else_block* as the child block.

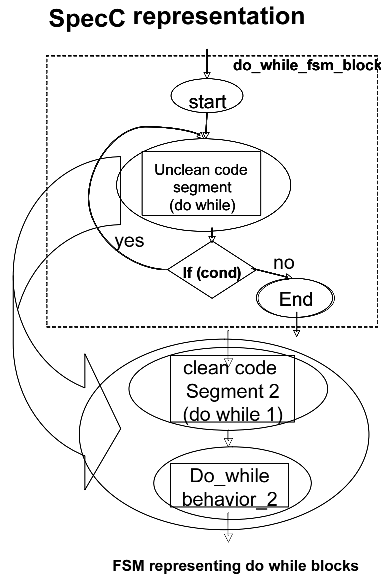

Figure 4.13: Step 2 - Do While Block 1.

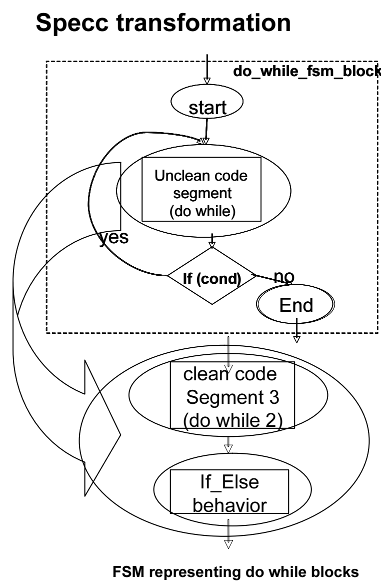

Figure 4.14: Step 3 - Do While Block 2.

4.4.4 Translation: Step 4

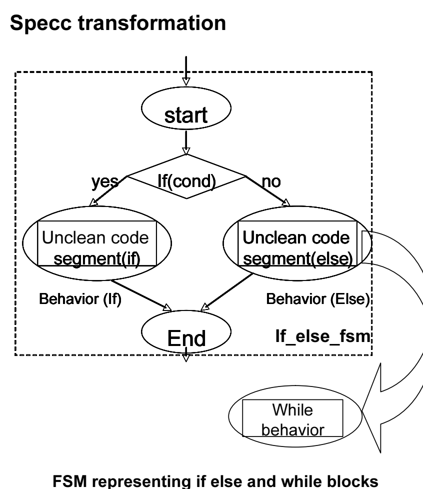

Figure 4.15: Step 4 - If Else Block.

If_Else_block (Figure 4.15) has *While_block* as the child block and Figure 4.15 depicts the difference from Figure 4.2 in 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.

4.4.5 Translation: Step 5

Step 5 (Figure 4.16) depicts the inner most while block. This while block has a sequence of two behaviors named *leaf_behavior_1* and *leaf_behavior_2*. Figure 4.16 depicts the difference from Figure 4.3 in that there is a composite sequential behavior containing *leaf_behavior_1* and *leaf_behavior_2*.

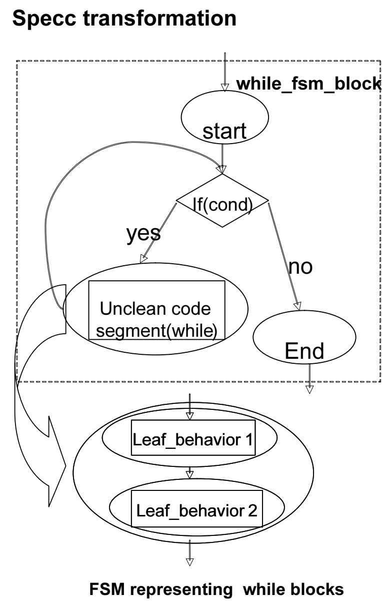

Figure 4.16: While Block.

Acknowledgments

This manual is based on work by the authors, including contributions by Kiran Ramineni [4, 5, 6].

In addition, the details of the SpecC design flow in general were shaped and influenced to a large extend by all the authors of the tool set that is the basis of SCE: Samar Abdi, Lucai Cai, Junyu Peng, Dongwan Shin, and Haobo Yu.

Finally, SpecC technology would not exist at all today without the vision of Prof. Daniel D. Gajski.

References

[1] A. Gerstlauer, R. Dömer, J. Peng, D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[2] R. Dömer, A. Gerstlauer, D. Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium (STOC), Japan, December 2002.

[3] SpecC Technology Open Consortium. http://www.specc.org.

[4] A. Gerstlauer, K. Ramineni, R. Dömer, D. Gajski. System-On-Chip Specification Style Guide. CECS Technical Report 03-21, Center for Embedded Computer Systems, Irvine, June 2003.

[5] A. Gerstlauer. SpecC Modeling Guidelines. CECS Technical Report 02-16, Center for Embedded Computer Systems, Irvine, April 2002.

[6] K. Ramineni, D. Gajski. C To SpecC Conversion Style. CECS Technical Report 03-13, Center for Embedded Computer Systems, Irvine, April 2003.