Working paper, April 1998

Control Generation for Embedded Systems Based on Composition of Modal Processes [*]

Pai Chou, Ken Hines, Kurt Partridge, and Gaetano Borriello

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

Abstract

In traditional distributed embedded system designs, control information is often replicated across several processes and kept coherent by application-specific mechanisms. Consequently, processes cannot be reused in a new system without tailoring the code to deal with the new system's control information. The *modal process framework* provides a high-level way to specify the coherence of replicated control information independently of the behavior of the processes. Thus multiple processes can be composed without internal tailoring and without suffering from errors common in lower-level specification styles. This paper serves two purposes: to describe the synthesis of the *mode manager*, the runtime code that maintains control information coherence and to describe the semantics of modal process interaction.

1 Introduction

To handle the ever-increasing complexity of distributed embedded systems, modern design methodologies must support system *composability*. For this reason, most distributed embedded systems are modeled as communicating *processes*. Process composition has been particularly successful in data-dominated applications because a set of dataflow processes can be composed as long as they agree on the protocol and data format of their communication.

However, the process model is less suitable for applications that require distributed control information. Under the process model, control information shared among multiple processes must be encoded as data and communicated using messages. Transmissions, receipts, and tests of control information must then be "sprinkled" throughout the data-processing code. This approach is error-prone. For example, an update may be accidentally omitted and deadlock or other synchronization problems may occur. Furthermore, although processes with control information are composable, they are not very modular. Any change involving shared control information requires changing multiple processes [3].

Thus, code is rarely reused as is. Since a process must make fixed assumptions about what control interface it wishes to have, it must anticipate the control interactions of any other process it is composed with. If its interface does not match what is expected by other processes, it cannot be composed with them. Instead, it or some of the other processes must be modified, or an application-specific translation process must be inserted between them. Modification is sometimes impossible for intellectual property reasons. Translation processes are inefficient. Moreover, both techniques require an intimate understanding of what control is shared, when it is shared, and how it is shared, thus potentially introducing new coherency maintenance errors every time supposedly "reusable" processes are composed.

Our *modal process framework* addresses these problems [4]. All control information is represented using boolean *modes* that guard run-to-completion handlers. Rather than keeping modes coherent by communicating their values at the application level, the designer specifies *constraints* between modes of different modal processes. A runtime *mode manager* insures that the constraints are maintained by communicating with other processes as necessary. Because the constraints handle all the synchronization of control information through the mode manager, the modal processes are free to focus on specific modular, reusable behaviors. Modal processes also enhance *retargetability* because the runtime system can be synthesized for a specific distributed target architecture (potentially with different allocation of processes to different processors) without requiring the designer to write low-level synchronization primitives.

This paper describes the semantics of modal processes and the process of synthesizing modal process mode managers. For synthesis, the coherence requirements are expanded into basic constraint primitives and checked for consistency. Depending on the constraint topology, various optimizations are possible for greater run-time efficiency in terms of both space and time.

2 The modal process model

A modal process contains a set of code segments called *handlers*, which can be triggered by *events*. Examples of

[*]This work is supported by PYI

events are notifications of elapsed times and message arrivals. The handlers execute with run-to-completion semantics, such that once a handler begins execution, no other handler in that process may execute until it completes. In addition, a modal process also has a number of *modes* that govern the behavior of the process. Each mode can be in one of two states, *active* or *inactive*. Which state a mode is in is referred to as its *status*. Each mode also defines a binding relation between events and handlers. When an event occurs, all handlers bound to that event by an active mode are invoked. A vector that represents the active/inactive status of all modes is known as a *configuration*. Associated with each configuration is a *scheduling policy* that manages the processing of events.

2.1 overview of a configuration change

At any given time, each modal process has a *current configuration*. The current configuration is changed only after a handler finishes executing. Because the system may be running on a distributed architecture, changes to the configuration on one modal process may affect the configuration of another modal process. Hence configuration changes are negotiated using a mechanism called a *vote*. When a handler finishes execution, it may return a new vote. In a single-processor architecture system, the vote may be processed immediately, but in an distributed architecture, multiple votes may be requested simultaneously, and they must be resolved before being processed.

A vote contains a set of pairs, each of which names a mode in the handler's modal process, and the desired new value for the mode. Formally, a vote V is defined to be a subset of $M \times \{A, D\}$ where A indicates that the mode should be activated, and D indicates that it should be deactivated. M is the set of modes of the modal process in which the handler resides. Any modes in the modal process unmentioned by the vote are treated as "don't cares." However, as we shall see later, these modes as well as modes of other modal processes may still be affected by this vote through *constraints*.

2.2 mode constraints

Mode constraints (or simply *constraints*) completely define how modes are coordinated within and between modal processes. When a modal process is designed, constraints will be established that govern its internal behavior. New constraints can also be imposed within the modal process and between other modal processes according to the requirements of the application.

To simplify both the implementation of the mode manager and the specification of the system, a customizable, higher-level language is transformed into a more primitive set of constraints. Two features distinguish the higher-level language from the primitive constraints: *transitive constraints*, whose transitive closure expands into primitive constraints, and *abstract control types* (ACTs), which allow common patterns of primitive and transitive constraints to be defined and applied.

Primitive Force Constraints	
constraint	meaning (when unguarded)
AA$_p(m_1, m_2)$	if $(m_1, A) \in V$ then $C'(m_2) := A$
AD$_p(m_1, m_2)$	if $(m_1, A) \in V$ then $C'(m_2) := D$
DA$_p(m_1, m_2)$	if $(m_1, D) \in V$ then $C'(m_2) := A$
DD$_p(m_1, m_2)$	if $(m_1, D) \in V$ then $C'(m_2) := D$

Table 1: The primitive force constraints assign status of a mode m_2 the new configuration C' based upon the vote V for mode m_1.

Primitive Guarding Constraints	
constraint	meaning
ABlockC$_p(m, c)$	if $C(m) = A$, block constraint c
DBlockC$_p(m, c)$	if $C(m) = D$, block constraint c
APermitC$_p(m, c)$	if $C(m) = A$, permit constraint c
DPermitC$_p(m, c)$	if $C(m) = D$, permit constraint c
ABlockV$_p(m, v)$	if $C(m) = A$, block vote v
DBlockV$_p(m, v)$	if $C(m) = D$, block vote v
APermitV$_p(m, v)$	if $C(m) = A$, permit vote v
DPermitV$_p(m, v)$	if $C(m) = D$, permit vote v

Table 2: The primitive guarding constraints

Primitive constraints are the only constraints understood by the runtime mode manager. There are two kinds: primitive force constraints and primitive guard constraints.

Primitive force constraints determine what status a mode will have in the new configuration based upon the status of another mode in the current vote. Table 1 lists the different primitive force constraints. Primitive force constraints can be one of four polarities, AA, AD, DA, or DD. The first letter (the *source polarity*) signifies that the constraint applies when a vote contains the pair (m_1, x) where x is the source polarity. The second letter (the *destination polarity* determines the status to be assigned to m_2 in the next configuration.

A primitive force constraint does not indicate a persistent relationship; even if m_1 remains active and the constraint AA$_p(m_1, m_2)$ is imposed, m_2 may be made inactive by some other vote. Nor does a primitive force constraint indicate a transitive relationship: if both AA$_p(m_1, m_2)$ and AA$_p(m_2, m_3)$ are present, then a vote for m_1 will not necessarily turn on m_3. Under some circumstances, transitive behavior might not be desirable, hence it is left to the control of the transitive constraints.

Primitive guarding constraints restrict the application of primitive force constraints or votes by testing the current status of another mode m. A *blocking* guarding constraint inhibits the behavior of another constraint c or vote v. If inhibited, the constraint or vote has no effect on the next configuration. Since a vote is a pair $(m, x), x \in \{A, D\}$, activation and deactivation can be inhibited independently. If n multiple blocking constraints are imposed on c or v, then if any one of them has the appropriate value for their source mode m, then c or v is blocked. A *permit* constraint is the dual: if its mode m has the correct value, then c or v is permitted, otherwise it is blocked. If n permit constraints are imposed, then if any of their source modes m has the appropriate value, c or v is enabled.

2.2.1 *transitive force constraints*

Unlike the primitive constraints, transitive force constraints are not understood by the runtime mode manager. Instead, they are used to add primitive force constraints according to transitive closure rules. Each primitive force constraint has a corresponding transitive force constraint denoted by substituting the "p" subscript with "t" (e.g. AA$_t$). Two transitive force constraints can be *transitively composed* if their "middle" modes and status match. For example, transitive force constraints AA$_t(m_1, m_2)$ and AA$_t(m_2, m_3)$ can be transitively composed.

It is possible for constraints to *conflict*, e.g. if both AA$_p$ and AD$_p$ constraints are present. Because constraints can be built from higher level structures (see next section), such situations can arise inadvertently. Consequently, each constraint has a *constraint priority*. Any conflict is resolved in favor of the constraint with the higher priority. Primitive force constraints are not added when they conflict with higher priority constraints, and existing primitive force constraints are removed if implied higher priority constraints are added. Two conflicting constraints that have the same priority is an illegal situation and is detected statically. This check is conservative because two equal-priority conflicting constraints may never conflict in any execution trace of the system (for example, one may be blocked whenever the other applies).

2.3 abstract control types

Even with the benefit of transitive force constraints, specifying all the constraints in an application would be very tedious. Therefore, we allow patterns of constraints to be named and applied much in the way macros are expanded in traditional programming languages. These patterns are called Abstract Control Types (ACT's). ACTs can also be used to guarantee that transitive force constraints are always paired with primitive force constraints.

Some simple ACTs are defined in Table 3, although in general much more complicated control patterns could be defined.

The **pc**$_{tp}$ constraint establishes a fundamental relationship between m_1 and m_2 called "parent-child." If m_2 is voted active, then m_1 will be activated as well. If m_1 is voted inactive, them m_2 will be deactivated. Since the transitive constraints are also present, any other constraint that activates m_2 will also activate m_1, and similarly for deactivation of m_1. Note that if m_1 is voted inactive or m_2 is voted active, no change is made to the other mode. This fundamental relationship can be used as a building block for many other ACTs.

pc$_{tp}$**(**m_1, m_2**)**	/* parent/child */
AA$_p(m_2, m_1)$, DD$_p(m_1, m_2)$	
AA$_t(m_2, m_1)$, DD$_t(m_1, m_2)$	
mutex$_{tp}$**(**m_1, m_2**)**	/* mutally exclusive */
AD$_p(m_1, m_2)$, AD$_p(m_2, m_1)$	
AD$_t(m_1, m_2)$, AD$_t(m_2, m_1)$	
unify$_{tp}$**(**m_1, m_2**)**	/* merge states */
pc$_{tp}(m_1, m_2)$, pc$_{tp}(m_2, m_1)$	
dseq($m_1, m_2, ..., m_k$**)**	/* deactivation sequence */
$(\forall i, 1 < i < k)$: DA$_p(m_i, m_{(i+1) \bmod k})$	
kill($m*, m_1, m_2, ..., m_k$**)**	/* $m*$ preempts each m_i */
$(\forall i)$: AD$_p(m*, m_i)$	

Table 3: Examples of abstract control types (ACTs).

mutex$_{tp}$ and **unify**$_{tp}$ are two other fundamental ACT's. They express the relationships (respectively) that two modes may not be simultaneously active, and that two modes must always have the same value. **unify**$_{tp}$ is constructed from the ACT **pc**$_{tp}$.

For more details on ACTs, please see the expanded version of this paper [5].

3 Constraint transformations

The constraints on modes do not have execution semantics *per se*, but the handlers supply the actual execution and the constraints effectively "steer" the requested votes by restricting the allowed configurations of the system. It is possible to translate the combined handlers, modes, and their constraints into an equivalent (flat, parallel, or hierarchical) FSM. However, it is much more compact to interpret the constraints at run time with a *mode manager*. The interpretation algorithm, which is described in the next section, can be implemented very efficiently and offers the flexibility for extensions to distributed architectures.

This section presents the steps in transforming a specification-level system of constraints into one suitable for the mode manager. We formulate constraint transformation as a graph problem. The two goals of the transformation steps are to express all constraints in terms of the basic primitives and to statically detect and resolve conflicts.

3.1 graph formulation

The system of constraints can be represented using a graph-like data structure. $G(M, E)$ consists of a set of vertices M, which corresponds to the set of modes being constrained, and the edges E represent constraints. G can have more than one directed edge between a given pair of vertices, although they must have distinct and compatible labels.

Each edge is labeled with a constraint of the form (*op*, i, j, p). The field *op* is one of the force or guard constraints. After transformations, all *op*'s must be primitive constraints. The vertex $i \in M$ represents the mode that exerts the constraint; the vertex j, on the other hand, can actually be either a vertex or another edge. If j is a vertex then *op* must be either a force or a vote guard. If j is an edge then *op* must be an edge guard, and we currently require j to represent a force constraint. Finally, p is an integer priority of the constraint.

Additionally, the system assumes that the graphs are reduced such that all modes connected by explicit unify constraints or equivalent combinations of primitive constraints into single *supermodes*.

3.2 ACT expansion

An ACT acts as a constraint macro that adds a new set of edges to the constraint graph. However, unlike a macro, constraints are not added all at once. Instead, the expansion is divided in two phases. The first phase expands constraints by applying the ACTs independently. The second phase is "post-processing" to take into account the interactions between the constraints.

In the first phase, each ACT is expanded by calling its definition routine to add constraints either explicitly enumerated or algorithmically. Each ACT may actually be defined in terms of other ACTs, and they are expanded recursively into force and guard constraints. In addition, an ACT may perform structural modifications to the constraint graph. For example, even though a unify ACT can be expressed in terms of symmetric AA and DD constraints, an efficient alternative is to merge the modes into a supermode. An ACT may also introduce additional modes and handlers that are associated with either new modes or existing modes. For example, a parent-child ACT may either expand constraints transitively such that hierarchical transitions can be performed in a single step, or implement the transitivity with a synthesized helper handler that invokes a mode's parent or child. All such transformations are abstracted from the end user, although advanced designers may wish to be exposed to such knowledge in order to better evaluate implementation tradeoffs.

As an example, consider the composition of the bumper and wheels processes in Fig. 1. The bumper process is internally constrained as a composition of a dseq(F, R, W, T) at priority 1 and kill(R, F, W, T) at priority 2. It is possible to apply ACTs across the processes, such as the pc constraint that designates R and W of the bumper process as the children of mode B of the wheels process. These constraints are merged in as shown at the top.

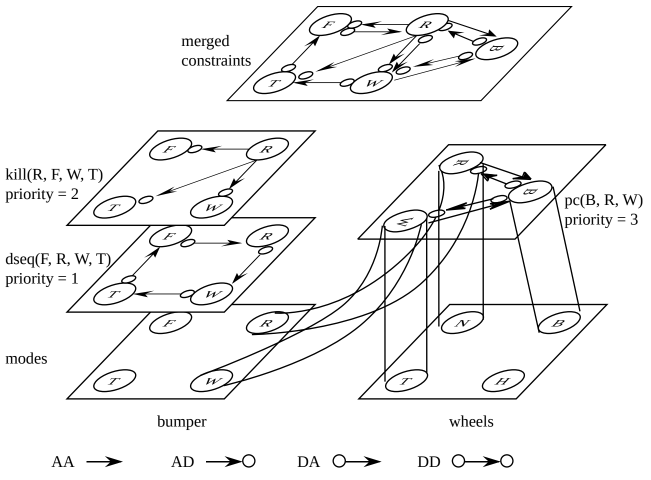

Figure 1: example of ACT expansion on the bumper and wheels processes

The second phase of ACT expansion computes the *transitive closure* on the graph output by the first phase. The edges can be divided into two sets, force and transitive edges. Various ACTs mark the edges to be either transitive or intransitive. All transitive edges are transitive, but force edges can be either. Transitive closure is performed on the transitive subgraph so that constraints instantiated by different ACTs can relate to each other. After the transitive closure is evaluated, the transitive edges are removed.

3.3 post-processing

This "post-processing" algorithm is invoked after all ACTs have been applied, so that interactions between ACTs can be processed. The algorithm (3.1) is divided into two parts. The first part computes the transitive closure on the force constraints, and the second part adds the induced guard constraints on the newly transitive edges added by the first part.

The transitive closure of a graph can be computed using a modified version of the Floyd-Warshall algorithm [6]. Extensions are necessary for several reasons: to handle multiple edges between a pair of vertices, to handle vertex-to-edge edges, to adapt the special transitivity rules for this problem, and to statically resolve constraints that conflict. An edge e has been slightly augmented with several attributes. First, an *iteration-number* attribute, $I[e]$, to record in which iteration it was added. An edge with an iteration number of 0 means it is one of the original constraint edges before transitive closure is applied. The algorithm should either produce a graph that is free of conflicts, or report to the designer that the constraints are ill-posed.

The guard induction is applied to all those constraints whose causality is dominated by a guarded vote or edge. One

Algorithm 3.1 The post-processing algorithm for transitive closure and guard induction

```
// apply transitive closure to force constraints
$n := 0$
foreach ($k \in M$)
  $n := n + 1$
  foreach (force $e_{ik} \in$ transitive IN[$k$])
    $e_{ik} = (< w, x >, i, k, p_{ik})$, $I[e_{ik}] < n$
    foreach (force $e_{kj} \in$ transitive OUT[$k$]$||_{<x,\_>}$)
      $e_{kj} = (< x, y >, j, k, p_{kj})$, $I[e_{kj}] < n$
      $e_{ij} = (< w, s >, i, j, p_{ij})$
      if ($e_{ij} \notin E$) then
        add $e_{ij} = (< w, s >, i, j, p_{kj})$;
        set $I[e_{ij}] := n$;
      else if ($x \neq z$) {
        // contradiction: need to resolve thru priority
        if ($p_{ij} \leq p_{kj}$) { // ill-posed constraints
          throw malformed exception($e_{ij}, e_{kj}$);
        } // else skip closure
      } // expand guard constraints on transitive force edges
foreach (guard $e_{ij} = (< x, g >, i, j)$)
  if ($j \in forceE$)
    foreach (force $e_{kl} = (< y, z >, k, l), I[e_{kl}] > 0$)
      if ($j$ dominates $l$ from $k$)
        add guard $e = (< x, g >, i, e_{kl})$
  else if ($j$ represents a vote)
    foreach (force $e_{kl} = (< y, z >, k, l), I[e_{kl}] = 0$)
      add guard $e = (< x, g >, i, e_{kl})$
  else throw exception "$j$ is a guard edge"
```

vertex or edge b dominates a vertex c from a source a if all paths from a to c require voting or going through b. We therefore extend guards to all transitive constraints that are dominated by the guard.

Fig. 2 shows a subset of the robot example in the form of the constraint graph through the transitive closure process. The left (a) shows the graph assuming all ACTs have been expanded. For simplicity, we assume all edges are marked transitive. When a transitive edge is added, it is assigned the same priority as that of its incoming edge. For example, in the middle (b) figure, DD(F, T) has priority 2 because it acquires it from AD(R, T), while DA(F, T) has priority 1 because it is connected through DA(W, T). DD(F, T) and DA(F, T) are incompatible and is resolved in favor of DD(F, T). To induce the guard constraint by G, we enumerate all transitive edges dominated by the edge(s) it guards. In this case, DA(R, W) dominates the transitive constraint DD(F, W) and therefore we induce a new guard from G to this edge.

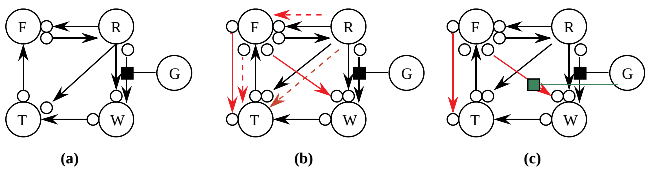

Figure 2: (a) shows the constraint graph of the bumper process, (b) after transitive closure, with conflicting edges shown in dashed arrows and removed, and (3) inducing the guard constraint on the transitive edge.

4 Centralized mode manager

Following the transformation of ACTs and constraints into their runtime form, the mode manager code that implements the constraints must be produced. The implementation depends significantly on whether the target architecture is a uniprocessor or a distributed architecture. The discussion of distributed architecture implementations is postponed to the next section.

The centralized mode manager has a notion of a discrete *step*, which defines a sequential boundary for a set of votes to be accumulated and resolved as a single externally visible change of configuration. We support two possible step semantics: event-triggered and time-triggered. Both share the same engine that computes the next configuration.

4.1 computing a new configuration

Algorithm 4.1 Centralized mode manager configuration selection.

```
foreach vote $V = \{(m_i, s_i) \in M \times \{A, D\}\}$
  foreach ($(m_i, s_i) \in V$)
    if ($m_i^{t+1}$ = undefined and voteGuard($m_i$) = true) then
      foreach ($e_{ij} \in$ OUT[$m_i$] $||s_i$)
        if (edgeGuard($e_{ij}$ = true))
          let $\langle s_i, u_j \rangle$ = constraint label of $e_{ij}$
          $m_i^{t+1}$.set($u_j$)
foreach ($m_i^t$ that is undefined)
  $m_i^{t+1}$.set($m_i^t$)
```

Algorithm 4.1 shows how the next configuration is computed. At the end of a step, the mode manager is given an ordered set of requests, or votes, to change either part or all of the configuration. Each vote is a set of tuples $V = \{(m_i, s_i) \in M \times \{A, D\}\}$, where m_i is the specific mode to change, and s_i indicates whether it should be activated or deactivated. These are ordered such that a higher priority vote is ordered earlier and effectively locks in its value, so that a lower priority vote cannot override it if the same mode it wishes to change has been set. Otherwise, a

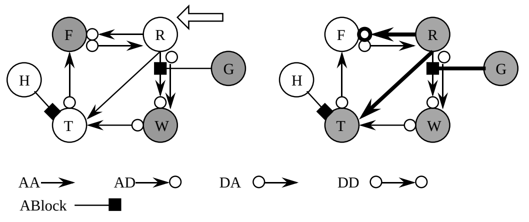

Figure 3: Example for computing the next configuration. All edges are primitive.

mode m_i can be transitioned to $s_i \in \{$A, D$\}$, and its constraints must be fulfilled.

To fulfill the constraints exercised by mode m_i on changing to s_i, the algorithm iterates over all outgoing force constraints from m_i that match the source polarity. In other words, if s_i = A (namely to activate m_i), then constraint edges of types $\{$AA, AD$\}$ are used; or if s_i = D then $\{$DA, DD$\}$ edges are used. Each constraint indicates whether m_j should be activated or deactivated, although the force constraint may be guarded by another mode. The change is made only if the guarding condition is satisfied. Guard evaluation can be as efficient as a single conditional test, because as individual modes change configurations, they can update the guarding condition by incrementing or decrementing the number of permitting or blocking modes.

Example

To illustrate the operation of the mode manager algorithm, we consider an example based on the bumper process of the robot (Fig. 3). Note that the mode manager maintains the configurations without using the knowledge about what processes the modes belong to. Therefore, the mechanism for managing modes within a process is exactly the same as that for a set of processes. For illustration purposes, we added a vote-blocking constraint from H to T and a edge-blocking constraint from G to W, and reversed a few constraint polarities.

Assume the current configuration is { F, W, G }, and a handlers requests for activation of R. The algorithm iterates over all outgoing edges of R labeled AA$_p$ or AD$_p$ in an attempt to apply force constraints. These edges are AD$_p$(R, F), AA$_p$(R, T), and AD$_p$(R, W) – but DA$_p$(R, W) is not applicable because the vote is for activation, not deactivation. F is deactivated, since its guard is trivially true. T is also activated, even though H appears to be blocking T because the block applies to votes for T, not on force constraints entering T. On the other hand, since G is true, G blocks AD$_p$(R, W) and therefore W does not change. The resulting configuration is therefore { R, T, W, G }.

4.2 voting steps

The execution of a modal process system with centralized control can be viewed as a sequence of discrete steps. All events generated during a step are consumed during a later, though not necessarily the next, step. Furthermore, no handler execution crosses a step boundary. Several handlers may be invoked in a given step. If they requests mode changes, the requests are queued until the end of the step when they are processed collectively for the next step. We provide the mechanism for defining a variety of steps, ranging from event-driven steps to dataflow and time-triggered steps.

The simplest step is defined by an event occurrence. That is, the designer may assume no simultaneous events and that a mode change request is serviced right after dispatching an event to a set of handlers. Discrete event models are more general in that events are not only completely ordered but can also be simultaneous, such that vote processing is performed after all (logically) simultaneous events have triggered their handlers.

Another way of defining steps is to mark certain event types as step-delimiting events. For synchronous dataflow (SDF) models, a reasonable step would be to process votes after an entire iteration of the dataflow graph has been invoked. This allows the dataflow graph to be invoked according to a static schedule without using the more expensive event dispatch mechanism. Although dataflow models are untimed, dispatching according to a static schedule can be extended for real-time systems by replacing dataflow events with timer events. In general, statically scheduled, time-triggered systems offer the best determinism and can make the strongest guarantee in meeting hard real-time constraints.

5 Distributed mode manager

When mapping a design to a distributed architecture, control may be implemented in a centralized or a distributed style. If the designer desires a centralized control process, the centralized mode manager described in the previous section can be used, with slight extension to communicate votes explicitly in a message. However, such an organization is not very efficient and defeats the very advantages offered by distributed architectures, because the centralized mode manager must handle and generate communication to all processes even if most are not affected by a localized mode change.

To exploit the architectural distribution, we support *distributed mode managers*, which maintain consistent mode configurations between processes residing on different processors—without centralized control. With distributed mode managers, each processor in the system is given its own mode manager and each of these coordinate activation and deactivations between themselves. In this case, however, there is no single notion of step. In fact, the rate of step progression may be different for each specific mode manager. To avoid overspecification, the modal process model does not impose specific *synchrony* semantics on the interactions between mode managers; instead, several synchrony options can be supported, as described in [2]. This section focuses on one on synchrony option called *mode synchronous* semantics, where a mode change blocks progress of only those processes whose modes are affected until their mode managers agree to it.

The synthesis steps for a distributed mode manager can be divided into graph partitioning, control communication synthesis, and local mode-manager synthesis. Local mode-managers are centralized mode managers whose inputs are their respective partitioned graphs. This section reviews the graph partitioning algorithm that has been described previously and addresses the extensions to the mode managers needed for distributed control coordination.

5.1 mode manager partitioning

In distributed implementations of a modal process system, it isn't necessary for all parts of the system to maintain the complete constraint graph. In fact, each subsystem needs only a projection of the constraint graph containing the portions relevant to the processes in that subsystem. Specifically, these are the the modes that occur within these processes (the local modes), and the modes that appear as the source end of a primitive constraint that terminates a local mode (see Figure 4 *b)* and *c)*). For more information, see [4].

5.2 control communication

Upon completion of the partitioning step, the mode manager residing on each subsystem is aware of which *voted* activations and deactivations of modes need to be transmitted to the rest of the system. Using mode synchronous semantics, the requesting subsystem is not able to perform the changes until each of the relevant subsystems acknowledges this request. Assuming reliable communication between all mode managers, there is a three phase handshake (request, acknowledge, commit) such that the requester transmits the desired activations and inactivations as a special vote to all relevant mode managers and it waits for the remote subsystems to acknowledge the requests.

The receiving subsystem mode managers include this vote in calculating the next configuration (based on this subsystems own version of steps). In considering this sort of vote local mode manager must determine whether there were any internally generated conflicting votes—and if there were and they had higher priority than the special vote, it must send a request of its own to the original requester before acknowledging the original request.

If it finds no such conflict, it simply acknowledges the request and places itself into a provisional state until it receives the corresponding *commit* message. From the perspective of the requester, the actual transition to a new configuration is blocked until all requests have been acknowledged. When the requester receives acknowledgments from all subsystems over all parts of the vote, it performs the action locally (provided there were no conflicts received before getting all acknowledgments) and sends *commit* messages to all relevant subsystems. If it received a conflict in the mean time, and it decided that the conflict had higher priority, then it sends *abort* messages to each of the participants.

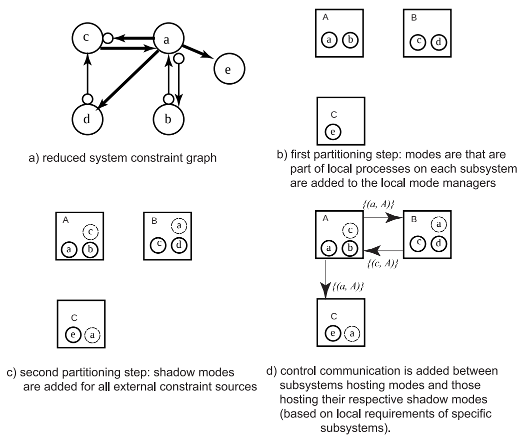
a) reduced system constraint graph
b) first partitioning step: modes are that are part of local processes on each subsystem are added to the local mode managers
c) second partitioning step: shadow modes are added for all external constraint sources
d) control communication is added between subsystems hosting modes and those hosting their respective shadow modes (based on local requirements of specific subsystems).
{(a, A)}
{(c, A)}

Figure 4: Shown are the steps involved in partitioning the constraint graph for individual mode managers (based on a preexisting process partition), and in synthesizing control communication.

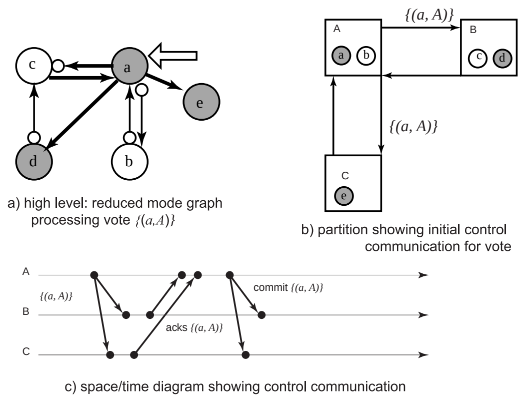
a) high level: reduced mode graph processing vote {(a,A)}
b) partition showing initial control communication for vote
{(a, A)}
commit {(a, A)}
acks {(a, A)}
c) space/time diagram showing control communication

Figure 5: a) shows a system constraint graph, b) shows the mapping of nodes from this to processes in a distributed implementation and c) shows the control communication required to insure mode synchrony.

To insure consistent choices in the event of several concurrent requests, there should be system wide total ordering of priorities that allow all subsystems to independently but consistently choose from among conflicting requests.

Identification of the required control communication is straightforward given the partitioning step. Any constraint such that the source mode is on one subsystem, and the terminal mode is on another implies a communication.

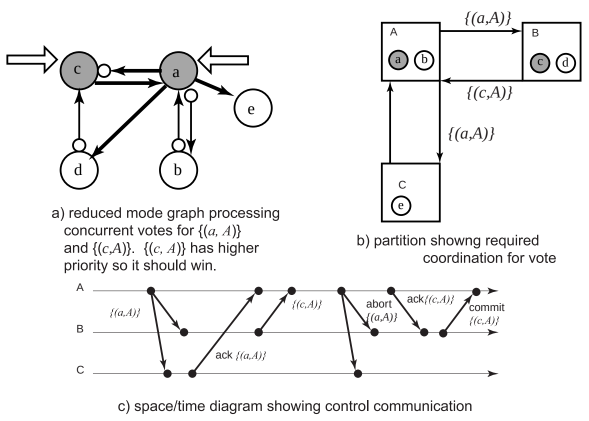
c
a
e
d
b
a) reduced mode graph processing concurrent votes for {(a, A)} and {(c,A)}. {(c, A)} has higher priority so it should win.
{(a,A)}
A
a
b
B
c
d
{(c,A)}
{(a,A)}
C
e
b) partition showng required coordination for vote
A
B
C
{(a,A)}
ack {(a,A)}
{(c,A)}
abort {(a,A)}
ack{(c,A)}
commit {(c,A)}
c) space/time diagram showing control communication

Figure 6: This shows the control communication between the processes in Figure 5 in the presence of an inter-process vote conflict.

All subsystem mode managers are essentially centralized mode managers, and can be synthesized as demonstrated in the previous section, with some minor modifications.

5.3 Examples

In Figure 4 we show a system constraint graph, and the steps required to build consistent distributed mode managers for this. First the mode graph is partitioned across the subsystems and then the control communication is synthesized.

Next we subject this system to votes various conditions that might occur in choosing a new consistent configuration (shown in Figures 5 and 6). In Figure 5 the system hosts a single request for mode activation, and this is easily resolved. In Figure 6 case, there are two concurrent requests for activation, where one request has a higher priority than the the other. In this case, each of the requesting managers must evaluate the relative priority of the requests, and independently (but consistently) choose the winner. The subsystem that requested the losing activation (subsystem A in this case) is then responsible for sending abort messages to all subsystems that received the original request.

6 Related Work

Esterel [1] is an imperative langauge for composing control hierarchically. However, Esterel's control state is encoded by the program counter and cannot easily be used to schedule and selectively activate handlers the way modal process modes can. Coordination in Esterel is handled by emitting events, which is similar in style to message passing, and also suffers from code sprinkling, fixed assumptions about the interface client, and error-prone maintenance problems.

Statemate, based on Statecharts [8], uses a process model on the activity chart level, and supports composition with a process using statecharts. Like Esterel, however, statecharts use events to define the interface between two composed state machines. ActivityCharts are forced to encode control into messages the same way the process model does.

Frølund's *synchronizers* [7] are more similar to our modal processes because they customize the behavior of the reusable code without requiring the designer to alter it. However, synchronizers are designed, not surprisingly, to assist synchronization. Consequently, they only support blocking of method invocation, and treat dispatching, scheduling, and real-time issues as orthogonal problems. Synchronizers also assume a shared-memory computation model, while our model is focused on a specific appliation domain, namely that of distributed real-time embedded systems.

D [9] is a distributed programming langauge framework build on aspect-oriented programming, a general mechanism that includes support for collecting code that would otherwise be scattered about a program. D also supports reusable synchronization, but like synchronizers, D also assumes a shared-memory computation model.

7 Conclusions

Control composition is an important problem in modeling and synthesis of complex embedded systems. Existing approaches hinder reuse of control behavior. If the behavior is structured as communicating processes, control handling is sprinkled everywhere and must be modified in an *ad hoc* manner when used in different applications. If the behavior is captured as hierarchical state machines, control may compose better but there is no modularity and they are difficult to retarget over different architectures. The modal process model shows promise in addressing these problems. It allows designers to effectively constrain the state space within and between processes using high-level primitives, called *abstract control types* that subsume hierarchical state machines and can better match designer's intuition similar to the *subsumption architecture*. By separating out composition-specific behavior and synthesizing the run-time system, we can make the modules more reusable over different applications.

The usefulness of a specification model can be diminished if it does not lend itself to efficient implementation on a variety of target architectures. We have shown that the basic primitives, namely *force* and *guard* constraints, do lend themselves to very efficient implementations and requires only as much time as the actual number of modes that change in a given configuration. Although we provided algorithms for a software implementation of the synthesized control, called the *mode manager*, the simplicity and the parallel nature will lend them to efficient hardware implementations as well. More importantly, modal processes can be mapped onto arbitrary distributed architectures through synthesis of distributed mode managers. By automating the highly error-prone low-level tasks of managing synchronization and control communication, designers can better spend their time with global design issues.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

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

[3] P. Chou and G. Borriello. Functional encapsulation vs. state encapsulation in the specification of reactive systems. In *Submitted to the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1998.

[4] P. Chou and G. Borriello. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Proc. Design Automation Conference*, June 1998.

[5] P. Chou, K. Hines, K. Partridge, and G. Borriello. Control generation for embedded systems based on composition of modal processes (extended version). Technical Report UW-CSE-98-04-01, University of Washington, 1998.

[6] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[7] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Seventh European Conference on Object-Oriented Programming (ECOOP)*, number LNCS 707. Springer-Verlag, July 1993.

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

[9] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-008 P9710042, Xerox Corporation, February 1997.