Customizing Event Ordering Middleware for Component-based Systems *

Gurdip Singh and Sanghamitra Das
234 Nichols Hall, Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506
singh@cis.ksu.edu

Abstract

Middleware services have been used extensively in the development of distributed real-time embedded (DRE) systems to address their communication requirements. The stringent performance requirements of DRE systems often require highly optimized implementations of middleware services. Performing such optimizations manually can be tedious and error-prone. The focus of this paper is on middleware services for ordering events. We propose a model-driven approach to generate customized implementations of event ordering services in the context of component based systems. Our approach is accompanied by a number of tools to automate the customization. Given an application App, an event ordering service Order and a middleware platform P, we provide tools to (a) Analyze high-level specifications of App to extract information relevant to event ordering, (b) Use the extracted application information to obtain a customized service, Order(App), with respect to the application usage, and (c) Exploit the properties of the specific communication mechanisms by provided by P to customize the implementation of Order(App) on P.

1 Introduction

Distributed real-time embedded (DRE) systems often involve a large number of components, distributed across several processing nodes, 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 aid in the design of such large-scale DRE systems [HDD+03, SKZ04, GNS+02, SZP+03]. Cadena is one such tool for modeling, analysis, development and deployment of CCM based systems. In Cadena, a system is defined by set of component instances and their interconnections between the ports of the components. Cadena tools provide capabilities to analyze such systems and generate the code templates for the components with business logic to be filled in and the configuration code necessary to deploy the system. An important part of the deployment phase is the implementation of the event and data connections between the components via the underlying middleware services. Event service middleware has been used extensively to implement

---

*This work was supported by DARPA PCES contract F33615-00-C-3044 and NSF awards EIA9980321 and CCR0098179.
event communication and is well suited for component based systems as it is anonymous in nature; that is, producers and consumers are not aware of one another [HLS97, CNF98, KR95, RVC95, FG00]. A number of QoS aspects such as real-time delivery and fault-tolerance have been explored in the literature and have been integrated into event service middleware as well.

This paper explores the generation of optimized implementations of event ordering middleware for component based systems. Different types of ordering requirements such as FIFO, causal and total ordering have been studied in the literature, and algorithms for implementing them have been proposed [BvR94, ADKM92, PRS97, KS98, MG97, JSM97, MMSA +96]. Consider the integration of an event ordering algorithm, say Order\_Alg, into an event service. In performing such an integration, designers are faced with two opposing choices. The first is to create a generic, reusable service by embedding Order\_Alg into the event service as shown in Figure 1(a) and enhancing the event service interface to allow specification of ordering requirements. Second, the stringent application requirements such as those of DRE systems may force the designers to develop optimized version of the service, customized to the specific application contexts. In the following, we discuss two aspects of this tradeoff:

1. To enable reusability, algorithms often do not make any assumptions regarding the application. As a result, a straightforward use of a causal ordering algorithm, for instance, may result in large amount of dependency information being propagated which never gets used. Similarly, traditional algorithms for total ordering operate with the "pessimistic" assumption that the application may issue events in any order and the algorithm must perform the necessary work to impose a total order. While this assumption may be true in general, a specific application may issue events in a predefined order (e.g., an event \(e_a\) may be issued only in response to event \(e_b\)). Using this information, it is possible to customize the ordering algorithms to a specific application usage. This, for instance, is similar a compiler optimizing a library function based on its usage (e.g., replacing a parameter by a constant \(c\) if all invocations to the function use \(c\) as the actual parameter).

2. To target a larger class of underlying platforms, algorithms often make weak assumptions regarding the communication mechanisms (such as "message delivery time is unbounded but finite"). In specific target platforms, however, stronger assumptions may hold. For example, in the event service used in the Boeing Bold Stroke system [Sha99], event communication between components co-located on the same server is implemented via synchronous method calls, which results in a smaller set of possible sequences in which the events can be interleaved and delivered. This restricted behavior of the underlying platform can be used to further customize the event ordering algorithms. This, for instance, is similar to a compiler optimizing the program during the code generation phase by exploiting the properties of the target architecture (e.g., rearranging the instructions to minimize delays).

Therefore, to use generic, reusable services, techniques to customize them for specific applications and platforms are needed. Such customization techniques based on static analysis can be applied to DRE systems as they are often closed systems, wherein the set of components and their interconnections, as well as the target platform are known in advance. For example, the Boeing Bold Stroke system [Sha99] for designing avionics applications uses the TAO’s real-time event service [HLS97], which is an instance of a general reusable middleware service. The Boeing engineers have identified several optimizations (such as the one discussed in (1) and (2) above) that can be exploited in specific application contexts to improve
Figure 1: Model-driven approach

performance. For small systems, such optimization opportunities can be identified and performed manually. However, for large systems, this exercise can be tedious and error prone. Therefore, automated techniques to optimize implementations based on the application usage are needed. To address this problem, we propose a model-driven approach to customize event ordering algorithms. As shown in Figure 1, the approach involves the following main steps:

- Our approach starts with the designer specifying the application, \textit{App}, in Cadena. We provide tools to extract the application’s event communication topology from its specification, which not only includes the inter-component communication edges but also intra-component dependencies specifying when events are generated. This information is made available in the form of a port topology graph, \textit{PTG}(\textit{App}), which is amenable to analysis.

- To illustrate our approach, we use two existing algorithms, \textit{Causal IDR} and \textit{Total Sequencer}, for causal ordering and total ordering respectively. Given \textit{PTG}(\textit{App}), we provide techniques to generate optimized versions of \textit{Causal IDR} and \textit{Total Sequence} with respect to \textit{App}. For each case, we present an analysis algorithm to determine aspect/functionality of the algorithm which is redundant with respect to the given application and provide techniques to optimize the algorithm by eliminating this redundant functionality. For \textit{Causal IDR}, the redundancy is in terms of dependency information which never gets used in the application, and for \textit{Total Sequencer}, it is the ordering already performed by the application itself (which is used to avoid the algorithm from redoing this work).

- To illustrate optimizations with respect to a target platform, we use the event service employed in the Cadena’s deployment framework, which is a Java version of the event service used in Boeing Bold Stroke system. This service provides two mechanisms for event communication, direct dispatching and full channel dispatching. We show that the properties of these mechanisms can be used to further optimize \textit{Causal IDR} and \textit{Total Sequencer}.

It is known that application semantics and network topology information can be exploited to improve performance, and several techniques such as integrated layer processing [AP93] and partial evaluation [MTC99] have been proposed for specific classes of programs. One of our main contributions is to provide the infrastructure support to automate the tool-chain for enabling such optimizations. This includes techniques to represent application information in intermediate representations amenable to analysis and tools to configure middleware services. Although we have used specific algorithms as case studies, our
infrastructure allows analysis of different types of the plugged in to optimize the implementation of DRE systems. We believe that this type of model-driven development can be used in many different contexts.

2 Overview of Cadena

In this section, we refer the aspects of Cadena which are relevant to event communication and our model-driven approach. Cadena is an integrated modeling environment for modeling and building CCM systems. It provides facilities for defining component types using CCM IDL, assembling systems from CCM components and producing CORBA stubs and skeletons implemented in Java. This system is realized as a set of components and the port connections. The overall development process of a distributed application in Cadena is shown in Figure 2. We will use as an example of a simple avionics system shown in Figure 3 to illustrate the steps.

- The first step in the specification of components in the CCM model. In Figure 4, we give the CCM IDL specified by the designer to define the component type BMLazyActive for the AirFrame component instance in Figure 3. CCM components provide interfaces to clients on ports referred to as facets, and use interfaces provided by other clients on ports referred to as receptacles. Components publish events on ports referred to as event sources, and consume events on ports referred to as event sinks. In the BMLazyActive component type of Figure 4, inDataAvailable1 is the name of an event sink of type DataAvailable, and outDataAvailable is the name of a event source of type DataAvailable.

- The next step is to assemble a system by identifying the instances of the component types and their interconnections. This is given by the Cadena Assembly Description (CAD), whose excerpts for the example system are shown in Figure 4 (b). In CAD, a developer declares the component instances that form a system, along with the interconnections between the ports. For receptacle and event sink ports, a connect clause declares a connection between a port of the current instance and a port of the component that provides the interface/event. For example, the connect clause in Figure 4 connects the outDataAvailable port of GPS to the inDataAvailable2 port of AirFrame.

- The next phase is the generation of code and the configuration metadata. Cadena uses the OpenCCM’s IDL to Java compiler to generate the component and container code templates from the component IDL definitions. This produces an implementation file for each component into which the designer is supposed to fill the business logic. From the CAD file, Cadena tools also generates configuration code as well as
XML metadata to deploy the system. The generation of this metadata involves executing a number of analysis algorithms and assignments of ids to components and ports.

- The final step is the deployment phase in which the application is deployed on a target platform. The deployment phase involves the following steps:
  - (a) A component server is first installed at each location (or processor).
  - (b) Within each component server, the components (along with their containers) assigned to each server are instantiated.
  - (c) The containers use the underlying middleware services to set up the event and data connections. The event service middleware used to implement the event connections is discussed in the following.

An event service is a middleware service which brokers communication between producers and consumers [Gro95, ASS+99, HLS97, HBBM96, MB98, FGS97, KR95, LM99, RRDC97]. 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. We have developed the Adaptive Event Service (AES), a Corba-based Java event service, whose architecture is shown in Figure 5. It is possible to use AES as a stand alone CORBA service. In this case, a single event channel is created and all containers (from all component servers) interact with the channel via the ORB.

Figure 3: A simple avionics system

```plaintext
#pragma prefix "cadena"
module modalsp {
    interface ReadData {
        readonly attribute any data;
    }
    eventtype TimeOut {};
    eventtype DataAvailable {};
    enum LazyActiveMode {stale, fresh};
    component BMLazyActive {
        publishes DataAvailable outDataAvailable;
        consumes DataAvailable inDataAvailable1;
        consumes DataAvailable inDataAvailable2;
        attribute LazyActiveMode dataStatus;
    };
    (a)
}
```

Figure 4: (a) CCM/Cadena artifacts, (b) Cadena Assembly Description for ModalSP (excerpts)

```plaintext
system ModalSPScenario {
    import cadena.common, cadena.modalsp;
    Rates 1, 5, 20; // Hz rate groups
    Locations 11, 12, 13; // abstract deployment locs
    ...
    Instance AirFrame implements BMLazyActive on #LAloc {
        connect this.inDataAvailable2 to GPS.outDataAvailable runRate #LArate;
        connect this.inDataAvailable1 to Navigator.dataOut;
    }
    Instance Display implements BMModal on l2 {
        connect this.inDataAvailable to AirFrame.outDataAvailable runRate 5;
        connect this.dataIn to AirFrame.outDataAvailable;
    }
    ...
    (b)
}
```
This, however, is inefficient as every notification will be a remote call via the ORB. Therefore, we adopted an architecture wherein an event channel is created in each component server as shown in Figure 5(a). The components on the same component server communicate via the local event channel (which is more efficient) whereas components on different component servers communicate via gateways. Let $\text{publish}(C)$ and $\text{consume}(C)$ denote sets of events published and consumed by component $C$ respectively. During the code generation phase, Cadena tools generate the connection metadata file (CMF) in XML format which contains the information regarding the $\text{publish}$ and $\text{consume}$ sets for each component. At deployment time, the containers use the CMF to configure the connections; that is, the container for component $c$ connects to the local event channel as publisher of events in $\text{publish}(c)$ and as a consumer for events in the set $\text{consume}(c)$.

To summarize, as shown in Figure 6, Cadena provides an end-to-end automated tool chain which starts at the modeling level and ends with deployable implementations (the solid lines denote the existing design steps). The tool-chain exposes several intermediate representations and configurable metadata at various points in the chain. The optimization techniques discussed in the following sections essentially are used to plug in analysis algorithms in the tool-chain (shown by dotted lines in Figure 6) which leverages the intermediate presentations to generate metadata to configure the middleware for the purpose of optimizing the implementations.

### 3 Event Ordering Specification

For an event $e$, we source $e_{\text{src}}$ to denote the port that published $e$, $e_{\text{pub}}$ to denote the event of $e$ being published and $e_{\text{deliver}}(p)$ to denote the event of $e$ being delivered on port $p$. We use $(p, q)$ to denote the
connection from an output port \( p \) to an input port \( q \). Let \( \rightarrow \) denote the happens before relation defined in [Lam78]. The FIFO ordering requirement is as follows:

- **FIFO**\((p,q)\): Let \( \langle p, q \rangle \) be an event connection. For all events \( e_i \) and \( e_j \) published on port \( p \), if \( e_i \cdot \text{pub} \rightarrow e_j \cdot \text{pub} \) then \( e_i \cdot \text{deliver}(q) \rightarrow e_j \cdot \text{deliver}(q) \).

Informally, this requirement asserts that events from the same port are delivered in the order published over a connection. For example, FIFO ordering over the connection from GPS to AirFrame in Figure 3 would ensure that data values are delivered to AirFrame in the order in which the readings are taken by GPS. The causal ordering requirement is as follows: Let \( p_1, \ldots, p_x \) be the input ports of a component.

- **Causal**\((p_1, \ldots, p_x)\): For any events \( e_1 \) and \( e_2 \) received on ports \( p_i \) and \( p_j \) respectively, where \( 1 \leq i, j \leq x \), if \( e_1 \cdot \text{pub} \rightarrow e_2 \cdot \text{pub} \) then \( e_1 \cdot \text{deliver}(p_i) \rightarrow e_2 \cdot \text{deliver}(p_j) \).

Informally, this semantics requires causally related events to be delivered in the order of occurrence. For example, in Figure 3, Navigator sends data in an event \( e_1 \) to AirFrame, which in turn updates its own data and sends an event \( e_2 \) to display. If \( e_2 \) is received by Display before \( e_1 \) (which violates causality) then this may result in Display using the older value from Navigator with new data from AirFrame (or displaying the “effect” before the “cause”). Enforcing causal ordering on events will eliminate such inconsistencies. However, causal delivery allows concurrent messages to be delivered in any order and the order may be different in different components. The total ordering requirement, on the other hand, guarantees the following property: Let \( p_1, \ldots, p_x \) be a set of input ports belonging to a set of components.

- **Total**\((p_1, \ldots, p_n)\): For any events \( e_1 \) and \( e_2 \) received on all ports \( p_1, \ldots, p_n \), if \( e_1 \cdot \text{deliver}(p_i) \rightarrow e_2 \cdot \text{deliver}(p_i) \) then \( e_1 \cdot \text{deliver}(p_j) \rightarrow e_2 \cdot \text{deliver}(p_j) \) for all \( 1 \leq i, j \leq x \).

Informally, total ordering requires that the common set of events received on ports \( p_1, \ldots, p_x \) be delivered in the same order on all ports. Total ordering may be required in cases where consistency is required across components. For example, consider the case where two components Display1 and Display2 both receive events \( e_1 \) and \( e_2 \), and depending on which event is received first, different actions (resulting in different displays) are taken by each component. If these events are not delivered in the same order to the two display components, then this may result in inconsistent displays being shown by the two components. Total ordering on \( e_1 \) and \( e_2 \) will eliminate such inconsistency.

Although one can require all events to be delivered in FIFO, causal or total order, the ordering requirements in a DRE application may vary for each component. In many systems, for example, data values in the events often have a validity interval associated with them. Based on this, in Figure 3, we may relax the causal ordering requirement by allowing the Display to use an older value from Navigator as long as the value is still valid. Thus, even if \( e_2 \) is received before \( e_1 \), Display can use a value from Navigator’s event \( e_1 \) with a more recent value from AirFrame as along as the value in \( e_1 \) is still valid. To allow this flexibility, we allow the designer to selectively specify ordering constraints for specific components using the predicates FIFO, Causal and Total discussed above.

### 3.1 Existing algorithms and problem motivation

To illustrate our model-driven approach, we study two existing algorithms, one for causal ordering and the other for total ordering.

- **Causal ordering** is typically implemented by propagating dependency information in the events [BvR94, PRS97, KS98]. In particular, [PRS97] proposed an algorithm, which we refer to as Causal JDR, based on
computing the immediate dependency relation (IDR). In this algorithm, each event $e$ only carries the set $IDR(e)$ containing its immediate predecessors. When $e$ is received by a component, its delivery is delayed until all events in $IDR(e)$ have been delivered. We will illustrate the main aspects of the algorithm using an example. Consider the scenario shown in Figure 7(a) where $e_1$ is published on the reception of $e_0$, and $e_2$ is published on the reception of $e_1$. In this example, when $e_1$ is sent, its IDR set will contain $e_0$. When $e_2$ is sent, $B$ includes $e_1$ in $IDR(e_2)$. However, $e_0$ is not included in $IDR(e_2)$ since it does not immediately precede $e_2$. Since $IDR(e_1)$ includes $e_0$, delivery of $e_1$ before $e_2$ will ensure that $e_0$ is delivered before $e_2$. However, to compute the IDR set, $O(N^2)$ integers may have to be sent along with each event in the worst case, where $N$ is the number of components (this involves information about the destination set of each event and the set of known concurrent events from other sites). Thus, a straightforward use of this algorithm can result in a large amount of information being propagated, especially in applications with a large number of components.

- Total ordering on events can be implemented using a central sequencer site to order the events. In this algorithm, which we refer to as Total Sequencer, each event to be ordered is sent to all components, including a central sequencer site. The sequencer site assigns timestamps in a linear order to all events it receives, and sends the timestamp of each event to all components. The events are then delivered in the timestamp order to all components. For example, in Figure 7, both $A$ and $B$ send events $e_1$ and $e_2$ respectively to $C$ and $D$. Since $e_1$ reaches the sequencer site first, it is assigned timestamp 1 whereas $e_2$ is assigned timestamp 2. Thus, even though $e_2$ reaches $C$ before $e_1$, it is delayed until $e_1$ is delivered.

DRE systems often have stringent QoS constraints and require highly optimized implementations, and a straightforward of Causal IDR and Total Sequencer may not satisfy these requirements. This mismatch in efficiency can be attributed to two reasons. First, the algorithms Causal IDR and Total Sequencer do not make any assumptions regarding the application structure. As a result, the algorithms attempt to capture information about the application structure on-the-fly. For example, in Causal IDR, data structures such as the destination set and the set of known concurrent messages which are sent along with each message essentially propagate information about the application structure. Second, some of the ordering performed by the algorithms may be redundant. For instance, [PRS97] gives an example of a case where the algorithm may propagate dependency information which is never used. Similarly, the total ordering algorithm may perform redundant work in ordering events that are already sent in a specific order by the application. Both of the sources of inefficiency can be eliminated if the application structure is known in advance. This is typically the case in DRE systems as they are often closed systems; even though components may
be switched in and out, the possible set of components and how they will be interconnected is known in advance.

Given the application structure and the middleware platform, it may be possible to identify and perform optimizations manually for small scenarios. However, for scenarios involving a large number of components, this can be tedious and error prone. Hence, automated tools are needed to configure the application. To address these issues, we propose a model-driven approach to optimize middleware for event ordering. The approach offers the following:

1. **Tools to derive application context information**: We provide automated analysis algorithms to derive the port topology graph (PTG) from the application specification. The PTG includes the dependencies between the ports of the components and presents the designers with an abstraction amenable to analysis.

2. **Tools for optimizing ordering algorithms with respect to the application structure**: We present algorithms to analyze the PTG with respect to the ordering requirements. Using the PTG, we show that how the computation of the immediate predecessor relation in CausalIDR can be optimized. We also show that we can determine the ordering already performed by the application, and how this information is used to optimize TotalSequencer by weakening the total ordering requirements.

3. **Tools for optimizing implementation with respect to the middleware communication mechanisms**: We present techniques to utilize specific properties of the communication mechanisms to optimize the implementation of an ordering algorithm. A general implementation of an algorithm may have to cope with arbitrary sequences in which events can be delivered by the underlying middleware. However, when specific mechanisms are used, only a subset of these sequences might be possible. By making use of this information, we show that the implementations can be optimized.

Several techniques such as integrated layer processing [AP93] and partial evaluation [MTC99] have been proposed to optimize implementations to a particular application usage. For example, to enable reusability, software is often organized as a set of layers. Given the protocol layers needed for an application, techniques in [AP93, MTC99] can be used to collapse the layers to eliminate redundant processing. There have also been proposals for optimizing ordering algorithms based on the application structure and network topology. For example, [RV95] uses the information about the network topology to determine when to propagate dependency information. The algorithm in [PRS97], on the other hand, attempts to gather information about the application structure on-the-fly. Our main contribution is the development of the infrastructure that makes such analysis and optimization possible in an automated manner. Our development methodology exposes the application information that can be used by analysis algorithms. Thus, in addition to the algorithms that we propose, existing analysis such as those in [RV95] can be plugged into our infrastructure.

4 **Derivation of the application structure in Cadena**

In this section, we first define the data structures to capture the application information necessary for optimizing ordering algorithms. The graphical interface of Cadena shows the component topology graph (CTG), which is denoted by \((V, E)\), where each vertex in \(V\) denotes a component and there is an edge from \(v_1\) to \(v_2\) if there exists an event connection from an event source port of \(v_1\) to an event sink port of
A port topology graph (PTG) of an application contains more information and is defined as a graph \((V, E)\), where each vertex in \(V\) denotes an event source or a sink port of a component. There is an edge \((v_1, v_2)\) in a PTG if

(a) \(v_1\) is a source port of a component and \(v_2\) is a sink port of another component and \(v_1\) is connected to \(v_2\) (inter-component edge), or

(b) \(v_1\) and \(v_2\) are sink and source ports of the same component respectively, and the receipt of an event on \(v_1\) can cause an event to be published on \(v_2\) (intra-component edge).

The intra-component edges are further classified into two types, deterministic and non-deterministic. An edge \((v_1, v_2)\) in component \(C\) is deterministic if in all executions, whenever an event is received on \(v_1\), \(C\) publishes an event on \(v_2\) without awaiting the occurrence of any other event. Otherwise, the edge is labeled non-deterministic. An example of a PTG is given in Figure 8. The solid edges denote the inter-component communication edges whereas the dashed edges denote the intra-component port dependencies, and the dashed boxes represents the components.

We now discuss the derivation of the PTG from the application specification. As shown in Figure 6, the Cadena infrastructure stores application information in intermediate representations that are exposed to analysis algorithms. In particular, one such representation is the Abstract Syntax Tree (AST) which provides information regarding the component instances and the interconnections between them. Our analysis algorithm traverses the AST to derive the inter-component edges in the PTG. The intra-component edges are derived from the Component Property Specification (CPS) files. In Cadena, a CPS file is associated with each component to specify dependencies between the ports and behavioral specifications for the event handlers. A fragment of the CPS file for the BMLazyActive component from Figure 3 is shown in Figure 9 (the grammar is given in [Cad]). The case statement in the CPS file specifies how an incoming event is processed. For example, the case statement in Figure 9 specifies that when an event on port inDataAvailable is received, if the variable modeVar’s value is enabled then a synchronous method call on dataport dataIn is made and an event on port outDataAvailable is generated (otherwise, the event is discarded). This, for instance, will result in the edge from inDataAvailable to outDataAvailable to be labeled as non-deterministic. We have written a program to derive this dependency information from the CPS files. The main subroutine used by the program is analyseCaseStatement(p), where \(p\) is a port-name, which analyzes the CPS file to return the set of output ports on which an event may be generated on receiving an event on \(p\). This information is then used to determine the intra-component edges as well as their types, deterministic or non-deterministic.
4.1 Optimizing algorithms using PTG

In the following sections, we present techniques to optimize the causal ordering and total ordering algorithms. To allow optimizations, the algorithms themselves must be customizable. Therefore, we first modify each of the algorithms into a form that can be customized at initialization time (this includes enhancing the metadata format to include the new information). We then show how our analysis algorithms can be plugged into the tool-chain (see Figure 6) to analyze PTG(app) and generate the enhanced metadata needed for customization.

4.1.1 Causal ordering algorithm

The algorithm for causal ordering in [PRS97] works by generating and propagating dependency information in the events. We have modified the algorithm, which we refer to as Causal IDR Optimized (CIO), which now uses two tables as input, a generation_rule table and a propagation_rule table, to compute and propagate dependency information respectively. These tables are defined for each component separately. Let q (r) be an input (output) port of a component B, and p be an output port of another component A. An element \((p, q, r)\) in the generation_rule table of B (see Figure 10(a)) implies that the events published on connection \(h_p;q\) must be included in the IDR set sent along with events published on port \(r\). This table determines when new information is introduced in the IDR sets (each port in the system is assigned a unique id in Cadena, and the tuples in the tables are in terms of these ids). To propagate information already received in an event, we define the propagation rules. An element \((p, q, r)\) in the propagation_rule table of B (see Figure 10(b)) implies that the events published from port \(p\) that is contained in the IDR set of events arriving on port \(q\) must be included in the IDR sets events published on port \(r\). Each component is provided with these sets at initialization time. When an event is published by a component, the propagation_rule and generation_rule tables are consulted to determine the dependency information to be propagated. In addition, when an event arrives at a component, the dependency information contained in the event is used to determine whether or not to delay the delivery of the event (depending on whether the preceding events have already been delivered).

We now describe the derivation of these tables using application information.

**Definition:** We say that a PTG \(G\) has a causal_cycle if there exists a component \(A\) with output port \(p\) and
a component $B$ with input ports $q_1$ and $q_2$ such that $p$ is connected to $q_1$ and there is a directed path from $p$ to $q_2$ (see Figure 10(c)). We will also refer to this as the causal cycle from $p$ to $(q_1, q_2)$.

Let $Causal(q_1, \ldots, q_x)$ be an ordering requirement for a component $B$. For each pair $(q_i, q_j)$, we first obtain the set of all causal cycles in the PTG from all ports to $(q_i, q_j)$. This is done using a variation of the depth-first traversal algorithm. Assume that there exists a causal cycle from port $p$ of $A$ to $(q_i, q_j)$. Let $ep = p.event$ and $eq = q_2.event$, and $p, in_1, out_1, \ldots, in_l, out_l, q_2$ be the path from $p$ to $q_2$ as shown in Figure 10(c). In this case, we know that it is possible for $ep$ to causally precede $eq$. Therefore, we need to propagate dependency information along this path. Therefore, we add $(p; in_1; out_1)$ to the generation_rule table of $in_1$’s component, and $(p; in_j; out_j)$, where $1 < j \leq x$, to the propagation_rule table of $e_j$’s component.

The rules described above add the tuples to the tables which are necessary to preserve causality. This eliminates the propagation of redundant dependency information. Since the original algorithm $Causal.IDR$ does not make any assumptions regarding the application, it always propagates the dependency information (which corresponds to having all tuples $(p, q, r)$, for each input $q$ and output port $r$ of the component, present in the tables). At run time, however, $Causal.IDR$ attempts to reduce the dependency information using additional data sent along each event. For the example in Figure 7(a), along with $e_1$, the destination set $\{B, C\}$ is also sent. From this information, $B$ knows that $C$ also received $e_1$; hence, when $e_2$ is sent to $C$, $B$ includes $e_1$ in the IDR($e_2$) so that causality can be enforced at $C$. Essentially, this can be viewed as attempting to acquire knowledge of the potential causal cycles at run-time. Our analysis algorithm, on the other hand, use the PTG to perform such optimizations statically. Furthermore, at run-time, one cannot determine whether some dependency information will be needed in the future and hence, one has to operate conservatively. However, in our framework, we eliminate some of this information statically (e.g., for events that are not part of any causal cycle) by analyzing the PTG.

### 4.1.2 Total ordering algorithm

As discussed in Section 3, the total ordering requirement is specified as a set of predicates of the form $Total(p_1, \ldots, p_x)$. For each such predicate $tp$, let $source(tp)$ denote the set of source ports from which events are to be delivered in a total order to all ports $p_1, \ldots, p_x$. The algorithm $Total.Sequencer$ takes the set $source(tp)$ as its input and imposes a total order on the events issued by ports in $source(tp)$. In the modified algorithm, $Total.Sequencer_Opt$ (TSO), the input to the algorithm is $\langle source(tp), \Rightarrow \rangle$ instead, where $\Rightarrow$ is the triggers relation satisfying the following properties:
(Ta): For each port $p_j$, there exists at most one port $p_i$ such that $p_i \Rightarrow p_j$, and

(Tb): The relation $\Rightarrow$ is acyclic.

Informally, $p_x$ triggers $p_y$ if the occurrence of an event on port $p_x$ always causes an event on $p_y$ to occur. For example, consider the case in Figure 3 where on receiving an event published on the $outDataAvailable_1$ port of $GPS$, the corresponding event handler of $AirFrame$ recomputes its data and send the updated data on an event from port $outDataAvailable$ to the $Display$. In this case, $outDataAvailable$ port of $GPS$ triggers the $outDataAvailable$ port of $AirFrame$. We will now use a small example to illustrate the concepts in TSO. Let $source(tp) = \{p_1, p_2, p_3\}$, and assume that $p_1$ triggers $p_2$. Thus, after each event publication on $p_1$, an event is subsequently published on $p_2$; however, events on $p_3$ can be issued concurrently with those from $p_1$ and $p_2$. In TSO, we take advantage of this information as follows. We first partition the set $source(tp)$ into a set of domains, where all ports within a domain are related by $\Rightarrow$. From properties $Ta$ and $Tb$, each domain has a single root port which is not triggered by any other port. To enforce a total order on the delivery of the events, TSO only orders the events issued by the root ports. That is, only these events are sent to the sequencer site and are assigned the timestamp. Events from other ports are sent directly to all components and are ordered with respect to the root events using a “deterministic merge” mechanism which merges events from different domains in a deterministic manner. One simple merge mechanism is atomic merge wherein all events of a domain are ordered immediately after the root event. In our example, there are two domains, $\{p_1, p_2\}$ and $\{p_3\}$. After an event $e_1$ from $p_1$ is received by a component $A$, we know that an event, $e_2$, from $p_2$ will be published. In this case, after delivering $e_1$, each component waits for $e_2$ to arrive before delivering any other event (thus, $e_2$ is scheduled for delivery immediately after its root event). Hence, $e_2$ does need to the timestamped by the sequencer site. To ensure total ordering, the main requirement is that the merge be deterministic and the same at all components. Since only root events are sent to the sequencer site, the optimized algorithm can lead to significant savings in the number of messages and latency.

We now discuss the derivation of $\Rightarrow$ from the PTG. For two ports $p_x$ and $p_y$ in $source(tp)$, $p_x \Rightarrow p_y$ if $p_y$ is reachable from $p_x$ via a path such that the following is true for each edge $(p_i, p_j)$ in the path:

($Tr1$): if $(p_i, p_j)$ is an intra-component edge, then it is deterministic and either $p_j = p_y$ or $p_j$ does not belong to $tp$,

($Tr2$): if $(p_i, p_j)$ is an inter-component edge then there is no other incoming edge for $p_j$.

Condition $Tr1$ states that $p_y$ must be reachable from $p_x$ via a path that does not involve any other port in $tp$ whereas Condition $Tr2$ ensures that no port other than $p_x$ can also cause $p_y$ to generate an event. We perform this computation via a depth-first traversal of the PTG using only the edges that satisfy the criteria $Tr1$ and $Tr2$. This information is then provided as metadata for configuration of the TSO.

For less structured applications (such as discussion groups), the components may issue events at arbitrary times, and very few ports may be related by the triggers relation. However, in DRE systems with pre-defined communication topologies, the triggers relation will contain more entries, and hence, greater will be savings in the number of messages and latency. Although the relation $\Rightarrow$ can be computed as described above, the application designer may want to relax $\Rightarrow$ based on other factors such as schedulability and latency (weakening $\Rightarrow$ does not impact the correctness). For instance, consider a case wherein there are two domains, one consisting of a single port $p_1$ and the other consisting of a large number of ports with the root as $p_2$. In this case, if an atomic merge mechanism is being used, then after an event on $p_2$ is received, the delivery of the event $e_1$ from $p_1$ might get delayed for a long time by events from the other.
domain (possibly violating $e_1$'s deadline). In this case, $\Rightarrow$ can be relaxed to break the second domain into smaller domains. An idea similar to that of deterministic merge was proposed in [AS00] where the fact that all components were producing messages periodically is used. These messages are not sent to the sequencer site but merged at the individual sites using a mechanism similar to deterministic merge.

5 Customizing algorithms using middleware information

In this section, we discuss the optimization of Causal IDR and Total Sequencer by exploiting the properties of the communication mechanisms in the underlying middleware. In the following, we give a brief description of the two mechanisms available for event notifications in our target platform:

- **Full Channel Dispatching** (FCD): To explain this mechanism, we first give a brief description of the threading architecture of the event channel, which is based on the Bold Stroke architecture [Sha99]. In this architecture, all threads reside in the event channel and the components are reactive in nature. Each port is associated with a rate group which denotes the rate at which the event handler for that port is invoked. Activities are triggered by timeout events generated by the timer threads in the event channel. The rates at which the timeout events are generated are 1Hz, 5Hz, 10Hz, 20Hz and 50Hz. One queue is maintained for each rate group in the dispatching module, and one dispatch thread, $Th_r$, for each rate $r$.

To explain the thread behavior, we use the simple example shown in Figure 11(a) wherein GPS receives a 5Hz event on an input port ($inData1$) and in response, publishes an event on its output port $outData$. This event from GPS is then consumed by AirFrame on its port $inData2$. In addition, we assume that all ports are associated with the same rate group (5hz). Thus, when a 5Hz event is generated by the event channel, a notification for GPS is placed in the 5hz queue. Each dispatch thread iteratively picks an event from its respective queue and invokes the handler of the specified consumer component. Thus, $Th_5$ will invoke the $inData1$ handler of GPS. The execution of the handler causes the publication of an event on the $OutData$ port. When this event arrives in the event channel, the subscriber list of this event is consulted and a notification for each subscriber is placed in the dispatch queue. In this case, a notification for AirFrame is placed in the 5hz queue (as shown in Figure 11(a), all of these tasks are done by $Th_5$). Subsequently, the call to publish the event completes and $Th_5$ resumes the execution of the GPS’s $InData$ event handler. On the completion of this handler, $Th_5$ processes the next event in $Queue_5$. 

![Figure 11: Full Channel and Direct Dispatching](image_url)
Direct Dispatching (DD): The direct-dispatch (DD) mechanism bypasses the event channel by direct communication between the producers and the consumers. In the Bold Stroke design process, to determine whether events from port \( p \) of a component \( A \) to port \( q \) of component \( B \) can be tagged as DD, the condition, \( DD(p, q) \), used is the following:

(a) \( A \) and \( B \) are colocated on the same server,
(b) Both ports have the same rate group, and
(c) The event published on \( p \) is not involved in event correlation.

In this case, since we know that the thread publishing event on \( p \) will also execute the event consumer handlers (even when the notification is via the event channel), one can optimize the notification with a direct method call which bypasses the expensive layers in the event channel (see Figure 11(b) where the event from GPS to AirFrame is direct-dispatched).

In the current BoldStroke design process, condition \( DD(p, q) \) discussed above is used to tag connections as either DD or FCD. BoldStroke designers have found that tagging connections as DD can result in a significant saving in latency. Only events that require additional services (such as correlation or thread switching) have to be sent via the full channel. However, in our case, since the event ordering algorithms TSO and CIO are embedded in the event channel, any event involved in the implementation of the ordering requirements must now also be tagged as FCD. For example, all events in a causal cycle must be tagged as FCD as we need to piggyback IDR information on the events. Thus, the condition (c) in \( DD(p, q) \) must be strengthened to also specify that the events on \( p \) are not involved in implementing ordering requirements. This not only imposes the additional overhead of sending events via FCD, it also involves the event ordering overhead such as computing IDR information or sending events to the sequencer site.

We now look at techniques to alleviate this overhead. The main idea is as follows. If the asynchronous model for event communication is assumed (wherein events are delivered within finite, but arbitrary amount of time), then the delivery of the events can be interleaved in a large number of ways and the ordering algorithms must account for each possible interleaved execution. However, when DD or FCD are used, certain interleavings do not occur. For example, consider the case where \( p, q1 \) and \( q2 \) in Figure 10 belong to the same rate group. If \( e_1 \) and \( e_2 \) are both dispatched via DD, then only the following interleavings are possible:

\[
\begin{align*}
& e_1.deliver(C); e_2.deliver(B); e_1.deliver(B) \\
& e_2.deliver(C); e_1.deliver(C); e_1.deliver(B)
\end{align*}
\]

On the other hand, if \( e_2 \) is FCD and \( e_1 \) is DD then the possible interleavings are:

\[
\begin{align*}
& e_1.deliver(C); e_1.deliver(B); e_2.deliver(B) \\
& e_1.deliver(B); e_1.deliver(C); e_2.deliver(B)
\end{align*}
\]

In the second case, \( e_2 \) is guaranteed to be delivered after \( e_1 \) at \( B \) (ensuring causality). Thus, we can take advantage of this information by dispatching \( e_1 \) via DD and \( e_2 \) via FCD (instead of dispatching both via FCD). This also eliminates the computation of the IDR information as causality is guaranteed. In the following, we identify one instance of such optimization with respect to each ordering algorithm:

Optimization of CIO: We say that \( group(p_1, \ldots, p_x) \) is true if all components associated with the ports \( p_1, \ldots, p_x \) are colocated on the same server and all ports have the same rate associated with them. Based on the properties of the dispatching mechanisms, we have the following lemma:

**Lemma 1:** causal \((q1, q2)\) holds if for each causal cycle from any port \( p \) to \((q1, q2)\), one of the following is true: (a) the notification from \( p \) to \( q1 \) is done before \( p \) to \( in_1 \).
(b) \((ep, q1)\) is DD, at least one connection \(\langle out_{y-1}, in_y \rangle\) in the path from \(p\) to \(q2\) is FCD, and 
\(\text{group}(p, q1, in_1, out_1, \ldots, in_y)\) holds.

The proof of Lemma 1 is delegated to the full paper. If the conditions specified in Lemma 1 hold, then no additional effort is needed by CIO to order events. Condition (a) can be ensured if \(q1\) appears before \(in_1\) in the subscriber list for \(p\). Although this can be achieved at configuration time, it requires more fine-grained control over the configuration process. Hence, we only concentrate on Condition (b). Let there be a causal cycle from \(p\) to \((q1, q2)\) in the PTG. By default, all events in this causal cycle will be tagged FCD. However, if \(DD(p, q)\) holds and there exists \(in_y\) such that \(\text{group}(p, q1, in_1, out_1, \ldots, in_y)\) is true then we tag all connections \(\langle p, in_1 \rangle, \langle out_1, in_2 \rangle, \ldots, \langle out_{y-2}, in_{y-1} \rangle\) as DD whereas \(\langle out_{y-1}, in_y \rangle\) is retained as FCD. This ensures that Condition (b) is satisfied. Furthermore, we optimize CIO by removing \(\langle p, in_1, out_1 \rangle\) from the \textit{generation rule} table of \(in_1\)'s component and \(\langle ep, in_j, out_j \rangle\), where \(1 \leq j \leq x\), from the \textit{propagation rule} table of \(in_j\)'s component.

- **Optimization of TSO**: Let \(tp = \text{Total}(p_1, \ldots, p_x)\) be an total ordering requirement and \(\text{source}(tp)\) be a set of source ports from which the published events to ports \(p_1, \ldots, p_x\).

\textbf{Lemma 2}: \(\text{Total}(p_1, \ldots, p_x)\) holds if all the following are true:

(a) All events from ports in \(\text{source}(tp)\) are direct-dispatched,
(b) All ports in \(\text{source}(tp)\) are the same rate group, and
(c) For all ports \(p_i\) and \(p_j\) in \(\text{source}(tp)\), there does not exist a path from \(p_i\) to \(p_j\) in the PTG labeled with only direct-dispatched edges.

The conditions in Lemma 2 ensure that each event published on a port in \(\text{source}(tp)\) is delivered to all its consumers without interleaving with delivery of any other event from the same set of ports, which guarantees total ordering on the events. Thus, if conditions (b) and (c) hold, and all ports \(p_1, \ldots, p_x\) are in the same rate group as ports in \(\text{source}(tp)\), then we tag the connections from ports in \(\text{source}(tp)\) to ports \(p_1, \ldots, p_x\) as DD (rather than FCD which would be the default). This eliminates the need for sending these events to the sequencer site.

In the discussion above, we have identified one instance of optimization for each of the algorithms. Other optimizations of similar nature can be identified and the corresponding analysis algorithms plugged into our infrastructure.

\section{Conclusion}

We have presented a model-driven approach to customize event ordering middleware. Our approach involves extracting application information in the form of data structures amenable to analysis, algorithms to analyze these data structures to optimize the ordering services with respect to application usage, and techniques to optimize the implementations with respect to the underlying platform. We have demonstrated the use of this approach in optimizing algorithms for event ordering. Analysis algorithms to optimize causal ordering and total ordering algorithms have been implemented. The infrastructure is general in nature and allows analysis algorithms for various types of optimizations to be plugged in.
References


