IPCHINOOK: An Integrated IP-based Design Framework for Distributed Embedded Systems

Pai Chou

Consystant Design Technologies, Inc.
Seattle, WA
chou@consystant.com

Ross Ortega, Ken Hines, Kurt Partridge, and Gaetano Borriello

Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA
{ortega,hineskj,kepart,gaetano}@cs.washington.edu

Abstract

IPCHINOOK is a design tool for distributed embedded systems. It gains leverage from the use of a carefully chosen set of design abstractions that raise the level of designer interaction during the specification, synthesis, and simulation of the design. IPCHINOOK focuses on a component-based approach to system building that enhances the ability to reuse existing software modules. This is accomplished through a new model for constructing components that enables composition of control-flow as well as data-flow. The designer then maps the elements of the specification to a target architecture: a set of processing elements and communication channels. IPCHINOOK synthesizes all of the detailed communication and synchronization instructions. Designers get feedback via a co-simulation engine that permits rapid evaluation. By shortening the design cycle, designers are able to more completely explore the design space of possible architectures and/or improve time-to-market. IPCHINOOK is embodied in a system development environment that supports the design methodology by integrating a user interface for system specification, simulation, and synthesis tools. By raising the level of abstraction of specifications above the low-level target-specific implementation, and by automating the generation of these difficult and error-prone details, IPCHINOOK lets designers focus on global architectural and functionality decisions.

1 Introduction

The complexity of modern embedded system design requires designers to leverage the reuse of both software and hardware IP modules. In this paper, we focus on software components (e.g., MPEG decoders, control algorithms, and user interfaces). These modules are designed with a particular API that cannot adapt well to new system contexts. Many assumptions about the way the components will be used are embedded in their implementation, making it difficult (if not impossible, due to concerns about proprietary information) and time consuming (to understand their code well enough) to make the appropriate modifications.

This work was supported by DARPA contract DAAH04-94-G-0272, a Mentor Graphics graduate fellowship, and PYI MIP-8858782.

DAC 99, New Orleans, Louisiana

This *IP integration* problem is exacerbated by the fact that many embedded systems use multiple processing elements and highly specialized communication topologies in the interests of meeting cost, power, and performance constraints. The modules must now not only be composed appropriately but their activities must also be coordinated across more than one processor. The communication and synchronization code required to do this cannot be generally anticipated. A common solution is to provide a general embedded operating system that can perform these functions. However, this still requires adapting the modules to the new run-time environment and opens up the design to inefficiencies due to the full generality of this operating system.

If we follow this IP-based approach to system design along its logical trajectory, we see that the design cycle is dominated by system integration time, rather than component design. In such a scenario, the task of generating the low-level interfaces between components and optimizing their coordination must be automated so that designers can investigate architectures and partitionings that better satisfy the design constraints. For designers' efficiency and for IP protection, tools are needed that provide specific support for system integration.

IPCHINOOK is an example of a new generation of IP-oriented system design tools. It is targeted to specifically address the three problems of:

- *IP composition*

  Component models of software IP suffer from some fundamental problems. First, their fixed APIs may not anticipate how the components will be used, thus limiting their applicability when designers cannot directly modify their code, and causing reverse engineering and maintenance problems when designers can. Second, composing multiple IP components often requires designers to spend an inordinate amount of time writing and debugging custom integration code. Finally, due to information hiding and imperfect anticipation of what internal component state needs to be exported, the integration code often ends up duplicating the state of the components (and their associated transition functions) in order to make it visibile. In IPCHINOOK, we formalize component coordination into a separate system assembly step that leads to reusable composition constructs. To accomplish this, we permit the explicit coordination of state/control between components in addition to an event-based model appropriate for data-flow and more traditional APIs.

- *Communication synthesis*

  Intermodule communication software must be tailored to the application by taking into account factors such as the capabilities of the processing elemenets, their interconnection mechanisms, the system's physical topology, and how functionality

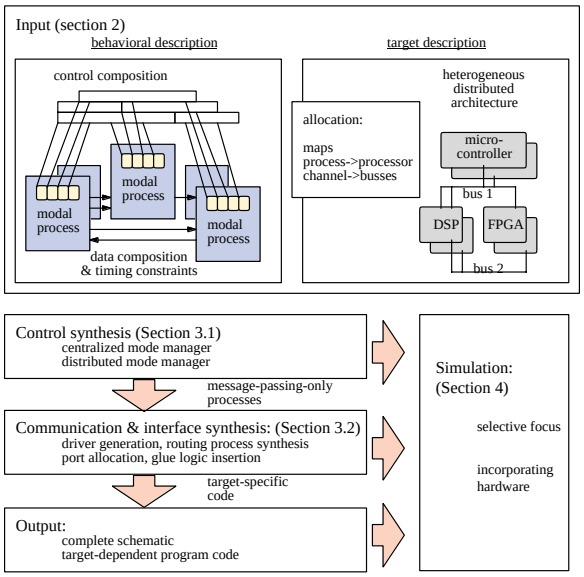

Figure 1: Overview of the IPCHINOOK design framework.

is distributed across the system. Automating this process radically shortens the design cycle and makes the specification portable to different architectures without compromising efficiency.

- *Rapid evaluation*

  Co-simulation is needed at different levels of the design process, from the original high-level specification of an integrated set of modules to detailed implementation. Speeding up simulation along with the ability to collect profiling information provides the feedback the designer needs to alter the target architecture and/or specification and explore the design space in fruitful directions. We apply a technique called selective focus that enables a simulation to expend its cycles where details are required rather than uniformly over the entire design.

This paper provides an overview of the IPCHINOOK framework (see Figure 1). It first describes the behavioral model of modal processes, which enable control composition using high-level primitives without intrusive modification. Next, the synthesis stages are described, including the mode manager (run-time coordinator) and communication and interface synthesis. Finally, the paper covers the Pia simulator, which provides the user interface and framework for the designers to perform most of their design and development tasks.

2 Specification Model

The input to IPCHINOOK consists of a behavioral description, a target description, and an *allocation function* that maps between them. The behavioral description contains the functionality of the system described as one or more concurrent interacting modules known as *modal processes*. The target description describes the available processors, I/O devices, communication busses, and topology of the hardware. The mapping between the behavioral description and target architecture determines which processor each modal process will run on. This structure allows functionality, hardware, and the distribution of functionality all to be changed independently of one another.

2.1 Behavioral description

2.1.1 Modal processes

A modal process consists of *ports*, *handlers*, and *modes*. Ports provide logical communication contact points for interprocess communication. Ports are connected by *channels*. Each channel connects a single output port to one or more input ports for transmitting messages. A message arrival at an input port is an *event*, which triggers the invocation of a handler. A handler encapsulates application code, sends messages on output ports, and returns mode change requests. Handlers have run-to-completion semantics, which saves the designer from having to worry about intercomponent synchronization [25].

The modes of a modal process can be independently active or inactive. Modes also specify a mapping from ports to handlers. When a mode is active then if a message is received on a port p, and a mapping exists in the mode from p to handler h, h will be invoked. Since multiple modes can be simultaneously active, multiple handlers can be called in response to a single message receipt, although each handler is invoked only once per message. Handlers are invoked in a statically-defined order.

The execution of modal process handlers is divided into discrete *steps*. In a step, messages queued on input ports are dispatched to the handlers that are enabled by the active modes. During its execution, a handler can send one or more messages on its output ports, but there is at least a one-step delay before the receiving handlers can be triggered, as all input buffers are at least one-deep.

Changes to active modes are made by a three-phase process. The first phase, *vote collection*, takes place immediately after all handles in a given step have finished executing. Each handler returns a set of *votes* that indicate which modes should be activated or deactivated. A handler can only request changes to modes local to its modal process, although a local mode change can have a global impact through ACTs, as described below. In the second phase, *vote reconcilation*, votes and ACTs are examined to determine what, if any, mode changes should be made at the end of this step. Conflicts between votes are reconciled by priorities assigned to votes by handlers or ACTs. After reconciliation, in the final phase, *vote distribution*, the new set of active and inactive modes is distributed to all affected modal processes. Different synchronization models affect the distribution as described in section 3.

2.1.2 Control composition

In order for components to coordinate with each other, they must agree on a protocol. If they do not already agree, then *adaptation* is necessary. A novel feature of IPCHINOOK is the use of high-level primitives called Abstract Control Types (ACTs) for control coordination. ACTs establish automatically maintained relationships between modes. ACTs cause votes to be augmented with additional mode requests to maintain the required relationships.

ACTs can be used for many purposes, such as using one mode to guard a mode change in another, correlating modes so they are always active or inactive at the same time, and establishing a mutual exclusion relationship between modes. More complex ACTs can form a hierarchical FSM similar to Statecharts [14] or Esterel [2]'s watchdogs. IPCHINOOK provides a rich library of ACTs and allows designers to define their own abstractions in a structured manner. More rigid composition frameworks force designers instead to use awkward constructs such as overlapping superstates.

An ACT can define a protocol for interprocess composition. Components may need adaptation to be composed with each other, but this organization enables adaptation of modal processes without intrusive modification. Note that this differs from the modularity

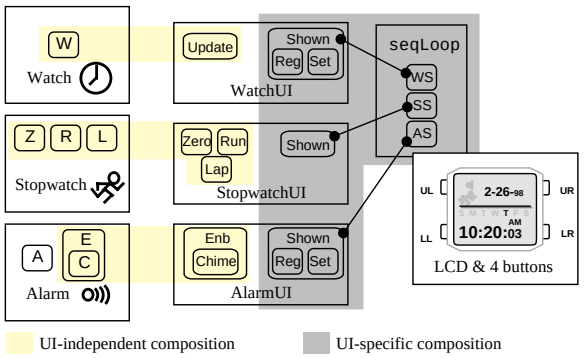

Figure 2: Example of Modal Process architecture for a digital wristwatch. Each functional component is coordinated with a UI component. UI components are coordinated by the `seqLoop` ACT.

provided by Statemate's ActivityCharts, which allows modules to communicate only using traditional message-passing.

Consider as an example application the digital wristwatch example used by Esterel and Statecharts. The behavioral description can be decomposed into the six modal processes: one to control the passage of time (watch), another to control the stopwatch, another to control the alarm, and three others corresponding to the user interfaces of these particular modal processes. Figure 2 depicts this structure graphically, including the modes involved. An ACT, `seqLoop`, constrains the `Shown` mode of each of the user interface processes to arrange for their successive activation as the user repeatedly pushes the "mode-change" button. A different activation pattern, such as changing the cyclic ordering or allowing multiple modes to be simultaneously active could be imposed simply by changing the ACT or its constraints. By incorporating this abstraction into the modal process structure, reuse is supported, composition behavior is cleanly separated from module behavior, and consistency of the replicated state is guaranteed. The original Esterel and Statecharts versions of the wristwatch example do not provide a clean module separation. Without support for coordination between modules, the application programmer must arrange to broadcast updates to state values whenever they are changed to all interested modules—an error-prone and difficult-to-maintain task.

For a more detailed description of ACTs, see [6].

2.2 Target description

A target description defines a desired target *architecture* and the *allocation* function that maps the modal processes and channels to the architecture. Unlike some other cosynthesis systems, IPCHINOOK does not attempt to automate partitioning and allocation; both are expected to be supplied by the designer or by automatic architecture generation tools [19]. An architecture is defined by its processors, operating system, and communication protocols. The allocation function maps elements in the high-level design to those in the architecture.

A processing element can be a microprocessor running software or a programmable logic block, such as an FPGA. IPCHINOOK is specifically designed to take advantage of commercial off-the-shelf solutions. The architecture specification also includes the communication topology, which describes how components are connected to each other and which communication protocols are used. To maximize the use of off-the-shelf parts, IPCHINOOK provides a rich library for supporting standard communication protocols, including I^2C, CAN, SCSI, USB, IrDA, and Ethernet.

The allocation function maps processes to processors, and logical communication channels to architectural communications links. The process-to-processor mapping is many-to-one: that is, processes are assumed to be indivisible, but each processor can run multiple processes. On the other hand, the communication mapping is many-to-many: a logical link can be routed through several physical segments, and each physical channel can carry multiple logical channels. Even though the specification uses a message-passing style, IPCHINOOK can synthesize code that runs on a shared memory system [21].

3 Synthesis

Synthesis is the transformation of a high-level design representation to a lower-level one, which takes the design one step closer to implementation. The synthesis stages include mode-manager synthesis and communication and interface synthesis.

3.1 Mode-manager synthesis

A set of modal processes composed using ACTs has many possible implementations. IPCHINOOK takes the approach of synthesizing *mode managers* for coordinating the processes. A mode manager is the part of the run-time system that manages control communications according to the ACTs. The mode manager ensures that the system always runs in a coherent context by handling the state maintenance task. When targeting heterogeneous distributed architectures, IPCHINOOK automatically synthesizes mode managers, one for each processor, to handle intricate synchronizations associated with the control communication.

Mode manager synthesis can be done before and after target mapping. Without a specific architecture, IPCHINOOK synthesizes a *centralized* mode manager that can be readily executed on a uniprocessor for simulation. The processes by default are assumed to run *mode synchronously*: all processes are blocked (and no handlers run) until the mode changes are resolved. For the distributed version, several available implementation options allow designers to make tradeoffs between space, performance, and determinism. IPCHINOOK supports several synchrony models, from the strict *mode synchrony* and *event synchrony* to loosely coupled *communication synchrony* and *dataflow synchrony* [7]. Mode synchrony is safe but inefficient; it requires handshaking for every change of state. Complete asynchrony (such as is used in [3] and [5]) can be more efficient, but is subject to glitches and livelock. These options can be specified for each architecture mapping, without having to modify the same high-level specification.

3.2 Communication and interface synthesis

Communication synthesis and interface synthesis implement an application-specific communication infrastructure so that the application's data messages and the mode manager's control messages are handled efficiently by the target architecture. Communication synthesis implements abstract communication protocols on the given target architecture allowing modal processes to exchange messages. Interface synthesis generates interfacing logic and low-level device-drivers to connect processing elements together.

Many different types of communication are used in system designs and at different stages in the design process. Abstract communication protocols are used in the target-independent system specification. These abstract protocols must be transformed and implemented on top of the low-level bus protocols mandated by the target architecture. In the target-independent specification, each output port must be annotated with both a blocking style (blocking or non-blocking) and a deadline constraint. Each input port must be annotated with the appropriate queuing semantics such as the queue

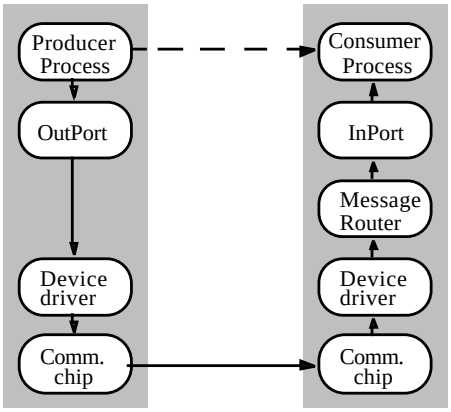

Figure 3: The designer is presented with the communication abstraction shown as the dashed line. The automatically generated communication infrastructure is shown by the solid lines.

size and overflow behavior. These annotations are specified by the designer for data ports, and by the mode manager synthesis code for its generated control ports. Communication synthesis refines these abstract communication protocols from the target-independent design into a target-dependent implementation, and tailors the mode manager's communication code for each processor in the target architecture. Figure 3 shows the communication abstraction presented to the designer and the actual communication infrastructure that realizes the abstraction. The communication synthesis tool accounts for the particular bus protocols, routing requirements, and timing constraints for all of the communication in the system. This approach, which takes a global view of communication, allows designers to map their high-level system specifications to target architectures composed of arbitrary bus topologies. The system architect is insulated from the tedious details of bus protocols, device-drivers, and various other operating system constructs necessary to achieve a working prototype.

The steps for communication synthesis in IPCHINOOK [21] are (1) multi-hop deadline distribution, (2) bus protocol attribute synthesis, (3) message router generation, and (4) device-driver instantiation. Interface synthesis then takes as input the target architecture and the instantiated device-drivers.

Multi-hop communication synthesis creates *hop processes* for situations where logically connected processes are mapped to processors not directly connected. The hop processes route the message through intermediate processors and busses to reach the destination process. Since the abstract communication link supports a unidirectional one-to-many connection topology, a message may travel on multiple paths to reach all of its destinations. The deadline associated with each message is distributed along the message's paths so that the bus protocol parameters for all messages can be effectively determined. The deadline partitioning algorithm takes a global view of communication by considering all the bus traffic in the system. Automating multi-hop communication synthesis with the distribution of deadlines and the insertion of hop processes is necessary to support target architectures that can have any bus topology.

Bus protocol attribute synthesis examines all of the messages that must be transmitted over a particular bus and determines the appropriate protocol parameters for each of these messages. These messages are then transformed into bus messages that incorporate the synthesized protocol-specific attributes. A protocol taxonomy based on the bus arbitration policy of the protocol has been defined [21] to enable the synthesis of these parameters. These bus protocol attributes may be message IDs, processor IDs, message or processor priorities, and queues, depending on the specific protocol. They are synthesized so that messages with tighter timing constraints have a higher priority when arbitrating for the bus. The bus message also includes routing information so that the customized real-time operating system executing on each processor can deliver the message to its destination.

The device-drivers, along with the bus messages containing the synthesized protocol attributes, abstract the designer from the peculiarities of a given bus protocol. For instance, the Controller Area Network (CAN) protocol [20] transfers only eight data bytes per message. The customized device-driver on the sending processor automatically divides larger messages into eight byte segments and the receiving processor reconstructs the message. The reconstructed message is passed to the synthesized message router, which examines the message's routing information and delivers the message to the appropriate input ports. The synthesized input ports incorporate the queueing semantics specified in the high-level system description.

The device-drivers are automatically instantiated from a protocol library. Interface synthesis customizes those device-driver routines that directly read and write the physical pins of the processor. These routines must reflect any introduced glue hardware logic. For the interface between a processor and hardware, multiple techniques can be used to physically connect the devices. Embedded microprocessors typically have I/O ports, which are pins connected to registers that can be accessed from software. The I/O ports are flexible and require little glue hardware. Interface synthesis allocates the I/O ports to maximize sharing the physical pins among the peripheral devices that need to be connected to the processor [10]. Memory-mapped IO is another common interfacing technique for embedded systems, where the software writes to “phantom” memory addresses. External address matching logic interprets these memory accesses as I/O communications. The phantom memory locations are called I/O addresses. When the I/O ports have been fully utilized, interface synthesis performs memory-mapped I/O by inserting multiplexing logic as necessary to realize the connection. To minimize the address matching logic, different encoding techniques are used to assign the I/O addresses [11].

Communication synthesis allows system architects to investigate the tradeoffs between different mappings, bus topologies, and communication protocols. Exploring the design space is crucial to creating a cost-effective implementation that satisfies all system-level constraints. Moreover, communication synthesis also provides an effective means for system-level analysis by instrumenting for simulation, without having to modify the high-level specification.

4 Simulation

Hardware/software co-simulation lets designers execute a model of their embedded system [23]. The model can range from the initial target-independent description to the output of intermediate synthesis stages. Many hardware/software design automation tools have co-simulation support [1, 22, 26], however IPCHINOOK supports a novel technique called *selective focus* simulation [15]. This approach simulates at the highest level of abstraction whenever possible to achieve the fastest simulation speed. Since designers may sometimes need to inspect low-level details in isolated regions of the system, our approach must allow this as well. Instead of simulating the entire system at the detailed level, selective focus performs detailed simulation only in the regions of interest; other parts of the system are simulated at a higher level to provide the workload needed to drive the simulation. Selective focus lets designers

dynamically zoom in and out of different regions in a given simulation run.

The simulation tool, Pia, also includes support for a style of debugging that exploits the higher levels of abstraction (i.e. modal processes and ACTs) enabled by IPCHINOOK[17]. This allows designers to step through a mode trace or event trace and observe the system's behavior.

4.1 Implementing selective focus

The simulator is able to obtain the selective focus effect by keeping track of several versions of entry calls for each interface, and by choosing the appropriate version based on the detail level at which a region of the system is operating. A set of versions for a consistent single level of detail is called a *runlevel*. Normally, each interface has two runlevels, one that goes through any lower level interfaces it uses, and another that goes directly to similar interfaces on other components. To illustrate this point, consider the example system, "WubbleU" [27]. WubbleU is a small PDA that is connected to the Internet through a wireless connection (in this case IrDA), and a software server on a host machine. Figure 4 shows the protocol stack used by the components on either side of the wireless link. Each interface in this protocol stack is given two runlevels; one for performing communications in the normal way—through the rest of the stack—and one for accelerating the simulation by communicating directly with the similar interface on the other component.

There are several issues to consider when implementing runlevels. For example, there must be a way for communicating interfaces to coordinate the runlevel at which they are currently operating. Also, when interfaces switch runlevels, they must not leave any residual state behind. In addition to these, each of the different runlevels held by any one interface must be essentially indistinguishable to the applications or other interfaces that use it. For example, in Figure 4, the link management interface should not be able to tell whether the link access layer is talking to the link access layer on another component directly, or through the physical layer. For the first two considerations, some simple rules suffice. For example, communication tokens can be tagged with the runlevel of their transmitting interface, such that recipients can ensure that they are handled in the appropriate manner, thus coordinating runlevels without handshaking. Also, the granularity for switching runlevels for a particular interface is an entry call, so that any request for a switch in runlevels will not be handled until it is safe to do so.

The last of these issues—namely generating alternate methods for different levels of detail—is not always quite so simple to resolve. Fortunately, it is usually possible to automatically generate runlevels based on information provided by the designer about the behavior of entry calls, and based on information derived from higher level specifications [16].

4.2 Including real hardware in simulation

Pia can incorporate real hardware to be used as part of the simulation [18], and it allows simulator nodes to be distributed across the Internet. The main advantage of this is that parts vendors can put real parts on the web, and allow designers to use these remotely through a simulator interface. In this way, designers can try out these parts in their designs without having to build a hardware prototype, or even purchase the parts. Other advantages are that IP providers can maintain tight controls on preliminary use of the IP by limiting access to only the external interface, and better performance can be achieved by mapping concurrent modules in a simulation to separate hosts.

The role played by selective focus in geographically distributed simulation can be quite substantial. By allowing the designer to dynamically trade abstraction for total communication overhead, better parallelism is achieved, without sacrificing any essential detail. This is because components can synchronize on larger scale communication actions rather than on all the inevitable low level actions required to perform them.

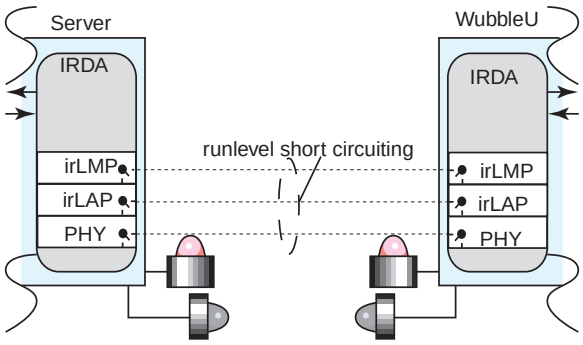

Figure 4: An IrDA protocol stack showing runlevel short-circuiting between the link management layers, the link access layers, and the physical layers.

5 Conclusions

IPCHINOOK is a comprehensive hardware/software cosynthesis framework for heterogeneous distributed embedded systems. It enables design space exploration by mapping a high-level design to the target architecture of designers' choice. The specification model enhances design reuse through control composition, and the synthesis tasks enhance retargetability through coordination, communication, and interface synthesis. Efficient simulation techniques let designers validate their design at different stages of synthesis without having to construct a hardware prototype. The simulator also gives them flexible observability to maximize simulation performance and a level of control that would otherwise be difficult to accomplish with hardware prototypes.

The current version of the IPCHINOOK user interface has been used to construct several system specifications with multiple mappings for each. The handlers that make up the executable code of the system specifications are written in a subset of Java (without dynamic creation and destruction of objects). In addition, the tools themselves are all written in Java.

IPCHINOOK presents a new model for IP-based design. Our future work will seek to further develop the modal process abstractions and composition methodology. Areas of future emphasis include (1) allowing verification of mode liveness and safety properties, (2) expanding debugging to directly use coordination information, (3) expanding the mode management framework to include hardware IP, and (4) broadening communication synthesis to support networked distributed embedded systems.

References

[1] BALBONI, A., FORNACIARI, W., AND SCIUTO, D. Co-synthesis and co-simulation of control-dominated embedded systems. *Design Autmation for Embedded Systems* (July 1996).

[2] BERRY, G. Programming a digital watch in Esterel v3.2. Tech. Rep. 1032, Instut National de Recherche en Informatique et Automatique (INRIA), May 1989.

[3] BERRY, G., RAMESH, S., AND SHYAMASUNDAR, R. K. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 1993), pp. 85–98.

[4] BOLSENS, I., DEMAN, H. J., LIN, B., ROMPAEY, K. V., VERCAUTEREN, S., AND VERKEST, D. Hardware/software co-design of digital telecommunication systems. *Proceedings of the IEEE 85*, 3 (March 1997), 391–418.

[5] CHIODO, M., ENGELS, D., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., SUZUKI, K., AND SANGIOVANNI-VINCENTELLI, A. A case study in computer-aided co-design of embedded controllers. *Design Automation for Embedded Systems 1*, 1-2 (January 1996), 51–67.

[6] CHOU, P. *Control Composition and Synthesis of Distributed Real-Time Embedded Systems*. PhD thesis, University of Washington, 1998.

[7] CHOU, P., AND BORRIELLO, G. An analysis-based approach to composition of distributed embedded systems. In *Proc. International Workshop on Hardware/Software Codesign (CODES/CACHE)* (1998).

[8] CHOU, P., AND BORRIELLO, G. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Proc. Design Automation Conference* (June 1998), pp. 88–93.

[9] CHOU, P., HINES, K., PARTRIDGE, K., AND BORRIELLO, G. Control generation for embedded systems based on composition of modal processes. In *Proc. International Conference on Computer-Aided Design* (1998).

[10] CHOU, P., ORTEGA, R., AND BORRIELLO, G. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proc. International Conference on Computer-Aided Design* (1992), pp. 488–495.

[11] CHOU, P., ORTEGA, R., AND BORRIELLO, G. Interface co-synthesis techniques for embedded systems. In *Proc. International Conference on Computer-Aided Design* (1995), pp. 280–287.

[12] DAVEAU, J.-M., MARCHIORO, G. F., BEN-ISMAIL, T., AND JERRAYA, A. A. Protocol selection and interface generation for hw-sw codesign. *IEEE Transactions on VLSI Systems 5*, 1 (March 1997), 136–144.

[13] ERNST, R., HENKEL, J., BENNER, T., YE, W., HOLTMANN, U., HERRMANN, D., AND TRAWNY, M. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems 20*, 3 (May 1996), 159–166.

[14] HAREL, D. StateCharts: a visual formalism for complex systems. *Science of Programming 8*, 3 (June 1987), 231–274.

[15] HINES, K., AND BORRIELLO, G. Dynamic communication models in embedded system co-simulation. In *Proc. Design Automation Conference* (June 1997), pp. 395–400.

[16] HINES, K., AND BORRIELLO, G. Optimizing communication in hardware-software co-simulation. In *Codes/CASHE '97* (1997), IEEE, ACM.

[17] HINES, K., AND BORRIELLO, G. Debugging distributed implementations of modal process systems. *Lecture Notes in Computer Science 1474* (1998), 98–107.

[18] HINES, K., AND BORRIELLO, G. A geographically distributed framework for embedded system design and validation. In *Proc. Design Automation Conference* (June 1998), pp. 140–145.

[19] ISMAIL, T. B., AND JERRAYA, A. A. Synthesis steps and design models for codesign. *IEEE Computer 28*, 2 (February 1995), 44–53.

[20] ISO 11898. *Road vehicles - Interchange of Digital Information - Controller Area Network (Can) for High-Speed Communication*, 1st ed., 1993.

[21] ORTEGA, R., AND BORRIELLO, G. Communication synthesis for distributed embedded systems. In *Proc. International Conference on Computer-Aided Design* (1998).

[22] PASSERONE, C., LAVAGNO, L., CHIODO, M., AND SANGIOVANNI-VINCENTELLI, A. Fast hardware/software co-simulation for virtual prototyping and trade-off analysis. In *Proc. Design Automation Conference* (1997), pp. 389–394.

[23] ROWSON, J. Hardware/software co-simulation. In *Proceedings of the Design Automation Conference* (1994), pp. 439–440.

[24] ROWSON, J. A., AND SANGIOVANNI-VINCENTELLI, A. Interface-based design. In *Proceedings of the Design Automation Conference* (June 1997), pp. 178–83.

[25] SELIC, B., GULLEKSON, G., AND WARD, P. T. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[26] VALDERRAMA, C. A., NACABAL, F., PAULIN, P., AND JERRAYA, A. A. Automatic generation of interfaces for distributed C-VHDL cosimulation of embedded systems: an industrial experience. In *7th International Workshop on Rapid Systems Prototyping* (June 1996).

[27] WubbleU hand held PDA benchmark for co-design, http://www.it.dtu.dk/jan/WubbleU.