Communication Synthesis for Distributed Embedded Systems *

Ross B. Ortega
Dept. of Computing and Software Systems
University of Washington
Bothell, WA 98021-4900 USA
ortega@u.washington.edu

Gaetano Borriello
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350
gaetano@cs.washington.edu

1. ABSTRACT

Designers of distributed embedded systems face many challenges in determining the tradeoffs when defining a system architecture or retargeting an existing design. Communication synthesis, the automatic generation of the necessary software and hardware for system components to exchange data, is required to more effectively explore the design space and automate very error-prone tasks. This paper examines the problem of mapping a high-level specification to an arbitrary architecture that uses specific, common bus protocols for interprocessor communication. The communication model presented allows for easy retargeting to different bus topologies, protocols, and illustrates that global considerations are required to achieve a correct implementation. An algorithm is presented that partitions multihop communication timing constraints to effectively utilize the bus bandwidth along a message path. The communication synthesis tool is integrated with a system co-simulator to provide performance data for a given mapping.

1.1 Keywords

communication synthesis, interprocessor communication, multihop communication, bus protocols, hardware/software co-synthesis, distributed heterogeneous embedded systems

2. 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, an HP LaserJet design has three different processors and two different busses connecting the processors as well as many point to point connections [12]. The designers selected the most appropriate interprocessor communication based upon the requirements of the functions mapped to each processor.

Designers of distributed systems are faced with many choices in connecting the various processors together. Upender and Koopman [20] list many standard bus protocols commonly used in embedded systems. It is increasingly attractive for designers to use a known protocol instead of creating an arbitrary or proprietary one. 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 (cores) which directly implement particular bus protocols. These products abstract away many of the physical low-level protocol details. However, to effectively use these protocol chips or cores, many application-specific details must be considered in deriving the remaining protocol parameters.

When designing a distributed embedded system, it is necessary to consider many different points in the design space to achieve the appropriate cost/performance ratio. Each new design point forces the system designer to rederive all of the application-specific protocol parameters and customize the communication subsystem to reflect the current architecture. Designers require tools to map the same high-level specification onto different architectures so that the various tradeoffs can be quickly and easily measured. Fully automating the design space exploration (target architecture and partitioning) is not feasible and fails to exploit the talents of system architects. However, automating the construction of the tedious and error-prone portions of the communication subsystem frees the designer to consider a larger number of potential solutions. Communication synthesis allows designers to investigate the tradeoffs between different allocations, partitionings, bus topologies, and bus protocols by managing the low-level protocol and real-time kernel details necessary to realize a complete implementation.

Consider the following design scenario. The system architect for a robot control system needs to evaluate the two different architectures shown in Figure 1. The first architecture consists of five processors connected by a Controller Area Network (CAN)[23] bus. The second architecture has two

ICCAD98, San Jose, CA, USA

*This work supported by NSF PYI MIP-885872 and DARPA DAAH04-94-G-0272

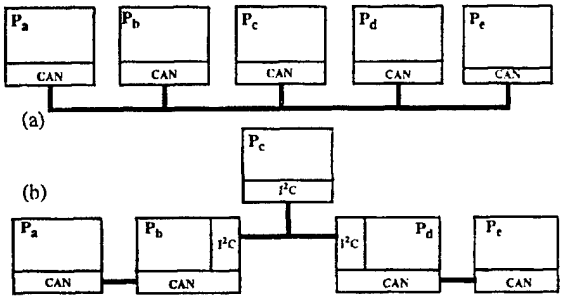


Figure 1. Two potential architectures for a robot control system. (a) has five processors on a shared CAN bus. (b) has an I^2C bus between two CAN busses.

separate CAN busses connected by an Inter-Integrated Circuit (I^2C) bus[15]. Without a communication synthesis tool, the designer must develop a unique communication infrastructure for both architectures. All of the low-level details such as the bus protocol parameters and formats for each interprocessor communication, the device-drivers, message routing information, and the timing constraints for all communications must be captured in executable code before the designer can begin evaluating a given architecture. If a mistake is made in any of the details, then changing one parameter may require a rederivation of all the communication parameters in order to meet the system's timing and performance constraints. Once this daunting task has finally been completed for one architecture, the designer must completely redo most of this work before evaluating the second architecture. This task is so time-consuming that system architects typically consider a very limited number of design alternatives and frequently only one.

Now consider the same scenario with a communication synthesis tool. Taking a high-level specification consisting of communicating processes, the designer assigns each process to a processor and maps the communication path for each message (see Figure 2). Communication synthesis generates a customized real-time operating system for each

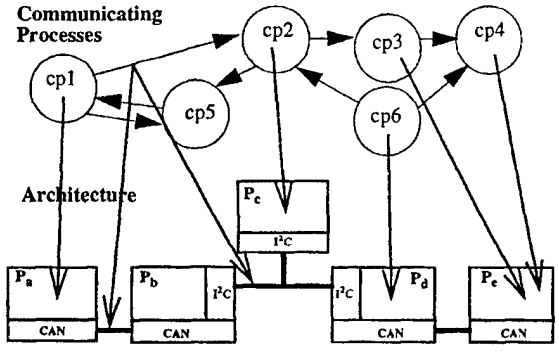


Figure 2. The system architect maps the behavioral specification of communicating processes to an architecture. Notice that the communication from cp1 to cp2 is mapped across two busses. Not all mappings shown.

processor taking into account the particular bus protocols, routing requirements, and timing constraints for all of the communication in the system

Even very simple systems can benefit from communication synthesis as shown in Figure 3. The high-level behavioral specification calls for the process *producer* to communicate with the process *consumer*. A designer evaluating the given architecture maps *producer* and *consumer* to processors P_a and P_b, respectively. Given the 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 device-drivers and the real-time kernels to match this configuration, and derives the protocol parameters to allow communication over the CAN bus.

The above example illustrates the synthesis of communication for two processes in isolation from the rest of the system. However, to correctly and effectively synthesize the communication for a bus protocol, global system analysis is required. All of the traffic on the various system busses must be considered so that timing constraints are respected. Bursty communication patterns may require local queues so that important events are not lost. If there is not a direct connection between communicating processes, *e.g.* cp1 and cp2 in Figure 2, then intermediate "hop" processes are 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 messages' timing constraints impact the allocation of these priorities. All of these details must be considered when creating a communication infrastructure.

Recently there has been much attention focused on the problem of communication synthesis for distributed real-time embedded systems[13]. Many of these efforts either do not consider the global properties of the communication links or map to non-standard protocols. Vahid and Tauro[21] and Ernst and Benner [6] both proposed using a communication library with a standard API (Application Program Interface). However, protocols based on message priorities require a unique allocation of all the priorities on the bus in addition to providing an API. Rowson *et al.* proposed a new methodology, Interface-Based Design, where the designer successively refines the communication from abstract tokens down to the final implementation. This

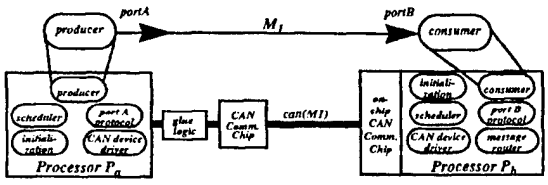
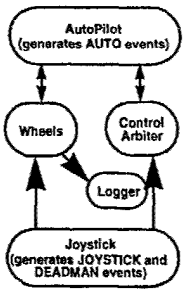


Figure 3. An example of communication synthesis. The designer maps process *producer* to processor P_a, *consumer* to P_b and message M_1 to the CAN bus. Communication synthesis generates a customized real-time operating system and glue logic to transmit and deliver the message.

paper can be viewed as automating this approach. Daveau *et al.* [5] take a behavioral description and automatically select a protocol from a library to implement the communication. They use non-standard protocols such as bidirectional handshake and dual fifo. Gajski *et al.* [7] consider all of the events on the bus, but they implement a non-standard bus protocol and do not address real-time kernel synthesis. Yen and Wolf[22] address the problem of heterogeneous processors connected via arbitrary bus topologies. However, they assume an abstract protocol based on processor priorities. CoWare[2] supports heterogeneous processors, but focuses on shared memory communication and non-standard protocols. Gasteier and Glesner[8] attempt to synthesize busses that do not require arbitration. This approach is more suitable for data-flow oriented systems with predictable communication patterns than for control-dominated systems. There has also been work done in the area of scheduling messages in a multiprocessor environment to meet real-time and quality of service constraints [10]. Message scheduling globally analyzes the communication requirements of the system to create an effective scheduler. However it is assumed that there is a simple API providing access to the bus. As discussed above, not all protocols can be implemented without a synthesis step prior to using the API. Another alternative is to use a real-time operating system (RTOS). Although an RTOS provides a flexible communication infrastructure, the designer must still derive and manage most of the details necessary to realize a given mapping. For instance, an RTOS permits interprocessor communication, yet no explicit support is given for multihop communication where a message must travel on multiple busses. The designer must keep track of the timing constraints and routing of each multihop message.

This paper addresses the problem of synthesizing the communication for an arbitrary bus topology specified by the system architect. Instead of optimizing designers out of the design process, this approach allows system architects to easily map high-level designs to different implementation architectures for evaluation. Designers can rapidly explore many more points in the design space than current techniques allow. The synthesis tool we have implemented requires a behavioral description and a mapping of high-level functions to the computational components of a particular architecture. All of the remaining details of system communication are automatically synthesized. The effect is that an application-specific real-time operating system is generated for each processor in the system. The communication synthesis tool has been fully integrated with a system co-simulator [9] to quickly provide designers with performance information for a given mapping.

Throughout this paper we will use the example of the robot control system shown in Figure 4. The robot has two fundamental modes of operation. In *joystick* mode, the robot is controlled by a joystick manipulated by the operator. In *auto-pilot* mode the robot is automatically 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 messages from the wheels process indicating the current heading and velocity.



Message	Destination	Constraint
AutoGetHead	Wheels	25ms
AutoGetVal	Wheels	25ms
AutoHead *data*	Wheels	25ms
AutoVel	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, AutoPilot	75ms

Figure 4. Communication requirements in a robot control system consisting of five communicating processes.

The next section presents a communication model appropriate for specifying real-time embedded systems. Section 4 discusses how we synthesize communication for a system described with this communication model. Section 5 discusses the integration of the communication synthesis tool with a co-simulator. Section 6 contains examples of mapping the robot to different bus protocols and topologies.

3. COMMUNICATION MODEL

We have developed a communication model suitable for reactive real-time embedded systems. The model is based on a set of processes that communicate by exchanging non-blocking messages. A non-blocking protocol is more appropriate for distributed real-time systems than a blocking protocol [11] partly because it decouples computation from communication. When a process executes a message *send*, 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. When the process is next invoked by the real-time scheduler, the received message may be made visible to the receiving process. Messages may have multiple destinations, but can have only one source.

A behavioral description consists of a set of communicating processes. 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. In the robot example, 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 process also 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.

The designer may specify receiving attributes on an input port 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 previously received, but as yet unconsumed, message is lost. Along with the queue size, the designer may 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. 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.

Similar to [10] various message attributes must be specified to enable global analysis. These attributes include the maximum size of any message generated on an outport, the maximum frequency at which the messages may be generated, and a required response-time constraint. The maximum size and frequency are necessary to calculate the bandwidth requirements of each message.

Even though the communication model is fundamentally one of non-blocking communication, the designer can designate any output port to have blocking semantics. The communication synthesis tool automatically generates acknowledge messages and ports and modifies the scheduler to implement blocking behavior.

When a process is granted the processor, the real-time scheduler calls one of the process's handlers. 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 only run for a bounded amount of time and executes with *run to completion* semantics [16]. 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. In this model, run to completion semantics eliminates the need for user-level semaphores which are difficult to use correctly and complicate timing analysis.

In addition to event-triggered handlers scheduled by message arrival, a process may contain time-triggered handlers specified with an invocation rate. For example, the wheels controller has a control loop that runs every 100ms.

4. COMMUNICATION SYNTHESIS

Communication synthesis is the process of implementing the communication links between the processes that exchange messages. Figure 5 shows the inputs and outputs of the communication synthesis tool. The designer provides

Figure 5. The inputs / outputs for communication synthesis

a behavioral specification using the communication model presented in Section 3. This behavioral specification consists of the processes, the connection of output ports to input ports, and the various attributes associated with each port. In addition to the behavioral specification, the designer provides an architectural specification which includes a list of processing elements, a bus topology with bus protocols, a mapping of the behavior specification processes to the processing elements, and a mapping of port connections to particular busses. Taking the inputs shown in Figure 5, the communication synthesis tool analyzes the communication patterns and then customizes a real-time operating system for each processor. Interprocessor communication is divided into single-bus and multihop communication. In single-bus communication the source and destination processors share the same bus. Intraprocessor communication is a special case of single-bus communication. In multihop communication, a message travels on multiple busses to reach its destination.

4.1 Distribution of Real-Time Constraints for Multihop Messages

Multihop messages clearly demonstrate the need for global analysis in communication synthesis. All of the bus traffic in the system must be accounted for in order to effectively partition a timing constraint among the various busses. The designer specifies an initial real-time constraint that is a deadline for the message to be delivered to its destination. The communication synthesis tool must distribute this deadline along the message's path so that the protocol parameters for all messages can be effectively determined. Previous work in determining the worst-case delay for transmitting a message such as [19] require restrictions which are incompatible with our communication model. For example, the assumption that a message's timing constraint must be less than the period of the sending process implies that a message can only have a send queue depth of one. No such restrictions exist in our model of non-blocking communication.

The overall approach for multihop communication synthesis is divide and conquer. A multihop message is divided into submessages for each hop that a message takes to reach its destination. First, we require the designer to explicitly enumerate every path a message must take from the source to each destination. Next, a heuristic algorithm, proportional effective bandwidth (PEB), calculates the deadlines for the submessages such that the sum of deadlines plus overhead on any path is less than or equal to the original deadline

The main idea behind PEB is to first optimistically

determine the time required to deliver a message to all of its destinations and then proportionally distribute any remaining time among the various hops taking into account all other messages that may compete for busses along the message's path. Each deadline can be divided into two components: the time required to transmit the message on the bus (*minXmitTime*) and the time remaining before the deadline expires (*extraTime*). For example, if a message has a deadline of 5ms and it takes 1ms to transmit it over a particular bus, then *minXmitTime* is 1ms and *extraTime* is 4ms. Note that minXmitTime is a function of the user-message size, protocol formatting fields such as headers and checksums and the raw bandwidth of the bus. After *minXmitTime* has been calculated for each hop on all paths, the optimistic worst-case path delay, *optDelay*, over all paths is computed. This delay is optimistic because it does not account for other bus traffic. It is the minimum time necessary to transmit the message to all of its destinations. A processor-specific delay, *hopDelay*, accounts for the time a processor takes to read in a message on one bus and transmit it on another bus. *ExtraTime* is defined as the original deadline minus the optimistic worst-case delay. More formally:

$$\text{ExtraTime} = \text{Deadline} - \text{OptDelay}$$

$$\text{OptDelay} = \max\left(\forall\text{paths}\left(\sum_{path_j} \text{minXmitTime}_i + \text{hopDelay}_j\right)\right)$$

$$\text{minXmitTime}_i = \frac{\text{messageSize}_i + \text{protocolOverhead}_i}{\text{bw}_i}$$

ExtraTime is proportionally distributed along each path taking into account contention on the busses. An effective bus bandwidth, *effBw*, is computed for each bus. A scaling factor taking into account the longest latency path at each hop allocates the extra time remaining, *ETR*, from the source to its destinations. Formally:

$$\text{deadline}_i = \text{minXmitTime}_i + \text{extraTime}_i$$

$$\text{effBw}_i = (1 - \text{utilization}_i)\text{bw}_i$$

$$\text{extraTime}_i = \frac{\frac{1}{\text{effBw}_i}}{\frac{1}{\text{effBw}_i} + \max\left(\forall\text{paths}\left(\sum_{x=i+1}^{\text{endOfPath}} \frac{1}{\text{effBw}_x}\right)\right)} \text{ETR}_i$$

$$\text{ETR}_i = \text{ExtraTime} - \sum_{k=1}^{i-1} \text{extraTime}_k$$

Consider a robot architecture where every process is mapped to its own processor. A portion of the topology is shown in Figure 6 where the Joystick processor sends a message with a deadline of 10ms to the Wheels and Arbiter processors. To simplify this example we assume the hopDelay and the protocol overhead is negligible. The minimum transmit times for a joystick message of 10 bytes are 0.8ms, 0.4ms, and 0.2ms for bus1, bus2, and bus3, respectively. OptDelay of 1.2ms is determined by the slower Joystick/Wheels (0.8ms+0.4ms) path. ExtraTime, 10ms - 1.2ms = 8.8ms, is proportionally distributed among the two hops. First consider the case where all three busses are lightly loaded so the effective bandwidth is essentially the raw bandwidth of the busses:

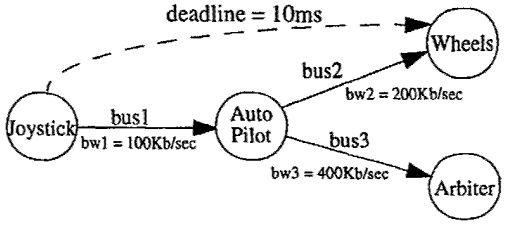


Figure 6. Distributing deadlines. A message is sent from the joystick processor over bus1 to the AutoPilot processor and then to the Wheels and Arbiter processors.

$$\text{extraTime}_1 = \frac{\left(\frac{1}{100\text{Kb/sec}}\right)(8.8\text{ms})}{\frac{1}{100\text{Kb/sec}} + \max\left(\frac{1}{200\text{Kb/sec}}, \frac{1}{400\text{Kb/sec}}\right)}$$

$$\text{extraTime}_1 = 5.87\text{ms}$$

$$\text{extraTime}_2 = \text{extraTime}_3 = 2.93\text{ms}$$

$$\text{deadLine}_1 = 0.8\text{ms} + 5.87\text{ms} = 6.67\text{ms}$$

$$\text{deadLine}_2 = \text{deadLine}_3 = 3.33\text{ms}$$

Now consider the case where bus3 has a utilization of 75% giving it an effective bandwidth of only 100Kb/sec.

$$\text{extraTime}_1 = \frac{\left(\frac{1}{100\text{Kb/sec}}\right)(8.8\text{ms})}{\frac{1}{100\text{Kb/sec}} + \max\left(\frac{1}{200\text{Kb/sec}}, \frac{1}{100\text{Kb/sec}}\right)}$$

$$\text{extraTime}_1 = 4.4\text{ms}$$

$$\text{extraTime}_2 = \text{extraTime}_3 = 4.4\text{ms}$$

$$\text{deadLine}_1 = 0.8\text{ms} + 4.4\text{ms} = 5.2\text{ms}$$

$$\text{deadLine}_2 = \text{deadLine}_3 = 4.8\text{ms}$$

The heavy load on bus3 causes the message to be delivered to the AutoPilot processor earlier than in the non-loaded case. The effect on the deadline for bus b2 is that there is more freedom to allow other messages to have a higher priority since from this path's point of view, the message has arrived earlier than necessary. Clearly, the performance of a bus on a different path can have a system-wide impact.

After the timing constraints have been partitioned for all multihop messages in the system, hop processes are automatically inserted where needed. The hop processes are treated as user processes for the duration of communication synthesis.

4.2 Single-Bus Interprocessor Communication

The system description that now consists of user and hop processes mapped to processors that communicate via single-hop messages with timing constraints. Single-bus interprocessor communication synthesis customizes this description to realize the selected bus protocols, introduces

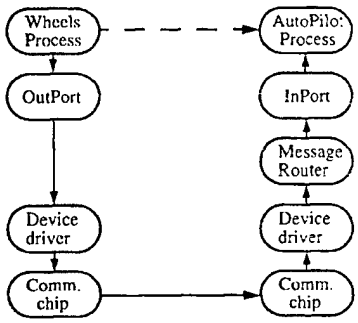


Figure 7. The Wheels process sends a message to the Auto-Pilot process (dashed line). The actual communication flow automatically generated is shown by the solid lines.

device-drivers to communicate directly with the bus or via a communications chip, enhances the real-time kernels on each processor to route messages to their destinations, and implements the message attributes regarding queuing and notification. The communication flow for a simple point to point communication is shown in Figure 7. The end result is that an application-specific real-time operating system is generated for each processor.

The first step groups all of the messages that are 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 [20] to focus on the attributes which are required for protocol synthesis. Our taxonomy considers protocols that base arbitration on message priority, processor priority, master/slave, time, time/processor priority hybrid, and non-priority schemes. The designer-specified bus protocol (*e.g.* CAN) is automatically placed into this taxonomy and the protocol attributes are determined according to the heuristics presented below.

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 deadlines of the individual messages. Messages with smaller deadlines have higher priority with ties broken arbitrarily but consecutively allocated.

Because of priority inversion [16], processor-based priorities are problematic for real-time systems and give the least flexibility to the synthesis tool. For example, consider a process that generates an infrequent and short deadline message M_1, but normally generates long deadline message 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 deadline of any message sent on the bus.

In a master/slave protocol, the master processor polls the slave processors to see if any require the bus. There are different higher-level protocols that can be implemented on top of this protocol. 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. An alternative 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.

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. The master sends out a heart-beat message and then the processors send out their message at a given delta time from this heart-beat.

After the protocol specific attributes have been determined, the behavioral specification is modified 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. Furthermore, 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 as in Figure 1a. The CAN protocol has message-based priorities with non-destructive contention for the bus. When using this protocol, all of the *send* subroutine calls in the high-level specification are automatically replaced with two new subroutines. The first routine takes the user-level message along with protocol attributes synthesized using the heuristics mentioned above and creates a new low-level bus message. The second routine calls the device-driver with the low-level bus message which passes the message to the communication chip.

The designer is abstracted away from low-level protocol details. For instance, the CAN protocol has a limit of eight data bytes. Messages larger than eight bytes are automatically divided into multiple CAN messages sharing the same id. The eighth data byte is filled with a constant indicating that more data for this particular message is pending. The CAN device-driver receiving the message builds up the original behavioral message before delivering it to the message router.

The next step customizes the user processes. An input port data structure is instantiated for each behavioral input port and implements the queuing semantics according to the

individual message attributes from the behavioral specification. Each port is given a unique id and at run time registers itself with the message router described below.

Once the processes have been transformed, the bus protocol device-drivers are instantiated from a protocol library. These device-drivers are written using the communication model from the previous section. The device-driver has three 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's handler. It stores a message in the device-driver's send queue and immediately returns (a non-blocking send call). The second entry point is an interrupt handler that sets a flag indicating that an interrupt occurred. The third entry point, the execute method called by the scheduler, is responsible for receiving packets and sending out any messages on the send queue.

If the processor has built-in support for a bus protocol, then the given interface to this internal peripheral only requires software instructions to access the particular control registers. However, it may be necessary to use an external communications chip such as the SAE 81C90[1]. In [4] and [3] it was shown how to automatically connect peripheral devices to a microprocessor by synthesizing any necessary 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 device-driver is also responsible for stripping out protocol specific attributes and re-constructing the original behavioral message from the received packets. After an entire behavioral message has been received, the device-driver passes the message to the processor's message router which delivers the message to each of the destination ports according to the message receive attributes of the port. The message router is customized for each processor. It contains a mapping of messages to the input ports for the processes (including hop processes) mapped to this processor.

5. CO-SIMULATION

At this point the communication synthesis tool has synthesized the necessary information to construct an application-specific communication architecture. To provide system architects with an integrated rapid-prototyping environment, the tool also generates all of the files needed to run a timing accurate co-simulation of the synthesized system. The designer can attach logic analyzer probes to individual busses and gather statistics about any message in the system. The communication synthesis tool generates code to automatically log the generation and reception of all messages. Analysis of the log file gives system architects performance information allowing them to quantify various architectural tradeoffs and validate the performance of the synthesized communication infrastructure.

6. EXAMPLES

The robot from Figure 4 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. The communication synthesis tool was run as an interpreted Java application on a 233MHz PowerPC 750. The execution time results are summarized in Table 1.

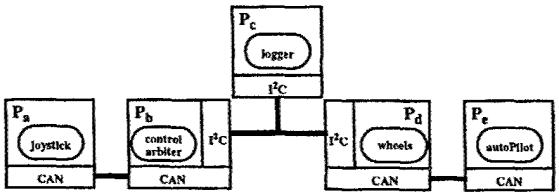


Figure 8. A mapping of the robot control system. After partitioning the timing constraints among the busses, hop processes are placed on P_b and P_d, and the protocol parameters are derived for the messages.

Mapping	# of synthesis files	Time to generate syn. files (s)	# of sim. files	Time to generate sim. files (s)	Total time (s)
CAN bus	23	5.03	7	1.47	6.5
I^2C bus	23	5.09	7	1.34	6.43
CAN/I^2C	35	7.46	7	1.48	8.95

Table 1: Execution time results for the communication synthesis tool on three mappings of the robot specification.

The first two mappings place each process on its own processor and all of the processors are connected via a common bus. The processors have an on-chip protocol processor. To go from the CAN mapping to the I^2C mapping required changing only 11 lines of code in the architectural description: 5 different processors, 5 different bus interfaces, and a different bus. These small modifications illustrate the ease of considering different mappings. In the first mapping a 1 Mb/sec CAN bus is used while the second one uses an 400Kb/sec I^2C bus. For each mapping the synthesis tool generated 23 Java files (a total of 3000 lines of code): 13 files output port class definitions, 5 enhanced user processes, and 5 message routers. For simulation, the tool generated an additional 7 files (5 simulated processors, a netlist, and the Makefile).

The next mapping uses two different CAN busses with an I^2C bus connecting them (see Figure 8). The joystick process generates the message *DeadManHalt* which has the shortest deadline of any message. It 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 is routed via P_b and its I^2C bus to P_d. Therefore, a hop process is placed on P_b. Since this highest priority message travels from P_b to P_d, processor P_b is assigned a higher I^2C priority. In a similar fashion the autoPilot process must communicate with the control arbiter causing a hop process to be placed on P_d. Within 9 seconds after the mapping, we were able to

begin simulating. Synthesis for this mapping takes longer because the deadlines are partitioned for multihop communication, generating files for the hop processes (6 messages are multihop) and their corresponding outports.

Each of the architectures was executed in the Pia co-simulation environment[9]. The joystick process was modified to send out periodic commands. After 20 commands, the logging of the system was halted. A few of the more interesting statistics are shown in Table 2. This

Mapping	DeadManHalt (min, max) (µs)	Joystick (min, max) (µs)	WheelsVel (min, max) (µs)	AutoPilots (min, max) (µs)
CAN bus	47, 47	29, 155	97, 165	38, 134
I^2C bus	99, 197	97, 5022	616, 960	144, 339
CAN/I^2C	29, 309	29, 12275	97, 132	38, 438

Table 2: Evaluation of the three robot mappings. Min and max transmit times are shown for selected messages.

type of data can be used by system architects to quantify architectural tradeoffs. *DeadManHalt* is sent from the joystick processor to the wheels and arbiter processors. It has a faster minimum delivery in the CAN/I^2C architecture because of bus contention in the other architectures. The message was sent when another message was being transmitted. Since *DeadManHalt* has the highest priority, it is the next message on the bus. When the CAN/I^2C system has bus traffic there is a longer delay of 309µs. Evaluating the performance of these three architectures, the system architect can choose the most appropriate architecture.

7. CONCLUSION

Designers of distributed embedded systems require tools to explore in detail different points in the design space. Communication synthesis allows designers to investigate the tradeoffs between different architectures by managing the low-level protocol and routing details required to implement system communication. A global view of communication is necessary to map to those fixed protocols which are most suitable for real-time systems. The communication model presented allows for retargeting to different protocols and architectures. Designers can map high-level specifications to arbitrary architectures. The communication synthesis tool is fully integrated with a co-simulator so that designers can gather performance statistics to evaluate tradeoffs between different architectures.

8. ACKNOWLEDGEMENTS

We would like to thank Ken Hines for his support during the integration of the communication synthesis tool with Pia.

9. REFERENCES

[1] SAE 81C90/91 *Stand Alone Full CAN Controller, Preliminary Data.* Siemens, 1996.

[2] I. Bolsens, H.J. DeMan, B. Lin, K. Van Rompaey, S. Vercauteren, D. Verkest. Hardware/software co-design of digital telecommunication systems. In *Proceedings of the IEEE*, 85(3):391-418, March 1997.

[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 co-synthesis techniques for embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, November 1995.

[5] J-M. Daveau, G.F. Marchioro, T.Ben-Ismail, and A.A. Jerraya. Protocol selection and interface generation for hw-sw codesign. *IEEE Trans. on Very Large Scale Integration* (5)1:136-144, March 1997.

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

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

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

[9] K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proceedings of the 34th Design Automation Conference*, pp. 395-400, June 1997.

[10] D.D. Kandlur, K.G. Shin, and D. Ferrari. Real-time communication in multihop networks. *IEEE Trans. on Parallel and Distributed Systems* (5)10:1044-1055, October, 1994.

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

[12] 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.

[13] R.B. Ortega, L. Lavagno, and G. Borriello. Models and methods for hw/sw intellectual property interfacing. In *1998 NATO ASI on System-level Synthesis*.

[14] R.B. Ortega and G. Borriello. Communication synthesis for embedded systems with global considerations. In *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, March 1997.

[15] I^2C Peripherals for Microcontrollers. Philips Semiconductors, 1992.

[16] R. Rajkumar, L. Sha. and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1988.

[17] J.A. Rowson, A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the Design Automation Conference*, pp 178-83, June 1997.

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

[19] 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.

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

[21] F. Vahid and L. Tauro. An object-oriented communication library for hardware-software codesign. In *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, March 1997.

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

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