Modal Processes: Towards Enhanced Retargetability through Control Composition of Distributed Embedded Systems [*]

Pai Chou and Gaetano Borriello
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA
{chou,gaetano}@cs.washington.edu

Abstract

To explore different points in the design space of an embedded system, it is important to be able to compose a design from reusable design components, and then map the resulting system description onto several possible target architectures with different partitionings of functionality. Today's specification models support composition styles that work well for data communication but not for control communication between concurrent processes to be mapped onto a distributed architecture. We propose a new retargetable system specification model that combines the best properties of process-based and hierarchical-FSM-based methods for modular composition of data and control. The model lends itself to automated synthesis of the run-time system for coordinating tasks on different processors in the system. The model and synthesis method are illustrated with several examples of embedded systems.

1 Introduction

Embedded systems are increasingly being implemented as distributed systems with heterogeneous processors. Distributed architectures are motivated by applications that must interact with multiple elements of the environment concurrently. For example, typical automobiles are now controlled by a distributed system that coordinates everything from the braking system and engine to the dashboard and climate control. These systems also tend to be heterogeneous, so that designers have more flexibility in optimizing the design to their specific objectives of cost, size, power, and performance. It is often imperative that designers explore many points in the design space, but at the same time, they are given less time to complete their designs. To meet these conflicting goals, successful designers must maximize design reuse and work at the highest possible level of abstraction.

[*]This work was supported by DARPA DAAH04-94-G-0272

Today, most designs are written in *implementation languages* such as C, Ada-95 (for avionic or military applications), or Java. These languages allow reuse through a *procedural interface*, and object-oriented languages further enable specialization and extension through inheritance. They work well for a wide variety of algorithmic descriptions on a single processor. However, they become very difficult to manage when several concurrent processes must interact with each other on a wide variety of target architectures. Programs that use threads are often very difficult to debug and do not behave consistently on different platforms. The run-time system also consumes high overhead that can be prohibitive on low-cost embedded processors. Furthermore, today's methodology relies heavily on legacy code and components that severely limit the optimizations a designer can explore.

What makes the development of a retargetable specification method difficult is the underlying dichotomy between data and control flow. Virtually all practical embedded systems contain both control and data aspects of behavior, and a good model for one aspect is awkward for the other. The most successful methods so far have taken a domain specific approach by fixing their assumptions about data and control. These models can be roughly divided into communicating processes and hierarchical state machines. We propose a new model that combines the best features of the two approaches.

We envision a design style where the designer composes a behavioral specification with reusable modules, and uses automated tools to map it onto several target architectures. These building blocks may be designed either in-house or be intellectual property. To adapt reusable components to the specific application, we propose a new way of customizing the behavior by *control composition*, instead of modifying individual components. By deriving a complete implementation of the system on the target architecture, designers would be able to closely evaluate not only the mix of components and their communication topology and protocols, but also the allocation of system tasks to different processors. Such tools would support systems with arbitrary topologies that use a rich set of components and interfaces.

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50 DAC98 - 06/98 San Francisco, CA USA

2 Motivating examples

Embedded systems are very diverse. We have selected two examples with different features that will allow us to illustrate our specification model.

2.1 Tessellator robot

The first example is a simplified version of the space shuttle's heat-shield tile tessellator robot. It can be controlled either manually with a joystick or automatically to inspect and replace tiles on the hull of the spacecraft. The mode of operation is selected by a toggle switch, and defaults to manual mode. For safety reasons, the robot moves only when a human operator is present and holding the deadman switch on the joystick. The robot must always halt whenever the switch is released. The joystick controls the heading and acceleration of the robot. In autonomous mode, the robot uses its sonar and bumper sensor for maneuvering. It moves forward normally, but if the sonar detects an obstacle, then the robot turns 45 degrees and continues forward. If its bumper is hit then the robot goes in reverse until two seconds after the bumper has been released continuously, and then turns 45 degrees and moves forward in the same manner as the sonar's reaction. The bumper overrides the sonar, switching from auto to manual mode overrides both, and releasing the deadman button stops all activities.

2.2 controller for a human-powered airplane

The RAVEN human-powered airplane [11] includes an embedded system to control its rudder and elevators. It runs one of several control algorithms based on user commands and regularly samples sensor values. The flight data can be logged in a flash memory module and transfered over a serial line for analysis after landing. The logger has two timing resolution options. The LCD-based user interface, which has seven command screens, is the most complex part of the design because it interacts with all other components in the system. Some commands are local to the user interface (*e.g.* changing screens, editing parameters) while others affect the system's modes of operation.

3 Related work

Existing models offer ways of composing modules to form a complete design. The ways existing models organize behavior can be divided into two styles: *functional encapsulation* and *temporal (state) encapsulation*.

3.1 functional encapsulation

Most systems are composed in a style we call *functional encapsulation*, as exemplified by process based models. Each process or module is an encapsulation of logically related functionality. Processes are a natural way of organizing design components, and they can be composed by message passing or signaling. Examples include synchronous dataflow (SDF) and dynamic dataflow (DDF) [6], CSP (strictly message passing) [9], and many concurrent FSM variants like CFSM [1] and SDL [12]. For design space exploration, processes define the granularity for partitioning. A designer can experiment with different assignments of processes to processors, and tools can help with system integration by synthesizing interprocessor communication [10]. Process models promote a modular design style – as long as the composition is limited to *data*. The modularity breaks down for control composition.

Data is concrete, while control is more abstract and can have many implementations: implicitly with the program counter or explicitly encoded and manipulated as data. Currently, control must be *encoded* as a command or a named signal in order to be communicated, and the receiver must be ready to interpret the command and change its control flow accordingly. This has been done successfully for specific domains, such as the subsumption architecture [4], where processes in a chain either send their own commands to the lower level or pass commands from above when overridden. Outside the specific domain, though, the problems with this approach are that data primitives are too low level for control and control is handled in an *ad hoc* manner by user code.

Control composition inherently requires processes to make use of a common set of states. To be modular, each process must maintain its own copy of the shared states. Each process is burdened with the responsibility of ensuring the coherence of their replicated states. Existing functional encapsulation models lack high-level primitives for expressing the state coherence requirements on the processes. Instead, state-coherence is handled imperatively in terms of transmitting state changes, interpreting the commands, and actually making those changes. Such low level code is embedded and scattered throughout each process and can be a major source of error.

3.2 temporal encapsulation

An alternative to functional encapsulation is to focus on the composition of control based on hierarchical state machines, as exemplified by hierarchical state machine models like StateCharts [7] and scoped watchdogs like Esterel [2]. Instead of embedding states in a process (in terms of the program counter or in state variables), explicit states and transitions are used to structure the behavior. Explicit states describe the behavior of the entire system and can be related in several ways: as parent/child, mutually exclusive, or parallel. Control composition by temporal encapsulation can be more modular than functional encapsulation because neither the preempting nor preempted state needs to know about each other, and different compositions do not require modifications to the behavior by explicitly anticipating the transition command and interpreting it. Also, there is no state coherence problem because there is only one set of states.

Temporal encapsulation can be difficult to partition for mapping onto a distributed architecture. It breaks process modularity because logically related functionality must be scattered in different states. Also, these models rely on the *perfect synchrony hypothesis* for control composition, but it is impractical for many distributed systems. As a result, languages that support temporal encapsulation also support process-like, parallel composition style using *choice* and signaling statements that use asynchronous composition at the system level, such as CRP [3] and ModuleCharts [8]. However, these are effectively processes without control modularity.

4 The modal process model

We propose a model that provides primitives for composing control independently of the communication semantics. The unit of composition is a *modal process*. It encapsulates functionality and defines the granularity of partitioning. Its behavior is organized into *modes*. Modes can be viewed as states of an incompletely specified state machine and serve as the interface for modular control composition. A distinguishing feature is that modes are managed by the run-time system code that we synthesize, according to the coherence requirements on the modes specified explicitly, rather than being embedded in user code and tied to the message passing semantics. By separating the specific synchronization requirements of the application from reusable behavior, this approach not only helps eliminating a whole class of bookkeeping errors related to state coherence, but also enhance module reusability in different applications and over different architectures.

4.1 modal processes

In our terminology, a **process** is a container of event handler routines. It may be derived from another model that assumes shared memory, message passing, event-driven, time-driven, dataflow, *etc*. In general, events may include I/O, interprocess communication, timing, dataflow, and mode transitions. We assume nonblocking communication, but the *synchrony* can be parameterized.

A **modal process** is a process with several modes, where a **mode** is, informally, a way of handling events. Each mode defines a collection of handlers to use while the mode is active. Different modes may pick different handlers to handle an event or ignore it. Modes allow a modal process to have several possible behaviors.

Although modes are similar to states, there are several differences. A traditional FSM has a transition function δ that maps a state $q_i \in Q$ to the next state q_{i+1}, or $\delta : Q \times I \to Q$. The state space of a modal process is a subset of the powerset of its modes, that is, $Q \subseteq 2^M$. When a handler finishes execution, it may return its transition request (A, D), where $A \subseteq M$ is a set of modes to activate and $D \subseteq M$ is a set to deactivate, and $A \cap D = \emptyset$. The requests from different handlers are merged.

For syntactic convenience, designer do not need to actually specify all of the modes to activate and deactivate on every transition, but they can rely on *mode constraints* to automatically activate or deactivate a set of modes. In fact, hierarchical state machines are a special case where two states can be mutually exclusive or related as parent and child. Mutual exclusion is an automatic deactivation of the current mode on

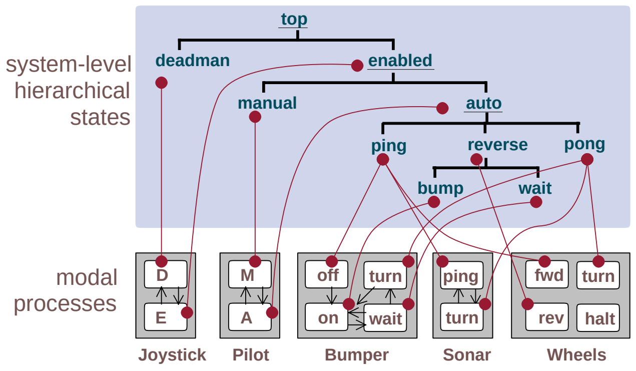


Figure 1: Control composition of the robot specification.

activating the new mode. A parent is automatically activated on activating a child, and a child is automatically deactivated on deactivating its parent. Unlike a state machine, a modal process is *incompletely specified*: it defines the maximum allowed state space and the minimum transition space. In fact, it may define no transitions at all. Control composition will restrict the state space and expand the transition space, as described in the next subsection.

Robot example

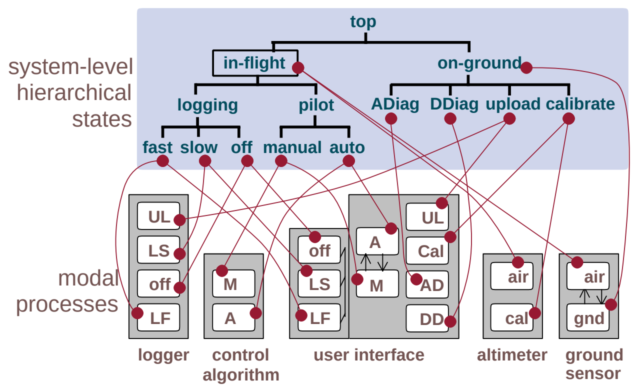
The robot has five processes: Joystick, Pilot, Bumper, Sonar, and Wheels (Fig. 1). Joystick has two modes, D for deadman and E for enabled. The Pilot process has two modes: M for manual and A for auto pilot. The Sonar process has two modes: Ping to pulse the sonar every two seconds and Turn for turning after obstacle detection. The Bumper process has four modes: off, on when the bumper is hit, wait when waiting for two more seconds, and turn like the Sonar. The Wheels process has modes for going forward, reverse, turning, and halt, but it does not define any transitions of its own.

RAVEN example

This system has five processes. The logger is capable of logging fast LF, slow LS, uploading data UL, or being off. The altimeter can measure air altitude or perform calibration. The control algorithm process has two modes for manual (M) or autopilot (A). The user interface process has two sets of mutually exclusive modes. One set is for setting the logging modes, and the other set for plane operations: manual, autopilot, upload logger data, calibrate, analog diagnostics, and digital diagnostics. The ground sensor detects whether the plane is in the air or on the ground.

4.2 system-level states

Control composition is done by *binding* modes to common states that can be organized hierarchically. A *system-level control state*, or simply **state** for short, is a boolean variable. When one process requests to activate a mode, it also causes



†Transition edges not completely shown between modes of the user interface process. Also, processes that perform analog/digital diagnostics are not shown. A framed box around a state, such as in-flight, indicates a fork (parallel children states). Data composition is not shown.

Figure 2: Control composition of the RAVEN human powered airplane controller.

other processes to activate their modes that are bound to the same state. A transition consists of an exit phase followed by an entry phase. A process may choose to be notified of the transition with a special internal mode-exit event to give it a chance to clean up, or a mode-entry event to set up.

Robot example

(Fig. 1) The system's state hierarchy has two states at the top level for deadman and enabled. In enabled, we have two nested states manual and auto. The auto state has three nested states: forward, turning, and reverse. The reverse state has two modes for bumped and wait. Note that Bumper and Wheels processes do not fully specify the transitions between their own modes; instead, they acquire some transitions through binding. By binding Bumper's turn mode to the same state as Sonar's turn mode, Bumper effectively reuses Sonar's behavior, which will take it back to off mode when Sonar returns to ping. The Wheels process has four behaviors but does not decide when to transition between them.

The binding between states and modes is many-to-many and may be asymmetric. A state activates *all* modes that are bound to it. On the other hand, a mode can be bound to several states, and *any* of them can activate the mode.

4.3 synchrony

Synchrony is the characterization of their relative progress or correlation to the events in the system. Different semantics can be obtained by changing the synchronization behavior.

Transition-synchronous semantics means that all parallel state machines make progress in lock-step, such as Esterel. **Event-synchronous** semantics, used by discrete event systems, means that all parallel state machines see the same set of events simultaneously, but it makes no restrictions on how many transitions each state machine can make within a time step. It is possible to combine the two. Synchronous composition can yield deterministic and predictable implementations, though they may be impractical for distributed architectures.

In **asynchronous** composition, each process can make arbitrary amount of progress asynchronously to other processes. Mode transitions are asynchronous to the communication. A special case is **communication synchronous**, where processes make progress asynchronously until they need to communicate. Asynchronous semantics is more realistic for distributed systems, and in fact synchronous models have come to rely on asynchronous compositions at the system level. For example, CRP [3] is essentially a set of locally-synchronous Esterel components that are composed asynchronously as CSP processes at the system level. StateMate [8] offers similar composition: locally-synchronous StateCharts components are connected together asynchronously in the Module-Charts.

Mode synchronous semantics requires different processes to synchronize on a mode change if their modes are affected, so that the composed system behaves in a coherent manner. This is particularly useful for distributed systems because most of the time the systems make independent but coherent progress by running in the same mode context without synchronization. The occasional synchronizations ensure that the entire system (or subsystem) is in a coherent new context before it is allowed to make further progress. In **data-synchronous** semantics, mode changes happen synchronously to dataflow. It allows the system to operate in a logically coherent manner by pipelining mode changes along dataflow, thereby relaxing the instantaneous state-coherence requirement and eliminating the need for additional synchronization.

Examples

Although both the robot and RAVEN are control oriented and have similar auto-pilot and manual modes, they have different synchronization requirements. The robot can be implemented correctly with asynchronous composition, while the RAVEN needs to be at least data-synchronous. On the robot, the sonar and bumper processes are not synchronized with each other or with the joystick, but the only requirement is respose time. The RAVEN, however, operates on streams of sensor/actuator values from the user interface to the control algorithm and the logger, similar to SDF. Theoretically the mode can change every iteration, but an implementation only needs to ensure that each process operates on the data in consistent modes. A rendezvous type of synchronization would be unnecessarily costly and even preclude pipelined implementations.

4.4 timing constraints

Another use of modes is to scope timing constraints. A system may have different sets of timing requirements depending on what mode it is in. Event detection may impose polling constraints in one mode, but in other modes, the event may be ignored and therefore need not be dispatched at all. Event sensitivity information can be extracted from the specification at compile time and is used by the real-time scheduler to determine the work load for each mode configuration. Handlers may be constrained by rate or related by precedence, with minimum or maximum separation requirements between the times of two observable events. Our modal process model is not tied to any particular scheduling model, and most types of timing constraints can be specified, as long as the available schedulers support them. These include both priority-driven schedulers, such as EDF and rate monotonic, and static scheduling techniques that can meet more complex intramodal and intermodal constraints [5].

5 Synthesis

We implement the modal process abstraction with a mix of compile-time transformations and synthesized run-time support. The first step partitions the hierarchical states of the system onto the processors and determines interprocessor communication needed for mode transitions. We also synthesize *mode managers* as run-time support for maintaining state coherence among modal processes that can reside on different processors.

5.1 representation

The system states are represented as a graph $H = (S, E, r)$. S is the set of vertices that represent states. $E \subset C \times S \times S$ is the set of edges that constrain the activation. C is the set of activation relations {AA, AD, DA, DD}: (AA, a, b) requires that when a is activated (denoted $a\uparrow$), b must be activated, too (*i.e.*, $a\uparrow \Rightarrow b\uparrow$), while (AD, a, b) requires b to be deactivated instead, namely $a\uparrow \Rightarrow b\downarrow$. Similarly, (DA, a, b) means $a\downarrow \Rightarrow b\uparrow$ and (DD, a, b) is $a\downarrow \Rightarrow b\downarrow$. Finally, $r \in S$ is the root (initial) state.

A modal process $\pi_i \in \Pi$ is (M_i, T_i), where M_i is the set of modes and $T_i \subseteq M_i \times M_i$ is the set of transitions between modes. We define $M = M_1 \cup M_2 \cup \ldots M_{|\Pi|}$. The binding between the modes and the states is a set of relations $B \subseteq S \times M \cup M \times S$.

A *partitioning* is a function that maps a process to its processor number $[1, n]$. Because a mode is contained by a process, we overload the function P to map a mode to its processor ID, without ambiguities, namely $P : M \to [1, n]$.

5.2 state partitioning

State partitioning involves projecting the system states onto the individual processors. Some replication may be required, although it is not always necessary to duplicate those states that have no binding to any of the modes on the given processor except those required structurally. In addition to projecting the states, this procedure also returns E_C, the interprocessor communication edges for mode transitions. The algorithm is shown in Fig. 3.

We make a copy of a state if one of the modes on the processor has a binding to that state or its descendent. The resulting

```
PartitionState($H = (S, E, r)$, $P : M \to [1, n]$,
                              $B \subseteq S \times M \cup M \times S$) {
    // copy states and edges for processor $i$
    **for** ($i = 1$ **to** $n$) {
        // project states onto processor i
        $S^i := \{s^i | \forall(s, m_i), (m_i, s) \in B\}$
        $E^i := \{(c, s^i, t^i) | (c, s, t) \in E\}$
    }
    // edges for interprocessor mode transitions
    $E_C := \{\}$
    **for** ($i = 1$ **to** $n$, $j = 1$ **to** $n$, $j \neq i$) {
        $E_C := E_C \cup \{(c', s^i, v^j) | (c, s^i, t^i) \in T^i$, where
                        $s \to u$ and $v \to t$ are the shortest paths
                        and $c = c'' \cdot c'$, $(c'', s, u)$
    }
    **return** $H^1 \ldots H^n, E_C$
}
```

Figure 3: State partitioning algorithm.

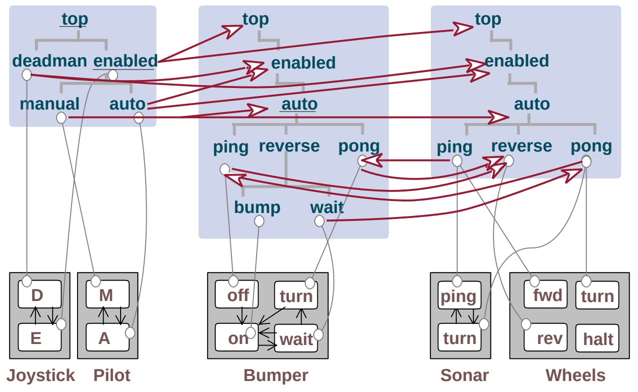


Figure 4: Control communication of the robot as partitioned onto three processors

graph is necessarily connected as a tree. The edges E^i are exactly those needed to connect the replicated states locally. The second part of the procedure builds the set of edges E_C for interprocessor mode transitions. It projects the local *transition path* (c, s^i, t^i) to a remote host j to obtain (c', u^j, v^j). If the projected path is empty then the mode change has no effect on that processor, and therefore no communication is necessary. The projection can be obtained by taking the transitive closures of the paths. The activation operators can compose as follows:

$\cdot$	AA	AD	DA	DD
AA	AA	AD	$\emptyset$	$\emptyset$
AD	$\emptyset$	$\emptyset$	AA	AD
DA	DA	DD	$\emptyset$	$\emptyset$
DD	$\emptyset$	$\emptyset$	DA	DD

Robot example

Consider partitioning the Robot onto three processors. To project the (manual, enabled, auto) path on the bumper's processor, we add the edges (manual^1, auto^2) and (manual^1, auto^3) since they are the terminal vertices. On the other hand, the path (auto, enabled, manual) terminates at enabled on processors 2 and 3, because they do not have the replicated manual state. Paths like (wait, reverse, bump) yield empty projections on processor 1 and therefore no edges are added from processor 2 to 1.

5.3 mode manager

The mode manager is synthesized as part of the run-time support for modal processes. It maintains the state hierarchy on its own processor and services the mode change requests. When an event is dispatched, the mode manager calls those handlers that are mapped by the local processes' modes. The order in which these handlers are called can be defined statically by the user or by the scheduling algorithm. It is possible that one or more handlers can request a mode transition, and they are serviced and resolved after all pending handlers are called. Local transitions can be implemented with either static path enumeration or dynamic path generation. The tradeoffs are determinism vs. code size.

By default, interprocess mode transitions assume mode-synchronous semantics, which is implemented with a three-phase synchronization. After the sender communicates the mode change, all receivers must acknowledge the completion of the exit phase. A receiver may also have a pending transition, and may need to either override the sender's request with a NACK or nullify its own transition and ACK. Transition conflicts can be resolved in a number of ways, and we currently support static priorities. For example, for a control-synchronous version of the robot, suppose the sonar tries to transition from pong to ping at the same time the bumper wants to go to bumped. This can be statically resolved in favor of the bumper.

If a sender receives a NACK or decides to nullify its pending transition, it continues collecting ACKs and sends a consolidated ACK to the receiver. If a sender receives all of the ACKs (statically determined count) then it sends out a "go-ahead" event that tells all receivers to proceed with the entry phase of the transition. This may be very strict but it can be used to implement the semantics of Esterel and StateCharts.

6 Conclusions

We propose the modal process model for capturing control and data behavior with real-time constraints by composing *modal processes* and automatically synthesizing run-time support software. Modes provide not only an interface for control composition between processes, but also a way of systematically scoping timing constraints. By enforcing a separation between system states and process modes, this model combines the data modularity of processes with the control modularity of hierarchical state machines. The decoupling between control and data enhances reuse and allows us to automate the most error prone aspects of distributed systems implementation, namely, synchronization and real-time. We believe that this approach will lead to a higher level of retargetability for distributed embedded systems by enabling the designer to better explore the design space. We are currently developing analysis tools to help the designer determine the best choice of synchronization mechanism and integrate hardware components into the modal process model.

References

[1] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proc. 33rd Design Automation Conference*, pages 568–571, June 1996.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[3] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, Jan. 1993.

[4] R. A. Brooks and J. H. Connell. Asynchronous distributed control system for a mobile robot. In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 727, pages 77–84, 1987.

[5] P. Chou, E. A. Walkup, and G. Borriello. Scheduling issues in reactive real-time systems. *IEEE Micro*, pages 37–47, August 1994.

[6] G.C.Sih and E. A. Lee. A compiled-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel and Distributed Systems*, 4(2), February 1993.

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

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403–414, April 1990.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] R. Ortega and G. Borriello. Communication synthesis for embedded systems with global considerations. In *Proc. Codes/CACHE*, pages 69–73, 1997.

[11] *The RAVEN Project*. http://ihpva.org/Raven/, September 1997.

[12] The SPECS Consortium and J. Bruijning. Evaluation and integration of specification languages. *Computer Networks and ISDN Systems*, 13:75–89, 1987.