COMS W4115

Programming Languages and Translators

Event Driven State Language (EDSL)

“It’s not like your daddy’s car”

Proposal

Christopher D. Sargent
cds2131@columbia.edu
September 15, 2007

Introduction

In an era in which the prices of highly capable microcontrollers are plummeting as technology continues to improve at a rapid rate, the extra development costs associated with hardware-only or low-level, highly-optimized languages for devices requiring state machines can be mitigated by utilizing a compiled language specifically designed for the creation of event-driven state machines.

As hardware becomes more programmable and abstract, as is the case with some of the newer military radios, which are software-based and intended to replace the fixed-waveform legacy radios of the past, languages that allow the user to design hardware dependent functionality without relying on specific hardware are becoming more practical.

Languages like HDL and VHDL which can be used for this purpose have existed for years, but their focus in the way that variables are defined and utilized has always been from the hardware aspect. The optimizers for these languages have not progressed at the same speed as those of the software programming languages, such as C++.

As many simpler electronic devices rely on events and states, an easy-to-program language tailored for state machines is beneficial not only from the stand-point of bringing a product to market earlier, but also from the stand-point of being able to fix a flawed design with a product without requiring a hardware swap or even worse, a complete recall.

An example of an electronic device that would be a good candidate for this technology is a coffee maker with a timed shut-off. The coffee maker has several states: off, on-making coffee, and on-warming coffee. A problem that can occur with coffee makers with timed shut-offs is that a design problem or lack of redundancy can cause the timing device within the coffee maker to malfunction, possibly creating a situation in which a fire hazard can occur. Being able to replace the software in this device to fix a design flaw is a rapid and cost effective way of mitigating risk. And depending on the level of technological advancement, the coffee maker may have the capacity of receiving the update via the internet.

Assumptions

A language of this nature cannot entirely abstract its hardware dependency, and the hardware unlike that of a personal computer, must be tailored to some degree for its particular application in the interest of cost savings amortized over the potentially large number of production units. What this necessitates is adaptive or multi-use hardware capable of producing input events and triggering output events.

But intelligent hardware is not all that is necessary to bridge the gap between an imperative programming language and a mass-produced device. The compiler itself must be somewhat intelligent and it must have an understanding of the hardware that it is marrying the state machine with, to the extent that the programmer only needs to know a minimum of the specifics. The programmer need not know whether an on button is a physical button that is pushed by the finger of a user or a button on a screen that the user clicks.

Requirements

This language has both programming language constraints and hardware application constraints. The internals of the compiler combine these two constraints in optimal fashion.

As previously stated, the optimizing compiler and the hardware application interface are identified as assumptions which are out of the scope of the tasks related to this project. This is partly because of the complexity involved with mapping the intentions codified in a higher level program language with the lower level hardware limitations. Additionally, this technology may rely on state-of-the-art multi-use hardware such as that required by software-based radios.

The scope of this project is therefore limited to the development of the language itself, and the specifics of what it could be done with additional libraries designed for specific hardware are left for future development.

Concepts

Before a specification can be written for the state language, general concepts relevant to the design of a state machine are identified. From the top, down, these concepts are identified as the domain, the events, the state machines, and their states.

The domain is the entire space within which all objects exist. All objects including the state machines and events exist within the domain. The domain is, for all intents and purposes, everything.

The events are not specific to any particular state machine as they can be utilized by more than one. Events are very much like interrupts in that they function as triggers when their states change. They are also very much like variables in that they have a higher level type associated with them. An event can be classified as being in one of three categories:

a. Input Events – An input event as triggered by a user or by another device. An example of a simple input event is a button that a user can press with the intention of triggering an action. In this example, the event would have a Boolean type.

b. Output Events – An output event is probably not triggered by a user, but is instead triggered by a state transition. As is the case with an input event, an output event has a type associated with it and the typed value is passed to the hardware or software associated with it.

c. Internal Events – An internal event is a construct available to the programmer for internal use within the domain. It does not receive input from the user nor does it provide output. Internal events differ from input and output events in that they must be started much like processes and when they terminate, they trigger an event. For the sake of language simplicity, only the timer internal event exists to trigger a Boolean event when time elapses. This does not preclude the possibility of identifying additional internal events or constructs for internal events to be specified at the programmer level.

A state machine is at the core of what this language is intended for. Unlike input and output events, the functionality of the state machine is not necessarily apparent to the user. Inside the state machine are an arbitrary number of states. Initially, the state machine starts at the first state. During each state, a set of actions may be executed. Depending on the events, a state change may occur.

Language

The language is merely a reconstruction of the concepts in a format that is understandable by the end user and is not overly difficult or time-consuming to compile. As was stated in the above section, there is only one domain and as such, there is not a need for it to be named. Within the domain which includes everything within the input file, there are *n* state machines and *m* events. Additionally, comment lines can be inserted to provide additional clarity.

Events are identified by their category:

input – An input such as a button, switch, or keypad.
output – An output such as a signal or message.
inout – An input and output capable device.
internal – An internal event such as a signal designed to trigger a state change.

Inputs, Outputs, and Inouts are external events, meaning that they are associated with equivalent hardware inputs and outputs. It is necessary to segregate internal events from external events, because the compiler must reconcile external events with the hardware inputs and outputs. Following the category is the type of event and following that is the event name.

State machine constructs are identified by the open and close braces, *{* and *}*. There is no need to assign a name to a state machine, because this is not an external concept. Comment lines are identified by the traditional C++ comment line syntax *//*. The semi-colon (*;*) serves to delimit statements, in case statements span multiple lines. A very simple example of a domain specification is as follows:

```
// Events
input bool ibInput1;
input bool ibInput2;
output bool obOutput;
```

```
inout float iobValue;
internal timer timTicker;

// State machine
{
     // State machine internals
}
```

Events must be defined before the one or more state machines that use them and they are defined for the entire domain. Each state machine has states that exist within it and are defined by an identifier, followed a colon. The first state is always the initial or start-up state. If all of the expressions are executed within a state and the flow is not altered by a change to another state, then a transition is automatically made to the next state—unless the current state is the last state, in which case, this state is automatically re-executed unless the flow is altered. A state machine can have *n* number of states. A very simple example of a state machine is as follows:

```
// State machine
{
// First state
State1:
     // State expressions

// Second state
State2:
     // State expressions
}
```

Each state can have *n* number of statements or expressions. A very simple example of a state is as follows:

```
// First state
State1:
     // Trigger output event
     obOutput = false;

     // Trigger internal event
     timTicker = 30;

     // Stay in this state if:
     // - Input1 is false –or-
     // - Timer is at zero
     if (ibInput == false || timTicker != 0)
          State1;
```

Because this language is intended for optimization for the hardware, it can be assumed that the instructions are executed in order. This assumption simplifies the source design for the programmer, but allows the optimizer to make assumptions based on the hardware that may allow pieces of the machine code to execute simultaneously.

A condition is specified in the form of an if-else statement, similar to C. If the condition is met, than either the single very next expression is executed or the expressions contained within the braces are executed. If not, then it is skipped. Complex logic is allowed within the if-else statements.

Example

A real picture of how this language works is given by a plausible example. The following represents a state machine utilized for a coffee maker:

```
// Power button event
inout bool iobPowerButton;

// Water empty event
input bool ibWaterEmpty;

// Water heating element event
output bool obHeatingElement;

// Hot plate event
output bool obHotPlate;

// Automatic shut-off event
internal timer timShutoff;

// Associated state machine
{
// Initial state
PowerOff:
     // Initialize the heaters to off
     obHeatingElement = false;
     obHotPlate = false;

     // Stay in this state if the power is off
     if (iobPowerButton == false)
          PowerOff;

// Make the coffee
MakeCoffee:
     // Make sure the power isn’t off
     if (iobPowerButton == false)
          PowerOff;

     // Check to see if there is water
     if (ibWaterEmpty == false)
     {
          // Turn on the heating element
          obHeatingElement = true;

          // Stay in this state
          MakeCoffee;
     }

// Heat the coffee
HeatCoffee:
     // Make sure the power isn’t off
     if (iobPowerButton == false)
          PowerOff;

     // Turn on the hot plate
     obHotPlate = true;

     // Initialize the timer for 20 minutes
     timShutoff = 1200;

// Wait until the pot is shut off or timer runs out
Standby:
     // Make sure the power isn’t off
     if (iobPowerButton == false || timShutoff == 0)
     {
          // Turn the power off
          iobPowerButton = true;

          // Move to the power off state
          PowerOff;
     }

     // Stay in this state
     Standby;
}
```