ABSTRACT
The flow of events and data on the connections between components in an application may have to be controlled at run time based on the application state to optimize performance. For example, a set of components (or a subsystem) may be inactive in a given state and events flowing towards such inactive components can be eliminated. This paper presents a model-driven approach to dynamically control the flow of events on event connections. The proposed model-driven tool chain consists of analysis algorithms which analyze the application to derive metadata for event flow control, and an adaptive event communication framework which provides configurable event communication middleware. The metadata derived by the analysis algorithms is used to automatically configure the middleware for application state-aware propagation of events. The analysis algorithms are especially useful in large scale distributed real-time embedded systems where deriving such metadata manually can be tedious and error-prone. We have applied the proposed mechanisms to application scenarios from the Boeing BoldStroke system.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.10 [Software Engineering]: Design—methodologies

1. INTRODUCTION
Distributed real-time embedded (DRE) systems often involve a large number of components interacting with one another in complex ways via synchronous as well as asynchronous communication. A number of tools based on component-oriented frameworks such as the CORBA Component model (CCM), EJB and DCOM have been developed to design large-scale DRE systems [9, 5, 19]. Cadena is one such tool for modeling, development and deployment of CCM based systems [9, 18]. Figure 1 shows the different stages of the application design process in Cadena. An application is assembled by specifying the instances of the components to be created and the connections between the ports of the components. During the deployment of the application, the component instances are created and the connections between the ports are set up using middleware services.

The focus of this paper is on adaptive event flow on event connections. The specification in Figure 1 shows the edges of the event communication topology for an example application. The flow of events on these edges, however, may depend on the system states. For example, in the Boeing BoldStroke system, a platform to develop avionics applications [17], the application scenarios are often described in terms of system modes, where only a subsystem may be active in a specific mode. In Figure 1, for instance, the subsystem consisting of components C and D may be active only in a specific mode. In this case, C may want to receive events from B only when it is active. In the traditional deployment schemes (such as the current deployment scheme in Cadena and the BoldStroke system), all connections are established statically at initialization time and the mode-based behavior is incorporated in the business logic of the components. Thus, C would remain connected as a consumer of B’s events and will receive notifications from B irrespective of its state. This results in unnecessary overhead as the CPU and bandwidth resources consumed by an event notification that is eventually dropped by C are wasted. To optimize performance, such events can be eliminated before they consume resources.

We are developing FraMES, a Framework for Model-Driven Event DiStribution, with the goal of model-driven middleware synthesis and optimization. In this paper, we propose a model-driven approach to dynamically control event communication in FraMES. As shown in Figure 2(b), our approach starts with the analysis of the application to generate a Mode-Spec file containing XML metadata identifying how events should be controlled, and then uses the metadata to configure the middleware. As opposed to the traditional approach (Figure 2(a)), our approach takes the control logic of the components and pushes it into the event connection implementations so that event notifications can be discarded earlier in the notification paths. The main elements of the approach are the following:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*This work was supported by DARPA PCES contract F33615-00-C3044 and NSF award CNS0098179.

3rd Workshop on Adaptive and Reflective Middleware Toronto, Canada
Copyright 2004 ACM 1-58113-949-7 ...$5.00.
- FraMES provides two schemes to control the event flow. The first scheme (shown by dotted lines in Figure 2) employs a set of mode-managers, synthesized from the ModeSpec file, which monitor the current state of the system and instruct a set of configurator modules to suspend or resume events flows on the various connections using the event service API. The second scheme uses a mode-aware event service that is configured using the ModeSpec file. The mode-aware event service monitors the flow of events and controls it as per the metadata specification. Thus, the first scheme uses the middleware API to control the event flow whereas the second scheme configures the middleware itself.

- An important part of tool chain is the generation of the ModeSpec file needed to configure FraMES. This information has to be derived from the mode-based behavior of the components. For scenarios with small number of components, it might be possible to derive this information manually. However, for large scale systems, automated techniques are needed. We have developed tools to analyze the application behavior to generate the ModeSpec file.

To summarize, the contributions of the paper are:
- An adaptive mode-aware event service to control the flow of events based on application semantics.
- A connection management mechanism to suspend/resume event connections based on the application state. Change management frameworks [12, 11] have been proposed to dynamically add and delete components. However, the mechanisms proposed in this paper deal with CCM based systems and focus on temporary suspension of event flows on connections.
- Analysis algorithms to derive control metadata from the application specification.
- Experiments to demonstrate the usefulness of our model-driven approach. Our experiments show that:
  (a) Controlling event flow in the middleware leads to better performance as compared to embedding control logic in the components alone.
  (b) Fewer components need to be made mode-aware, thereby reducing the overhead of mode propagation.

2. EVENT COMMUNICATION OVERVIEW

In this section, we give a brief overview of Cadena emphasizing aspects relevant to event communication. Cadena is an integrated modeling environment for modeling and building CCM systems, and uses the OpenCCM CCM implementation [4]. A system in Cadena is realized as a collection of components coupled together via data and event connections. CCM components publish events on ports referred to as event sources, and consume events on ports referred to as event sinks, and event connections are specified by coupling event source and sink ports. The containers in the CCM code architecture use the underlying middleware services to provide communication support for event connections (see [6, 20] for a detailed code architecture description). For implementing an event connection, we have configured the OpenCCM containers to use two schemes: event service dispatching and direct dispatching.

- **Event Service Dispatching:** An event service is a middleware that brokers communication between producers and consumers [7]. A component can register with the event service as a producer of an event or as a subscriber of an event. Whenever a producer produces an event, all current subscribers for that event are notified of the event occurrence. Event service has been used extensively in information dissemination services [1], distributed systems [8, 10, 15, 3, 13], real-time systems [14] and parallel computation [16]. We have developed the Adaptive Event Service (AES), a Corba-based Java event service, and have modified the OpenCCM containers to use AES. The containers use the connect and disconnect interface methods of the event service to establish event connections.

- **Direct Dispatching:** The direct-dispatching (DD) mechanism bypasses the event service by direct communication between the producers and the consumers. In this case, the subscriber lists are maintained in the producer containers which are used to make direct notifications. The DD mechanism is a light-weight alternative to event service dispatching when the producer and consumer are on the same node (the notification can then be optimized as a non ORB-call). This optimization has been used in the BoldStroke platform as well [17].

3. CONTROLLING EVENT FLOW

The event communication topology (ECT) of an application is defined by the connections between the event source and sink ports of the various components. ECT contains edges for all possible event communication that can take place in the application. In many frameworks such as Cadena and the Boeing BoldStroke system, connections for all edges in the ECT are set up during initialization. That is, a component subscribes at initialization time for all possible events it may need during its lifetime. However, the communication over the ECT edges may have to controlled at run time. For example, consider the ModalSP scenario shown in Figure 3. In this scenario, the system operates in two modes: one in which the TacticalSteering component is enabled and
the other in which the NavSteering component is enabled. If the connections are set up statically, then NavSteering and TacticalSteering will subscribe to events irrespective of the current system state. Thus, NavSteering for instance will receive events from AirFrame and NavSteeringPoints even if it is disabled (such events are simply dropped by the component on their reception). To optimize performance, events flowing towards such inactive components can be eliminated. For large scenarios, automated analysis may be needed to identify events to be dropped. For example, analysis can identify components that only publish events consumed by inactive components in certain modes; such components can be made inactive as well in that mode.

In this paper, we propose a model-driven approach to dynamically adapt the flow of events on event connections. As opposed to the traditional approach where the control logic is embedded inside the business logic of the components, our model driven approach embeds the control logic inside the event communication middleware. The main elements of our approach are the following:

- The metadata specifying the event flow control.
- Two mechanisms to control the event flow in FraMES.

The first mechanism, termed flexible connection management (FCM) approach, manages the flow by suspending and resuming the event flow on connections using the middleware API. The second mechanism, termed mode-aware event service (MES) approach, directly configures the middleware to control the event flow.

3.1 Metadata for event control

To control the flow of events on the ECT edges, we need metadata specifying which connections to control and under what conditions. This metadata is specified in the Mode_Spec file in an XML format. In this file, the “drop statement” is used to specify when to drop an event or an event notification. These statements include an application condition based on the mode variables of the components. For example, for the ModalSP scenario of Figure 3, the notification of AirFrame’s event to TacticalSteering must be dropped if the condition “TacticalSteering.modeVar = disabled” is true.

3.2 Flexible Connection Management

The modules used in the FCM approach are the global and local configurators, mode managers and the containers. As shown in Figure 4, these modules are configured using the information in the Mode_Spec and component assembly description (CAD) files. In the following, we explain each of these modules.

- **Mode Managers**: The mode managers keeps track of the current modes of the components. From the Mode_Spec file, each mode-manager determines the set of mode variables that it has to keep track of and the conditions associated with events to drop them. Each container is initialized with the reference of the local mode-manager, and the list of mode
announcements to be made. Based on these announcements, the mode-managers evaluate the conditions and instruct the configurators to enable/disable the connections.

- **Global and Local Configurators**: We employ a set of configurators arranged in a hierarchical manner. The task of a local configurator is to perform all reconfigurations local to the server. Reconfiguration of connections between components located on different servers are performed by the global configurator. Each configurator provides an interface method to receive requests for connection changes (calls labeled 2 and 4 in Figure 4). The configurators, in turn, forward the requests to the containers.

- **Container Configuration**: On receiving a request from a configurator, a container invokes the appropriate methods of the event middleware depending on the type of connection. The Corba Event service specification defines the `suspend` and `resume` operations where the suspend operation allows a consumer to temporarily suspend its subscription without removing any of the proxies, and the resume operation simply enables the flow of notifications on a suspended connection. The suspend/resume operations incur a much smaller overhead as compared to connect/disconnect operations. Therefore, we have extended the OpenCCM containers to incorporate the suspend/resume interface for DD notification. The AES also provides a similar interface. Given this interface, the following sequence of actions takes place to reconfigure a connection:

  1. A mode change is announced by a container.
  2. On receiving a mode announcement, the mode manager evaluates the conditions and makes connection change requests to the local configurator.
  3. If the connection is local, then the local configurator interacts with the local containers. If the local connection is a DD connection, then the `suspend` (or `resume`) method is invoked on the producer’s container. Otherwise, the `suspend` method is invoked on the consumer’s container, which in turn suspends its subscription with the event channel. If the connection is not local, then the request is forwarded to the global configurator, which acts on the request in the same manner as the local configurator.

### 3.3 Mode-aware event service

We have developed Adaptive Event Service (AES), a Corba-based Java event service [18]. To control event flow, we have added the mode awareness aspect to AES by parameterizing it with a module, `ModeModule`. An instance of this module is created in each event channel instance. The task of the `ModeModule` is to observe the flow of events and to constrain this flow. The code for the `ModeModule` is generated from the `ModeSpec` file and its structure is similar to the code for the Mode Manager discussed in the previous section. Essentially, `ModeModule` remembers the values of the mode variables to be tracked and it evaluates the conditions associated with event control whenever a mode change event arrives (the containers are configured to publish mode change events for this approach). As a result of this evaluation, if a connection is to be suspended or resumed, the `ModeModule` interacts with the appropriate consumer proxies to suspend or resume the flow of specific event notifications. In Figure 1, for example, even though C remains connected to the event service as a consumer, AES would monitor the current mode of the system and propagate events from B to C based on this mode information.

### 3.4 Infrastructure for generating metadata

The control logic for handling the events is embedded inside the business logic of the components. For small scenarios (such as the ModalSP of Figure 3), it may be possible to derive this information manually. For large systems, however, this can be a tedious and an error-prone task. Hence, automated tools are needed to analyze the application to derive the `ModeSpec` file. In Cadena, we allow the designer to associate a Component Property Specification (CPS) file with a component to specify dependencies between ports and behavioral specifications for the event handlers and other methods (the formal grammar is given in [2]). In particular, for each event, a case statement is used to specify how an incoming event is processed in different modes or states. We have come up with a set of methods to derive the event flow information from the CPS files. These methods analyze the case statement inside an event handler specification to determine the conditions under which an incoming event has no impact on the system state (this determines when it is safe to drop an event). Cadena also provides a number of helper functions for flow analysis to navigate the communication topology of the system. Together with the functions to analyze the component behavior and to navigate the interconnection topology, backward flow analysis can be performed iteratively, starting from modal components, to determine the set of connections in the entire system that can be suspended for specific combination of mode values.

Our framework also supports the specification of the event flow control by the designer as an aspect. For example, although the original application may have been written without modal behavior, the designer can superimpose the modal behavior by specifying which events to drop or delay based on application conditions. This information can be specified directly in the XML format of the `ModeSpec` file or a higher level language can be designed whose specification can be translated into the XML format.

### 4. EVALUATION

We have performed a number of experiments to demonstrate the usefulness of our model driven approach. The experiments were carried on a machine running XP with In-
<table>
<thead>
<tr>
<th>Event Communication Configuration</th>
<th>Average time per iteration (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD + AMM</td>
<td>1.059</td>
</tr>
<tr>
<td>DD + FCM</td>
<td>.691</td>
</tr>
</tbody>
</table>

Figure 5: ModalSP scenario experiments with DD

<table>
<thead>
<tr>
<th>Event-Communication Configuration</th>
<th>Average time per iteration (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES + AMM</td>
<td>1.059</td>
</tr>
<tr>
<td>AES + FCM</td>
<td>.902</td>
</tr>
<tr>
<td>AES + MES</td>
<td>.889</td>
</tr>
</tbody>
</table>

Figure 6: ModalSP scenario experiments with AES

System’s MediumSP scenario shown in Figure 7. In this scenario, in addition to the TacticalSteering and NavSteering components, we now have several intermediate components. The TrackSensor (TS) components are triggered by timeout events. Event from the TS components in turn trigger the Track (TR) components. The ModeSource component updates the modes of the modal components. Another goal of this experiment was to demonstrate the use of the analysis techniques. The first row in Figure 8 corresponds to the case where only NavSteering and TacticalSteering are modal components. For the second row, we perform one step of analysis by inspecting the components whose connections can be dropped, given the modal behavior of NavSteering and TacticalSteering. This analysis results in inferring the modal behavior of the Track components (Track1, . . . , Track10). For example, Track1 can drop event from TrackSensor1 if NavSteering is disabled. The third row corresponds to the case where another step of the analysis algorithm is carried out to infer the modal behavior of the TrackSensor components as well. Thus, each step of the analysis results in getting more information regarding connections that can be dropped. For comparison, the second column in Figure 8 shows the performance for the AES + AMM approach. We make the following observations regarding this approach:

1. This approach requires modifying the Track and TrackSensor components to make them modal.
2. The mode-source component must update the mode of each modal component. Hence, as the number of modal components increases, this overhead also increases.
3. The third column shows the performance of the AES + MES approach. In this case, we do not have to change the Track and TrackSensor. Furthermore, the overhead of mode updates remains the same (as a single notification announc
Table 1: Configuration Modal Components Average time per iteration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Modal Components</th>
<th>Average time per iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD + AMM</td>
<td>Steering, TR, TS</td>
<td>7.952</td>
</tr>
<tr>
<td>DD + FCM</td>
<td>Steering, TR, TS</td>
<td>6.570</td>
</tr>
</tbody>
</table>

Figure 9: MediumSP scenario experiments with DD

that a subsystem is inactive is sufficient). As can be seen, the MES approach performs better than the AMM approach. The fourth column shows the performance for AES + FCM approach. This approach share the same advantages as MES over the AMM approach. However, MES performs better than FCM as embedding control logic inside the event service is more efficient. Figure 9 shows the results of some of the experiments when the communication is using the DD scheme rather than AES. Our experiments essentially confirm our prediction that our model-driven approach that embeds the application control logic in the middleware leads to better performance than the traditional approach relying on modifying the application alone. For AES, MES performs better than the FCM approach. Hence, a combination with FCM for DD connection and MES for AES connection can give the best performance.

5. CONCLUSION

We have presented a model-driven approach to dynamically control the flow of events on event connections. As part of this approach, we proposed the FraMES framework which provides mechanisms to control the event flow. We have developed algorithms to analyze the application behavior to generate the metadata specifying how to control events. These algorithms are specially useful in large scale DRE systems where deriving such information manually is tedious and error-prone. We have applied the proposed mechanisms to application scenarios to demonstrate the benefits of dynamically controlling the flow of events.

6. REFERENCES


8. REFERENCES