Communication Synthesis for Embedded Systems with Global Considerations *

Ross B. Ortega and Gaetano Borriello
Department of Computer Science & Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350
{ortega,gaetano}@cs.washington.edu

Abstract

Designers of distributed embedded systems require communication synthesis to more effectively explore the design space. Communication synthesis creates or instantiates the necessary software and hardware required to allow system components to exchange data. This work examines the problem of mapping a high-level specification to an arbitrary, but fixed architecture that uses particular bus protocols for interprocessor communication. The approach detailed in this paper illustrates that global considerations are necessary to achieve a correct implementation. A communication model is presented that allows for easy retargeting to different bus topologies and protocols. The effectiveness of this approach is demonstrated by mapping a high-level specification to different architectures.

1 Introduction

With the decreasing cost of microprocessors, designers of embedded systems routinely consider a distributed system as the solution for their application. These systems are characterized by having heterogeneous processors connected by heterogeneous busses. For instance, the HP LaserJet design has three different processors and 2 different busses connecting the processors as well as many point to point connections [9]. The designers selected the most appropriate connections between the processors based upon the communication requirements of the functions mapped to the processors.

Designers of distributed systems are faced with many choices in connecting the various processors together. Upender and Koopman [13] list many standard bus protocols commonly used in embedded systems. Microprocessors targeted toward the embedded market incorporate support for the most popular protocols directly on chip. Semiconductor companies manufacture dedicated communication chips, chip sets, and hardware macros which directly implement particular protocols. Such products abstract away many of the low-level protocol details making it attractive for designers to choose a known protocol instead of creaing an arbitrary or proprietary one.

When designing a distributed embedded system, it is necessary to consider many different points in the design space. Designers require tools that allow them to map the same high-level specification onto these different architectures. Communication synthesis allows designers to investigate the tradeoffs between different allocations, partitionings, bus topologies, and bus protocols by managing the low-level protocol details necessary to realize an implementation. Consider the example shown in Fig. 1. The high-level behavioral specification calls for process S_a to communicate with S_b. A designer evaluating an architecture with two processors connected by a Controller Area Network (CAN) [15] bus maps S_a and S_b to processors P_a and P_b, respectively. Given the designer mapping, communication synthesis allocates a communication chip and interfaces it to P_a, uses the built-in CAN controller of P_b, modifies and optimizes the the device-drivers and the real-time kernels to match this configuration and allow communication over the CAN bus.

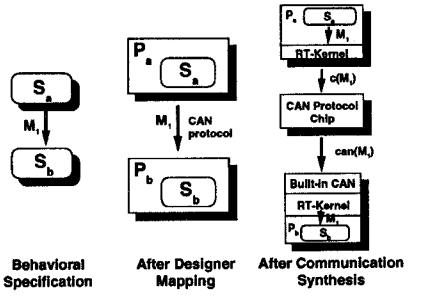
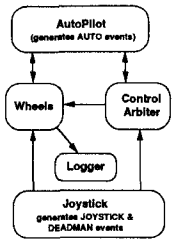


Fig. 1: Communication Synthesis

The above example considers the communication of two processes in isolation from the rest of the system. However, to effectively synthesize the communication for a fixed protocol, global system analysis is required. This analysis includes the frequency and type of events that will be transmitted on the bus and the bus topology. Bursty communication patterns may require local queues so that important events are not lost. If there is not a direct connection between communicating components, then intermediate processes will be required to relay the data from one bus to another. Protocol details such as basing bus arbitration on message or processor priorities along with the

*This work was supported by ARPA contract DAAH04-94-G-0272

0-8186-7895-X/97 $10.00 © 1997 IEEE



Message	Destination	Response Time
AutoGetHead	Wheels	25ms
AutoGetVel	Wheels	25ms
AutoHead data	Wheels	25ms
AutoVel data	Wheels	25ms
AutoPilot data	Arbiter	20ms
Joystick data	Wheels,Arbiter	100ms
DeadManHalt	Wheels,Arbiter	5ms
DeadManGo	Wheels,Arbiter	10ms
WheelsHead data	AutoPilot,Logger	50ms
WheelsVel data	AutoPilot,Logger	50ms
ArbiterAuto	Wheels,Arbiter	75ms
ArbiterJoystick	Wheels	75ms
ArbiterIdle	Wheels	75ms

Fig. 2: Communication in the robot

messages' response time constraints impact the allocation of priorities. A poor allocation may yield an infeasible system whereas a different allocation with the same target architecture may give a realizable implementation.

Recently there has been much attention focused on the problem of communication synthesis for distributed real-time embedded systems. These efforts either do not consider the global properties of the communication links or consider the global properties but map to one non-standard protocol. Ernst and Benner [5] proposed a communications library with a standard API (Application Program Interface). However, protocols based on message priorities require an allocation of the priorities in addition to providing an API. Gajski *et al.* [6] consider all of the events on the bus, but they implement a simple low-level bus protocol and do not address real-time kernel synthesis. Yen and Wolf [14] address the problem of heterogeneous processors connected via arbitrary bus topologies. However, they discount standard protocols as uncommon in embedded systems and assume an abstract protocol based on processor priorities. Gasteier and Glesner [7] attempt to synthesize busses which do not require arbitration. This approach is more suitable for data-flow oriented systems that have more predictable communication patterns than control-dominated systems. All of these approaches attempt to synthesize the bus topology.

This paper addresses the problem of synthesizing the communication for an arbitrary, yet static, bus topology. Instead of optimizing designers out of the design process, this approach allows designers to easily map their high-level designs to various implementation architectures for comparison. Designers can rapidly explore many more points in the design space than current techniques allow. The synthesis tool requires a behavioral description and a mapping to a particular architecture. All of the remaining details of system communication are synthesized.

Throughout this paper we will use the example of the robot control system shown in Fig. 2. The robot has two fundamental modes of operation. In *joystick* mode, the robot responds to a joystick manipulated by the operator. In *auto-pilot* mode the robot is controlled by a program running in the auto-pilot process. If at any time in either mode the operator releases the *dead man* switch, the robot immediately halts. A control arbiter process determines the operational mode of the system. The logger process records the periodic broadcasts from the wheels controller indicating the current heading and velocity.

2 Communication Model

We have developed a communication model suitable for reactive real-time embedded systems. The model is based on a set of processes which communicate by exchanging non-blocking messages. A non-blocking protocol is more appropriate for distributed real-time systems than a blocking protocol [8] partly because it decouples computation from communication. When a process executes a send statement it returns immediately after passing the message to the real-time kernel. Messages from other processors are received asynchronously via an interrupt indicating a message arrival. The real-time kernel performs minimal processing of the message and returns control to the previously executing process. Upon a time-triggered system interrupt, the received message is made visible to the receiving process. This is similar to synchronous digital logic where the inputs are not valid until a clock edge latches them.

A message is composed of an event name with optional data. In the robot, the dead man switch sends a *DeadManHalt* message with no data. The wheels controller sends a *WheelsHead* message with the current heading included in the data field. Messages may have multiple destinations.

Processes may specify receiving attributes that state how large a queue the system should allocate for a particular message. A queue size of one indicates an overwrite policy. If a different instance of the same message type arrives, then any previous unreceived message is lost. Along with the queue size, the designer can specify the behavior in case the queue becomes full. The choices available are: drop the incoming message, queue the incoming message and drop the message at the queue's head, or send a queue full message to the application with the message to be dropped. The final attribute that may be specified is a response time constraint. Similar attributes may be specified for the sending of messages. In the robot, all command messages have an overwrite policy without notification. Only the logger process has a queuing policy with notification. This particular notification routine simply records that data was lost.

A behavioral description consists of a set of communicating processes. A process contains state information that may be used for intraprocess communication. In the case of the wheels controller, the state variables include the current heading and velocity. A process has output ports for sending generated messages and input ports for receiving messages. There is a unique message port for each message type. The wheels controller has two output ports, *WheelsHead* and *WheelsVel*, and many input ports for the messages generated by the joystick, auto-pilot, and control arbiter.

A handler is a subroutine invoked to perform a service on behalf of a message. The typical handler consumes the triggering message, modifies state variables, generates outgoing messages and terminates. A handler may runs for a bounded amount of time and executes with *run to completion* semantics [10]. That is, once a handler begins executing it has the illusion of running without preemption. No other handlers from the same process may begin until the currently running handler terminates. Therefore, even though a handler may be preempted, the state of the process remains constant while the handler is not executing. The real-time kernel may preempt a handler for two reasons. First, an incoming message may need to be retrieved or an outgoing message may need to be sent. Second, the scheduler may allow a handler in a different process to execute.

A process may contain time-triggered handlers

specified with an invocation rate. The real-time kernel schedules these handlers at the appropriate times. The wheels controller has a control loop that runs every second. It checks the current heading with the target heading and if necessary turns the wheels a fixed increment to correct the heading.

A mode is a mapping of incoming messages to the handlers that consume them. The mode definitions simplify analysis for the scheduler by defining mutually exclusive behavior and allow the filtering of unwanted messages. In the robot's manual mode, the wheels controller responds to joystick messages and ignores auto-pilot messages.

3 Communication Synthesis

Communication synthesis is the process of realizing the communication links between the processes that exchange messages. The designer provides a behavioral specification using the communication model presented earlier. This behavioral specification includes the processes, a mapping of the generated messages to their destination processes, and the response time constraints on the messages. In addition to the behavioral specification, the designer provides an architectural specification which includes a list of processing elements, a mapping of the processes to the processing elements, a bus topology with bus protocols, and a mapping of messages to particular busses. Communication synthesis is divided into interprocessor communication and intraprocessor communication.

3.1 Interprocessor Communication

Interprocessor communication synthesis customizes the behavioral description to realize the appropriate bus protocol, introduces a device-driver to communicate either directly with the bus or via a communications chip, and enhances the real-time kernel to implement the message attributes regarding queuing and notification. The first step is to insure that there is a direct link between each message source and its destination. If not, then the designer must specify a path from the message's source to its destination. A routing process is automatically inserted on each processor along this path. There is at most one routing process per processor.

Routing a message from one bus to another illustrates the need for a global approach to communication synthesis. The designer places an initial response time constraint relative to the time the message is generated. The communication synthesis tool must calculate a new response time for this message so that it can effectively determine the appropriate priorities along the message's path. Previous work in determining the worst-case delay for transmitting a message such as [12] require restrictions which are incompatible with our communication model. Currently we use optimistic estimates and leave for future work algorithms that give more realistic timing results.

The next step groups all of the messages that will be sent on a particular bus. Protocol attributes are assigned to the messages and processors based on the arbitration scheme of the bus. We have modified the taxonomy in [13] to focus on the attributes which are required for protocol synthesis. Our taxonomy considers protocols that base arbitration on message priority, master/slave, time-based priority, and processor priority. This taxonomy also includes protocols not suitable for global prioritization such as Ethernet. The designer-specified bus protocol (*e.g.* CAN) is automatically placed into this taxonomy and the protocol attributes are determined accordingly.

Message-based priority protocols give the most flexibility to the synthesis tool in meeting the timing requirements of the system. Priorities are assigned according to the response time requirements of the individual messages on the bus. Messages with smaller response times have higher priority and ties are arbitrarily broken but consecutively allocated.

In a master/slave protocol the master processor polls the slave processors to see if any require the bus. Under this protocol, there are different higher-level protocols that can be implemented. For instance, it is possible to have message priorities by having the master poll all of the slaves and grant the bus to the slave with the highest priority message to send. However, such a protocol has a high overhead. A different protocol is to grant the bus to each slave in a round-robin or some other pre-determined order. We are investigating metrics to automatically select the most appropriate policy based on the global analysis of the designer's specification. Under both policies the bus master is chosen to be the processor with the least utilization. When implementing polling in a pre-determined order, the order the slaves are granted the bus is based on the frequency and response time constraints of the generated messages.

A variation of the master/slave protocol is one based on time. Under this protocol, time is conceptually the master and all of the processors are slaves. Each processor is granted a time slice during which it can send messages over the bus. Similar to the master/slave protocol above, the processors are granted the bus in a fixed order with the timing master selected as the processor with the lowest utilization.

Processor-based priorities are problematic for real-time systems. For instance consider a process that generates an infrequent yet short response time message M_1, but normally generates long response time messages M_2. If the processor is given a high priority to guarantee the timing constraint of M_1, then all of the M_2 messages inherit this high priority, potentially causing a priority inversion with messages from other processors on the bus. Currently we allocate processor priorities according to the shortest response time of any message sent on the bus. As timing analysis techniques improve it may be possible to perform a better allocation of processor priorities. If the allocation results in an infeasible solution, the designer may either repartition the processes to different processing elements or manually assign the processor priorities.

After the protocol specific attributes have been determined, the behavioral specification may require modification to reflect these attributes. For message-based priority protocols, the priority must be incorporated into the message send. Note that simply having a send API (subroutine call) is insufficient to realize the protocol because the message priorities are not determined until after the communication synthesis tool has analyzed all of the messages on the bus. The processes may come from reusable modules so assigning static priorities at the behavioral level is not possible. The tool must modify the send call to incorporate this additional information. Consider the following example from the robot where all of the processes are mapped to their own processor and communicate via a CAN bus (see Fig. 4a). The CAN protocol has message-based priorities with non-destructive

```
class CANDriver {
    // send handler with data
    canSend (CanPriority priority, Byte data, Byte size) {
        CanMessage m (priority, data, size);
        disableCanChipInterrupt();
        writeToChipOrSendQ(m);
        enableCanChipInterrupt();
    }
    canChipHandler () {
        // called when either message sent or arriving
        if (messageArriving()) {
            CanMessage m = getMessage();
            placeInKernelBuffer (m);
        } else sendNextMessageOnQ()
    }
}
```

Fig. 3: CAN device-driver

contention for the bus. When using this protocol, all of the send subroutine calls in the high-level specification that send messages without data are automatically replaced with the new call canSend (canPriority). Sends for messages that contain data are replaced with the call canSend (canPriority, canData, dataSizeInBytes). The joystick process sends a *DeadManHalt* message to the wheels process. Because this message has the smallest response time, it is assigned the highest priority, *CanP0*. The joystick handler *deadManHandler* contains the statement send (DeadManHalt) which is replaced with the subroutine canSend (CanP0). The wheels handler *reportHeading* broadcasts the current heading of the robot and makes the call send (WheelsHead, heading). This statement is replaced with the call canSend (CanP7, heading, 2). The CAN protocol has a limit of eight data bytes. Messages larger than eight bytes are automatically divided into multiple canSend calls with consecutive priorities.

After the send calls have been modified, the next step is to allocate the queues according to the individual message attributes from the behavioral specification. The queues have a common API called by the device-drivers for the particular bus protocol. This allows the the real-time kernel to send and receive messages independently from the execution of the processes.

Once the queues have been inserted, the bus protocol device-driver is instantiated from a protocol library. These device-drivers are written using the communication model from the previous section. The device-driver has two primary handlers that execute during the normal operation of the system. The first one is the protocol specific send routine which executes in the application handler. For the CAN protocol this is the canSend routine from above. The role of the send handler is to create a protocol specific message packet and write this packet directly to the communications chip or to the kernel's send buffer. The second entry point is responsible for receiving messages and sending messages that are on the send queue. The handler that directly interacts with the communications chip responds to messages (actually interrupts) from the chip. In the case of the Siemens SAE 81C90 Stand Alone Full CAN Controller [1], the chip generates interrupts when either a message arrives or is transmitted. For the CAN device-driver, this handler is canChipHandler (see Fig. 3). For communication chips that do not generate transmit interrupts, the chip's handler in addition to responding to receive interrupts, is time-triggered to insure that the send queue is emptied. Such details are encapsulated in the protocol library.

The high-level device-drivers instantiated above make low-level calls that interact with the communications chip. If the processor has built-in support for a bus protocol, then the

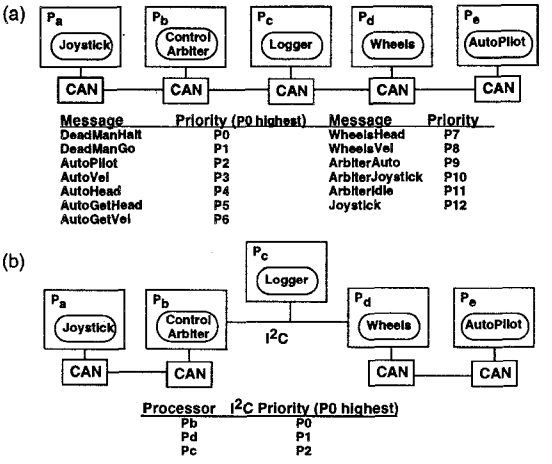


Fig. 4: Robot mapped to different architectures

given interface to this internal peripheral involves reading and writing particular registers or memory locations. However, it may be necessary to use an external communications chip such as the SAE 81C90. In [4] and [3] it was shown how to automatically connect peripheral devices to a microprocessor by synthesizing glue logic and reflecting the new hardware interface to the device in the low-level device-driver. Using these techniques, we can synthesize a bus interface for processors which do not internally support a given protocol.

The final step in interprocessor communication synthesis is to modify the real-time kernel to deliver messages to the appropriate processes. When a message arrives it is placed on a system queue according to the message's receive attributes. A time-triggered kernel handler checks the incoming queues, strips out protocol information encoded in the message and passes the scheduler a list of handlers that are ready to execute. The scheduler selects a handler to execute by passing it an incoming message.

3.2 Intraprocessor Communication

Synthesizing communication for intraprocessor messages involves similar steps as interprocessor messages. However, all intraprocessor communication occurs via shared memory. Since the handlers execute with run to completion semantics and there is only one source for a particular message type, there is no contention for an internal channel. Therefore, it is sufficient to use a simple API that implements the appropriate queue semantics respecting the high-level message attributes. The real-time kernel maps the sending messages to the corresponding internal destination channels.

4 Examples

To demonstrate the usefulness of this approach to communication synthesis, the robot described above was mapped to different bus topologies and protocols. We used these mappings as a proof of concept and did not attempt to achieve a minimal cost system. Since we focused on the problems associated with synthesizing the communication, abstract microprocessors are used in the mappings. However, using the techniques in [4] and [3], it is not difficult to use off-the-shelf processors or processor cores.

The first mapping places each process on its own processor and all of the processors are connected via a CAN bus. Portions of this implementation were discussed above and are shown in Fig. 4. The priorities were allocated according to the response time attributes on the individual messages.

The next mapping uses two different CAN busses with an I^2C bus connecting them (see Fig. 4b). The joystick process generates the message *DeadManHalt* which has the shortest response time of any message in the system. This message must be delivered from P_a to the wheels process on P_d and the control arbiter process on P_b. The designer indicates that this message will be routed via P_b and its I^2C bus to P_d. Routing processes are inserted on I^2C processors, P_b and P_d. Since this highest priority message travels from P_b to P_d, processor P_b is assigned a higher I^2C priority. This example clearly shows that global considerations are necessary when synthesizing the communication.

The next example maps the robot to processors connected by the NuBus [2] protocol. NuBus arbitrates based on processor priority but uses a fair arbitration scheme. All processors wanting the bus assert a bus request when the bus is idle. All of these requesting processors will be granted the bus once in priority order before any other processor can request the bus. The highest priority processor has the largest processor identity (ID) code. Since this is a processor-based priority protocol, IDs are assigned according to the shortest response time message on each processor. Because the joystick process generates *DeadManHalt*, processor P_a is assigned the largest ID.

The finai example maps the robot to processors on a Firewire (IEEE 1394 High Performance Serial Bus) backplane bus [11]. Firewire supports multiple arbitration schemes. Under fair arbitration the protocol is similar to NuBus. Under isochronous access, a timing master begins of a timing cycle with a nominal period of 125µs. Upon detecting a cycle start, those processors with isochronous data arbitrate for the bus. For the robot, all data transfers occur via isochronous arbitration. The Control Arbiter has the smallest utilization so processor P_b is made the timing master (highest priority processor) running the time-triggered *timingMaster* handler with a period of 125µs.

5 Conclusion

Designers of distributed embedded systems require tools to explore in detail different points in the design space. Communication synthesis allows designers to easily investigate the tradeoffs between different architectures by managing the low-level protocol details required to realize the communication among the system components. A global view of communication synthesis is necessary to map to those fixed protocols which are most suitable for real-time systems. The communication model that was presented allows for easy retargeting to different protocols and architectures. This approach allows designers to map their high-level specifications to arbitrary architectures. These ideas were validated by mapping a high-level specification to different bus topologies and protocols.

The next step in providing the designer with feedback regarding a particular implementation is to incorporate the synthesized system into a timed simulator. This will allow the designer to obtain performance statistics necessary to evaluate different designs. There are many improvements possible regarding interprocessor communication. One such optimization is to bundle multiple messages into a single message to reduce bus overhead. Another research area is to more clearly identify the core attributes that communication protocols share making it easier to incorporate new protocols into a protocol library. We are currently investigating new algorithms to give better derived response time constraints for messages routed from one bus to anther. We are also looking at other protocols such as Universal Serial Bus and PCI bus to insure that these ideas are applicable to the widest range of bus protocols.

References

[1] SAE 81C90/91 *Stand Alone Full CAN Controller, Preliminary Data. Siemens*, http://www.siemens.de/Semiconductor/products/ICs/34/pdf/81c90.pdf, 1996.

[2] Apple Computer. *Designing Cards and Drivers for the Macintosh Family.* Addison-Wesley, 1990.

[3] P. Chou, R. Ortega, and G. Borriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1992.

[4] P.H. Chou, R.B. Ortega, and G. Borriello. Interface cosynthesis techniques for embedded systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1995.

[5] R. Ernst and T. Benner. Communication, constraints, and user-directives in COSYMA. Technical Report TM CY-94-2, Technical University of Braunschweig, June 1994.

[6] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[7] M. Gasteier and M. Glesner. Bus-based communication synthesis on system-level. In *Proceedings of the International Symposium on System Synthesis*, November, 1996.

[8] H. Kopetz et al. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25-40, February 1989

[9] A.H. Mebane IV, J.R. Schmedake, I-S Chen, and A.P. Kadonaga. Electronic and firmware design of the HP LaserJet Drafting Plotter. *Hewlett-Packard Journal* 43(6):16-23, December 1992.

[10] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[11] M. Tenner. A bus on a diet - the serial bus alternative - an introduction to the P1394 High Performance Bus. In *Digest of papers COMPCON*, Spring 1992.

[12] K. Tindell and A. Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. Technical Report YCS-94-229, University of York, 1994.

[13] B.P. Upender and P.J. Koopman Jr. Communication protocols for embedded systems. *Embedded Systems Programming*, 7(11):46-58, November 1994.

[14] T.-Y. Yen and W. Wolf. Communication synthesis for distributed systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1995.

[15] H. Zeltwanger. An inside look at the fundamentals of CAN. *Control Engineering*, 42(1), January 1995.