Esterel 인터프리터를 위한 문맥지시적 디버거

A syntax-directed debugger for Esterel interpreter

Abstract

As a useful tool for embedded system codesign approach, it’s necessary to make a custom-built interpreter for the system description verification. Usually, designers need to write their program to simulate the environment their system works in. Sometimes making the simulation environment consumes designers more time and energy than describing their embedded system. The interpreter saves the cost that is spent on making such an environment. In this paper, the necessity and motivation of the interpreter will be introduced first, and then the details about each part of it will be illuminated.

1. Introduction

Most embedded systems have a predominant digital component consisting of a hardware platform which executes software application programs. Real time reactive embedded software may need to be described as collections of concurrently running processes and executed base on a real time operating system (RTOS). While an RTOS provides very flexible functionalities, the overhead of concurrency can be huge, furthermore, the interprocess communication mechanisms can easily become unwieldy and lead to behavior that is unpredictable. Therefore many synchronous programming languages, like Esterel, signal, and luster, provide the deterministic, timing-predictable concurrency for concurrent computation. Concurrent threads in these synchronous languages execute in certain periodic global clock cycles and communication between them is implicitly synchronized to the clock. Hence synchronous languages provide a good choice for people who are facing this kind of problems.

Esterel, one of the synchronous languages, provides a good tool for reactive deterministic concurrent embedded software programming. It’s used to describe the behavior resemble digital circuits so that it can be either implemented in hardware or in software. Esterel programs can contain multiple parallel threads, all of which are executed in fixed clock cycles and communicate using a broadcast mechanism. The broadcast mechanism requires that the information of each present signals is sent to everywhere of the program in and only in current clock cycle. Signals in Esterel may be pure or valued. Programs compute outputs exactly according to current states of input signals from outside and accessorial data within a part of the codes.

Recently, the new generation of CEC (the Columbia Esterel Compiler) [4], [5] made a substantial progress on generating more efficient and faster simulation codes for Esterel. Base on CEC and codesign concept, we are trying to develop our codesign approach with building a codesign development environment [1], [2], [3]. My interpreter is stressed on being a useful tool of this environment.

In the following part, I will briefly introduce our codesign approach and the motivation of my interpreter, which is a convenient and helpful tool that makes programmers verify their CEC style Esterel program easier. And then, in the third part, I will illustrate how to implement it.

2. Motivation

The term hardware/software codesign tries to meet system-level objectives, like performance, power, size etc, by exploiting the synergism of hardware and software through their concurrent design. Since Esterel grammar has implicitly required programmer make the hardware software partition, our codesign approach is straightforward. As Figure 1 shows, normally the first step is to specifically describe an embedded system as programmer’s desire. This system, may include both hardware component and software component, is either specified by only Esterel or specified by combined C with Esterel, which depends on the system partition. We usually think that the software part described by C will be running on a microprocessor system as an assistant for the hardware part implemented in digital circuit. Software description and hardware description are both made in
programming style, so that designers don’t need having any knowledge about the connection detail of digital circuit.

**System Description**

Specification

### CEC

**Partition**

- **HW**
- **SW**

**Hardware**

- VHDL

**Software**

- C

**Prototype**

**Simulation**

- C++ simulator

**Interface**

- Hardware: (FPGA)
- Software: (Microprocessor)

**Verification**

Description of system control logical and function

After that, the CEC compiler takes care of the translation and compiling work and generates a prototype from designer’s system description. Then the hardware circuit connection specified in VHDL and the software program can be generated from the prototype. CEC compiler also offers designers the C codes that can be thought as a program that simulates system behaviors.

Even though designers can get both the product codes and the simulation codes from CEC, this approach is still thought pretty rough so that we need to address more work on making it convenient, efficient and friendly.

As we can see in figure 1, verification for such a system is a little arduous. To verify a prototype by examining it’s corresponding C simulation codes, designers have to write a particular program which simulates the environment that the system is working at, and make an interface connecting the simulation codes and the verification environment, probably with control functions. It could be a lot of extra work besides describing the system in Esterel, and consumes designers more time and energy than making system prototype. So an interpreter which facilitates design verification of their system description by only clicking the step-in button and setting the input signals is necessary and useful. According to our codesign approach, this interpreter must exactly follow the Esterel language style required by CEC compiler.

### The interpreter

The interpreter has a well defined GUI interface to let users edit their Esterel source codes, control the execution and examine the behaviors. It consists of a parser that makes the attribute syntax trees, an interpreter core that interprets the behaviors, a debugger that deals with source codes reconstruction and high-lighting present statement works, and a communication interface between debugger and interpreter core.

As showed in Figure 2, the interpreter works like this: it imports Esterel source codes from GUI. After user decides to test his codes, it parses the source codes into a normal syntax tree, and then walks the tree through to extract some useful information and build an attribute syntax tree. This tree will be duplicated into the debugger for program reconstruction, and the interpreter core. Here, I defined an interface for communication between the debugger and the interpreter core. By this interface, in each step, or said clock cycle, the debugger sends the control message and states of input signals to the interpreter core, and the interpreter core sends back the present statement and states of output signals. So the debugger can highlight the current executed statement in the reconstructed program and set the states of output signals. Once user clicks the step-in button, this process will be executed one times. So users can examine the behavior of their system description by only clicking the step-in button and setting the input signals.

#### 3. The interpreter

The interpreter has a well defined GUI interface to let users edit their Esterel source codes, control the execution and examine the behaviors. It consists of a parser that makes the attribute syntax trees, an interpreter core that interprets the behaviors, a debugger that deals with source codes reconstruction and high-lighting present statement works, and a communication interface between debugger and interpreter core.

As showed in Figure 2, the interpreter works like this: it imports Esterel source codes from GUI. After user decides to test his codes, it parses the source codes into a normal syntax tree, and then walks the tree through to extract some useful information and build an attribute syntax tree. This tree will be duplicated into the debugger for program reconstruction, and the interpreter core. Here, I defined an interface for communication between the debugger and the interpreter core. By this interface, in each step, or said clock cycle, the debugger sends the control message and states of input signals to the interpreter core, and the interpreter core sends back the present statement and states of output signals. So the debugger can highlight the current executed statement in the reconstructed program and set the states of output signals. Once user clicks the step-in button, this process will be executed one times. So users can examine the behavior of their system description by only clicking the step-in button and setting the input signals.

#### 3.1 Building the parser and the tree walker

As a facility of our codesign approach based on CEC, the purpose of the interpreter is helping designers to verify their system. Since system prototype is compiled by CEC, the grammar I implement adjusts a little bit to CEC style. For example, the valued signal in original Esterel grammar now is not implemented in CEC, so it’s cut off here. Following this idea, a parser called Esterelminus is made by the ANTLR parser generator [6]. A snippet of ANTLR specification is showed below:

```plaintext
sequence: atomicStatement ( -> ^( SEQUENCE atomicStatement) |
( ';' atomicStatement)+ -> ^( SEQUENCE atomicStatement+))

atomicStatement : 'nothing' -> ^( 'nothing')
| 'pause' -> ^( 'pause')
| 'halt' -> ^( 'halt')
| sustain
| present
```

After input all the grammar I picked, ANTLR generates the parser in Java language. It also generates a tree walker in the same way. Some attributes will be attached in the attribute syntax tree by the tree walker.

#### 3.2 The communication interface

The communication interface connects the debugger and
the interpreter core. As we know, the two parts both have an identical copy of the attribute syntax tree, so it’s not necessary to pass the present subtree to interpreter core. All the debugger needs to do is waiting for the feedback information after passing the states of input signals to the interpreter core.

Since the two components work in two threads, the arriving message from debugger implicitly means that the interpreter core should move forward for one step and send back the result. The message passed in the interface is quite simple:

Class MessagePackageFromDebugger {
    SignalAndStates[] inputs
}

Class MessagePackageFromInterpreter {
    int index[];
    SignalAndStates[] outputs
}

The message from debugger shows interpreter core the input signal states, and then it will send the index of present subtree and the output signal states back as response.

3.3 The debugger

The debugger is consisted of three parts: the state tracking module, the tree walker for reconstruction and the GUI that is built based on Java Swing package. The debugger takes care of all the edition functions, control functions, program reconstruction and highlighted source codes displaying work. After getting the message from interpreter core, it reconstructs the whole program with highlighting the statement pointed by the index number. Finally, it shows all the output signal states and reconstructed source codes, then releases the control to users as the sign of one step completion.

3.4 The interpreter core

The interpreter core encapsulates all the interpretation details. Once the interpreter core steps into the parallel statement, one thread becomes several threads and one present statement becomes several present statements. Among these present statements, some of them may be “emit” statement and some are other kinds of statement. The only principle interpreter core needs to ensure is that it should “emit” a signal before other statements implement it. As long as the interpreter keeps doing the “emit” actions first, it’s impossible for the test statements like “present” make decision on a wrong condition because all the operations on signal states are done before anyone tests them.

4 conclusion and future work

As a facility of our codesign approach, it actually saves a lot of unnecessary work. For example, a simple vacuum robot case is showed in figure 4. Without the interpreter, more than one thousand lines C++ codes are taken to verify only 126 lines Esterel program.