Decomposing the Dashboard Example for a Distributed Implementation

EE249 FINAL REPORT
December 10, 1999

Jason Shamberger
Mentor: Dr. Alberto Ferrari

Introduction

Motivation
Modern consumers are demanding more electronics in their automobiles. As cellular phones, internet communications, and navigation systems are being added to a dashboard which already contains audio systems, engine information and driving controls, the designer is faced with an increasingly complex system to verify and implement. Since the electronics may be located throughout the car, the systems must be properly modeled as a distributed one, with multiple processors. It is important to model the communications between the processors at an early stage in the design flow, since it is necessary to ensure any real-time constraints will be met.

Goals of Project
There are two goals to this project. First, we would like to use POLIS to model a distributed system. As an example system, the dashboard demo which is included in the POLIS distribution will be redesigned. The new system will have an architecture consisting of two processors, the first of which will take input signals from the various sensors in the car and compute values to be displayed on the dashboard. We will call this the sensor CPU. The second processor will receive the updated information from the sensor CPU and display it on the dashboard for the driver. We will call this the display CPU. A CAN (Controller Area Network) bus will be used to carry communications between the two processors.

The second part of the project is to synthesize code for the two processors. Even though POLIS does not support multiprocessor code generation, after modeling the entire system to verify the functionality and evaluate the performance, the sensor and display code can be generated separately. To interface with the CAN controllers, we will generate the correct API function calls in the C code generated for the real-time operating system.
Background

**CAN Protocol**

The CAN protocol is a serial communications protocol that was developed by Robert Bosch GmbH in the early 1980’s [1]. It is designed for distributed real-time control systems, where temporal constraints are placed on the delivery of data. The protocol is designed without any centralized control, so devices can be added or removed from the network easily. The only requirement is that all devices have a unique identifier.

The CAN protocol only specifies the transport layer, and leaves the physical layer open to different implementations. However, the medium of transmission must have “dominant” and “recessive” bits. As an example, it could be implemented as open-collector wired-AND gates, where a low voltage would always pull down a high voltage. This property is used in assigning identifiers to determine which messages should be given priority in transmission. In this wired-AND gate example, the dominant bits would be logic zeros. Thus, the lower number identifiers are assigned to messages with higher priorities.

Every transmitter synchronizes its clock by listening to the data sent on the CAN bus. If a transmitter has a message to send, it waits until the bus is free and then it begins transmission. The first part of the transmission is the arbitration phase, during which multiple devices which began sending data simultaneously can determine which has the highest priority. Each transmitter begins by sending its own identifier. Since each transmitter is monitoring the line as it sends data, if it ever transmits a recessive bit but observes a dominant bit on the line, it knows a device with a higher priority is transmitting at the same time. At this point the lower priority device gives up, and waits for the bus to be free again. This ensures that the highest priority device is able to send its message without any loss of time. However, in this preemption scheme, once a transmitter has gained control of the bus, it cannot be interrupted until it has finished sending the current packet.

There is no concept of source and destination addresses in the CAN protocol, all the packets are broadcast on the bus and the identifier serves to describe the type of data. Each device on the CAN bus has a filter to decide whether to accept or ignore packets based on their identifier. In this project we only have two devices communicating, but in a larger system, information can be multicast from one device to multiple other devices.

The total length of data that can be sent in one packet is 8 bytes. In addition to the identifier broadcast during arbitration, there is a CRC and some control bits. Since the transmitters keep their clocks synchronized by reading data on the bus, bit stuffing is used to avoid long periods of constant voltage levels. If five consecutive bits of identical value are to be sent, a complementary bit is inserted after these five bits by the sender. Though this increases the length of the packet to be sent, it will be ignored in this modeling. Since all the packets of dashboard information need only one byte of data, their total length is 54 bits including the overhead. 24 of these bits are in fixed-length fields which do not use bit stuffing, so the total length of the packet could increase by 6 bits at most, which is only a 10% increase in length.

Another feature which is not implemented in this project is error handling. There are error handling features built into the protocol to deal with situations where the bit
stuffing rules are broken, or a device detects a bad transmission once it has gained control of the bus. I will assume that all communication is successful and there are no transmission errors which arise.

**Original dashboard decomposition**

The original dashboard model consists of a number of modules to monitor inputs from different sensors, and compute the outputs to display on the dashboard. Seven different values are displayed: alarms for the seatbelt, fuel level and water temperature, a speedometer, a tachometer, and an odometer which displays total distance traveled as well as the partial distance traveled since the last reset. This was all mapped to one CPU.

A drawback of this implementation is that it requires separate wires for each input signal from the sensors all the way to the dashboard. It is clear that in a distributed system where computation units are physically separated around the automobile, an advantage of shared communications is that it will reduce the number of wires which are needed to transmit information from one device to another.

**Strategy taken**

**Block diagram**

For this project the behavior will be split into two groups of modules, one group to sense and interpret the data and generate the desired output, and the second group to display this output on the dashboard. These two groups of modules will share a communication channel to transmit the data.

![Block Diagram of Behavior](image)

**Figure 1: Block Diagram of Behavior**

It should be noted that the sensing might be done in different locations around the car (i.e. the engine speed is measured in a different physical location than the fuel level). If this is the case, there could be several different CPUs performing the sensing work, which all send data to one CPU performing the display work. The sensing modules would be mapped to these different CPUs, which would all communicate over the same network with the display. For this project, I will just divide the behavior into two parts, though further division would not be any more difficult.

It is also important to note that this implementation does the least amount of processing at the dashboard. This reduces the amount of traffic that needs to be sent over the network. For example, instead of sending the fuel level every time it changes, we can just send an alarm when it is too low. In the event that the dashboard is displaying data from different sources, it could become a bottleneck if significant computation was required.
**Proposed architecture**

As an architecture for implementation, we will use two CPUs, one for the sensing and processing, and another for the display. Each processor will have a CAN controller on its processor bus, and these controllers will communicate via a CAN bus. For the purpose of this model, we will assume that the CPU and CAN controller are alone on the CPU bus, and we need not deal with an extra traffic on the CPU bus. There would most likely be other devices sharing the CAN bus, however, so we will model traffic due to them.

![Proposed architecture](image)

**Figure 2: Proposed architecture**

**Communications refinement**

Once we have chosen this architecture, we can refine the behavioral model of the communication we originally abstracted as an arrow. The CPUs on each side of the communications channel will use a network interface to transmit and receive data on the channel. The network interface shown below performs two tasks: it will mux the different modules communicating on the network to the one shared communication channel, and it will also serve as an arbiter of the channel, letting the message with the highest priority be sent first. These two parts of the behavior will be mapped to two different parts of the architecture. The muxing of different modules communicating on the network channel occurs in the software on the first CPU, when it uses the API to tell the CAN controller to transmit a message. The arbitration on the CAN bus occurs in hardware when the CAN controller chooses which message to send and then tries to send it on the bus. The network interface on the receiving end has two similar tasks, the first of which is to decide whether to accept a packet transmitted on the CAN bus. This is mapped into the hardware of the CAN controller. The second task is to demux the accepted messages, and deliver them to the appropriate software modules. This is performed in software, when API calls are used to poll the CAN controller to see if new messages have arrived.

![Communications refinement](image)

**Figure 3: Communications refinement**
Behavioral modeling

Description

For behavioral modeling, the block diagram above was implemented in POLIS. Most of the code for the sensing side could be modified from the code written for the original dashboard example. Rather than connecting the output directly to the displays, however, the output was sent to the network interface. This module, NETWORK_MUX simply waits for any input, then sends the data and identifier to the channel (see Esterel code in Appendix A).

To model the delay in the channel, a star was written directly in Ptolemy. This is acceptable since this part of the system will not be synthesized, it is only needed for simulation. This delay star has an array to store the input data in, with a buffer allocated to store one message for each identifier. When the channel is free and ready to transmit a message, the star looks through the array of waiting messages for the one with the highest priority identifier. It then calculates the delay needed to send this packet, and emits it after waiting for the appropriate amount of time.

The network interface on the other end of the channel is similar to the first network interface, except it demuxes the messages by their identifiers, and sends them to the correct parts of the display.

Results of simulating with traffic

In an automobile, the CAN bus will be shared by many different devices. Traffic was added to the model to account for the congestion on the bus. This was achieved by creating extra ports on the delay star to take traffic messages as inputs. Their priorities, frequency and length can all be changed easily to account for different types of traffic. Since I did not have sample traffic from a real implementation, I created three different types of traffic: a high frequency stream of small packets of data, a stream of small packets of data with a Poisson distribution of arrival times, and a low frequency stream that sent several packets of data in each burst.

For the dashboard application, the critical measure of performance is that all the information gets through. This criterion is not met when the transmission of one packet is delayed enough such that the next packet with the same identifier arrives before it can be sent, and the original packet is overwritten in the buffer. In the same way that a designer can explore different architectures with a tool like POLIS, a designer can explore different network design decisions. Different priorities can be assigned to different data streams to try and minimize the number of packets lost. In my example, the CAN bus has a bandwidth of 125Kbps, and the dashboard data uses about 28Kbps at its maximum. I added traffic to bring the total bus usage to 75% of the bandwidth before I started observing message loss. By shifting the priorities of the dashboard signals and the traffic, I was able to increase the bus utilization before observing any packet lost. (See table below). The number of overwrites is measured over a simulated time of 10 seconds, and includes overwrites of both the dashboard data and the traffic data. The traffic packets which are lost are important, since we do not want the dashboard data we are sending on the bus to disturb the other systems which are communicating. The initial priority scheme gave the traffic having the highest priority, and the dashboard signals were ordered as follows: belt alarm, fuel alarm, water alarm, engine speed, wheel speed,
partial tenths, total tenths. However, the critical signal is the one that is most frequently sent, and in this case that is the engine speed signal. Thus I revised the priorities to give the engine speed signal the highest priority amongst the dashboard or traffic messages. This resulted in a much higher bandwidth utilization (90%) before any packet loss began appearing.

<table>
<thead>
<tr>
<th>Bus Utilization</th>
<th>93 Kbps (75%)</th>
<th>113 Kbps (90%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Overwrites</td>
<td>Percentage</td>
</tr>
<tr>
<td>Original Priorities</td>
<td>38</td>
<td>0.3 %</td>
</tr>
<tr>
<td>Revised Priorities</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 1: Packet loss

Synthesis

Description of CAN API

One advantage of using a tool such as POLIS is that once a designer has modeled the behavior and explored different design decisions, he or she can move directly to synthesis. In this case, there is the added complexity of generating the code to interface with the CAN controller. The CAN controller has an API which is implemented as a set of C functions. Calls to these functions perform the various operations needed to manage the controller. There are functions to initialize the controller, check for new messages, load data in a message buffer, send a message, and so forth. The designer’s code calling these functions can then be linked with the object code implementing the API.

For this project I used the Motorola msCAN driver for the CAN modules which is available on-chip in versions of the 68HC08 and 68HC12 [2]. This implementation has one buffer allocated for each identifier, and if another packet is sent before the first is transmitted, the first packet is overwritten, as modeled above.

Method

The standard method of synthesizing code from the Esterel input files in POLIS can thus be used. It is only necessary to add calls to C functions from the Esterel files. The API functions could be called directly from an Esterel file, but it is a better abstraction to create wrapper functions which in turn call the API functions of the CAN controller (see api_lib.c in Appendix A).

The only problem with this method of synthesis is that the same file cannot be used for both behavioral simulation and synthesis. For simulation, the muxing module of the network interface must emit signals to the star modeling the channel delay, while for synthesis the module does not emit anything, but rather makes calls to API functions of the CAN controller. I found it necessary to create two separate versions of this file, and swap them depending on whether I was simulating or synthesizing. There are two problems that arise from this method. The first is that estimates of code size and
execution time will be off, since the code used for simulation is not the actual code used for implementation. The second problem is that requiring the designer to translate the behavioral code to the implementation code could introduce bugs that were not present and went therefore undetected in the simulation phase. A solution for this would be to automate the procedure to eliminate human error.

Conclusions

POLIS was used successfully to model and implement a distributed system. The performance of the system including the network used for communication was simulated. From the results of this simulation, it was possible to determine the correct priorities to assign to the different messages sent on the network.

Once the system performance was satisfactory, code was synthesized. The generated code included the function calls necessary to interface with the network controller.

This project has shown that POLIS is a useful tool for designing such a system, yet there is room for improvement. Greater support in POLIS for multiprocessor systems would be beneficial, namely allowing synthesis of code for multiple processors. Currently, it is necessary to first model the system as a whole, then split it into the two subsystems for the sensors and display and synthesize code for each half of the system separately.

Future Work

There are several ways in which I would like to expand the work done in this project including the following:

• Look at a more complex example, with more than two devices communicating on the CAN bus. The example of two devices was sufficient to show the modeling is possible, but a more interesting example would include more devices, and one-to-many type communications.
• Implement error handling for overrunning buffers. In the model developed, overwrites were counted, but no information was sent back to the sender. It would be interesting to use feedback to request retransmission of information which was lost.
• Improve the method of moving from simulation to synthesis, by automatically generating the code needed to call the API functions. The method of exchanging files for simulation and synthesis worked fine in this small system, but in a larger example this might become too complex. It would be desirable to have the designer write the behavioral model, and have the code for synthesis automatically generated from this.

References

Behavioral NETWORK_MUX module (from file NETWORK_MUX_BEHAV.strl):

module NETWORK_MUX:
  input
    RESET,
    BELT_ALARM : boolean,
    FUEL_ALARM : boolean,
    WATER_ALARM : boolean,
    ENGINE_SPEED : integer,
    WHEEL_SPEED : integer,
    PARTIAL_TENTHS : integer,
    TOTAL_TENTHS : integer;
  output
    DISPLAY_TYPE: integer, % identifier for CAN bus
    DISPLAY_VALUE : integer; % data for CAN bus
  constant
    BELT_ALARM_CODE : integer,
    WATER_ALARM_CODE : integer,
    FUEL_ALARM_CODE : integer,
    ENGINE_SPEED_CODE : integer,
    WHEEL_SPEED_CODE : integer,
    TOTAL_TENTHS_CODE : integer,
    PARTIAL_TENTHS_CODE : integer;
  var
    belt_alarm_val : integer,
    water_alarm_val : integer,
    fuel_alarm_val : integer
in
  % Infinite loop
  loop
    % When a RESET happens quit loop
    abort
    await [BELT_ALARM or FUEL_ALARM or WATER_ALARM or ENGINE_SPEED or
    WHEEL_SPEED or TOTAL_TENTHS or PARTIAL_TENTHS];
  present
    case BELT_ALARM do
    % output a belt alarm
    if(?BELT_ALARM)
      then belt_alarm_val := 1;
    else
      belt_alarm_val := 0;
    end if;
    emit DISPLAY_TYPE(BELT_ALARM_CODE);
    emit DISPLAY_VALUE(belt_alarm_val);
    %
    % other cases follow, they are omitted here to save space
    %
  end present;
  when RESET;
  end; % loop
end; % var
Synthesis NETWORK_MUX module (from file NETWORK_MUX_SYN.strl):

module NETWORK_MUX:
function mscan_init() : integer;
function mscan_send(integer, integer) : integer;
input
  RESET,
  BELT_ALARM : boolean,
  FUEL_ALARM : boolean,
  WATER_ALARM : boolean,
  ENGINE_SPEED : integer,
  WHEEL_SPEED : integer,
  PARTIAL_TENTHS : integer,
  TOTAL_TENTHS : integer;
output
  %DISPLAY_TYPE: integer,
  %DISPLAY_VALUE : integer;
constant
  BELT_ALARM_CODE : integer,
  WATER_ALARM_CODE : integer,
  FUEL_ALARM_CODE : integer,
  ENGINE_SPEED_CODE : integer,
  WHEEL_SPEED_CODE : integer,
  TOTAL_TENTHS_CODE : integer,
  PARTIAL_TENTHS_CODE : integer;
var
  err_code : integer,
  belt_alarm_val : integer,
  water_alarm_val : integer,
  fuel_alarm_val : integer
in
  err_code := mscan_init();
% Infinite loop
loop
  % When a RESET happens quit loop
  abort
    await [BELT_ALARM or FUEL_ALARM or WATER_ALARM or ENGINE_SPEED or
    WHEEL_SPEED or TOTAL_TENTHS or PARTIAL_TENTHS];
  present
    case BELT_ALARM do
    % output a belt alarm
    if(?BELT_ALARM)
      then belt_alarm_val := 1;
    else
      belt_alarm_val := 0;
    end if;
    err_code := mscan_send(BELT_ALARM_CODE, belt_alarm_val);
    %
    % other cases follow, they are omitted here to save space
    %
  end present;
  when RESET;
end; % loop
end; % var
Wrappers to call API functions (from file api_lib.c):

#include "api_lib.h"

// Dummy file to allow compilation. For synthesis, the designer
// will include the header file for the API object code here.

// Initialize the mSCAN buffers for sending:
int mscan_init() {
  UINT8 i;
  // first perform a soft reset on the CAN module
  if(CAN_Init(FAST, 0) == ERR_OK) {
    for(i = 5; i < 12; i++) {
      // set up 7 buffers with ids 5 thru 12 to transmit data
      CAN_ConfigMB(i, TXDF, i, 0);
    }
  }
  return 0;
}

// Send a message containing the integer "data"
int mscan_send(int id, int data) {
  // create an array for the data - first byte holds length of array
  UINT8 data_buf[2];
  data_buf[0] = 1;
  data_buf[1] = data;

  // first load the data in the message buffer, then send it
  if(CAN_LoadMB(id, data_buf, 0) == ERR_OK) {
    CAN_TransmitMB(id, 0);
  }
  return 0;
}