APPLICATION SEMANTICS BASED OPTIMIZATION OF DISTRIBUTED ALGORITHM

by

SANGHAMITRA DAS

B.E., University College of Engineering, India, 1999
M.S., Kansas State University, 2003

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2012

Abstract

To increase their applicability, distributed algorithms are typically written to work with any application on any network. This flexibility comes at the cost of performance since these 'general purpose' algorithms are written with the worst case scenario in mind. A distributed algorithm written for a specific application or a class of application is fine tuned to the properties of the application and can give a better performance when compared to the general purpose one. In this work, we propose two mechanisms in which we can optimize a general purpose algorithm - *Alg* based on the application - *App* using it.

In the first approach, we analyze the specification of *App* to identify patterns of communication in its communication topology. These properties are then used to customize the behavior of the underlying distributed algorithm *Alg*. To demonstrate this approach, we study applications specified as component based systems where application components communicate via events and distributed algorithms to enforce ordering requirements on these events. We show how our approach can be used to optimize event ordering algorithms based on communication patterns in the applications.

In the second approach, rather than analyzing the application specification, we assume that the developer provides application properties - I_{App} which are invariants for the optimization process. We assume that the algorithm is written and annotated in a format that is amenable to analysis. Our analysis algorithm then takes as input the application invariants and the annotated algorithm and looks for potential functions in the algorithm which are redundant in the context of the given application. In particular, we first look for function invocations in the algorithm whose post-conditions are already satisfied as a result of the application invariants. Each such invocation is considered as a potential redundant module. We further analyze the distributed algorithm to identify the impact of the removal

of a specific invocation on the rest of the algorithm. We describe an implementation of this approach and demonstrate the applicability using a distributed termination detection algorithm.

APPLICATION SEMANTICS BASED OPTIMIZATION OF DISTRIBUTED ALGORITHM

by

SANGHAMITRA DAS

B.E., University College of Engineering, India, 1999
M.S., Kansas State University, 2003

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2012

Approved by:

Major Professor
Dr Gurdip Singh

Abstract

To increase their applicability, distributed algorithms are typically written to work with any application on any network. This flexibility comes at the cost of performance since these 'general purpose' algorithms are written with the worst case scenario in mind. A distributed algorithm written for a specific application or a class of application is fine tuned to the properties of the application and can give a better performance when compared to the general purpose one. In this work, we propose two mechanisms in which we can optimize a general purpose algorithm - *Alg* based on the application - *App* using it.

In the first approach, we analyze the specification of *App* to identify patterns of communication in its communication topology. These properties are then used to customize the behavior of the underlying distributed algorithm *Alg*. To demonstrate this approach, we study applications specified as component based systems where application components communicate via events and distributed algorithms to enforce ordering requirements on these events. We show how our approach can be used to optimize event ordering algorithms based on communication patterns in the applications.

In the second approach, rather than analyzing the application specification, we assume that the developer provides application properties - I_{App} which are invariants for the optimization process. We assume that the algorithm is written and annotated in a format that is amenable to analysis. Our analysis algorithm then takes as input the application invariants and the annotated algorithm and looks for potential functions in the algorithm which are redundant in the context of the given application. In particular, we first look for function invocations in the algorithm whose post-conditions are already satisfied as a result of the application invariants. Each such invocation is considered as a potential redundant module. We further analyze the distributed algorithm to identify the impact of the removal

of a specific invocation on the rest of the algorithm. We describe an implementation of this approach and demonstrate the applicability using a distributed termination detection algorithm.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Dr Gurdip Singh, my advisor without whose help I would not have been able to complete this dissertation. He has provided me guidance in academia, research and teaching and has instilled in me confidence and skills that I use everyday. I hope to be able to continue to seek his advice throughout my career.

I would also like to thank my committe members Dr Pinner, Dr Robby, Dr DeLoach, Dr Hatcliff and Dr Das for thier time, patience and advice to make this work better.

It is hard to express in enough words the support my family has given me. Sanjeev, thank you for your patience and your pride in me. Thank you for the gallons of coffee you have made for me at the oddest hours. My little girl Meghna, you have been such a trooper.

Ma, thank you for your continued encouragement and moral support. My brother Sougat, thanks for the comic relief. Finally, I would like to acknowledge my biggest chear leader, my father. I can never do anything wrong in his eyes. Thank you Baba, for your (sometimes misplaced) confidence and support. I would not be here without you.

Dedication

To Ma, who was not given the option to complete her studies and to Meghna, may your work be better than mine.

Chapter 1

Introduction

A structure of a typical distributed system can be decomposed into three layers: application layer, middleware layer and the communication layer. The communication layer provides the basic services such as those for point-to-point or multicast communication. The middleware layer implements distributed algorithms to provide more enhanced services to the application layer. These may include services for addressing problems such as synchronization, mutual exclusion and replication. A typical application consists of multiple processes that may communicate with each other by sending and receiving messages and utilize the services provided by the middleware layer to achieve a specific task. With the increased deployment of communication infrastructure, a number of areas such as sensor networks and peer-to-peer computing are emerging in which distributed programming is the natural way to program systems. However putting together such systems is a non trivial task. Typically, each layer in such systems is programed by a different group of people, namely application developers, middleware developers and the network designers. Many development frameworks have been proposed[4] with the goal of isolating the application developers from the details of the lower layers. Middleware developers accomplish this by providing services with well-defined interfaces for the application layer. In order to increase the applicability, middleware developers often develop algorithms to be generic in nature so that they can be used by a large class of applications and will work on a wide variety of platforms. We can call such algorithms as 'general purpose' algorithms. However, when used in the context

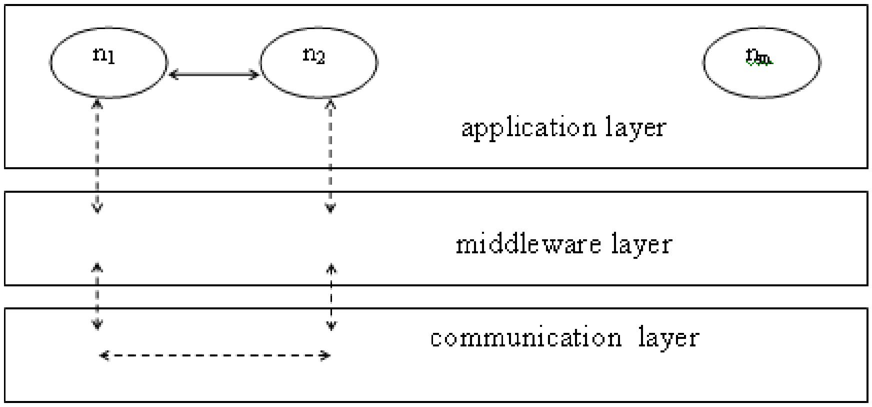


Figure 1.1: *Layers of interest in a distributed system.*

of a specific application, a general purpose algorithm may not perform efficiently and may even fail to meet stringent QoS requirements of real time distributed systems. For example, an application may not require a part of the service but the algorithm may perform all communication irrespective of the context in which it is being used. The extra communication overhead may deteriorate the performance and contribute to failure in meeting QoS requirements. The above problem often forces middleware algorithm developers to design and develop algorithms specifically for a particular application or a particular class of applications. They may tailor the algorithm to obtain an *optimized version of the service* by customizing it to work in the context of the application. Such algorithms may exhibit better performance and be able to meet QOS requirements. However, this customization can be a tedious and error-prone exercise.

The contributions of my work can be summarized as follows. Designer often have to manually customize distributed algorithm to suit the requirements of specific applications. We propose to develop methodologies and supporting tools to automate the process of customizing distributed algorithms. Furthermore, we show that the automatically customized algorithms exhibit better performance, and often similar to the ones which have been designed manually.

In the next section we will demonstrate with examples that certain parts of a ‘general purpose’ algorithm becomes redundant when used in a particular context. We can get a version of the ‘general purpose’ algorithm which works more efficiently in a system if we are able to identify and remove these redundancies. In chapters 2 and 3 we will show that we are able to identify the redundancies by first obtaining properties of the application and underlying network and then using these properties to remove redundant input data and actions taken by the ‘general purpose’ algorithm.

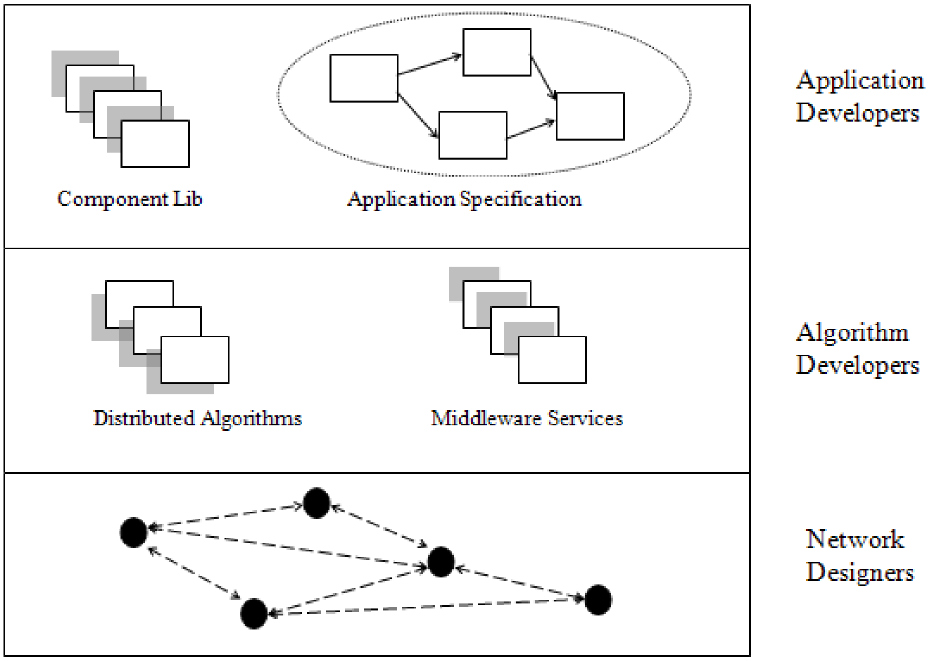


Figure 1.2: *A distributed computing framework*

1.1 Background and motivation

Figure 1.1 shows the structure of a distributed system. A distributed algorithm typically perform a set of tasks or subtasks on the request of the application, and accomplishing these tasks requires exchanging messages between the nodes in a system. The number of exchanged messages contribute substantially to the performance of an algorithm. We can get performance improvements if we are able to identify tasks or subtasks in a general purpose algorithm that do not apply to a particular system or are able to identify message exchanges within a task that are not required for the application.

> **Example 1** : Consider a system consisting of 5 nodes as shown in Fig 1.3 whose underlying network is fully connected. Assume that entities of an application running on this system access a shared resource R. To accomplish this, the system may use a mutual exclusion algorithm which can prevent multiple nodes from accessing the shared resource at the same time. In the classic request-response based mutual exclusion algorithm[23], the requests are ordered based on the time at which the requests are made. For this purpose, a request (and messages corresponding to the request) is marked with a time stamp indicating the time at which the request is made. The algorithm allows nodes to access R based on the timestamps - the node with an earlier request is allowed to access before others.
>
> In this algorithm, a node can have four states :
>
> *'requesting'*: The node is in the 'requesting' state when it wants to use the resource and is waiting for permission from the other nodes in the system.
>
> *'critical'*: A node is in the 'critical' state if it is using the resource.
>
> *'releasing'*: A node is in the 'releasing' state when it has finished using the resource and is sending out release messages to the other nodes.
>
> *'other'*: A node is in the 'other' state when it is in none of the previous states.
>
> The mutual exclusion algorithm works as follows. A node in the 'other' state wanting to use R sends out requests to all other nodes in the system and enters the 'requesting' state. Every node maintains a request queue that holds all the request messages it has received and the requests it has made. The queue is sorted according to the timestamps. When a node receives a request, it stores the request in its request queue and

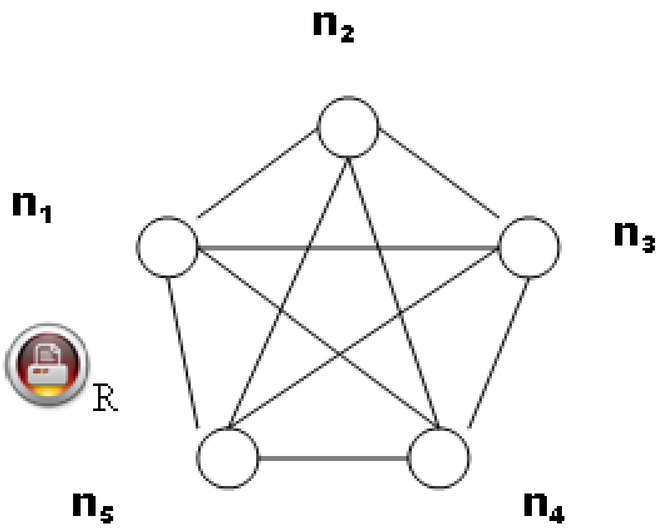
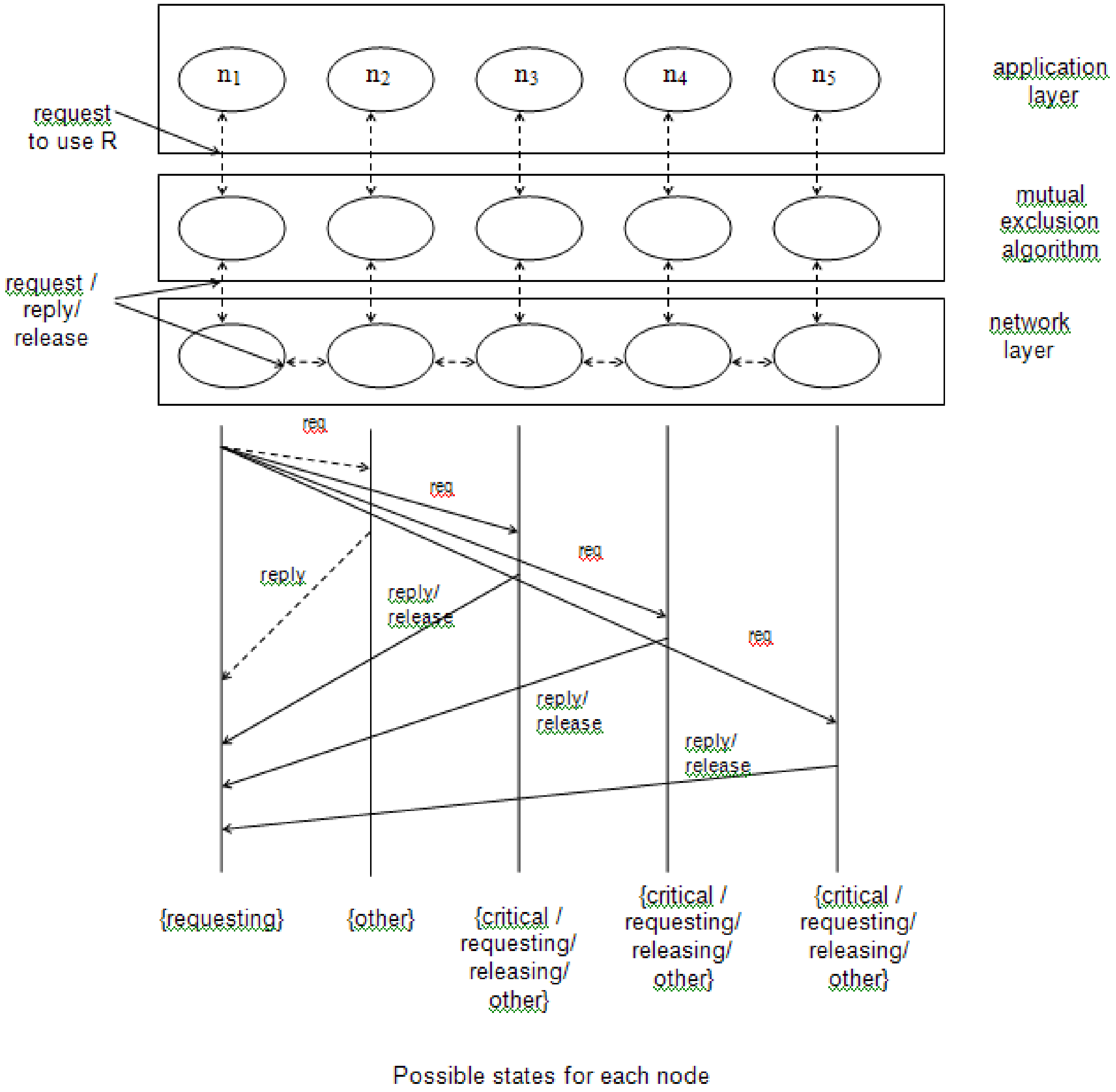


Figure 1.3: *Fully connected nodes*

sends a reply message if it is not using the resource, i.e. the node is not in the 'critical' state. When a node has finished using the resource, it sends out a release message to all nodes in the system. A node can use R i.e. enter the 'critical' state when (a) it has received either a reply or a release message from all other nodes in the system whose timestamp is greater than the timestamp of its request and (b) its request has the lowest timestamp in the queue. The mutual exclusion algorithm ensures that a node enters the 'critical' state when no other nodes is in the 'critical' or in the 'releasing' state. The messages being sent out by the algorithm at each node are essentially reporting the node's state ('requesting','critical','releasing' or 'other') to the other nodes in the system. This algorithm can be used with any application in which the participating nodes are making requests in any order. In certain applications, however, it may be the case that the requests for the resource R are made in a predefined or a predictable order. For example, it may be the case that node n_2 makes a request only after n_1 finishes using the resource and vice-versa. In such a scenario, when n_1



application layer
mutual exclusion algorithm
network layer
request to use R
request / reply/ release
n_1
n_2
n_3
n_4
n_5
req
req
req
req
reply
reply/ release
reply/ release
reply/ release
{requesting}
{other}
{critical / requesting/ releasing/ other}
{critical / requesting/ releasing/ other}
{critical / requesting/ releasing/ other}
Possible states for each node

Figure 1.4: *A scenario of message exchanges in the mutual exclusion algorithm*

wants to request permission to use R, we know that n_2 will be in the state ‘other’. In other words, when n_1 enters the ‘requesting’ state, the application is ensuring that n_2 is in the ‘other’ state. We can therefore say that *when n_1 enters the requesting state, it already has the knowledge of the state of n_2 from the application.* This knowledge makes the communication between n_1 and n_2 (where n_1 is seeking permission to use R from n_2) redundant and the subtask of determining the state of n_2 can be removed. If the nodes of an application make requests for R in a pre-determined order, then we can treat this behavior as a property of the application. Since the application is ensuring this property, we can treat it as an invariant, I_{App}, when analyzing algorithms in the middleware layer. The optimization process can use I_{App} to analyze the tasks and sub tasks of the algorithm Alg to eliminate redundancies.

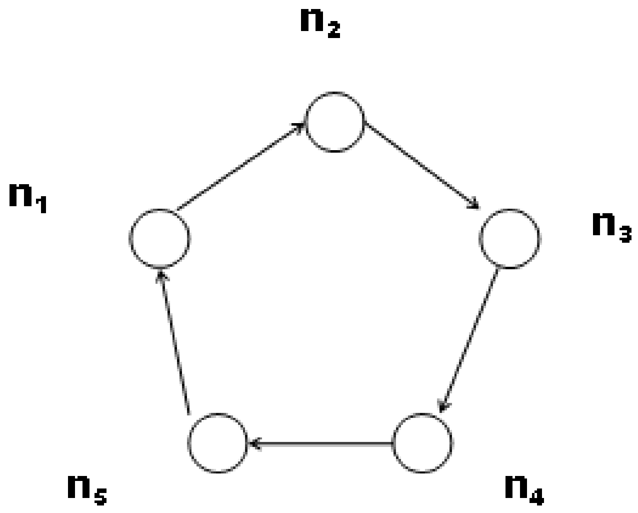
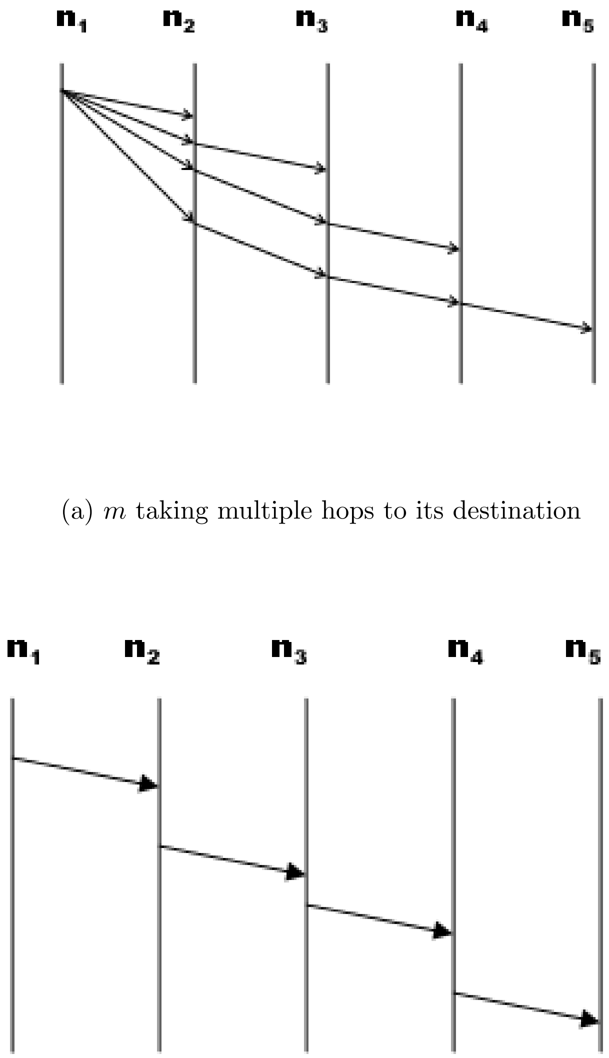


Figure 1.5: *Nodes connected in a unidirectional ring*

Example 2 : There may be properties other than those of the application that can be exploited to identify potential optimization opportunities. For example, the network topology of a system determines the actual number of messages sent by an algorithm. If the network is a fully connected such as in our example system, then every message has to travel only one hop. For other topologies where a node may not have a direct link to another node in the system, messages may have to travel multiple hops before reaching their destinations.

Consider the modification of the the network topology of our example system of 5 nodes such that the underlying network now connects all the nodes in a uni-directional ring(Figure 1.5). Assume that in a distributed algorithm, a node has to send out the same message m to all other nodes. Figure 1.6(a) shows the scenario if node n_1 is sending out a message to all other nodes in the system. Because of the way the nodes are connected to each other, when n_1 sends m to n_3, the message is first send to n_2 and then forwarded to n_3. The algorithm however has also sent a copy of the same message to n_2. So instead sending a message to n_3 which has to make two hops to reach its destination, we can forward n_2's message to n_3. Thus, we can modify the algorithm so that every node forwards the message to its neighbor[Fig 1.6(b)] instead of sending out multiple copies of the same messages.



(a) m taking multiple hops to its destination

(b) m being forwarded by the nodes

Figure 1.6: *Broadcast of a message m*

From the above two examples, we see that we can reduce the messages exchanged in a general purpose algorithms if we can identify and remove redundancies. This allows the possibility of the modified algorithm executing more efficiently with better performance.

The modifications in the above examples are determined by either the application behavior or the network topology. We will refer to the properties of the application and the network as 'system properties' in the rest of this text. Some more examples of system properties that can help identify redundancies in our example system are:

(1) The application in node n_i never requests to use R. In such a case, any communication between n_i and the other nodes of the system is redundant because the application property implies that the state of n_i is always 'others'.

(2) The requests to resource R by all nodes in the system is made in the following order : n_1, n_2, n_3, n_4, n_5. This sequence is repeated by the nodes. In this case node n_i can use a resource when it receives a release message from n_{i-1} without any additional message exchange.

(3) The underlying network has the property that the time taken by a message to reach its destination has an upper limit. In such cases, the absence of a message can indicate the state of the sending node.

The system properties can exist in two levels:

- *Single system level* where the property holds for a specific system only.
- *System class level* where the property holds for a group of systems. For example, in sensor networks, a node can snoop on messages on the network that are destined for its neighboring nodes.

1.2 Related work

To design distributed algorithms with better performance, traditionally different versions of algorithms for the same problem have been proposed. For example, in[5] Bapat and Arora proposed an optimized version of the termination detection algorithm for sensor networks. Termination is an important property for this class of applications which run in phases. Having synchronized phases is required when phases may not be backward compatible or when the next phase may depend on the completion of the previous one. In termination detection algorithms, nodes are assumed to exhibit reactive behavior. The state of a node turns from idle to active when it receives an application message. Although, in general networks, a node may receive application messages at any time,[5] exploits the following properties of sensor networks:

- A node remains active only if it periodically receives an application message every T time units. If it does not receive any message within the time period of T, then it goes back to the idle state. If an idle node does not receive a message within time T, it goes into the terminated or the passive state.
- In a sensor network, a node is able to snoop for messages intended for its neighboring nodes which are within its reception range.

The algorithm first identifies 'reporter' nodes. A node becomes a reporter if has not heard from any other reporter nodes. The reporters nodes then snoop for messages over broadcast channel and detect local termination. Global termination is declared when all reporters have reported local terminations. The algorithm uses network localization data to determine whether all reporters have reported their terminated status.

This algorithm is a tailored version which gives better performance on sensor networks as compared to a general purpose termination detection algorithm. The optimized algorithm, however, works only for a specific class of systems and has been re-written with the above properties in mind. Re-writing distributed algorithms is a time consuming and error prone effort and the resulting algorithm only works for a specific system or at the most for a class of systems who share the properties based on which the optimized algorithm was written. An automated solution to this problem will provide a considerable ease in terms of effort and exposure to errors. Any such automated process should be able to take as input system properties and make changes to a general purpose algorithm. To be able to make a change, the algorithm itself must be written in a way which will allow it to be customized. Therefore, the automated process should include a grammar for writing general purpose algorithms amenable to modifications. A solution along similar lines was proposed in[22] by V.

Kolesnikov where they tackle the optimization problem by introducing a framework to first define algorithms which can be modified. These algorithms can then be optimized statically or dynamically. They use programming abstracts when writing distributed algorithms which are customizable and expose the communication related design knowledge of the algorithm. The programming abstracts are based on the concept of interaction sets. When writing a general purpose distributed algorithm, an algorithm developer defines interaction sets at various points in the algorithm where communication happens with other nodes. However, in specific applications, all processes may not need to be a part of an interaction. Using InDiGo[14], a designer can restrict the membership of the interaction sets based on application properties.[22] also defines membership criteria for each interaction set and rules to make dynamic updates to them. Rules for dynamic updates are written to make use of information gathered as a result of message passing between processes. Given a application *App* and an algorithm Alg, the analyzer statically determines the initial membership of all interaction sets in Alg with respect to *App*. It can further constraint the membership of each set at run time using the dynamic update rules. The framework also uses knowledge of the network topology to remove redundant messages when the same message is sent to multiple processes.

Our approach to solve the problem specified in section 1.1 has similarities with techniques for partial evaluation. A partial evaluator performs a mixture of execution and code generation actions. It produces a specialized program when a part of its input is known[12,13]. It takes the set of known inputs $in1$ and tries to optimize a program $p1$ with respect to $in1$ to produce p_{in1}. The evaluator performs $p1$'s calculation which depend on the input $in1$, which may result in elimination of some code segments (*e.g.*, if the condition of the if-statement will always evaluate to true given $in1$, then the else part of the if-statement can be eliminated). The resulting code only depends on unavailable input $in2$. The optimized program p_{in1} should produce the same result as $p1$ would when given the rest of $p1$'s input $in2$.

Partial evaluation can be done with online and offline specialization. Offline specialization begins with binding-time analysis which provides annotations to the specializer which instructs it to evaluate expressions, unfold functions, generate residual code and generate residual function calls. Online specialization in turn computes program parts as early as possible and takes decisions ‘on the fly’ using available information[3,7].

Our approach can be seen similar to offline specialization. A generic middleware algorithm Alg can work with any application App on any network topology N. Intuitively, we can think of App and N as inputs to Alg since Alg is written to work with any application and on any network. We show how to perform optimizations if parts of App and the network topology N is known. We provide a grammar to specify N and the properties I_{App} which describe the application App.

We see a generic Alg as performing a set of tasks and subtasks. The optimizations we target is to reduce the number of tasks and subtasks within the context of App and N. We see a function call as a unit of work and its post condition as the result the function is trying to achieve. We make optimizations by removing function calls whose post conditions are deemed redundant with respect to I_{App}.

1.3 Our approach

To optimize a distributed algorithm based on application properties we need to obtain: (1) the properties of the application and (2) make the algorithms amenable to changes. Application properties can either be obtained from its specification or the application developers can write invariants to describe the application behavior. To be able to optimize a distributed algorithms based on these properties, an algorithm can either be written in a way that it can be changed or it should be able to take the application properties as input.

We have approached the problem of optimizing a distributed algorithm based on application properties in two ways(Figure 1.7). In the first approach (Chapter 2), we analyze the application specification and look for patterns of communication in its communication topology. These properties are then used to customize the behavior of the underlying distributed algorithm. To demonstrate this approach, we study applications in anonymous component based systems in which application components communicate via events and underlying distributed algorithms to enforce ordering requirements on these events. An application may rely on the middleware algorithms to order the delivery of events. For example, if an event $e2$ published by component A is caused by an event $e1$ published by B, then the application may require that $e1$ be delivered before $e2$ at all components which subscribe to both $e1$ and $e2$. Several algorithms have been in proposed in the literature to implement such ordering constraints. While these algorithms are efficient in general, they may be conservative in nature and may not exploit application-specific properties. Our approach to customization involves the following steps. As shown under Method 1 in Figure 1.7, we first transform an existing algorithm for ordering events to obtain a modified algorithm which takes a set of tables as input and is able to customize its behavior based on this input. We assume that an application is specified in the integrated development environment Cadena[11]. Our next step shown in Figure 1.7 is to perform the analysis step. In this step, from the specification of the application, we show how to build an intermediate data structure called a Port Topology Graph (PTG) which captures the communication topology of the application. We then analyze the port topology graph to identify patterns called 'causal cycles'. By identifying these patterns, we are able to capture ordering already enforced by the application. From the PTG, we then generate tables which are taken as input by the ordering algorithm. Using these tables, we show that the ordering algorithms can then reduce the amount of dependency information which needs to be propagated along with the events to ensure proper ordering at the receiving components. This reduces the size of the messages considerably and hence results in more efficient execution. We also show that properties of the delivery mechanism used by the event service can introduce an order in which events are delivered. We show ways to use this information to reduce messages exchanged by ordering algorithms.

In the second approach (Chapter 3), rather than analyzing the application specification, we assume that the developer provides application properties which are invariants for the optimization process. Furthermore, rather than manually transforming the algorithm as in Method 1, we assume that the algorithm is written and annotated in a format that is amenable to analysis. The main part of Method 2 is the Analysis step shown in Figure 1.7. In this step, our analysis algorithm then takes as input the application invariants and the annotated algorithm and looks for redundancies between the application and the algorithm. In particular, we first look for function invocations in the algorithm whose post-conditions are already satisfied as a result of the application invariants. Each such invocation is considered as a potential redundant module. We further analyze the distributed algorithm to identify the impact of the removal of a specific invocation on the rest of the algorithm. Based on this analysis, we determine whether or not the invocation can be removed. If the invocation can be removed, we also identify the code transformation that is needed to perform this removal. We study a general purpose termination detection algorithm designed for a bidirectional ring topology to demonstrate this approach. We consider the use of this algorithm for an application which only involves unidirectional communication. We show that the algorithm can be automatically optimized to eliminate redundant messages by taking advantage of the application invariant which specifies unidirectional communication. The optimized algorithm performs its functions by exchanging less number of messages as compared to the original algorithm.

In chapter 4 we show techniques to optimize an algorithm based on the network topology. We identify some basic patterns of message exchanges. When an algorithm sends and receives messages in these patterns over a specific network topology, we are able to reduce the number of messages sent out by the algorithm.

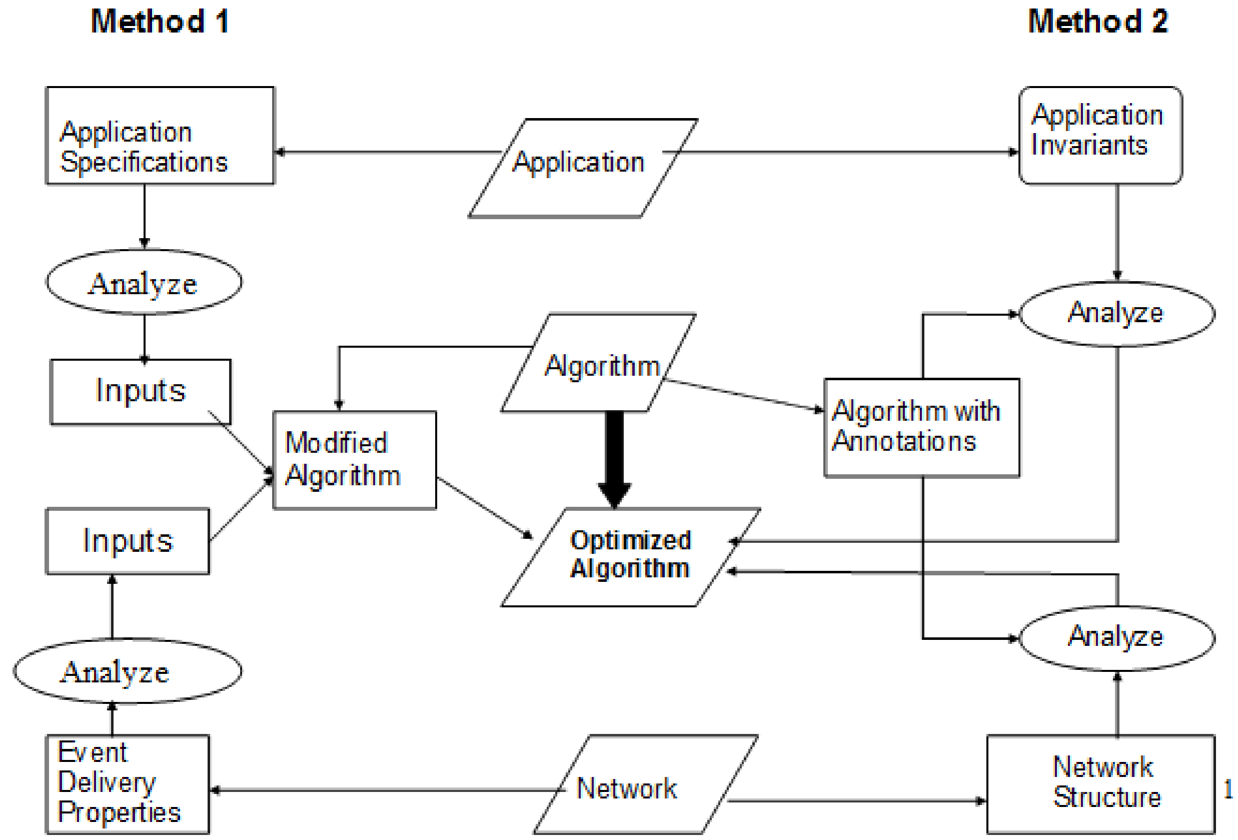


Figure 1.7: *Two different approaches to optimize distributed algorithms based on application properties*

Chapter 2

Customizing Event Ordering Middleware

Event service middleware, shown in Figure 2.1, has been used extensively for communication in component based systems. In these systems, the components communicate anonymously with each other via events – that is, the sender is not aware of who the receiver it and vice-versa. A component may register with the event service as a producer of an event or as a consumer of an event. A producer component publishes or pushes events to the middleware service and a consumer component consumes or pulls events from the middleware service. When a producer publishes an event, the middleware service notifies all consumers subscribed to that event (Figure 2.1).

A number of tools that have been developed to aid in the development and deployment of component based systems[9,11] use event service as an underlying middleware service. Cadena[11] is one such tool with an integrated development environment. This tool allows designers to specify components with ports. A port may be a source port on which events are published, or a sink port where events are consumed. Applications are defined by instantiating component instances and specifying connections between these ports of the component instances. Cadena also provides capabilities to analyze, generate code and deploy these systems.

Applications defined in such component based systems often have different ordering

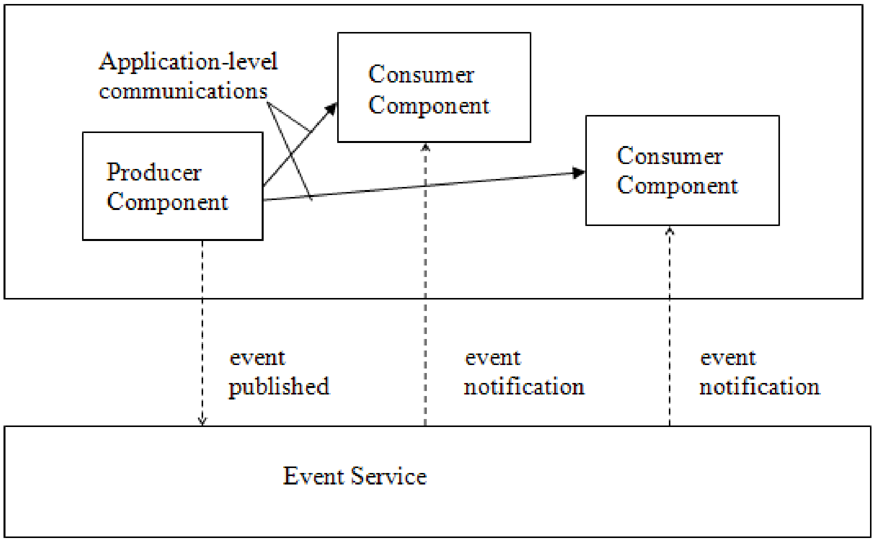


Figure 2.1: *Event Service Middleware*

requirements such as FIFO, causal ordering and total ordering. For example, consider the case where a component $C1$ produces event $e1$ which is consumed by both $C2$ and $C3$. Further, assume that after the consumption of $e1$, $C2$ produces an event $e2$ which is also consumed by $C3$. In a generic event service, it may be the case that $e2$ may be delivered at $C3$ before $e1$ even though $e1$ caused the occurrence of $e2$ (for instance, if $e2$ is a "reply" to a "question" raised in $e1$, then $C3$ will be delivered the reply before the question). The causal ordering requirement eliminates this possibility by requiring events to be delivered in an order consistent with causal ordering - that is, if e causes e' then e must be delivered before e'. Similarly, total ordering requires that events belonging to a specific category be delivered in the same order to all consumers consuming the same set of events.

Many different algorithms for ordering have been proposed in the literature[16,18]. These algorithms are general purpose can be used by any system and make no assumptions regarding the application or the target platform. 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. If such application behavior is known, then the performance can be optimized. For example, it may be known that e_2 is always produced in response to event e_1, we can reduce or eliminate some work or information exchanged by the ordering algorithms.

Similarly, the ordering algorithms in general make very weak assumptions about the underlying event delivery mechanisms. For example, typically the time for an event to be delivered is assumed to be variable. Also the algorithms take into consideration all possible inter leavings between concurrently produced events. However we may have additional information about the way the events are actually delivered from one component to another in a particular system. For example if the components are co-located in a processor, it is possible that the event delivery is a sequence of synchronized method calls. In such cases we can make stronger assumptions on the number of possible inter leavings and identify a smaller set of possible inter leavings.

2.1 The optimization approach

The following section describes the three main layers of the system which are involved in the optimization process as identified in Figure 1.1.

The application

Our approach starts with the designer specifying the application, *App*, in Cadena. As mentioned before, a component based distributed system can be defined in Cadena using a set of components with ports. Events are published and consumed on the ports. In Cadena, a designer can specify the communication between components by connecting the ports on which events are published to the ports on which they are consumed. These ‘inter-component’ connections specify the communication topology of the system from which we can derive event dependency information.

From the specification of the components in Cadena, we can also extract 'intra_component' dependencies. For example, a component always publishes event e_2 on its outgoing port after receiving event e_1 on its incoming port. We provide analysis algorithms which make all these information available in form of a 'Port Topology Graph(PTG)'.

The algorithm

Let Alg represent a general algorithm providing event ordering services to the application. The following are the typical properties of Alg

- An ordering algorithm typically works by propagating dependency information along with events. To maximize re-usability, such algorithms do not made any assumptions about the application and therefore, they propagates all dependency information irrespective of whether it may get used or not.
- To target a large number of deployment platforms, ordering algorithm do not make any specific assumptions about the mechanism of message delivery.

Although an algorithm having the above characteristics can be used with multiple systems, it may not provide the best performance when compared to an ordering algorithm written specifically for a system. For example, Alg may propagate dependency information which never gets used or the underlying platform may be implemented in a way that results in a smaller set of possible sequences in which the events can be interleaved and delivered.

We aim to modify the algorithm so that we can use application properties to reduce the dependency information flowing in the system. Similarly, we want to take advantage of the properties of the platform which reduce the interleaving of events due to its delivery mechanism.

To illustrate our approach, we use two existing algorithms, $Causal\_IDR$ and $Total\_Sequencer$, for causal ordering and total ordering respectively. We provide techniques to generate optimized versions of $Causal\_IDR$ and $Total\_Sequence$ with respect to an application. For each case, we present an analysis algorithm to determine the information used by the algorithms which is redundant with respect to the given application and provide techniques to optimize the algorithm by eliminating the redundancy.

The platform

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[21]. There are two mechanisms for event delivery in this system : direct dispatching and full channel dispatching. We show that the properties of these delivery mechanisms can be also be used to optimize the ordering algorithms.

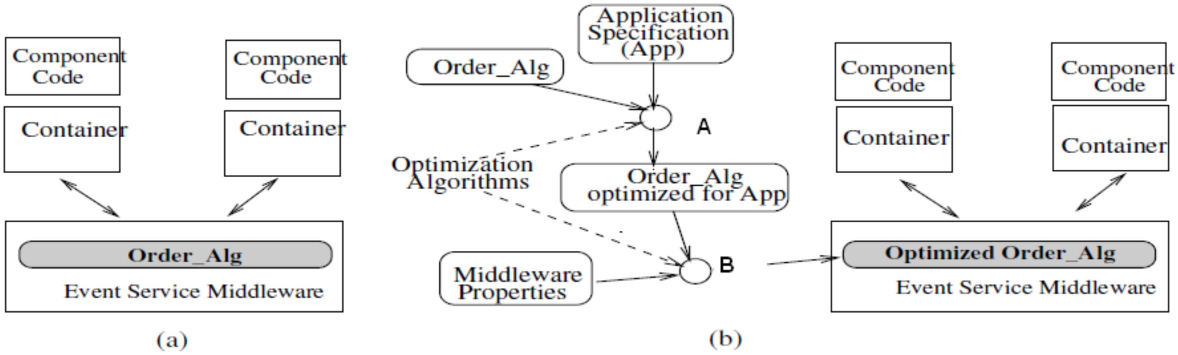


Figure 2.2: *Model driven approach*

Figure 2.2 shows our approach to optimizing the middle layer which is the algorithm layer using the properties of the upper layer or application layer and the lower or the network layer. The process starts with the specification of the system in Cadena using components and ports. We provide analysis tools to extract event communication patterns and the over all communication topology from the system specification. This information is then stored in a Port Topology Graph which is used as an input for further analysis. We look for specific patterns in the PTG and use them to optimize *Alg*. After this, at point (A) in Figure 1.7, *Alg* will be optimized with respect to *App*. We can now use take the properties of the underlying event delivery system and further optimize *Alg*. Therefore at point (B), *Alg* is optimized with respect with *App* and the event service.

2.2 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. We will use as an example of a simple avionics system shown in Figure 2.3 to illustrate the development process in Cadena.

• **Component Design**: The first step in the specification of components in the CCM model. In Figure 2.4, we give the CCM IDL specified by the designer to define the component type `BMLazyActive` for the `AirFrame` component instance in Figure 2.3. 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 2.4, `inDataAvailable1` is the name of an event sink of type `DataAvailable`, and `outDataAvailable` is the name of a event source of type `DataAvailable`.

• **Scenario Specifications**: 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 2.4 (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

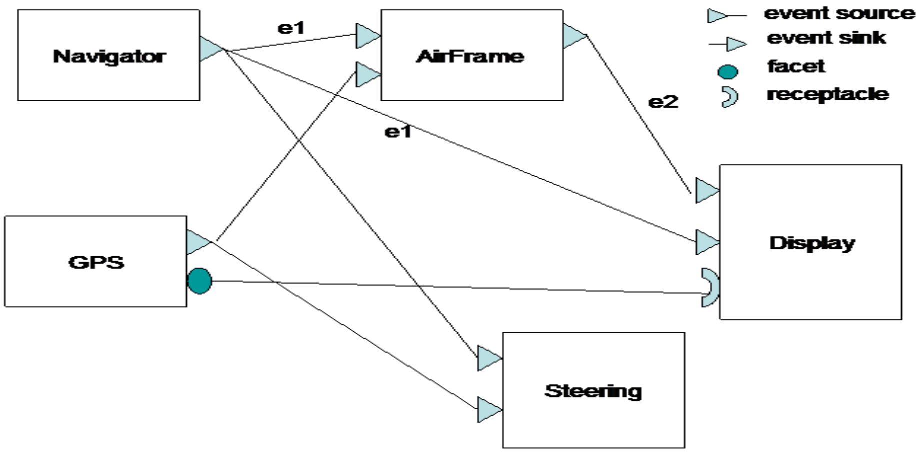


Figure 2.3: *A simple avionics system*

`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 2.4 connects the outDataAvailable port of GPS to the inDataAvailable2 port of AirFrame.

- **Code and Configuration Metadata Generation**: The next phase is the generation of

```
\#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)

```
system ModalSPScenario {
 import cadena.common, cadena.modalsp;

 Rates 1, 5, 20;        // Hz rate groups
 Locations l1, l2, l3;  // 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)

Figure 2.4: *(a) CCM/Cadena artifacts, (b) Cadena Assembly Description for ModalSP (excerpts)*

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.

• **Deployment**: The final step is the deployment phase in which the application is deployed on a target platform. Prior to deployment, a number of parameters may have to be specified. One such parameter is the *location* attribute, which maps the component to a physical processor. In early modeling stages, this attribute is a logical value but it has to be mapped to a concrete value before deployment. The second important attribute is *rate*; each port has a *rate* associated with it which specifies the rate at which events are consumed or published at that port. The *rate* attribute is subsequently used to assign threads at appropriate priorities to execute the event handlers (discussed below). 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.

2.2.1 Event Service Middleware

An event service is a middleware service which brokers communication between producers and consumers[6,10,15]. 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. The architecture of *Adaptive*

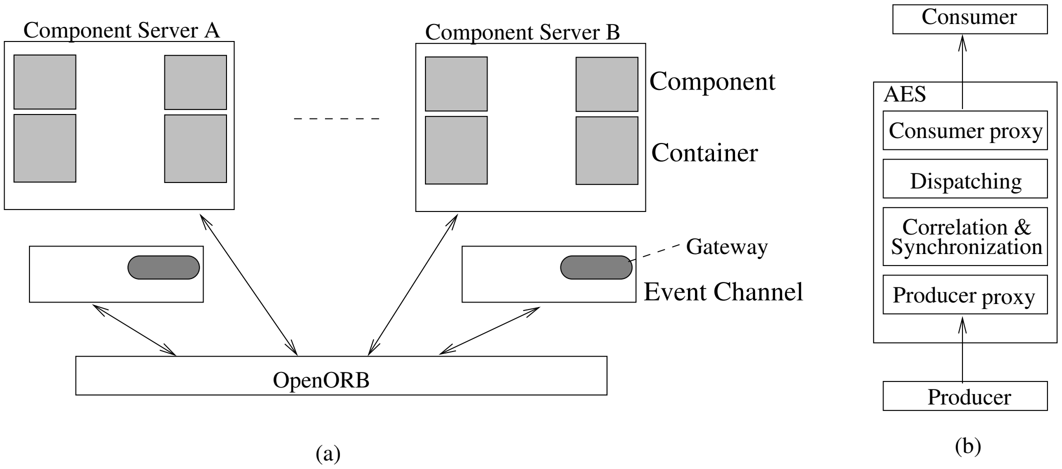


Figure 2.5: *CCM code architecture*

Event Service (AES), a Corba-based Java event service is shown in Figure 2.5. 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. 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 2.5(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 $publish(C)$ and $consume(C)$ 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 *publish* and *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 $publish(c)$ and as a consumer for events in the set $consume(c)$.

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

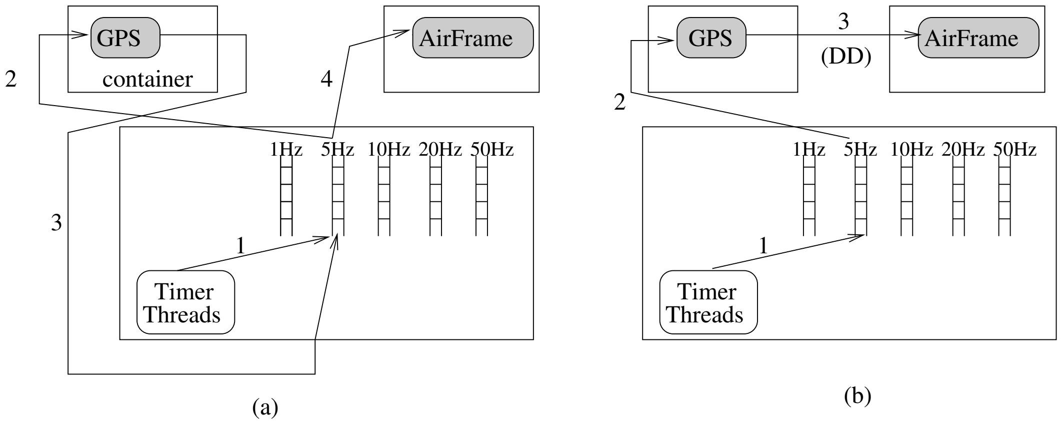


Figure 2.6: *Full Channel and Direct Dispatching*

description of the threading architecture of the event channel, which is based on the Bold Stroke architecture[21]. 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 2.6(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 2.6(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 $Queue_5$.

• **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 co located 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 2.6(b) where the event from GPS to AirFrame is direct-dispatched).

2.3 Event order specification

There are a number of events and ports associated with the publishing and consuming of an event in a system described above . Below is a list of the some of those which are of interest to us and the notations used in this text to refer to them.

For an event e:

1. *e.src* : This is used to denote the port that published e.

2. *e.pub* : This is used to denote the event of e being published.

3. *e.deliver(p)*: This is used to denote the event of e being delivered on port p.

4. $< p, q >$: This is used to denote the connection from an output port p to an input port q.

5. Let → denote the *happens before* relation defined in[17].

A tele-teaching application: Let us consider a simple tele-teaching application in which enables a single instructor and a group of students to hold a question-answer session. The instructor as well as each student is subscribed to the events published by all others in the group. The instructor initiates the question-answer session by first publishing the question. The students can then respond to the question by publishing their own answers.

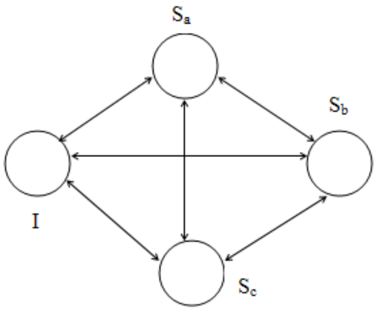


Figure 2.7: *Teaching application*

Using the above notations and the tele-teaching application we specify and explain the ordering requirements below.

- The **FIFO(p,q)** requirement is as follows: Let $< p, q >$ be an event connection. For all events e_i and e_j published on port p, if $e_i.pub \rightarrow e_j.pub$ then $e_i.deliver(q) \rightarrow e_j.deliver(q)$. Informally, this requirement asserts that all events published which travel over a connection $< p, q >$ are delivered at q in the order in which they were published at p. In our teaching application this requirement ensures that if a student S_a publishes two responses to a question, then the responses will be delivered in the order in which they were sent out.

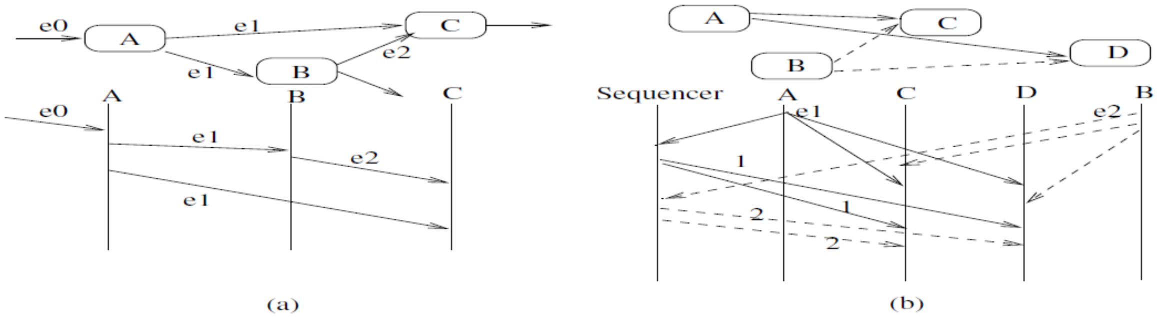
- The **causal ordering requirement** $Causal(p_1, ..., p_x)$ is as follows: Let $(p_1, ..., p_x)$ be a set of input ports. For any events $e1$ and $e2$ received on ports p_i and p_j respectively, where $1 < i, j < x$, if $e1.pub \rightarrow e2.pub$ then $e1.deliver(p_i) \rightarrow e2.deliver(p_j)$.

Informally, this property requires that events which are causally related to each other are delivered in the order in which they occur. Let us consider the following situation in our teaching application. The instructor publishes a question q_i and student S_a immediately responds to it by publishing the answer a_i. The causal ordering requirement ensures that all members of the group will first receive the question q_i and then the answer a_i since these two events are causally related to each other. No member of the group will see the answer (the effect) before the question (the cause). Causal delivery allows concurrent messages to be delivered in any order and the order may be different in different components. For example if two students in the group send out answers a_i and a_j in response to the question q_i, other members in the group may receive a_i and a_j in any order. As another example, in Figure 2.3, $Navigator$ sends data in an event $e1$ to $AirFrame$, which in turn updates its own data and sends an event $e2$ to $display$. If $e2$ is received by $Display$ before $e1$ (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.

- The **total ordering requirement,** $Total(p_1,,,,p_n)$, is as follows: For any two events $e1$ and $e2$ received on all ports $p_1,,,,p_n$, if $e1$.deliver(p_i) $\rightarrow$ $e2$.deliver(p_i) then $e1$.deliver(p_j) $\rightarrow$ $e2$.deliver(p_j) $\forall 1 < i, j < x$. Informally, total ordering requires that the common set of events received on ports $p_1,,,,p_x$ be delivered in the same order on all ports. Total ordering may be required in cases where consistency is required across components. In our teaching example total ordering will ensure that all members of the group will see events in the exact same order. For example if two students in the group publish answers a_i and a_j in response to the question q_i, every member of the group will receive a_i and a_j in the same order.

2.4 Ordering Algorithms

In the following, we describe existing ordering algorithms that we have used in our work.



(a)

(b)

Figure 2.8: *Ordering Algorithms*

Causal-Order Algorithm Causal ordering is implemented by propagating dependency information in the events[16,18]. In particular,[18] proposed an algorithm, which we will refer to as $Causal\_IDR$ in this text, is based on computing the immediate dependency relation or IDR. Every event e carries with it a set $IDR(e)$ which contains all the events it depends on. When e is received by a component, it is buffered and its delivery is delayed till all events in $IDR(e)$ have been delivered. We will illustrate this using the scenario in Figure 2.8(a). Assume that event e_1 is published in response to the event e_0, and e_2 is published on the reception of e_1. In this case, IDR(e_1) will contain e_0. Similarly, IDR(e_2) will contain e_1. Note that IDR(e_2) does not contain e_0 since the event e_0 is not an immediate predecessor. Since IDR(e_1) will contain e_0, the event e_0 will have been delivered before e_1 and therefore before e_2. However, for a system with N components the size of the IDR set in the worst case will be $O(N^2)$ which will include the information about the destination set of each event and the set of known concurrent events from other sites.

Total Ordering Algorithm Total ordering can be implemented using a central site to put a timestamp on each event in the system. This algorithm, which we will

> refer to as *Total_Sequencer* works as follows: each event to be ordered is sent to all components, including the central site which is called the sequencer. The sequencer site assigns timestamps in a linear order to all events it receives, and the sends the timestamp of each event to all components. At each component the events are then delivered in the order of the time-stamps. For example, in Figure 2.8(b), both A and B send events $e1$ and $e2$ to C and D respectively. Since $e1$ reaches the sequencer site first, it is assigned timestamp 1 whereas $e2$ is assigned timestamp 2. Thus, even though $e2$ reaches C before $e1$, it is delayed until $e1$ is delivered.

As seen in the above description, both *Causal_IDR* and *Total_Sequencer* do not make any assumptions about the order in which events are produced by the application. The algorithm *Causal_IDR* captures dependency information on the fly. Information such as the destination sets for events are essentially application properties which are being propagated by the algorithm. It may be the case that an application produces events in a pre-determined order. In such cases in the causal ordering algorithm some of the dependency information being propagated can be obtained statically by analyzing the application structure at compile time rather than gathering them during run time. Similarly for the total order algorithm the sequencer might be doing redundant work if some events are always send in a specific order.

2.5 Deriving application properties

To perform optimizations, we need to provide information regarding the application against which the optimizations can be performed. We derive these properties from the application specification in Cadena (shaded in figure 2.9). We then use these properties to create inputs for the modified version of the causal order algorithm as described in section 2.6.1.

In the following, we describe a series of topological structures representing an application, each of which refines the previous one with more fine grained dependency information.

Component Topology Graph (CTG) : The graphical interface of Cadena shows the

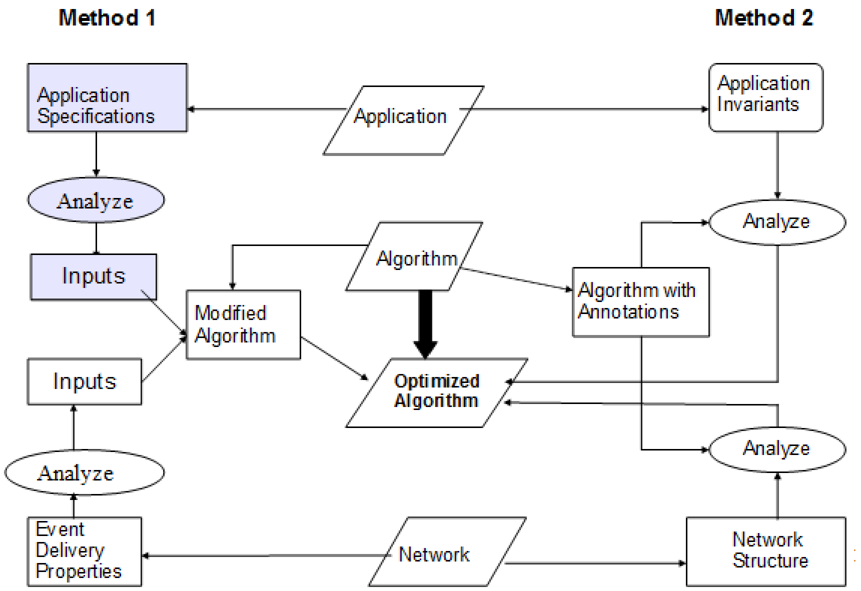


Figure 2.9: *Optimization based on application properties*

component topology graph $CTG = (V, E)$, where each vertex in V denotes a component. An edge from $v1$ to $v2$ in exists in E if there is an event connection from an event source port of component $v1$ to an event sink port of component $v2$.

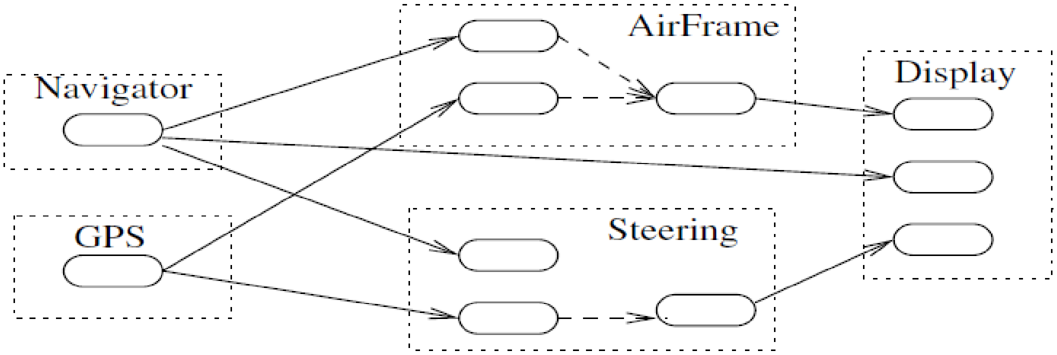


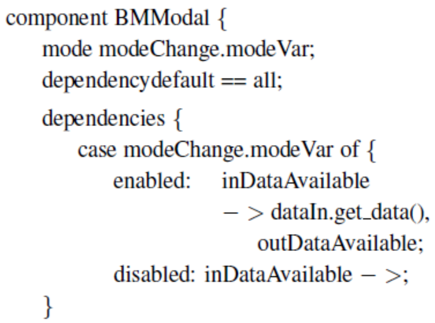
Figure 2.10: *Port Topology Graph*

A **port topology graph (PTG)** of an application contains more information and is defined as a graph (V, E), where each vertex in V is an event source or a sink port of a component. There are two types of edges in a PTG. An edge $(v1, v2)$ exists in a PTG if :

(a) $v1$ is a source port of a component and $v2$ is a sink port of another component and $v1$ is connected to $v2$. This type of an edge is called an inter-component edge
(b) $v1$ and $v2$ are sink and source ports of the same component respectively, and the receipt of an event on $v1$ can cause an event to be published on $v2$. This type of an edge is called an intra-component edge.

The intra-component edges are further classified into two types, deterministic and non-deterministic. An edge $(v1, v2)$ in component C is deterministic if in all executions, whenever an event is received on $v1$, C publishes an event on $v2$ without awaiting the occurrence of any other event. Otherwise, the edge is labeled non-deterministic. An example of a PTG is given in Figure 2.10. 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.

2.5.1 Obtaining a PTG from a CTG



```
component BMModal {
    mode modeChange.modeVar;
    dependencydefault == all;
    dependencies {
        case modeChange.modeVar of {
            enabled:    inDataAvailable
                        − > dataIn.get_data(),
                            outDataAvailable;
            disabled: inDataAvailable − >;
    }
```

Figure 2.11: *Sample CPS file*

The Cadena infrastructure stores application information is intermediate representations which can be analyzed for dependency information. One such representation is the abstract syntax tree (AST) which holds information regarding the components and their interconnections. Our analysis algorithm traverses the AST and gathers information for all $inter - component$ edges. $Intra - component$ edges are obtained from the component property specification (CPS) file. Each component has a CPS file which specifies dependencies between ports and the behavior of event handlers. A fragment of a CPS file is shown in Figure 2.11. The case statement in the CPS file specifies how an incoming event is processed. For example, the case statement in Figure 2.11 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.

Our process derives this dependency information from the CPS files. The main subroutine used by the program is analyseCaseStatement(p), where p is a portname, 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.

2.6 Optimizing the algorithms using the PTG

In this section we present techniques to optimize the causal ordering and the total ordering algorithms. To be able to optimize these algorithms we must first modify the algorithms so that they can be customized at initialization time.

2.6.1 Causal Ordering Algorithm

As discussed earlier, the causal ordering algorithm in[18] works by generating and propagating dependency information of events(that is, information about its causal predecessors) so that an event can be delayed at a consumer until all of its predecessor events have been received. We modify $Causal\_IDR$ to $Causal\_IDR\_Optimized$ (CIO) which takes two tables as inputs, (a) generation rule table , (b) propagation rule table. These tables are defined for each component and are used to compute and propagate dependency information.

Generation rule table: This table determines when new information must be added to the IDR sets. Let q and r be an input and an output port of component B (Figure

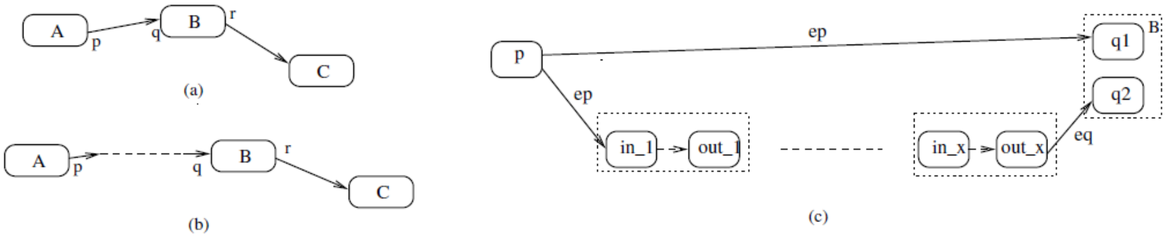


Figure 2.12: *Causal Cycle*

2.12(a)). Let p be an output port in component A. An entry $< p, q, r >$ in the generation rule table means that events published on the connection $< p, q >$ must be included in the IDR set of events published at port r

Propagation rule table : An element $< p, q, r >$ in the propagation rule table of component B (see Figure 2.12(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.

The above tables are provided to each component at initialization time. When an event is published by a component, these tables are consulted to determine the dependency information which needs to be propagated.

The entries to these tables are made from the analysis of the PTG. The main pattern we identify in a PTG is the *causal cycle*. 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 $q1$ and $q2$ such that p is connected to $q1$ and there is a directed path from p to $q2$ (Figure 2.12(c)). We also refer to this as the causal cycle from p to $(q1, q2)$.

Let $Causal(q_1, , , q_x)$ be an ordering requirement for a component B. For each pair (q_i, q_j), we use a variation of the depth first traversal to obtain the set of all causal cycles in the PTG from all ports to (q_i, q_j). Let us assume that there is a causal cycle from port p of component A to (q_i, q_j). Let ep be an event published at port p, and $p, in_1, out_1, , , , , in_x, out_x, q2$ be the path from p to $q2$ as shown in Figure and eq be the last event in the path which is consumed at port $q2$. 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 $1 < j < x$, to the propagation rule table of in_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. This 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 Fig 2.12(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). Essentially, this can be viewed as attempting to acquire knowledge of the potential causal cycles at runtime. 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.

2.6.2 Total Order Algorithm

As discussed in section 2.3, the total ordering requirement is specified as a set of predicates of the form $Total(p_1, , , , p_x)$. 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 $p_1, , , , p_x$. 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 $< source(tp), \Rightarrow >$ 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 $p_i \Rightarrow p_j$,

(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. We will now use a small example to illustrate the concepts in TSO. Let $source(tp) =< p1, p2, p3 >$, and assume that $p1$ triggers p2. Thus, after each event publication on $p1$, an event is subsequently published on $p2$, however, events on $p3$ can be issued concurrently with those from $p1$ and $p2$. 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 events from the root port 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, $p1, p2$ and $p3$. After an event $e1$ from $p1$ is received by a component A, we know that an event, $e2$, from $p2$ will be published. In this case, after delivering $e1$, each component waits for $e2$ to arrive before delivering any other event (thus, $e2$ is scheduled for delivery immediately after its root event). Hence, $e2$ 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 px and py in $source(tp), px \Rightarrow$

py if py is reachable from px 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 $p_j = py$ or pj 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 py must be reachable from px via a path that does not involve any other port in tp whereas Condition (Tr2) ensures that no port other than px can also cause py 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 our tele-teaching application), the components may issue events at arbitrary times, and very few ports may be related by the triggers relation. However, in 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.

2.7 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 (figure 2.13).

In the BoldStroke design process, condition $DD(p, q)$ is used to tag connections as either Direct Dispatch(DD) or Full Channel Dispatch(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

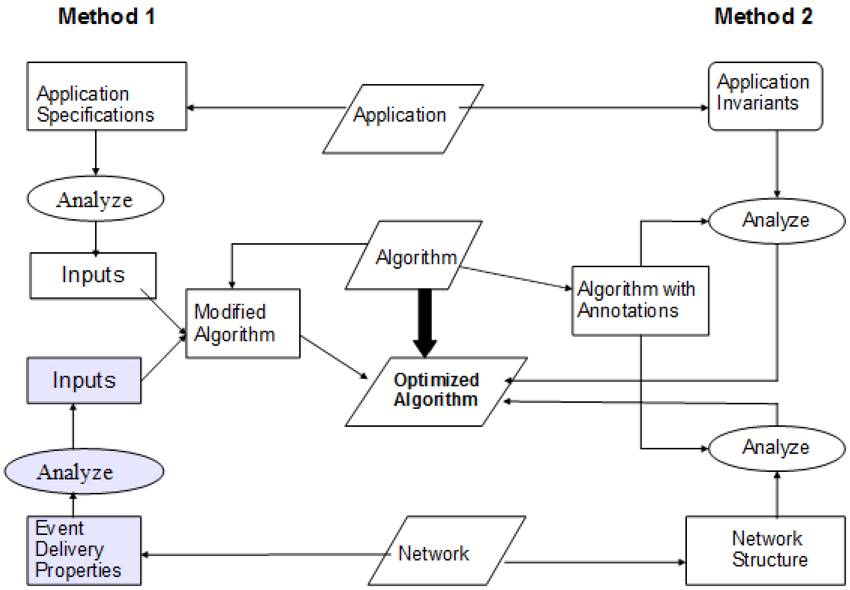


Figure 2.13: *Optimizing based on communication mechanism*

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 inter leavings do not occur. For example, consider the case where p, $q1$ and $q2$ in Figure 2.12 belong to the same rate group. If e_1 and e_2 are both dispatched via DD, then only the following inter leavings are possible:

$$e_1.deliver(C); e_1.deliver(B); e_2.deliver(C)$$

$$e_1.deliver(B); e_2.deliver(C); e_1.deliver(C)$$

On the other hand, if e_2 is FCD and e_1 is DD then the possible inter leavings are:

$$e_1.deliver(C); e_1.deliver(B); e_2.deliver(C)$$

$$e_1.deliver(B); e_1.deliver(C); e_2.deliver(C)$$

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 co located on the same server and all ports have the same rate associated with them. Our optimizations rely on the following property of DD:

Property 1: Let $e1$ and $e2$ be two events published on ports $p1$ and $p2$ respectively, $S1$ and $S2$ be a set of input ports on which they are delivered respectively, and $group(p1, p2, S1, S2)$ holds. In an execution, let the first delivery event among those delivery of $e1$ and $e2$ on $S1$ and $S2$ respectively correspond to $e1$. If (a) $e1$ and $e2$ are DD and $e1$ does not cause $e2$ (that is, $\neg(e2 \rightarrow e1)$ or (b) $e1$ causes $e2$, $e1$ is DD and $e2$ is FCD, then all delivery events for $S1$ will appear before all delivery events for $S2$ in that execution.

This property follows from the fact that once a thread starts dispatching for an event, it will not be interrupted by dispatching of another independent event belonging to the same rate group. If delivery of $e1$, say on port q causes event $e2$ to be published, the thread for this rate group will proceed with delivery of $e2$ if $e2$ is also DD. This possibility, however, is eliminated by the condition required in Property 1.

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) $\langle ep, q1\rangle$ is DD, at least one connection $\langle out_{y-1}, in_y\rangle$ in the path from p to $q2$ is FCD, and

$group(p, q1, in_1, out_1, \ldots, in_y)$ holds.

Proof: We need to prove that in all executions, ep will be delivered before eq. We know that event ep causes the event ey on the connection $\langle out_{y-1}, in_y\rangle$. Since $\langle out_{y-1}, in_y\rangle$ is FCD, from Property 1, we know that all events on the connections $\langle p, q1\rangle$, $\langle p, in_1\rangle$, $\langle out_1, in_2\rangle, \ldots, \langle out_{y-2}, in_{y-1}\rangle$ will be delivered prior to ey. Since ey is dispatched before eq, we know that ep will be delivered before eq.

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 $group(p, q1, in_1, out_1, \ldots, in_y)$ 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 (p, in_1, out_1) from the *generation_rule* table of in_1's component and (ep, in_j, out_j), where $1 \leq j \leq x$, from the *propagation_rule* table of in_j's component.

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

Lemma 2: $Total(p_1, \ldots, p_x)$ holds if all the following are true:

(a) All events from ports in $source(tp)$ are direct-dispatched,

(b) All ports in $source(tp)$ are the same rate group, and

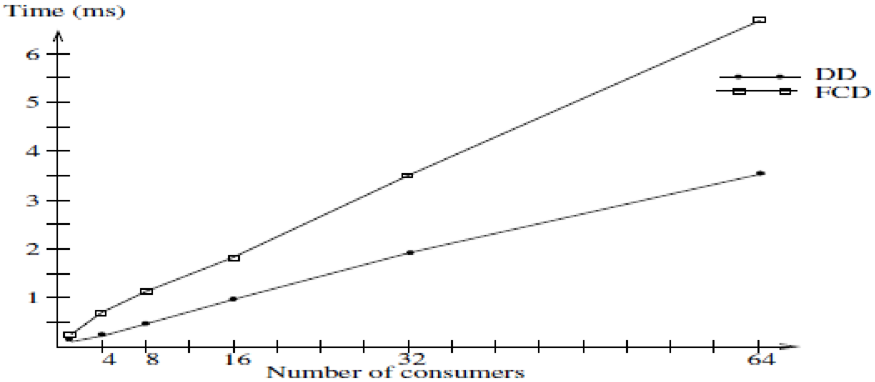
(c) For all ports tp_i in $source(tp)$, there does not exist a path from any port p_j to tp_i in the PTG labeled with only direct-dispatched edges.

Proof: Let $tp1$ and $tp2$ be two ports in $source(tp)$, $ep1$ and $ep2$ be two events published on these ports respectively, We must show that $ep1$ and $ep2$ are delivered in the same order on all ports $p_1, \ldots, p_x$. In an execution, assume that the first delivery event of $ep1$ is delivered before $ep2$ on one of these ports. From condition (c), we know that if $ep1$ causes $ep2$ then there must exist a path from a port p_j to $tp2$ such that at least one connection on this path is FCD. Then, from Property 1, we know that the delivery of all events for $ep1$ will appear from the event on this FCD connection is delivered. Since this FCD event will occur from $ep2$, delivery of all events in $ep1$ will appear before $ep2$.

The conditions in Lemma 2 ensure that each event published on a port in $source(tp)$ 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 $source(tp)$, then we tag the connections from ports in $source(tp)$ 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.

2.8 Evaluation



The performance benifit obtained are as follows. The first is the reduction in the amount of dependency information propagated and the latency reduction due to tagging of events as DD instead of FCD. We have identified several instances of such reductions in dependency information and latency by using variations of the applications from the Bold Stroke system. We expect this to result in performance improvements as DD is more efficient than FCD (see Figure 2.8). Second, we expect a reduced latency due to the decrease in the number of messages being sent to the sequencer site in TSO. To evaluate this, we have performed experimentation by implementing TSO on an simulation platform. The application used consists of five components, A,...,E, where A and B send events e_a and e_b respectively to all components. When e_a is received, each of C, D and E, in turn, publish a response event $resp_{a,c}$, $resp_{a,d}$, and $resp_{a,e}$ respectively. Thus, event e_a triggers $resp_{a,c}$, $resp_{a,d}$ and $resp_{a,e}$. Similarly, when e_b occurs, the events $resp_{b,c}$, $resp_{b,d}$, and $resp_{b,e}$ are published. The following table gives the results of the experiment wherein both A and B send 5 events each. The time reported in the table is the average time taken (in milliseconds) until all of the events are delivered. Rows 1 and 2 are average time for requesting components (A and B) whereas Rows 3 and 4 are for the responding components (C, D and E). We varied the number of responding components (1 Responder assumes only C is present, whereas 2 Responders assumes C and D, and so on). As the number of responding components is increased, we observe an increase in the number of total time taken; however, the increase is more for Total Sequencer as compared to TSO.

Time(ms)	1 Responder	2 Responder	3 Responder
Total_Sequencer(requesters)	37772	57893	77706
TSO(requesters)	16289	17357	20187
Total_Sequencer(responders)	38075	58030	78036
TSO(responders)	20340	20372	20505

Table 2.1: *Total Sequencer vs TSO*

The approach of optimization specified in this chapter obtains the application properties from the application specification. We then use these properties to either reduce the message size as we demonstrated with CIO or eliminate messages as we demonstrated with TSO. In this method the identification of the properties is specific to the algorithm being optimized. This technique is best suited for middleware algorithms which tag information on to application messages. In the next chapter we show a more general methodology for optimization.

Chapter 3

Automated optimization of Distributed Algorithms based on Application Properties

In the previous chapter, we presented a process to optimize ordering algorithms that are used in an event service middleware. To be able to get better performance from $Causal\_IDR$ and $Total\_Sequencer$, we had to first create their 'optimized' versions, $Causal\_IDR\_Opt$ and $Total\_Sequencer\_Opt$ respectively. These versions of the algorithms were able to take as input the information gathered from the analysis of the port topology graph. For $Causal\_IDR\_Opt$, the inputs were in form of the generation and propagation tables, and for $Total\_Sequencer\_Opt$ the input was partitioned sets of events. Using the approach described in the previous chapter, we see that a general purpose algorithm has to be modified so that it can take the application properties as an input. In this chapter, we propose an alternative way in which an algorithm writer can write the general purpose algorithms in a way that they can be optimized automatically. In this approach, instead of analyzing the application specification, the application developers provide invariants which describe the application behavior and properties.

In our framework, an algorithm Alg is specified as a composition of a set of algorithm flows $F^1, \ldots, F^m$, where each flow accomplishes a specific subtask. A flow F can be viewed as set of communicating processes, $F_1, \ldots, F_n$, executing at different nodes. A process F_i is written as a set of functions and message handlers. We require the algorithm developer to specify a proof outline for F. A proof outline involves specifying a pre- and a post-condition for each function and message handler in F_i. In giving the proof, the designer may make use of *interface variables*. *Interface variables* are used across the application-algorithm layers and are used to express application layer properties. The designer must ensure that the proof outline is correct and that the proof outlines of the different flows are non-interfering (which is discussed in more detail later).

The optimization engine optimizes the algorithm with respect to a specific application. In particular, we leverage at invariant properties of an application running on the top layer. For example, in an application, a node may never make a request for a shared resource [section 1.1] under certain conditions. We can treat such properties as invariants. Let *I_App* denote an application invariant involving the interface variables. The optimization engine uses the proof outline of each flow to eliminate functions calls whose post-conditions are ensured due to *I_App*. This involves a partial analysis of each flow to ensure that the removal of a function is safe (that is, the removal does not invalidate any other assertion or causes a deadlock in a flow).

We divide the framework into three parts. The first part describes how algorithm writers can write distributed algorithms which can be optimized automatically. The second part involves application developers and network designers to come up with properties of their respective layers and the third is an optimizer that takes input from the first two parts and produces an optimized version of the distributed algorithm.

3.1 Example

To illustrate the different steps and aspects of the framework we introduce an example system. The system has five nodes and is connected in a ring topology. Each node in the system has two neighbors and has a bi-directional connection to both (Fig 3.3).

The system uses an algorithm for the termination detection problem. In this problem,

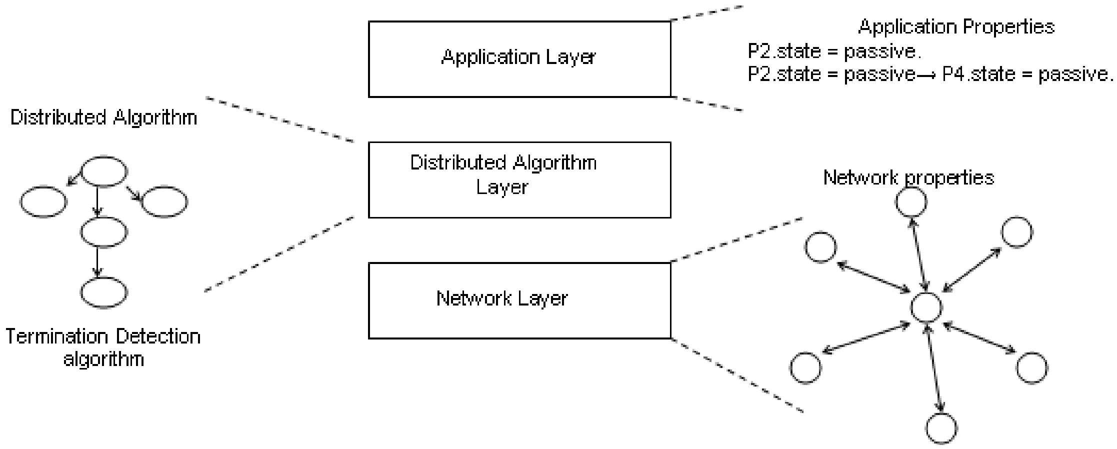


Figure 3.2: *Layers*

a node in the application can be in the ‘active’ or in the ‘passive’ state. When a node is performing a computational task, it is in the active state. A ‘passive’ state, on the other hand, is not performing any computation, but it can become ‘active’ on receiving a message from an active node. Furthermore, it can then send out messages as part of its computation which may activate other nodes. In many systems, it is required to detect termination of a computation - when all nodes have finished their computational tasks and no further computational activity will take place. A computation has terminated when all nodes are passive and all channels are empty(ensuing that no node will become active again). It is typically required in systems using diffusion computations or phase based systems where a computational phase can only begin when the previous one has ended.

The algorithm for termination detection must detect whether the system has reached a state in which all nodes are passive and all messages channels are empty. We describe a termination detection algorithm below that detects termination in a bidirectional ring. Let the node initiating this algorithm be referred to as the Initiator node(I1). In this algorithm, the initiator node associate each initiation of the termination algorithm with a sequence number. All messages carry this sequence number and a node drops any message with an older sequence number. We assume that the sequence numbers are bounded. The algorithm

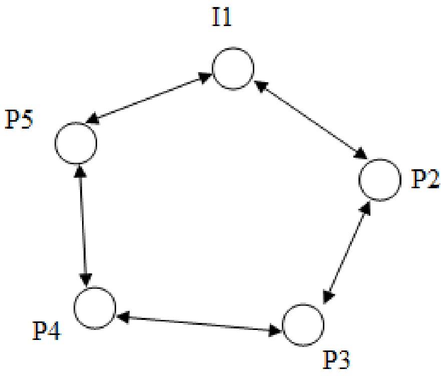


Figure 3.3: *Example System*

proceeds in two phases.

Phase I :

When the Initiator node (I1) becomes passive, it begins the first phase by sending a marker message to its neighbor P2 (its neighbor in the clockwise direction). The marker message contain the sequence number associated with this initiation. When a node in the ring receives a marker message in the first phase, it takes the following actions:

If the node is passive, it:

- Saves the timestamp when it received the first marker message.
- Saves the sequence number of the message.
- Forward the marker message to its other neighbor (its neighbor in the clockwise direction).

If the node is active:

- The node waits until it becomes passive and then forwards the marker.

Thus, the marker message proceeds in the clockwise direction around the ring. When the initiator node receives the marker message back from P5, it checks whether it is currently passive and has remained passive since initiating phase I,; if so, it proceeds to the second phase.

Phase II:

In this phase, the initiator sends out a marker message with the same sequence number as the marker message it sent out in the first phase but in the anticlockwise direction to P5. When a node in the ring receives the second marker message, it checks if the message contains the same sequence number as the last marker message it saw. If the message has an older sequence number, then the message is dropped. If the sequence numbers match, it checks if it is passive and has remained passive since the last time it saw a marker message. If this condition is true then it forwards the marker message in the anti-clockwise direction. Otherwise, it does not take any action.

In the second phase, if the marker message traverses the entire ring and the initiator node receives the message back from P2, it checks the following: If the initiator is currently passive and has remained passive since it initiated phase one then it declares termination. Otherwise, the system may not have terminated. If the marker message is not received, the initiator node will time out. In either case, it can initiate the first phase of the algorithm again with a fresh sequence number.

3.2 Optimization Framework

In the following sections of this chapter, we will describe how to write the termination detection algorithm described above and its proof (represented by the shaded rectangle in Figure 3.5). We will also describe how to write application invariants which specify the properties of the application executing on a network (represented by the shaded rectangle in Figure 3.8). We will then describe the optimization process in detail (represented by the shaded oval in figure 3.9).

The optimization process first makes an intermediate data structure called a 'call graph' for the distributed algorithm being analyzed. This data structure exposes the calling structure of the algorithm. Using the call graph, the optimization process then compares the application invariant to the post condition of each function call. If the invariant implies the post condition of a function call, the optimizer then considers this function call to be redundant and removable. It is then removed if its removal will not cause a deadlock in the system. By removing a function call, we save on two fronts :(a) We save on execution time of the function call and any other function it calls and (b) We potentially reduce the number of messages being sent out by the algorithm by eliminating the messages sent out by the removed function and any other function it calls.

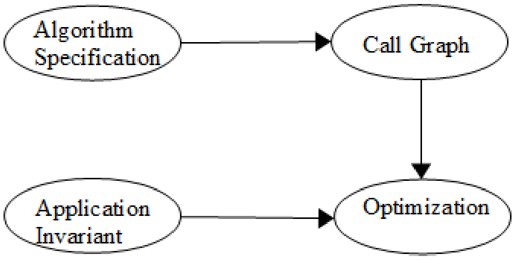


Figure 3.4: *Optimization Framework*

3.2.1 Algorithm Specification

Algorithm Definition

In this section we describe how to specify the middleware algorithm (Figure 3.5).

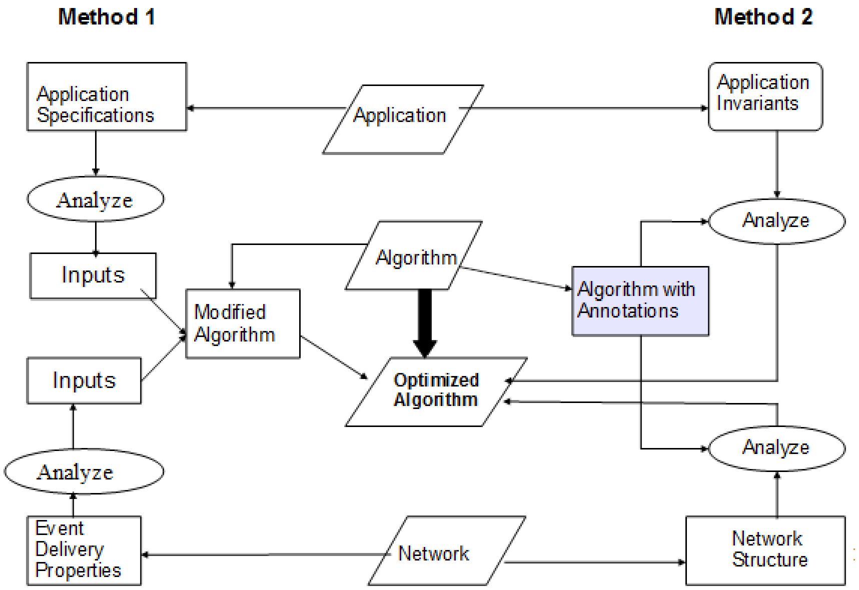


Figure 3.5: *Algorithm Specification*

Let system S consist of N nodes, $S = \{ n_i, \mathit{0} \leq i \leq \mathit{N\text{-}1} \}$. The algorithm layer at any node n_i in S is triggered by one of the following :
(1) By the application: This is done by the application layer entity invoking an interface method provided by the algorithm at node n_i . This may result in the algorithm performing a local action and/or sending messages to other nodes in the system. The other nodes may in turn send out more messages to gather information or evaluate a condition requested by n_i.
(2) By the receipt of a message from another node which was either triggered by the application or by the receipt of a message.

Many distributed algorithms such as global state determining algorithms or diffusing computation algorithms can be naturally decomposed into a set of flows, each of which is initiated by the application process. As a simple example, consider a set of ten nodes connected in a linear topology with nodes numbered $1, \ldots, 10$ (node 1 is on the left end). Assume that each node i has a local variable x_i and the goal is to compute the sum of all local variables. For node i to accomplish this, we can design an algorithm wherein it sends a message *initiate* to its right neighbor which then forwarded all the way to the right end. When node 10 receives this message, it sends an $ack(x_{10})$ message to node 9. On receiving this message, node 9 adds x_9 to the value in the ack message and propagates it to the left. In this manner, the accumulated sum is propagated until it reaches node i. After receiving this message, it sends a message to its left neighbor to compute the accumulated sum of all variables belonging to nodes on its left hand side. In this problem, it is possible for more than one node to initiate the algorithm concurrently. To accommodate for multiple initiations, one can formulate the following rule: When node i receive an initiate message from its left neighbor j, and i has already initiated the algorithm and sent a message to its right neighbor, then node i waits for the ack message to be received from its right neighbor before responding to j. When it receives the ack message, it sends the accumulated sum to j. A similar rule is applied when receiving an initiator node receives a message from its right neighbor.

Assume that two nodes, say node 3 and 7, initiate the algorithm concurrently. Each initiation can be considered as a separate flow. Each flow is independent of the other. The flow corresponding to 3's initiation can complete on its own and does not depend on $7'$ initiation (that is, it would have completed on its own if 7 had not initiated the flow). The only interaction between the flows is that one flow can benefit from the work done by another flow. In this case, 3 can benefit from the fact that 7 has already send a message to compute the accumulated sum of its right hand side. As we will see later, with such a structure, optimizations in one flow do not result in a deadlock in the other. For instance, assume that from the application invariant, node 9 knows that the value of x_{10} is always 0. In this case, we can optimize the flow initiated by 7 so that in this flow, x_9 does not need to send a message to 10. This optimization, however, does not impact flow 3. That is, when 7 receives the *initiate* from 6 corresponding to the flow initiated by 3, it will return the accumulated sum to 6. Therefore, viewing an algorithm as a set of flows drastically reduces the portion of the state space which needs to be analyzed when optimizing a flow.

We assume that a middleware algorithm $\textsc{Alg}$ is a composition of a set of flows $(F^1, \ldots, F^n)$, where F^i is initiated by application at site $Init(F_i)$. Each flow F^i is a composition of a set of processes $F^i_1, \ldots, F^i_n$, executing at different sites. Multiple process types belonging to different flows running at the same site may share a set of global variables, $\textsc{Alg}.global$. Thus, $\textsc{Alg}$ $= < Comp(F^1, \ldots, F^n)$, $\textsc{Alg}.global >$. We assume that the composition operator is local in nature. Thus, the algorithm at site i, $\textsc{Alg}_i = < Comp(F^1_i, F^2_i, \ldots, F^n_i), \textsc{Alg}.global_i >$. Each flow may be instantiated multiple times. However, each initiation must complete before the next instance is started.

We assume that each flow in an algorithm is deadlock free and completes when executed in isolation (separate from other flows). This implies that a flow is never waiting on any updates made to global variables by other flows in the system. We also assume that the algorithm *Alg* obtained by composing the flows is also deadlock-free. This implies that one

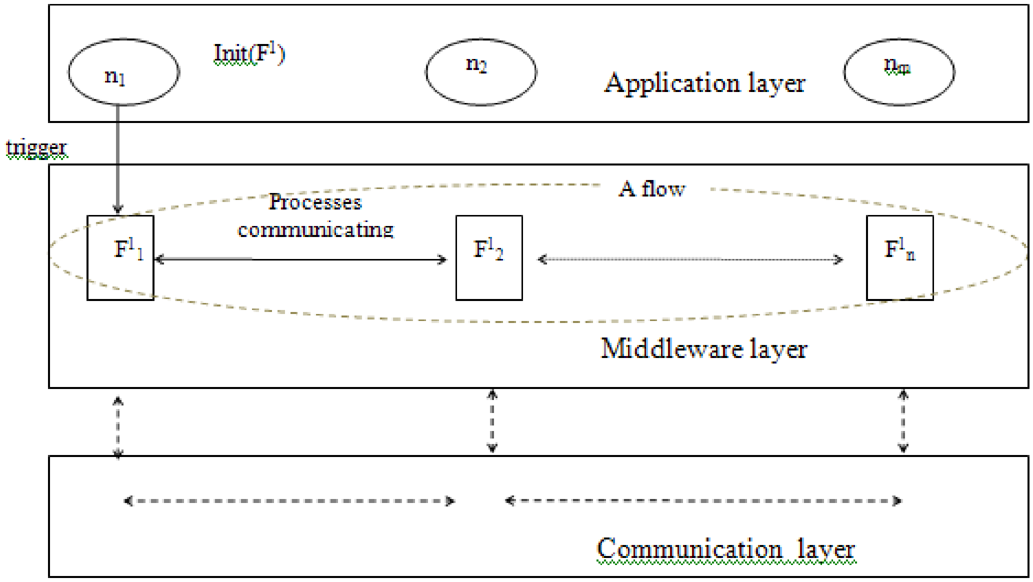


Figure 3.6: *A flow*

flow does not update global variables in a manner that creates a deadlock in another flow.

3.2.2 Specification of a flow

The code running in the middleware layer at different nodes of a system may be different. For example, in a system using a termination detection algorithm, some nodes may play the initiator role by executing the initiator code while other nodes would be responding to these messages by running the responder code. We call each such code a *process type*. A set of such process type instances makes a flow F. A flow F^i is defined as the parallel composition of $F_1^1 \mid\mid F_2^1 \ldots \mid\mid F_n^1$, where F_j^i indicates a process of F^i running on site j. Each flow F^i has a unique initiator process type $Init(F^i)$. $Init(F^i)$ is triggered by the application while the process at other nodes are triggered by the receipt of messages.

The specification of a flow F^i consists of the specifications of the process type for each site. Each process type is implemented as a module which may have message handlers, internal functions, and a set of global variables to which are shared with modules of the other process types at the same site. Each module is required to have a *primary message handler* with the signature `receive(Message m,int processName)`. All incoming messages are first processed by the primary message handler. A module can have one or more *secondary message handlers* written to process specific message types. Therefore, the process type specification for F_j^i comprises of a primary message handler, global variables, secondary message handlers and internal functions and is denoted by $\{mh_{primary}$, $global\_variables$,set of $mh_{secondary}$, set of functions $fn\}$

An algorithm containing multiple process types is declared as follows :

```
algorithm TermDect {

// Variable declarations
GlobalVariable_declaration;

// Message declarations
Message_declaration;

// Process declaration
thread ProcessType_name (Outgoing Channels, Incoming Channels)
{
LocalVariable_declaration;

// Primary message handler
receive(Msg m)

// Secondary message handlers
([trig_])$mType_x$_handler(Message $msg_{mType_x}$)
...
([trig_])$mType_y$_handler(Message $msg_{mType_y}$)

// Functions
$function_1(..)$
...
$function_n(..)$
}
```

The structure of each function $function_i$ is as follows:

```
pre_condition;
return_type (trig_)functionName(parameter_declaration) {
statement_block;
} post_condition;
```

Variable types There are two kinds of variables allowed in the algorithm: node_level global variables and local variables. Local variables are local to a function or a message handlers, whereas global variables can be accessed by all functions and message handlers at a site (note that they are still local to a site and not shared across sites).

Message Handlers

The primary message handler, which processes all incoming messages, is allowed to read but not write to any global variable of the algorithm. It may contain book-keeping code related to the processing of a message only. For example, it can have code to compare sequence numbers or to determine whether a message is old and needs to be discarded.

We will use the signature *mtype_handler* to denote secondary message handler for message type *mtype*. It is executed after the primary message handler has processed the message corresponding to its type.

Functions A function is written as a block of statements. The block of statements may contain the following: assignment, if-then-else, while, for, wait, function call and send statements. We do not allow recursive function calls. All statements except the function call and wait statements are allowed in the primary message handler.

A function can wait for a message to arrive using the wait statement. The wait statement is written with a wait condition which is a boolean expression on global variables, `wait(boolean cond)`. When the execution reaches a wait statement, it waits until the wait condition in the argument becomes true. For each wait condition *cond*, if a variable appearing in *cond* is updated in a message handler *msg_handler* then we assume that there exists a signal statement to indicate possible change in the truth value of *cond*. We assume that the execution of a process type can not be interrupted by other process types running on the same site. The execution is only switched when a executing process type encounters a wait statement.

The algorithm developer also needs to identify a subset of functions called trigger functions in the algorithm. A `trigger function` is the first function to be executed when a process type starts executing. Trigger functions are indicated by adding the *trig_* prefix to a function name or a secondary message handler name. Although the primary message handler is the first function executed when an process type is triggered by a message, we mark the secondary message handler as the trigger function (if the receipt of the message is the trigger for the process type) as it has been specifically written for its type.

The separation of message handling from the functionality of the algorithm helps us compose, analyze and optimize the functionality of an algorithm without affecting the processing of incoming messages. For example, assume that a process P is waiting for a message m via a wait condition. The wait condition may become true either on the arrival of a message or may become true due to optimizations and composition of P with other processes at the node P is running. If the later happens, then m will still have to be consumed by the algorithm's message handlers in spite of the fact that P is no longer waiting for it. The separation of message handling from the wait statement enables us to receive messages irrespective of whether or not they will make a wait condition true.

Execution of a flow

An event in a distributed system is defined as an action which changes the state of the system. Following are some examples of types of events:

- method call
- method return
- send statement
- assignment/ condition evaluation

Let EventType be a set of event types containing the following :

- method call or a message handler being triggered
- method return or message handler return
- send statement
- wait statement
- assignment statement

An execution $Ex = e_1, e_2, \ldots$ of a flow F^i is a sequence of events which can occur starting from the initial state. A flow consists of a set of histories, where a history is defined as $\{Ex, \longrightarrow\}$, where $\longrightarrow$ is a partial order defined as follow. Let *Ex.Event* be a set of events, e which occur in an execution *Ex* such that $e \in$ EventType. We define a relation *Event* $\longrightarrow$ *Event* as follows: $e_1 \longrightarrow e_2$, such that :

- e_1 and e_2 are method calls to functions $fn_1()$ and $fn_2()$ respectively and the function call to $fn_2()$ is made within the body of $fn_1()$

- e_1 is a function call to $fn_1()$,e_2 is a send statement and e_2 is executed within the body of $fn_1()$

- e_1 is a function call to $fn_1()$,e_2 is an assignment statement and e_2 is executed within the body of $fn_1()$

- e_1 is a function call to $fn_1()$ and e_2 is the return of the same function

- e_1 is a send statement and e_2 is the invocation of a message handler at the receiving process.

- e_1 is a signal, e_2 is a wait statement and e_1 ends the wait for e_2.

- e_1 and e_2 are events within the same function or message handler body and e_1 occurs before e_2.

Each linearization of an history h of F^i is an execution of F^i.

The relation $\longrightarrow$ is transitive and the transitive closure of $\{e, \longrightarrow\}$ where $e \in Event$ will be the set of all events which are directly or indirectly triggered by e. Therefore if e does not happen then none of the elements in the transitive closure of $\{e, \longrightarrow\}$ happens.

Let fn be the function call in the process p. Let e_i be the event of the function call. Let e_r be the event of the return statement of fn. We define causal(e_i) to be the transitive closure of e_i where any event e in causal(e_i) which is not in e_r, $\longrightarrow$. The causal set has the following property:

Property 1: Given an execution Ex of a flow F: e_1,..,e_i,..,causal(e_i),e_x,..,e_n of Alg, if we move the events in causal(e_i) after e_x then the resulting execution is also a valid execution of F. This means that the event e_x is not implicitly waiting for an event in causal(e_i) to happen and cannot be a 'wait' statement waiting for a event in causal(e_i).

Example: Figure 3.7 gives a termination detection algorithm written according to the rules of our framework. This algorithm corresponds to the description given in section 3.1. There are two process types in the algorithm, one for the initiator and the other for the responders. The initiator has six functions and message handlers while a responder process has two message handlers. Let the initiator process run on node 'I1' in figure 3.3. The responder processes run on the other nodes.

The algorithm is initiated by the application by calling the trigger function, `trig_detectTermination(in)` at node '1'. This function calls two main methods, initiatePhase1 and initiatePhase2, which initiate the first and the second phase of the termination detection algorithm respectively. At node 'P2', the secondary message handler, `trig_marker_handler(..)` processes the marker messages it receives in both of the phases. On receiving a marker message, it checks its own state. If it is in a passive state then it forwards the marker message to its other neighbor.

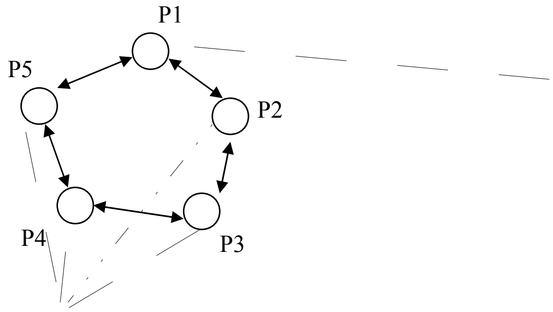
3.2.3 Algorithm Proof

Let $F^i = \{F^i_1, \ldots, F^i_n\}$ be a flow. A proof for F^i_j is developed as follows:

- For each function fn in F^i_j, the designer must associate a precondition $pre(fn)$ and a postcondition $post(fn)$ with it.

- For each message handler mh in F^i_j, the designer must associate a precondition $pre(mh)$ and a postcondition $post(mh)$ to it.

- The designer may specify a set of invariant $Inv(F)$ for the flow. The designer can use *interface variables* along with node-level global variables for the invariant and the pre and post conditions. For example processName.state ∈ {active, passive} is an interface variable in Algorithm 2.1. Here the set {active, passive} are the possible values of the variable processName.state. This variable is used in the proof annotations to describe properties (the state of the node processName being active or passive) of the application layer components.



```
RESPONDER
global:
startTimeStamp
stateChangeTimeStamp
self_state;
sequence_number

trig_marker_handler(marker,processName):
        if startTimeStamp = null
                startTimeStamp :=  currentTime()
        if processName = 4
           neighbor :=  1
        else
          neighbor := processName + 2
        endif
        if self_state = "passive"
           send(marker,neighbor)
        else
                if processName = 1 neighbor := 4
                else   neighbor := processName - 2
                endif
        endif
        if self_state = "passive" ^  !checkChangeOfState()
              send(marker,neighbor)
        endif
        endif
        return

receive(msg,processName):}
        If msg.sequence\_number < sequence_number
          drop msg
        else
         sequence_number := msg.sequence_number
        endif
        return

boolean checkChangeOfState():
        stateChanged := false
        if stateChangeTimeStamp > startTimeStamp
                stateChanged := true
        endif
        return stateChanged
```

```
INITIATOR
  global:
  startTimeStamp
  stateChangeTimeStamp
  recievedMarker
  phase1Complete
  phase2Complete
  currentState_of_self
  sequence_number

 trig\_detectTermination(n):
        terminated := false
        resetStateInfo();
         initiatePhase1();
        if (phase1Complete := true
                initiatePhase2();
        endif
        return

initiatePhase1():
        send(marker, 2);
         wait(recievedMarker = true)
        phase1Complete := checkChangeOfState()
        return

initiatePhase2():
        send(marker, 5);
        wait(recievedMarker = true)
        phase2Complete := checkChangeOfState()
        return

marker_handler(fromProc)
        recievedMarker   := true

 boolean checkChangeOfState()
          stateChanged := false
 if stateChangeTimeStamp > startTimeStamp
    stateChanged := true
 endif
 return stateChanged

receive(msg ,processname):
return;
resetStateInfo():
startTimeStamp := currentTime()
recievedMarker := false
sequence_number := sequence_number + 1
return
```

Figure 3.7: *Termination Detection Algorithm*

- The designer must come up with a proof-outline for each function and message handler to show that the specified post-conditions are satisfied and each assertion in $Inv(F)$ is an invariant. A proof outline of a function fn involves associating an assertion with each control point (which essentially gives us a precondition $pre(s)$ and a postcondition $post(s)$ for each statement in fn). We must prove that the triple $\{pre(s)\}\ s\ \{post(s)\}$ is true.

We are proposing annotating algorithms at the function level rather than a more granular level, for example after every statement. This is sufficient for our purpose as we are attempting to perform optimizations at the level of eliminating messages and function calls. For example, for a message type, the post condition of the corresponding message handler gives a complete description of the purpose of the message and its effects. While post-conditions of individual statements provide more details, they are not necessary in our optimization steps.

Composition of flows

`Union Composition`: We use this operator to compose processes at each site. This composition operator simply takes the union of all message handlers and functions at a site. It is done with respect to a set of shared variables. That is, the composition must identify the variables to be shared between the activities (so that they are labeled with the same name in each of the activities). The composition requires the following conditions to be satisfied:
(a) Functions with the same name must have identical signatures and function bodies. In addition, they must have the same pre- and post-condition.
(b) Message handlers for the same message type must have identical signatures and function bodies. In addition, they must have the same pre- and post-condition.
(c) For any two process types at the same site F_k^i and F_k^j, any action or statement in F_k^i should not invalidate any annotated assertion in the proof outline of F_k^j. That is, We require that the proofs be interference free.

3.2.4 Annotated Termination Detection Algorithm

INTERFACE:

processName.state ∈ ”active”,”passive”

clockwiseChan_1..n.state ∈ ”empty”,”not_empty”, counterClockwiseChan_1..n.state ∈ ”empty”,”not_emp

sysTerminated

Termination Detection Algorithm: Initator

GLOBAL:

startTimeStamp

stateChangeTimeStamp

recievedMarker

phase1Complete

phase2Complete

currentState_of_self

sequence_number

```
trig_detectTermination(n):
terminated ← false
resetStateInfo();
initiatePhase1();
if phase1Complete = true then
   initiatePhase2();
```

```
  end if
  return

  initiatePhase1():

  send(marker, 2);
  wait(recievedMarker = true)
  $phase1Complete \leftarrow checkChangeOfState()$
  return
Post: $phase1Complete \land counterClockwiseChan\_1.state = "empty" \land counterClockwiseChan\_2.state = "empty" \land counterClockwiseChan\_3.state = "empty" \land counterClockwiseChan\_4.state = "empty" \land counterClockwiseChan\_5.state = "empty" \longrightarrow systemTerminated = \mathbf{true}$

  Pre: $phase1Complete = \mathbf{true}$
  initiatePhase2():
  send(marker, 5);
  wait(recievedMarker = true)
  $phase2Complete \leftarrow checkChangeOfState()$
  return
Post: $counterClockwiseChan\_1.state = "empty" \land counterClockwiseChan\_2.state = "empty" \land counterClockwiseChan\_3.state = "empty" \land counterClockwiseChan\_4.state = "empty" \land counterClockwiseChan\_5.state = "empty"$

  marker_handler(fromProc):
  $recievedMarker \leftarrow \mathbf{true}$
```

```
// This method checks if the state of the initiator has changed since the initiation of the algorithm.
boolean checkChangeOfState():
$stateChanged \leftarrow$ false
if $stateChangeTimeStamp > startTimeStamp$ then
  $stateChanged \leftarrow$ true
end if
return  stateChanged
```

```
receive(msg,processname):
return
```

```
resetStateInfo():
$startTimeStamp \leftarrow currentTime()$
$recievedMarker \leftarrow$ false
$sequence\_number \leftarrow sequence\_number + 1$
return
```

Termination Detection Algorithm: Responder

```
GLOBAL:
// array of size n where n is the number of nodes in the system
startTimeStamp
stateChangeTimeStamp
self_state;
sequence_number
```

```
**trig_marker_handler(marker,processName):** // secondary message handler
if $startTimeStamp = null$ then
  $startTimeStamp \leftarrow currentTime()$
  if $processName = 4$ then
    $neighbor \leftarrow 1$
  else
    $neighbor \leftarrow processName + 2$
  end if
  if self_state = "passive" then
    send(marker,neighbor)
  end if
else
  if processName = 1 then
    neighbor ← 4
  else
    $neighbor \leftarrow processName - 2$
  end if
  if self_state = "passive" ∧ ¬checkChangeOfState() then
    send(marker,neighbor)
  end if
end if
return

**receive(msg,processName):**
if $msg.sequence\_number < sequence_number$ then
  drop msg
```

```
**else**
  $sequence\_number \leftarrow msg.sequence\_number$
**end if**



**return**

// This method checks if the state of the initiator has changed since the initiation of the algorithm.
**boolean checkChangeOfState():**
$stateChanged \leftarrow$ **false**
**if** $stateChangeTimeStamp > startTimeStamp$ **then**
  $stateChanged \leftarrow true$
**end if**
**return** stateChanged
```

3.3 Specifying Application Properties

The second part of the framework requires the application developers to come up with properties describing the behavior of the application (Figure 3.8). These properties are treated as invariants by the optimization engine and the distributed algorithm is optimized with respect to them.

The application properties must be described using interface and global variables with the usual arithmetic, relational and logical operators(see Appendix C for the grammar). Interface variables are used to share information between the layers(application-algorithm).

The optimizer also needs the network structure as an input to the analyzer. The network structure specifies how the nodes are connected and the mapping between nodes and process types.

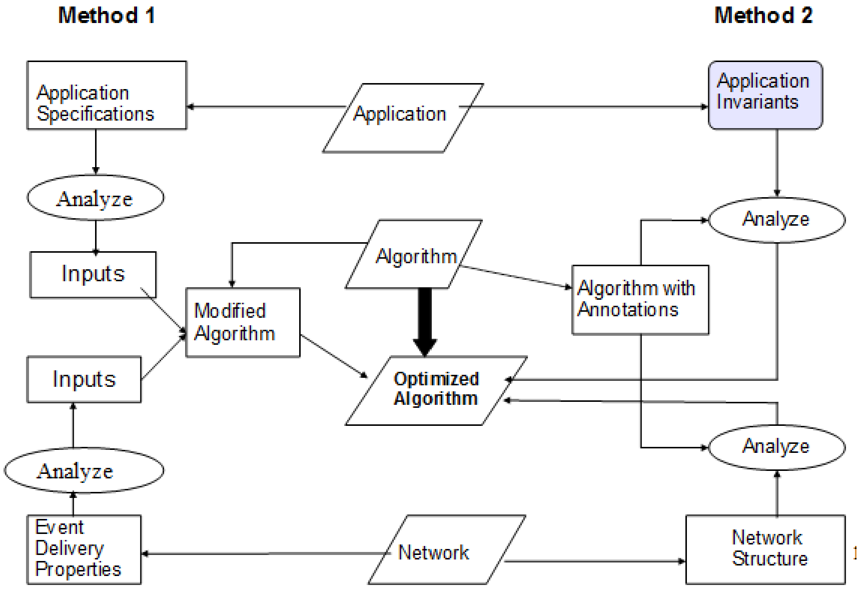


Figure 3.8: *Application Properties*

3.3.1 Example: application property

Consider an application running on a bi-directional ring network as shown in Example 3.1 which send messages only in the clockwise direction. This implies that all channels in the anticlockwise directions will remain empty. Therefore we can express this application property as :

```
counterClockwiseChan_1.state = "empty" ^ counterClockwiseChan_2.state = "empty" ^
counterClockwiseChan_3.state = "empty" ^ counterClockwiseChan\_4.state = "empty" ^
counterClockwiseChan_5.state = "empty".
```

The network structure of the system in Fig 3.3 consists of the following channels and mapping of nodes to process types. The clockwise channels: (I1,P2),(P2,P3),(P3,P4),(P4,P5),(P5,P1). The anticlockwise channels:(P5,P4),(P4,P3),(P3,P2),(P2,I1) and process type to node mapping: Initiator : I1, Responder : P2,P3,P4,P5

Using the grammar defined in Appendix B we write the network structure as :

```
System "ring" {
nodelist: 1,2,3,4,5;
channelList : 1->2, 2->3, 3->4, 4->5,5->1,1->5,5->4,4->3,3->2,2->1;
mapping : 1:"Initiator", 2: "Responder", 3: "Responder",4: "Responder",
5: "Responder";
}
```

3.4 The optimizer

This section explains the different steps the optimizer takes to analyze and optimize the distributed algorithm (Figure 3.9).The optimizing process consists of three main steps. In the first step we construct an intermediate representation of the algorithm which can be analyzed. In the second step we analyze the intermediate data structures with respect to the application/network properties to identify redundancies. The third and the final step involves removing the redundant parts and making the necessary transformation to the algorithm to obtain an optimized version of it.

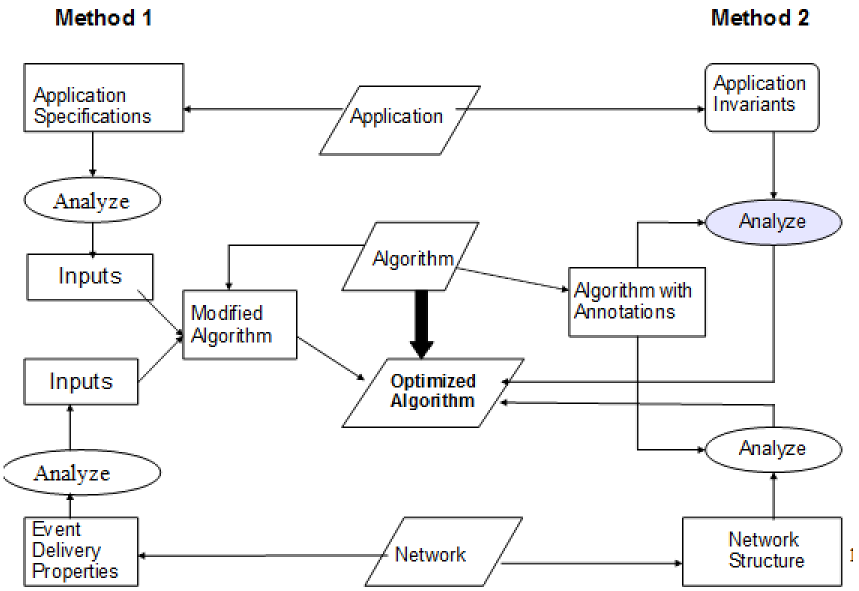


Figure 3.9: *Analysis*

3.4.1 Intermediate representations

We construct an intermediate representation from the algorithm specification. The following section give the descriptions for such a representation and we use the annotated termination detection algorithm 3.2.4 as an example in the explanations.

Call Graph

A call graph $G=\{V,E\}$ is a graphical representation of the call structure of the program F_j^i. It is a directed acyclic graph where each node in V represents a function or a message handler. Each edge in E is one of two types.

-*call edge*: It is a directed, labeled edge (a,b) : $a,b \in V$ representing a function call where function a calls function b.

-*wait edge*. It is a directed, labeled edge (a,b) : $a,b \in V$ which connects a node a containing a *wait(condition)* statement to a secondary message handler b which writes to one of the global variables in *condition*.

The following are some terminologies and properties associated with a call graph:

- Parent and child nodes : If there is a call edge *(a,b)* $\in$ E, then node a is referred to as the parent node of b and b is referred to as the child node of a. A node in the graph can have more than one parent node.
- Trigger node : The nodes representing the trigger function are called the trigger nodes. We assume a call graph has one trigger node.
- Leaf node : Any node without any outgoing call edges is called a leaf node.
- We assume that the call graph is acyclic.

We construct one call graph for each process which makes a flow.

Constructing the call graph:

The call graph for a process type is constructed from its specification. The following steps constructs the call graph G=$\{V$,$E\}$. As a running example we construct the call graph for the *initiator* program from Algorithm 3.2.4.

STEP 1: Add the primary message handler as a node to V. For Algorithm 3.2.4 we add a node to represent the handler receive(..).

STEP 2: Add a node to represent each secondary message handler to V. For the initiator process, we add a node for the secondary message handler marker_handler(..).

STEP 3: Identify the trigger function for the process type by the prefix trig_ in the function name. Add the trigger function as a node to V. In this example, we have only one trigger function called trig_detectTermination(..). We add this to V. Note that in many cases, the secondary message handler will be a trigger function. In such a case, we do nothing for this step.

STEP 4: Next starting with the trigger node n, for every function call to function n_c, we add a node n_c to V and an edge (n,n_c) to E. A newly added node n_c will be a child node of n. For every *wait(cond)* statement in n, we add a wait edge (n,n_s) to E which connects n to the secondary message handlers $n_s \in V$, where n_s writes to a global variable in *cond*. STEP 5: Repeat STEP 4 for all leaf nodes in the call graph(replace trigger node n with leaf node n).

In the call graph for the initiator, the trigger function trig_detectTermination calls three other functions, resetStateInfo(),initiatePhase1()and initiatePhase2(). These are added as nodes to represent them as children of the trigger node. The functions initiatePhase1() and initiatePhase2() have wait statements in it which wait for the global variable phaseComplete to become true. This variable signals the end of a phase and is modified by the secondary message handler marker_handler(). Therefore, we have two wait edges in this call graph.

The call graph for the responder process type in algorithm 3.2.4 contains two nodes to represent the primary and a secondary message handlers. The call graphs for both process types in the annotated termination detection algorithm (3.2.4) is given in Figure 3.10.

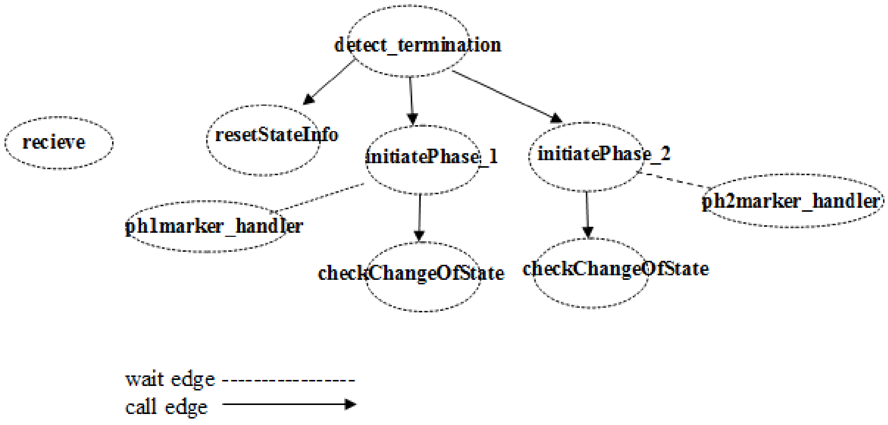


Figure 3.10: *Call Graphs for the initiator process in 3.2.4*

Analysis Algorithm

```
// Globals
```

```
flowScenario;
last_z; // Keeps track of global variable updates by functions at the initiator site.
functionAtInitiatorSite;
atInitiator; // Keeps track whether the function being analyzed is at the initiator site
waitqueue; // Keeps track of all unexplored message chains


**analyzeFlow(function, callGraph, invariant)**{
currentFn $\leftarrow$ function;
done $\leftarrow$ 0;
**while** done = 0 **do**
  **if** currentFn.pre $\land$ invariant $\rightarrow$ currentFn.post **then**
    // Start building a flow scenario
    // Determine the values for the input parameters of currentFn from invariant and precondition for currentFn
    parameterList $\leftarrow$ determineArguments(currentFn, invariant);
    // create a new flow scenario
    flowScenario $\leftarrow$ new FlowScenario();
    // call analyzeNode to analyze currentFn and all its dependent functions.

    functionAtInitiatorSite $\leftarrow$ currentFn;
    atInitiator $\leftarrow$ **true**
    analyzeNode(currentFn,parameterList);
    **while** waitQueue != empty **do**
      currentFn = waitQueue.pop();
      analyzeNode(currentFn,parameterList);
    end while
    functionRemoved := checkForRemovability(flowScenario) && checkForIncomingWaits(callGraph,flowScenario) ;
    if functionRemoved then
      transformCode(currentFn.return, currentFn.modify);
      // Continue depth first traversal from parent function
      currentFn ← callGraph.getParent(currentFn);
    end if
  else
    // Continue depth first traversal
    // the function below returns child functions connected via call edges and message handlers which can be connected to a message stub in fn.
    if (fn := callGraph.getUnvisitedFunction(currentFn)) != null then
      currentFn ← fn;

    else
      currentFn ← callGraph.getParent(currentFn);
    end if
  end if
end while

}


// Function Analysis Algorithm

// Input: Function to be analyzed , values of input parameters
```

```
Output: Updated flowscenario with assertions generated for the input function
and all its dependent functions
analyzeNode(function, parameterList):
if atInitatorSite = true; $\wedge$ function is of type MessageHandler then
    // left initator node for the first time
    atInitatorSite $\leftarrow$ false;
end if
// Add the function and a call edge to the flow scenario
flowScenario.addFunction(function);
state $\leftarrow$ new State(parameterList);
while (statement $\leftarrow$ function.getNextStatement()) != null do
    if statement is a "functioncall" then
        // find values of input parameters for the function from invariant and
        contents of the state
        parameterList $\leftarrow$ determineArguments(functionCalled, state, invariant);
        analyzeNode(functionCalled, parameterList);
        // If we are back at the initiator site, then record it.
        if functionAtInitiatorSite = function then
            atInitiatorSite $\leftarrow$ true;
        end if
    end if
    if statement is a "send(message,destination)") then
        flowScenario.addMessageEdge(message,function,destination);
        if atInitatorSite = true; $\wedge$ function is of type MessageHandler then
            // Leaving initator node
            // Remember at which function the analysis moved from the initiator
            node.
```

```
      functionAtInitiatorSite $\leftarrow$ function;
    end if
    // Find the correct message handlers which will process message. (from the call graph and network structure).
    for each messageHandler $\in$ flowScenario.messageHandlers(message,inv) do
      parameterList.add(message);
      analyzeNode(messageHandler,parameterList); // First analyze primary and then analyze secondary message handler
    end for
  end if
  if statement is a ”wait(condition)” then
    if condition != true then
      // add current position of analysis in the flow graph to waitqueue
      waitQueue.add(function,statement,state)
    end if
  end if
  if statement is an assignment ”var = exp” then
    // update the state variable with the result of the evaluated expression exp
    state.update(var,exp)
    // update last_z if function is at initiator node
    if atInitiatorSite then
      last_z.modify(var,exp)
    end if
  end if
  if statement is ”ifcond-then-else” then
    if state.evaluateCondition(cond) = true then
      analyze if block
```

```
        end if
        if state.evaluateCondition(cond) $=$ false then
            analyze else block
        end if
        if state.evaluateCondition(cond) $=$ true$\vee$ false then
            analyze if block and else block
        end if


    end if
end while
// Function analyzed, get function.return, function.modify, function.sent
from the state variable and add to flowscenario.
flowScenario.add(function,state);
// Check if the contents of function.return, function.modify, function.sent
has all required information to continue analysis
//
if function.singleReturnValue $\wedge$ function.singleGlobalUpdate $\wedge$ function.messagesSentPredicted
then
    if (function.pre $\wedge$ inv.applicationInvariant $\rightarrow$ function.post) $\wedge$ function.sent.size $= 0$
    $\wedge$ function.singleReturnValue $\wedge$ function.singleGlobalUpdate then
        function.label $\leftarrow$ 'REMOVED';
    end if
else
    STOP;
end if
```

3.4.2 Analyzing and Optimizing the call tree

The second step taken by the optimizer is the analysis of the call graphs. The analysis is driven by the properties of the application and the structure of the network. We want to reduce redundancies in the system by reducing or eliminating the work to be done at the algorithm layer based on properties of the application structure. For example, when using a global state gathering algorithm, if the state of a node n_i is known in advance, we can eliminate the message exchanges needed to determine the state of n_i by any other node in the system.

A function call can be seen as a unit of work which achieves a post condition. If this post condition is satisfied prior to the invocation of the function, then we can remove this invocation. A function may also update some variables (global variables and the local variables of the calling function via the return value) and can affect the state of a process at another site in the system via messages. Therefore, to be able to remove a function invocation, we need the following:

(a) Ensure that the post condition is satisfied by the application properties at the time of the function call

(b) Ensure that variable updates are preserved after the call is eliminated, and

(c) Ensure that the elimination does not impact computation at other sites in the system.

For a function *fn*

- Let *fn*.`return` represent the value of the return variable (this value is void if no return variable is present).

- Let *fn*.`modify` represent a set of key value pairs $(g_i = v_i)$, where g_i is a global variable modified by *fn* and v_i is the last value assigned to g_i.

- Let *fn*.`sent` represent a set of messages sent by *fn*.

Let *Inv* be an invariant which describes an application property of a system. Let the middleware algorithm used be $\textsc{alg} = (F^1, \ldots, F^n)$. We analyze each flow $F^i \in \textsc{alg}$ $0 \leq i \leq n$ and tag functions in F^i which can be removed to obtain an optimized version F^i_{opt}.

We will prove that the resulting algorithm $\textsc{opt}(\textsc{alg'}) = (F^1, \ldots, F^i_{opt}, \ldots, F^n)$ preserves all the properties of ALG with respect to the application.

To analyze a flow F^i, we use an analysis algorithm called *analyzeFlow*. We start the analysis at the trigger node of the call graph for the process running on Init(F^i). We traverse the call graph along the call edges in the sequence in which the functions are called. This traversal corresponds to the depth first traversal of the call graph. When we come across a function which sends a message, we use the information available about the network structure to identify and visit the message handler which processes that message and any of its dependent functions. For every function *fn* encountered in the traversal the following implication is tested:

Implication 1: $pre(fn) \wedge Inv \rightarrow post(fn)$

Lines 1-6 in the analysis algorithm declares some global variables used by *analyzeFlow*. The depth first traversal of the call graph starts on line 13. When a node is visited, we first test *Implication 1* as in line 14. If the function satisfies *Implication 1*, we start creating a *flow scenario*(line 19). If the visited function does not satisfy *Implication 1*, the algorithm continues with the depth first traversal (lines 35-44) by visiting the next unvisited node.

A flow scenario is a graphical representation of a full or a partial flow F^i. $Sc_i = \{\gamma_i,\epsilon_i\}$ for a flow $F^i = F^i_1 \mid\mid F^i_2 \ldots\mid\mid F^i_n$, where γ_i is a set of nodes from the call graphs of $F^i_j \in F^i$,0¡j¡(n+1) and ϵ_i is a set of directed edges. The directed edges are either call edges from $F^i_j, 0 < j < (n+1)$ or *message edges*. A *message edge* is a labeled directed edge *a,b* where *a* is a node which sends a message and *b* is a message handler of the receiving process type. Both nodes *a* and *b* belong to the call graphs of a process (**) in F^i. The label of the message edge indicates the type of the message being sent by node *a*.

Beginning with the first function to satisfy *Implication 1*, we use an algorithm called *analyzeNode* to (a) create the flow scenario, (b) test *implication 1* and (c) generate the sets *fn*.`return`, *fn*.`sent`, *fn*.`modify` at the end of every function encountered.

We call the site where the first function (fn) to satisfy Implication 1 is located an 'initiator site'. The flow scenario diagram starts with fn at the initiator site (3.11). At the end of the analysis, if this function is marked removed, then we will need the last update made by the function or any function it calls at the initiator site to make the correct code transformations.

We define the set $last_z$ is a set of key value pairs k-v, where the k is a global variable in process P to which fn belongs, and v is the last update made by fn or any child function of fn running at the initiator site.

The analyzeNode function begins in line 52. The analysis algorithm uses the variables last_z, functionAtInitiatorSite, and atInitiator to keep track of all the global variable updates being made at the initiator site. Following is the purpose of each variable:

last_z : This set holds the the key value pairs for the global variables at the initiator site and their values.

atInitiator: This is a boolean variable which is set to true when the function being analyzed by *analyzeNode* executes at the initiator site. When this variable is true then any changes made to global variables are recorded in the variable last_z. This variable is set to true before the analyzeNode function is called(Analysis Alg lines 25).

If any function has a send statement which sends a message m, then we will analyze the message handler which receives this message and its dependent functions. To find the message handler which processes m, we look at the network property which contains a map pairing sites to the process types deployed on them. Section 3.3.1 contains the specification of the network structure for the example system.

The algorithm analyzeNode generates the sets *fn*.`return`, *fn*.`modify` and *fn*.`sent` for every function fn in the flow scenario (Analysis Alg, line 110). A flow scenario may contain message chains. A message chain represents the path of execution within a flow when a sequence of messages $m_1, m_2..m_N$ occurs where m_i causes m_{i+1} to be sent out. It is represented in the flow scenario by a series of directed edges $\{e_1, ..e_i, ..e_n\}$ where each edge is either a call edge or a message edge. Any two consecutive edges in this sequence are of the form $e_i = \{a, b\}, e_{i+1} = \{b, c\}$ where a,b,c are function nodes or message handlers. The order of message edges in $\{e_1, ..e_i, ..e_n\}$ strictly follows the order of the messages of the chain it represents, $m_1, m_2..m_N$. The first edge in any message chain is always a message edge that is e_1 always represents message m_1.

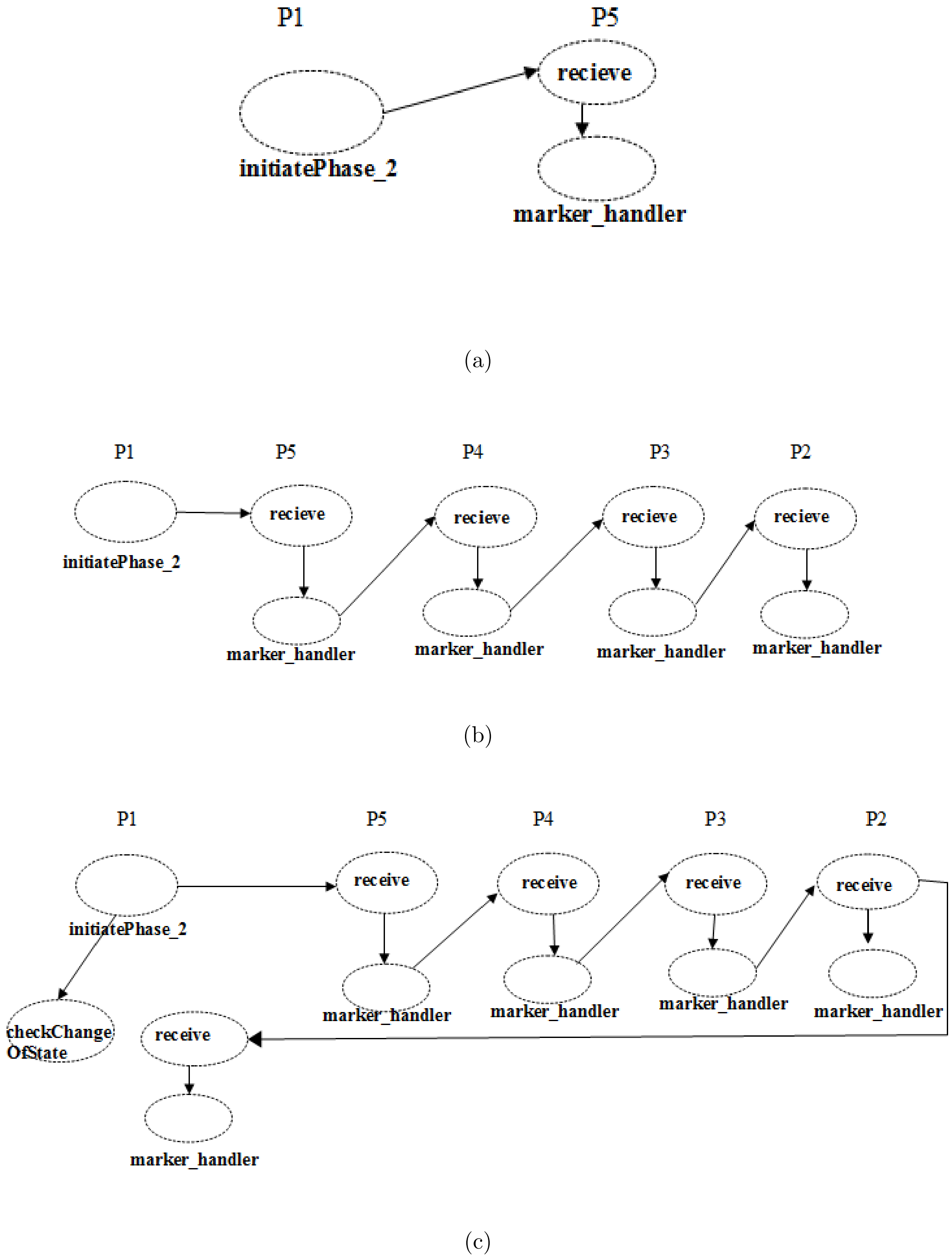
A function node is marked *removed* based on the following conditions(Analysis Alg, Lines 114):

Condition 1: A function is marked *removed* if *Implication 1* is satisfied and we determine from *fn*.`return` that the return variable has a unique value and all global variable updates are unique. The function also should not send out any messages.

Condition 2: *Implication 1* is satisfied for a function and we determine from *fn*.`return` that the return variable has a unique value and all global variable updates are unique. If the function sends a message, then we can mark it *removed* if all handlers processing the messages in that chain are marked *removed.* Removing any one message handler in a message chain when the invariant satisfies only its post condition will prevent the next messages in the chain from being sent. If the invariant does not imply the post conditions of the message handlers of the unsent messages the algorithm logic will be affected. Therefore, we are making sure that for any sequence of messages that can occur in a system, the post conditions of all the message handlers and functions processing them are fulfilled by the invariant *Inv* before removing any function/message handler from that chain.

At the end of the analysis of a function fn, we can transform the code to remove fn if we do not find incoming wait edges to any node in the flow scenario(line 28).

The nodes in our example system (section 3.1) are connected to in a ring topology. Let us assume the application executing on these nodes sends messages only in the clockwise



(a)

(b)

(c)

Figure 3.11: *Constructing the flow scenario*

direction. Therefore, the system will be 'terminated' when all the nodes are in the 'passive' state and all the clockwise channels are empty. The application property I_app is specified as : $counterClockwiseChan\_1.state = "empty" \wedge counterClockwiseChan\_2.state = "empty" \wedge counterClockwiseChan\_3.state = "empty" \wedge counterClockwiseChan\_4.state = "empty" \wedge counterClockwiseChan\_5.state = "empty"$.

The algorithm analyzeFlow finds that the application invariant implies the post condition of the initiatePhase2() function. The analyzeFlow algorithm now calls the analyzeNode algorithm with the input function 'initiatePhase2'. The algorithm analyzeNode now starts creating the flow scenario and generates the sets `return`, `sent`, `modify` for each of the functions in the flow scenario. The numbers at the end of each function in the figures (3.11) shows the order in which the functions are added to the flow scenario.

When a function node *fn* is removed, we transform the code of the function represented by the parent node (The application code has to be transformed if the trigger node has to be removed). This transformation involves removing the function call and replacing it with either a skip statement or a set of assignment statements. A function call will be replaced with a skip statement if no value is being returned by it and no updates are being made to the global variables within the function. If *fn* is returning a value or updating a global variable, the function call is replaced with assignment statements. Hence during the analysis we need to determine the values of the return variables and global variable updates by a function which is deemed removable. We also need to know the exact type and contents of any message sent by the function. This information is needed to determine the effect of *fn* in other nodes of the system.

3.5 Code Transformation

The third step in the optimization process involves removing the function call fn whose causal set was analyzed and determined to be removed. In this section we describe the different types of code transformations to be made when removing a function call.

Let function fn be marked removed by the analysis algorithm. Therefore fn.`return` and fn.`modify` tells us the values of any return variables and global variable updates made by fn. The code replacing fn depends on the information in fn.`return` and fn.`modify`.

Code transformation patterns:

- Pattern 1: fn is a function call with signature fn`()` where both fn.`return`, fn.`modify` are empty.

  Original code :

  `fn()`

  Transformed code:

  `skip;`

- Pattern 2: fn is a function call with signature `returnType` fn`()` where fn.`return` and fn.`modify` are not empty.

  Original code :

  `x = fn();`

  Transformed code:

  `x =` fn`.return`

  $g_1 = v_1; g_2 = v_2; ...$

- Pattern 3: fn is a function call with signature fn`(arglist)` . Here argList list of input parameters and Implication 1 from section 3.1 hold for $x_1 = v_1, x_2 = v_2...$ where v_i are specific values of the parameters x_i. fn.`return`, fn.`modify` are empty.

  Original code :

  `fn(`$x_1 = v_1, x_2 = v_2...$`);`

  Transformed code:

```
if($x_1 != v_1, x_2 != v_2 ...$)
        fn(arglist)
```

- Pattern 4 : fn is a function call with signature `returntype` fn`(arglist)`. Implication 1 from section 3.1 hold for $x_1 = v_1, x_2 = v_2...$ where v_i are specific values of the parameters x_i. fn`.return`, fn`.modify` are not empty.

  Original code :

  ```
  x = fn($x_1 = v_1, x_2 = v_2...$);
  ```

  Transformed code:

  ```
  if($x_1 != v_1, x_2 != v_2 ...$)
          x = fn($x_1 = v_1, x_2 = v_2...$);
  else
          x = $fn$.return
          $g_1 = v_1; g_2 = v_2; ...;$
  ```

3.5.1 Code changes in the parent node

When we analyze the effects of removing a function fn we do so in the context of the function call [line 17 in function analyzeFlow, section 3.4.1]. Because our grammar allows global variables, this context may be different for each call to fn. Consider a scenario in Figure 3.12. If our analysis determines that the function call to fn in 'path 1' is redundant and can be removed, we must change the parent function fp with one of the code transformations discussed above. However from the call graph we see that function fp is also called in $Path2$. A modified fp will produce a different execution in path 2 which has not been analyzed. Therefore instead of changing fp, we make a copy of the function fp_{copy} and make the code transformation in fp_{copy}. Then we replace the call to fp by a call to fp_{copy} in the parent function of fp in Path 1. The original function fp remain unchanged and therefore we do not affect other parts of the flow.

We define a transformation rule as follows: For any function f which requires a code transformation, a copy of f must be created if the callgraph for the process type where f has been defined has multiple vertices for the function.

For any optimization scenario the above rule must be recursively applied till we reach a function which has only one vertex in the call graph. The number of function copies we must make is finite since the algorithm specification requires that every processtype have a single trigger function. Typically in most distributed algorithms we have analyzed, the depth of the call graph has been small and the number of functions copies have been limited to a few.

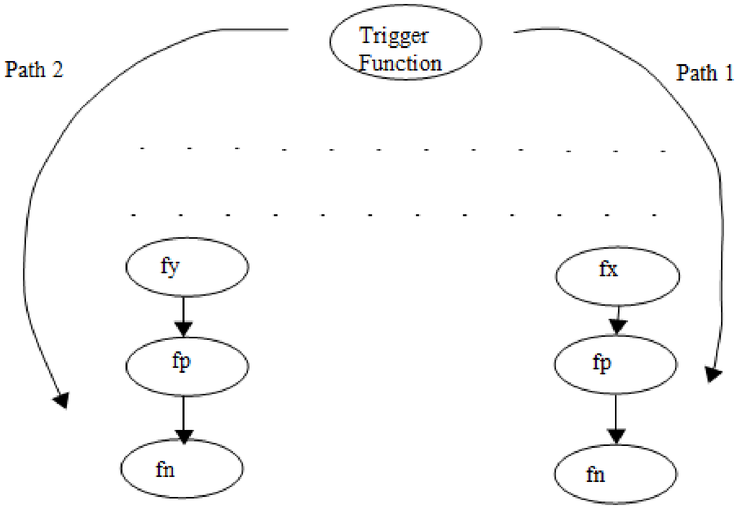


Figure 3.12: *Context of Optimization*

3.6 Proofs

Let Alg $=\{F^1, ..., F^n\}$ be a distributed algorithm deployed in a system with application properties in *Inv.* Forall i, 0<i<n+1 we analyze and optimize a flow $F^i \in$ Alg. Our optimizing technique replaces a function call with assignment statements. This results in the removal of the corresponding function call event from all executions of the flow. Therefore we need to understand the effects of the removed event on other events in the flow.

In flow F, let fn be a function that has been marked removed. Let e_i be the event which calls the function fn. From the analysis algorithm, we have the following:

(a) The post conditions of all functions calls in causal(e_i) are satisfied by I_{App}.

(b) There is no event $e \in$ causal(e_i) which is waiting for an event $e_x \notin$ causal(e_i). That is, there is no incoming wait edge from an event outside the causal set to an event inside the causal set.

(c) All last updates made to global variables in causal(e_i) at the site where e_i is the same for every execution of F.

Lemma 1: Given an execution Ex of a flow F: e_1,..,e_i,..,e_x,causal(e_i),..,e_n of Alg, if we move the events in causal(e_i) before e_x then the resulting execution is also a valid execution of F.

Proof:

Since $e_x \notin$ causal(e_i), we know from the definition of causal(e_i) that there does not exist an event $e \in \{e_i\} \cup causal(e_i)$ such that $e_x \rightarrow e$. Therefore, the event e_x is independent of any event in the set $\{e_i\} \cup causal(e_i)$ and can execute in parallel. Therefore $Ex' =$ e_1,..,e_i,..,causal(e_i),e_x,..,e_n is a valid execution of F.

In flow F, let fn be a function that has been marked removed and the parent node of fn contains the statement S : '$y = fn()$'. In any history h of F, let e_{fn} denote the event corresponding to a function call to fn, and ret_{fn} denote the event corresponding to the return of this call. Given a history $h = (Ex, \longrightarrow_h)$ of a flow F and an event e_{fn} in Ex, we partition the events in Ex into three sets :

(1) A set with one element e_{fn}, the function call event whose post condition is satisfied by `Inv` and has the potential to be removed.

(2) The causal set of e_{fn}, $causal(e_{fn}) = \{e : e_{fn} \longrightarrow e\}$.

(3) The set of all other events $Others = \{\text{e} : \text{e} \notin \text{causal}(e_{fn}) \cup e_{fn}\}$

Let $return\_value(fn)$ contain the value from *fn*.`return`. Let z,$z.val$ denote the key value pairs in $last_z$. For a variable $z \in last_z$, let $e_{z.val}$ denote the last event assigning a value $z.val$ to z in $causal(e_{fn})$. The sets *fn*.`return` and $last_z$ are computed by the analyzeNode algorithm (section 3.4.2).

Derivation Rule: To derive F_{opt}, we replace S by S_{opt} : "$y = fn\_opt()$". The function $fn\_opt()$ consists of assignment statements of the form $z := z.val$ for every pair in $last_z$ and it returns $return\_value(fn)$.

Given a history $h = (Ex, \longrightarrow_h)$ of F^i, $h_{opt} = \{Ex_{opt}, \longrightarrow_{opt}\}$ is an execution history of F^i_{opt}, such that $Ex_{opt} = (Ex - causal(e_{fn})) \cup \{e_{fn\_opt}, e1_{z.val}, ..., ret_{fn\_opt}\}$ where $e1_{z.val}, ...,$ are the assignment events in fn_opt(z:=z.val).

$e1 \longrightarrow_{opt} e2$ iff

(a) $e1 \longrightarrow_h e2$ and $e1, e2 \notin causal(e_{fn})$,

(b) $e1 \longrightarrow_h e_{fn}$ and $e2 = e_{fn\_opt}$.

(c) $ret_{fn} \longrightarrow_h e2$ and $e1 = ret_{fn\_opt}$.

We will denote an execution of a flow as follows:

- $P = p_1, p_2, \ldots,$ where p_i is an event.
- P_x is used to refer to an execution fragment of P up to the event p_x.
- $p_x.statement$ is used to denote the statement whose execution causes the event p_x.
- $p_x.state$ is used to denote the state before the occurrence of event p_x and it contains the values of all global variables at the site where fn executes.

From the definition of $last_z$, we know that there exists at least one assignment statement event in $causal(e_{fn})$ for each variable in $last\_z$.

Let $P = p_1, p_2, \ldots,$ be an execution of F^i in which all events in $causal(fn)$ appear immediately after e_{fn}. We refer to such an execution as an execution *atomic* with respect to S (or atomic to the statement of the event e_{fn}). From Lemma 1, we know that such an execution exists.

We define P^t as an execution obtained by removing all events belonging to $causal(e_{fn})$ except $e_{z.val}$, where $z \in last_z$ and replacing e_{fn_opt} and ret_{fn_opt} with e_{fn} and ret_{fn} respectively.

Lemma 2: For each execution $G = g_1, g_2, \ldots,$ of F^i_{opt}, there exists an execution $P = p_1, p_2, \ldots$ atomic with respect to S of F^i, such that $G = P^t$.

Proof:

We will prove this by constructing such an execution. Let $g_l = e_{fn\_opt}$. We can find an execution of F^i which is identical to F^i_{opt} *until the occurrence of* g_l. Hence, $P_{l-1} = p_1, p_2, \ldots, p_{l-1}$ is an execution fragment of F^i where $p_i = g_i$, $1 \leq i < l$. Since $S' = g_l.statement$ is the next statement to be executed in F^i_{opt}, the statement S can be executed in F^i. Hence, we let p_l $= e_{fn}$ to be the next event.

From Lemma 1, we can extend P_{l-1} to include atomic execution of $causal(e_{fn})$. Let, $frag$ $= p_1, p_2, \ldots, p_{l-1}, e_{fn}, c_{l+1}, \ldots, c_m$ such that $causal(e_{fn}) = \{c_{l+1}, \ldots, c_m\}$. Next, we obtain $frag^t$ (by removing all events in $causal(e_{fn})$ except assignment events $e_z.val$ for each $z \in last_z$). Let $frag^t = G_k = p_1, \ldots, p_{l-1}, e_{fn}, p_{l+1}, \ldots, p_y$ where the events $p_{l+1}, \ldots, p_y$ are the assignment events obtained from the key-value pairs of $last\_z$.

From the derivation rule, we know that the set of events for the assignment statements in fn_opt are equal to the assignments in the events $p_{l+1}, \ldots, p_y$ although they may not be in the same order. However, since the same assignments are made in both cases, we have that $g_{y+1}.state = p_{y+1}.state$. Since the state after the y^th event is the same in both F^i and F^i_{opt}, we have that $p_1, \ldots, p_y, p_{y+1}, \ldots$ is an execution of F^i.

Equivalent flows :

Let F be a flow with history $\{EX, \longrightarrow\}$. We define an equivalence relation between two executions of the same flow as follows:

Let $Ex = \{e_1, e_2, .., .., e_i, .., e_j, .., e_n\}$, where e_i and e_j are two function call events to functions fn_i and fn_j respectively. Let e_j.statement occur in the function body of fn_i. Therefore, $e_i \rightarrow e_j$. Furthermore, assume that e_j.statement has been identified by our analysis as removable. Let Ex' be an execution of the optimized flow with the following properties: Ex and Ex' are identical up to e_j. We replace e_j.statement in the function fn_i by assignment statements $z := z.val$ for $z \in last_z + fn_i.return$.

Then, we say that Ex and Ex' are equivalent ($Ex \Longleftrightarrow Ex'$). Ex is essentially the execution constructed from Ex_{opt} in Lemma 1. From the construction, we have know that for all events e which are present in both Ex and Ex_{opt}, $e.state$ is the same in both executions.

We optimize a flow $F \in$ ALG and the resulting flow is F_{opt}. In the optimizing process our analysis algorithm identifies a function call event e_{fn} and marks it removed depending on its effects on other events in the flow. The tagged function call is then removed by changing the code of its parent function node. We replace the call with assignment statements based on the contents of the sets fn.`return` and $last_z$. Therefore, for any execution Ex_{opt} of F_{opt}, there exists an execution Ex of F such that $Ex \Longleftrightarrow Ex_{opt}$.

3.6.1 Proof of correctness

The following conditions need to be satisfied when removing e_{fn_x} from F to obtain F_{opt}. Let $(EX, \longrightarrow)$ be the execution history of a flow F and $(EX_{opt}, \longrightarrow)$ be the execution history of the optimized flow F_{opt}.

1. **Let e_1 be the event representing a function call to fn_y and e_2 be the event representing its return. We know that in any execution of F, pre(fn_y) is true when fn_y is invoked, and post(fn_y) holds after e_2.** We must show that this is true in any execution of F_{opt} as well. On the contrary, let Ex_{opt} be an execution

of F_{opt} in which this is not true. There are two conditions under which this can happen.

Condition 1: In an execution Ex_{opt} of F_{opt}, pre(fn_y) is true but post(fn_y) is false. We know that there exists an execution (Ex,$\longrightarrow$) of F such that $Ex \Longleftrightarrow Ex_{opt}$. If fn_y is not a parent function of fn_x in Ex then the function remains unchanged. Therefore, since $e.state$ is the same for all events which are present in both Ex and Ex_{opt}, if post(fn_y) is false after e_2 in E_{opt} then post(fn_y) was false after e_2 in Ex as well, which is a contradiction.
In the case when fn_y is a parent function of fn_x, then the call to fn_x is replaced by assignment statements which exactly reflects the variable updates made by fn_x and its causal set. Therefore, in both the executions Ex and Ex_{opt} the same updates are made. Hence, if pre(fn_y) is true then post(fn_y) is also true in Ex_{opt} if it were same case in Ex

Condition 2: In an execution Ex_{opt}, pre(fn_y) is false when fn_y is invoked. Let Ex_{opt} $\Longleftrightarrow$ Ex. When optimizing we replace the event e_{fn_x} by assignment statements which makes the same updates as e_{fn_x} and causal(e_{fn_x}). If the function call to fn_x happened before the function call to fn_y in Ex, the same sequence of updates will take place in Ex_{opt} as well. Therefore, this will not affect pre(fn_y). If the actions of e_{fn_x} and causal(e_{fn_x}) happened after the function call to fn_y in Ex, any variable updates by causal(e_{fn_x}) and e_{fn_x} happens after the function fn_y executes and this does not affect the pre(fn_y) in either Ex or Ex_{opt}. Therefore in either case, for any execution Ex_{opt} of F_{opt} when pre(fn_y) is false there exist an execution Ex of F^i where pre(fn_y) is also false. This violates our assumptions.

2. F^i_{opt} **is deadlock free.** From the above argument we see that for any execution Ex_{opt} of F_{opt} there exists Ex of F^i and $Ex \Longleftrightarrow Ex_{opt}$, where the actions of causal(e_{fn_x}) + e_{fn_x} can happen before or after the execution of any function fn_y without affecting it. We also know from Lemma 1 that there does not exist a explicit or an implicit wait

between events in causal() and any other event in the execution.

Let us consider an execution $\{Ex_opt, \longrightarrow\} = e_1,..,e_i,..e_n$. If this execution is deadlocked then some event e_i is stuck. From the definition of the $\longrightarrow$ relation we see that such a case is only possible when e_i is a wait statement and there exists a signal event e_j, $e_j \longrightarrow e_i$. Therefore the deadlock will happen only if e_j the signal event has been removed. This is possible only if $e_j \in$ causal$(e_{fn_x}) + e_{fn_x}$. In such a case there has to be a wait edge between events in causal$(e_{fn_x}) + e_{fn_x}$ and the other functions. Since we have already established that such a wait edge cannot exist, when causal$(e_{fn_x}) + e_{fn_x}$ is removed from F to obtain F_{opt}, F_{opt} will be deadlock free if the original flow F was deadlock free.

3. Alg_{opt} **=$\{F^1$,..,F^i_{opt},..,$F^n\}$ is deadlock free**.

   From the definition of an algorithm we know that each flow is independent of the others and is written to execute and finish on its own. Therefore removing a function call from a flow F^i cannot cause a deadlock in any other flow of the algorithm.

3.7 Implementation

The algorithm in section 3.4.1 can be implemented in many ways. We have chosen to use existing tools such as ANTLRWorks[2], Yices theorem prover[8] and Bogor[19] to model and implement the different stages of analyzing a middleware algorithm Alg=$(F^1, \ldots, F^n)$ and identifying function calls in its flows which can be removed. This process can be broadly divided into two steps:

(a) Identify a function call fn in a flow whose post condition is satisfied by the application property and therefore has the potential to be removed

(b) Determine whether fn can be removed without causing deadlocks or affecting the rest of the flow.

The algorithm in section 3.4.1 is written as two functions analyzeFlow and analyzeNode. The function analyzeFlow identifies functions whose post condition can be satisfied by the application property (step (a) in the previous paragraph) and the function analyzeNode analyzes the causal set of fn to determine whether this function can be removed.

To implement analyzeFlow, we use code generated in Java from a parser generator ANTLRWorks[2] to build a call graph of each process type in the algorithm's specification. We then perform a depth first search and use a theorem prover[8] to check whether the application properties satisfies the post conditions of any of the function calls. Once a function call with the potential to be removed has been identified, we model the system in Bogor[19] and perform a state space search to make sure that the application property satisfies the post conditions of all function calls in the causal set of fn.

3.8 Inputs

The analysis process needs three inputs which must be specified by the algorithm writer:
(a) Specification of the algorithm.
(b) Specification of the network topology on which the algorithm will execute.
(c) The application property.

3.8.1 Grammar for inputs

We have specified grammars to write each of these inputs. The grammars was developed using ANTLRWorks which is an ANTLR parser generator. ANTLR[1] ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing actions. ANTLRWorks allows us to specify the grammar, check it for correctness and generate a lexer and a parser in Java.

Following is a brief description of the grammar for each input.

- `Algorithm`: The grammar[Appendix A] for the algorithm allows the algorithm writer to specify the flows in an algorithm which may consist of one or more process types. Each process type has its own functions and message handlers (as described in section 3.2.2). It must have one primary message handler and the algorithm writer may specify secondary message handlers if additional processing of a message is required for certain message types. Each function and message handler in a process type may have pre and post conditions. The algorithm writer can also specify interface variables and global variables.

- `Network Topology`:The algorithm writer can use the grammar [Appendix B] to define the network topology of the system on which the algorithm is going to execute. This input provides three pieces of information:
  (a) The number of nodes in the system being analyzed. The nodes are named and identified by integers.
  (b) The topology of how the nodes are connected to each other. A connection between two nodes represents a one way channel. A bi-directional channel between two nodes 1 and 2 is represented as $1 \rightarrow 2$ and $2 \rightarrow 1$.
  (c) The mapping between process types and nodes. This information tells us which process type is executing on which node.

- `Application Property`:The application property can be specified by the grammar specified in Appendix C. The application properties must be described using interface and global variables with the usual arithmetic, relational and logical operators.

3.9 Processing the inputs

We developed the above grammars in ANTLRWorks and generated a lexers and a parsers in Java for each of them. We have added actions in the parser for *Alg* to build a call graph. Once the call graph has been generated, we perform a depth first search of the call graph and use the Yices theorem prover to find if the application property implies the post condition of any function call in the graph.

If we find a function whose post condition is satisfied by the application property, we use Bogor to check if the post condition of all functions in the causal set of this function is also satisfied by the application property.

3.9.1 Bogor Model

The Bogor model is set up with information provided from all the three inputs. The model has two parts to it. The first part with input from the network topology specification as described in 3.8.1 sets up the nodes, channels and messages. The second part models the process types, its functions and message handlers.

Modeling the algorithm in Bogor.

Each process type from the specification of Alg is modeled as a 'thread' in Bogor. The threads are named with a combination of the process type name and the number identifying the node on which it is executing. This information is provided by the algorithm writer in the topology specification. For example a 'thread' representing a process type called $process$ executing on node identified by n is named as $process\_n$. The threads for our example system 3.1 is named as : $Initializer\_1$,$Responder\_2$ etc. Functions and message handlers of a thread are named with the thread name as a prefix. Therefore a function $func$ of a process type $process$ executing on node n is named as $process\_n\_func$. Figure 3.9.1 shows a snippet of Bogor code for our example system.

Modeling the system topology

Channels and messages are modeled in Bogor as records. Since Bogor does not allow the declaration of global variables to be shared between thread, we pass channels, global and interface variables to the threads as parameters and associated functions as parameters. Declaring channels, global and interface variables as records allows the threads to read and write into them and the changes be seen by other threads in the system. Following is how a channel and message is modeled in Bogor. The channel has been declared with a fixed size of 10 to limit the state space size.

```
record Channel {
        Message[] content;
        int wrap(0, 9) front;
        int wrap (0, 9) rear;
        // false specifies the empty state
        // true specified the non-empty state
        boolean state; }
record Message {
        int from;
        int to;
        int seqNo; }
```

Global and interface variables are also modeled as records. We name the records with the variable name declared in the algorithm specification. The value of the variable is stored in the field called ‘value’ in the record. For example the global variable $booleanphase1Complete$ is modeled as follows in Bogor:

```
record Phase1Complete{
      boolean value; }
```

When a system in Bogor is instantiated, its ‘MAIN’ thread executes first. Here we set up all the channels, create instances of global and interface variable and start the threads representing each process type. Channels are set up as follows : for every $x \rightarrow y$ present in the network topology we create a channel object named c_xy. (3.9.1).

```
active thread MAIN(){
        Channel c_12; //clockwise channel
        Channel c_15; //anti-clockwise channel

        loc init: do {
        c_12 := new Channel;
        c_12.content := new Message[10];

        c_15 := new Channel;
        c_15.content := new Message[10];

        // initialize global values
        s:=new SystemTerm;
        s.value:=false;

        // start thread
        start Initializer_1(c_12,....);
        start Responder_2(c_21,...);
        }


thread Initializer_1(Channel c_12,
Channel c_21,
Channel c_15,
...........,
SystemTerm t) {
```

```
TimeStamp initialized;
TimeStamp lastStateChanged;
Phase1Complete p1;

loc init: do {
             initialized := new TimeStamp;
             initialized.value := 0;
             p1:= new Phase1Complete;
          p1.value:=false;

              } goto phase1;
              loc phase1:
                    invoke Initializer_1_initiatePhase1(c_51,p1,c_51,
                       c_54,c_43,c_32,
                       c_21,t,initialized)
            goto checkState;
        loc checkState:
                     invoke Initializer_1_checkChangeOfState(
                      lastStateChanged,
                      initialized,p1) goto invokephase2;
        loc invokephase2:
                     when p1.value$==$true do {
              }goto phase2;
             when  p1.value$==$false do {} return;
        loc phase2:
            invoke Initializer_1_initiatePhase2(c_15,c_54,c_43,
             c_32, c_21) goto phase1;
```

```
}
```

3.9.2 Model Checking using Bogor

Once we have modeled our system and algorithm in Bogor we now need to find out if a function call fn can be removed. To remove this function we must make sure that the post conditions of all function calls in the $causal(fn)$ is implied by the application property. We model the expression $I_app-> post\_condition(fn)$ using the fun type in Bogor and check it using the $assert$ action. Bogor state space search with check the assertions in every execution possible. If the assert fails, a Bogor state space search will return an inconsistent state.

3.10 Evaluation

The optimization method described in this chapter takes three inputs (figure 1.7) and any optimizations identified depend on these inputs.

- *The application properties*: The communication topology and the application behavior determines the properties which can be identified by the application developers. Typically, one can identify stronger invariants for more structured applications.

- *The algorithm definition*: The optimization method relies heavily on the modularity of the algorithm specification. This modularity is in two levels. First, we want the algorithm developer to break down the tasks of the algorithm into independent subtasks, each of which is achieved by a separate flow. An algorithm definition consisting of multiple flows, each performing a small task, reduces the effort in analysis as we need to analyze a smaller state space. Second, the modularity in defining a flow as a collection of functions and message handlers determines the number of functions we can identify as removable. For example, a process type containing a function which performs many tasks will have a stronger post condition and we will need a strong set of application invariants to optimize a flow containing such a function call. An algorithm is more amenable for optimization when the process types are specified in a modular fashion with smaller functions.

- *The network topology* : The topology of a network often determines the logic in the algorithm definition. For example, the termination detection algorithm for a star network is defined differently from the termination detection algorithm for a bi-directional ring.

Following are a few case studies with different algorithms and application properties. We show that different optimizations which are obtained using different algorithms and application properties. We will describe the analysis effort is each case specifying the depth of the call graphs of each process type and the number of states in the state space search.

3.10.1 Case 1

Consider the termination detection algorithm written for bi-directional rings defined in Figure 3.7. We will use this algorithm with a tele-teaching application which sends messages along the clockwise direction in the ring. Let I1 be the instructor in Figure 3.3. Let the nodes R2, R3, R4 and R5 be the student nodes. A typical tele-teaching application may begin a session with a teaching phase which involves broadcasting audio/visual content followed by a question and answer session. The instructor poses a question which is followed by answers and discussion points made by all the nodes in the system. The instructor poses the next question when all discussion related to the previous one is over. A tele-teaching application may use a termination detection algorithm which is initiated by the instructor node to detect the end of a discussion so that the instructors can proceed to the next question.

Application Property: Since the application does not send any messages in the anti-clockwise direction, the application behavior assures that the anti-clockwise channels are empty. This property can be written as follows:

```
counterClockwiseChan_1.state = "empty" ^ counterClockwiseChan_2.state = "empty" ^
```

`counterClockwiseChan_3.state = "empty" ^ counterClockwiseChan\_4.state = "empty" ^ counterClockwiseChan_5.state = "empty".`

Algorithm : The termination detection algorithm is defined in figure 3.7. We describe how it works in section 3.1.

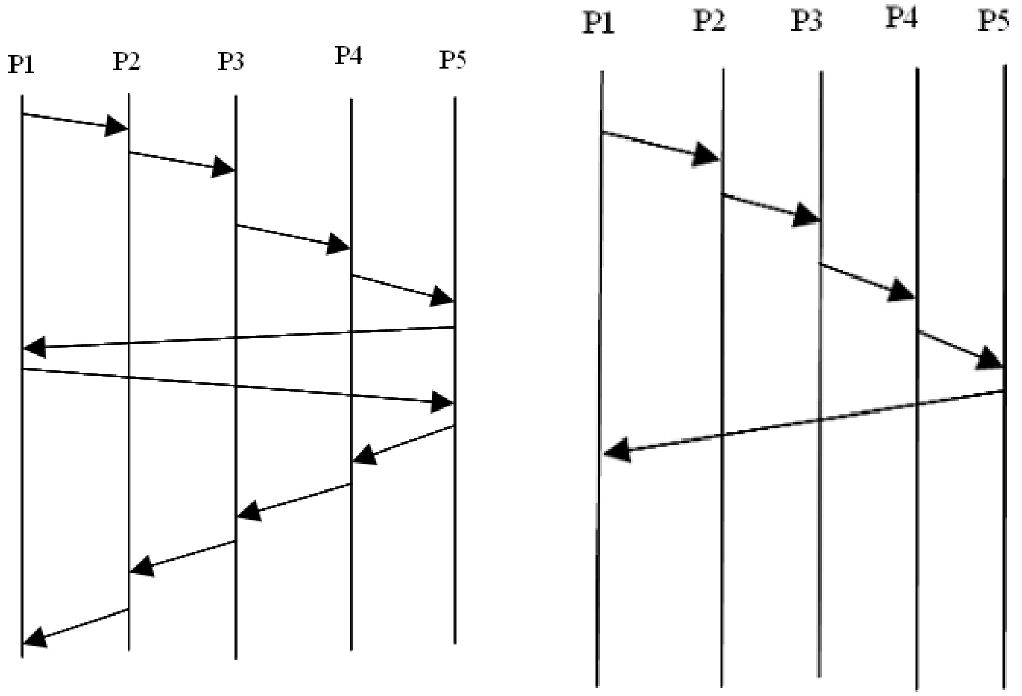
Optimization result: The application property implies the post condition of the method which initiates the second phase of the algorithm. The removal of this function is determined to be safe and its removal will result in no messages being sent in the second phase(Figure 3.10.1). By removing the method initiatePhase2() from the algorithm, we obtain the following optimizations:

(a) To determine termination with the original algorithm, this system would send out 10 messages. The optimized algorithm in turn will send out only 5 messages. Thus we reduce the message exchange by 50 percent.

(b) The optimized algorithm makes fewer function calls. The function initiatePhase2() and all the function calls in its causal set will not be called.

(c) The termination detection algorithm works in two phases. The second phase starts only after successful completion of the first one. If the original algorithm detected termination in time T, the equivalent execution in the optimized algorithm will detect termination in time less than T. This result is especially relevant in real time systems where using an optimized version of the middleware algorithm helps the system meet QOS requirements.

Analysis effort : To obtain the above optimization we have had to analyze the flow which contained the function call that initiates the second phase of the algorithm. The initiator process had a call graph with a depth of 2. The state space search of the flow consisted of 487404 transition, 84036 states searched, a max depth 180 and took a time of 1:48 minutes. The modular specification of the termination detection algorithm plays a crucial role in the optimization process. The algorithm is specified as a two-phase algorithm where the first phase determines whether messages are in transit in the clock-wise direction and the second phase determines messages in transit in the anti-clockwise direction. If the algorithm had been developed as a single phase, it would have been difficult to optimize using an automated process.



(a) Messages in generic algorithm

(b) Messages in optimized algorithm

3.10.2 Case 2

Consider the different termination detection algorithm as specified in D. This algorithm executes on a star network as shown in Figure 3.13. Let the node at the center of the star network I be the initiator node. The initiator node sends marker messages to other nodes in the system (A,B,C,D and E). The other nodes in the system respond to the marker message with their state. Termination is declared when all nodes respond with a passive message.

Application Property: Consider the use of this algorithm with an application which ensures that when the state of node I is passive, node A will also be passive. This property can be written as follows: $I.state = passive \rightarrow A.state = passive$.

Algorithm : The algorithm is specified in D.

Optimization result: The application property implies the post condition of the function call which determines the state of node A. The removal of this function is determined to be safe and its removal will result in no message exchanges between I and A. The message exchange which will be removed in the optimized algorithm is indicated in the figure 3.13. By removing the function call which determines the state of node A, we obtain the following optimizations:

(a) To determine termination with the original algorithm, this system would send out 10 messages. The optimized algorithm in turn will send out 8 messages. Here we have reduced the messages exchanged by 20 percent.

(b) The optimized algorithm makes fewer function calls. The function getState and all the function calls in its causal set will not be called.

(c) We may not save on time in this case. Termination is detected faster only in executions where the events corresponding to the message exchange between I and A happen after the events corresponding to all other message exchanges.

Analysis Effort : To obtain the above optimization we analyzed the instance of the flow which found the state of node A. This flow included the sending of a marker message from I to A and its response. The call graph of the process at I had a maximum depth of 2. The state space search included 786 transitions, 258 states searched, maximum depth 38 and took a time of 0:01 minutes.

3.10.3 Case 3

Consider the case where we use the termination detection algorithm for bi-directional ring from case 1 and the application property in case 2.

Application Property: We will use this algorithm with an application which ensures that when the state of node I is passive, node R2 will also be passive. This property can be written as follows: $I1.state = passive \rightarrow R2.state = passive$.

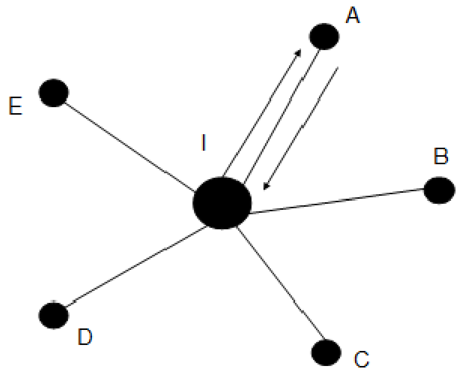


Figure 3.13: *Star Network.*

Algorithm : The algorithm is specified in 3.7.

Optimization result: We are not able to identify any optimizations in this scenario. The logic of the algorithm is written with a specific network topology in mind. I1 must send the marker messages to nodes R3, R4 etc via R2. Therefore message exchange between I1 and R2 cannot be removed in this case.

In this chapter we described a general methodology to obtain a optimized algorithm from a general purpose algorithm based on application properties. We provide grammars for the application developers to specify application properties and for the algorithm writers to write the algorithm. The framework uses these inputs to identify optimization scenarios. This technique is more general in nature when compared to the technique described in chapter 2 and is best suited for algorithms which send out control messages to perform its tasks (for example state gathering algorithms). In the next chapter we show techniques to optimize the algorithm based on the network topology.

Chapter 4

Network Patterns

As described in section 1.1, ‘general purpose’ algorithms which make no assumptions about the underlying network topology often send out more messages than needed for a particular network. Hence, there is an opportunity to save on the number of messages being exchanged. Let G be a graph which describes the network topology for a system. G can be defined as $< V, E >$, where V is the set of nodes in the network and E is the set of links connecting the nodes. A link e $\in E$ can be unidirectional or bidirectional.

In this chapter, we describe the optimizations which can be made to a general purpose distributed algorithm *Alg* based on the network topology(figure 4.1). Our optimizations are based on identifying ‘network patterns’. A network pattern is defined by a combination of a substructure in the network topology G and a pattern in the communication topology of the algorithm *Alg*.

Successful identification of network patterns described in this chapter will reduce the number of messages sent by an algorithm and hence will reduce the state space of the algorithm. Therefore to obtain an algorithm which is optimized with respect to the application *and* the network, we must first optimize with respect to the network topology followed by the optimization with respect to the application as described in the previous chapter.

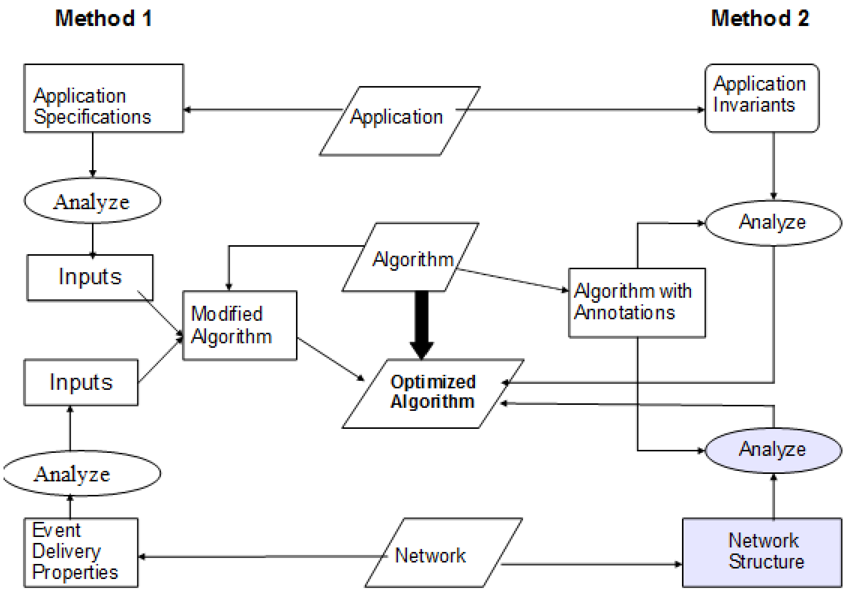


Figure 4.1: *Optimization based on network topology.*

A network pattern contains the following :

(a) A set of nodes $N = n_1, ...n_k$ and a corresponding set of links connecting them $L = < i, j >: i \in N$. The graph represented by $< N, L >$ is a subgraph of G where N is a subset of V and all edges in G between nodes in N are in L. The links in L can either be unidirectional or bi-directional.

(b) A set of of messages $m_1, m_2, .., m_n$ sent by Alg which are part of two or more message chains. The source and the destination nodes of each message in this set belongs to N.

We make the following assumptions about Alg and G :

1) The algorithm Alg is written using rules of our framework defined earlier.

2) Each node in the set V is identified by a unique integer.

4.1 Message forwarding pattern

Basic message forwarding pattern :

In this pattern, we have three processes $i, j, k \in V$ where process i needs to send the same message to processes j and k. The pattern consists of the following :

(a) Set of nodes N = i, j, k , Set of links L = $< i, j >, < j, k >$. The links can be either bidirectional or unidirectional.

(b) Set of messages : $\{mType_1, \ mType_2\}$, where both messages are instances of the same message *mType*. The source node for both the messages is i and the destination nodes for the messages are j and k respectively.

4.1.1 Identifying the pattern:

To identify such a pattern in a system, we need to check the following :

(a) The network topology has two links $< i, j >$ and $< j, k >$, and there is no link between nodes i and k. (b) The code to be executed in process i has the following send statements which execute one after the other.

```
send(mType, j)
send(mType, k)
```

Since there is no direct link existing from process i to process k, the action for the second send statement will be implemented in two steps. First, process i sends msg to j and then process j forwards it to k. Therefore, there will three messages sent in the network links $< i, j >$ and $< j, k >$ when the two sends execute as shown in the first diagram in figure 4.3.

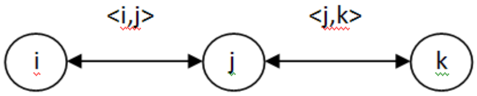
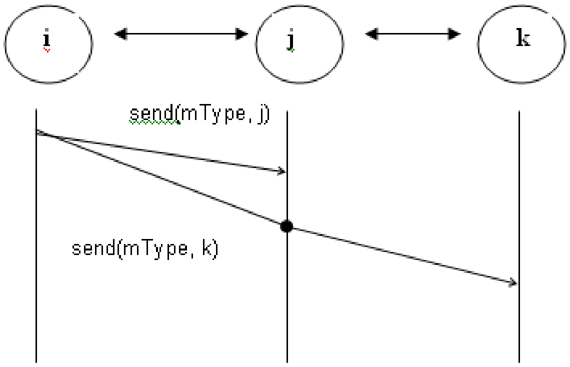
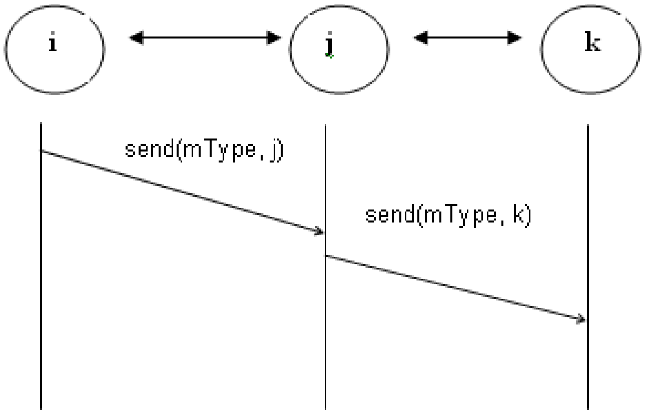


Figure 4.2: *Network connections between i,j,k*



(a) Message taking multiple hops to its destination



(b) Message being forwarded by j

Figure 4.3: *Messages sent before and after optimization*

4.1.2 Optimization transformation

Since both the send statements in this pattern send the same message to j and k and the network topology forces the message destined for process k to pass through process j, we make an optimization where the actions taken for the second send statement can take advantage of the actions taken for the first. That is, we can transform process j to forward a copy of the message it receives from process i to k.

Therefore the optimization made when this pattern is identified is that we remove the second send statement from the code of process i and change the code of the message handler for the message *mType* in j so that it forwards this message to k. The process codes running in process j and k should already have a message handlers for messages of type *mType* since these processes are expecting a message of this type in the original algorithm.

The optimized algorithm will have the following changes :

1. The process code of i will have the statement send(*mType*,*k*) removed

2. The process code of j will have an additional send statement added at the beginning of the message handler which processes *mType* :

   ```
   $pre_j$
   mType_handler(m,processname){
   send(m,k)
   $s_1$;
   ...
   $s_n$;
   }
   $post_j$
   ```

4.2 Extended message forwarding pattern

We can extend the pattern in the previous section to n processes where processes $1, .., n \in V$ are connected in a chain by the edges $< 1, 2 >, .. < i, i+1 >, .., < n-1, n > \in E$. This pattern consists of the following :

(a) Set of nodes N = $1, ...n$, Set of links L = $< 1, 2 >, .., < i, i+1 >, .. < n-1, n >$. The links can be either bidirectional or unidirectional.

(b) Set of messages : $mType_1$, $mType_2$, ... $mType_n$ where all the messages are of the type $mType$. The source node for all the messages is 1 and the destination nodes for the messages are $2, 3, ...n$.

4.2.1 Identifying the pattern:

To identify such a pattern in a system we need to check the following :

(a) The network topology has the links $< 1, 2 >, .. < i, i+1 >, .., < n-1, n >$ connecting the nodes $1, .., n$.

(b) The process code to be executed in process 1 has send statements to send the same message to process 2 to n. The sends are written to execute one following the other.

send($mType$, 2)

...

send($mType$, n)

4.2.2 Optimization transformation

The optimized algorithm will make the following changes :

1. In the code for process 1, it will remove all the statements send(msg,i), where $2 \leq i \leq n$.

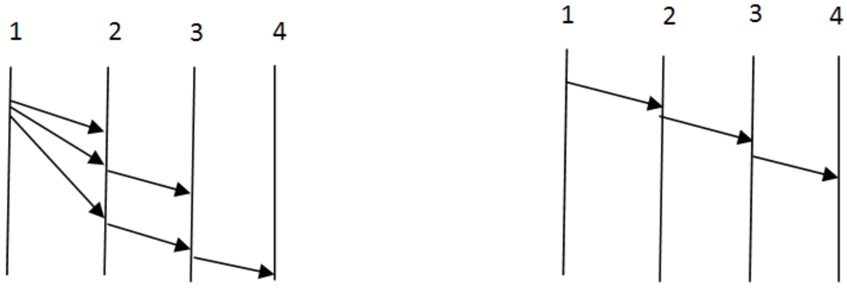


Figure 4.4: *Messages sent before and after optimization*

2. In the code for process 2, it will insert an additional send statement added at the beginning the message handler which processes *mtype* :

   pre_2

   `msg_handler(msg,processname){`

   `send(msg,3)` s_1;

   . . .

   s_n;

   }

   $post_2$

4.2.3 Example

Broadcasting a message in a ring: Consider a network with four processes $V = 1, 2, 3, 4$ and $E = < 1, 2 >, < 2, 3 >, < 3, 4 >, < 4, 1 >$ with the underlying network in shape of a ring where each link is unidirectional. We assume a simple broadcast algorithm which sends a message to all processes in the system, one message at a time. Let process 1 be the broadcasting process. Using the general purpose algorithm, the total number of messages sent for a broadcast will be 6. We can identify the extended message forwarding pattern in this example and by changing the code in all four processes, we get the optimized case where only three messages are sent.

4.3 Request-Reply pattern

In this pattern, we have three processes i,j,k $\in V$ where process i sends the same request message to processes j and k and waits for a reply back from each one of them. The pattern consists of the following :

(a) Set of nodes N = i, j, k , Set of links L = $< i, j >, < j, k >$. The links are bidirectional.

(b) Set of messages : $requestType_1$, $requestType_2$, $replyType_1$, $replyType_2$. The source node for the messages $requestType_1$, $requestType_2$ is i and the destination nodes for them are j and k respectively. The source nodes for messages $replyType_1$, $replyType_2$ are j and k and the destination is i.

Identifying the pattern:

To identify this pattern in a system we need to check the following :

(a) The network topology has two bidirectional links $< i, j >$ and $< j, k >$ between the processes i, j and k.

(b) The code to be executed in process i has the following send and wait statements.

send($requestType$, j)

send($requestType$, k)

wait(exp_j)

wait(exp_k)

(c) The code of j and k have the following send statement in either the message handler for $replyType$ or in any function called by the same message handler:

send(replyType,i)

As with the previous pattern, the absence of a direct link between processes i and k requires that the action for the second send statement and its corresponding reply will be taken in two steps for each. Therefore a total of six messages are sent and received(Fig 4.5).

Optimization transformation

Just as with the previous patterns, we want that the actions to be taken for sending a message from i to k take advantage of the actions taken for sending the message from i to j. In addition to that, in the request-reply pattern, we want to do the same for the replies as well. Therefore, instead of receiving two replies, process i will receive one message from j which contains the reply from j and the reply from k piggybacked on it. Since the process i was not written to receive such a message, we will add a new message handler in i. Similarly, we need to add a message handler in process j since was not written to expect a reply message from k.

Therefore, the optimized algorithm will have the following changes (Figure 4.6):

1. The code of i will have the statement send(*requestType*,k) removed.

2. The process code of j will have an additional send statement added at the beginning of the message handler which processes *requestType* :

   pre_j

   ```
   requestType_handler(msg,processname){
   send(msg,k)
   ```
   s_1;

   $\ldots$

   s_n;

   }

   $post_j$

3. The send(reply,i) in process code of k is replaced with send(replyVector,j). We introduce this new message type whose contents will include a vector which can hold more than one message. When this message is sent from process k to j, it holds the message $replyType_1$ from k

```
replyVector.add(reply)
send(replyVector,$j$)
```

4. We now add a new message handler in process j to processes the replyVector message k will send it. We also introduce a global variables in the process j called reply_from_k to store k's reply.

```
$pre_j$
replyVector_handler(replyVector,processname){
reply_from_k ← replyVector
}
$post_j$
```

5. The statement send(reply,i) in process code of j is replaced with the following :

```
replyVector.add(reply);
wait(reply_from_k != null);
replyVector.add(reply_from_k);
send(replyVector, $i$);
```

6. Next we add a new message handler in process i for the replyVector message. It calls the appropriate message handlers for the messages in the vector just received.

```
$pre_i$
replyVector_handler(replyVector,processname){
reply ← replyVector.get(0)
replyType_handler(reply,$j$)
reply ← reply_piggyback.get(1)
```

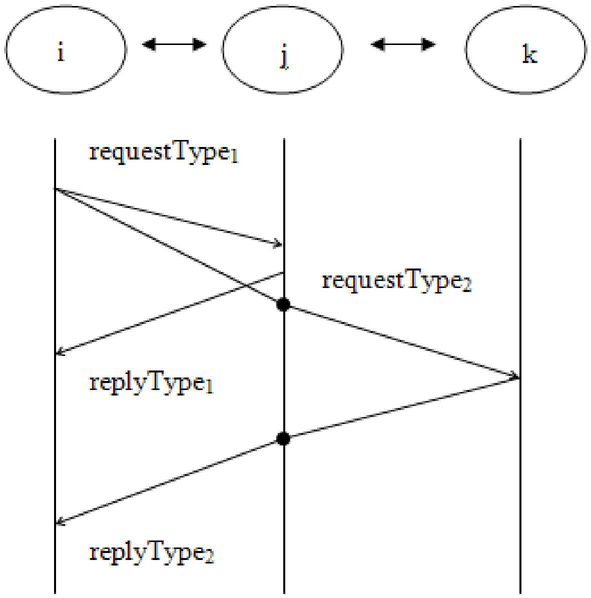


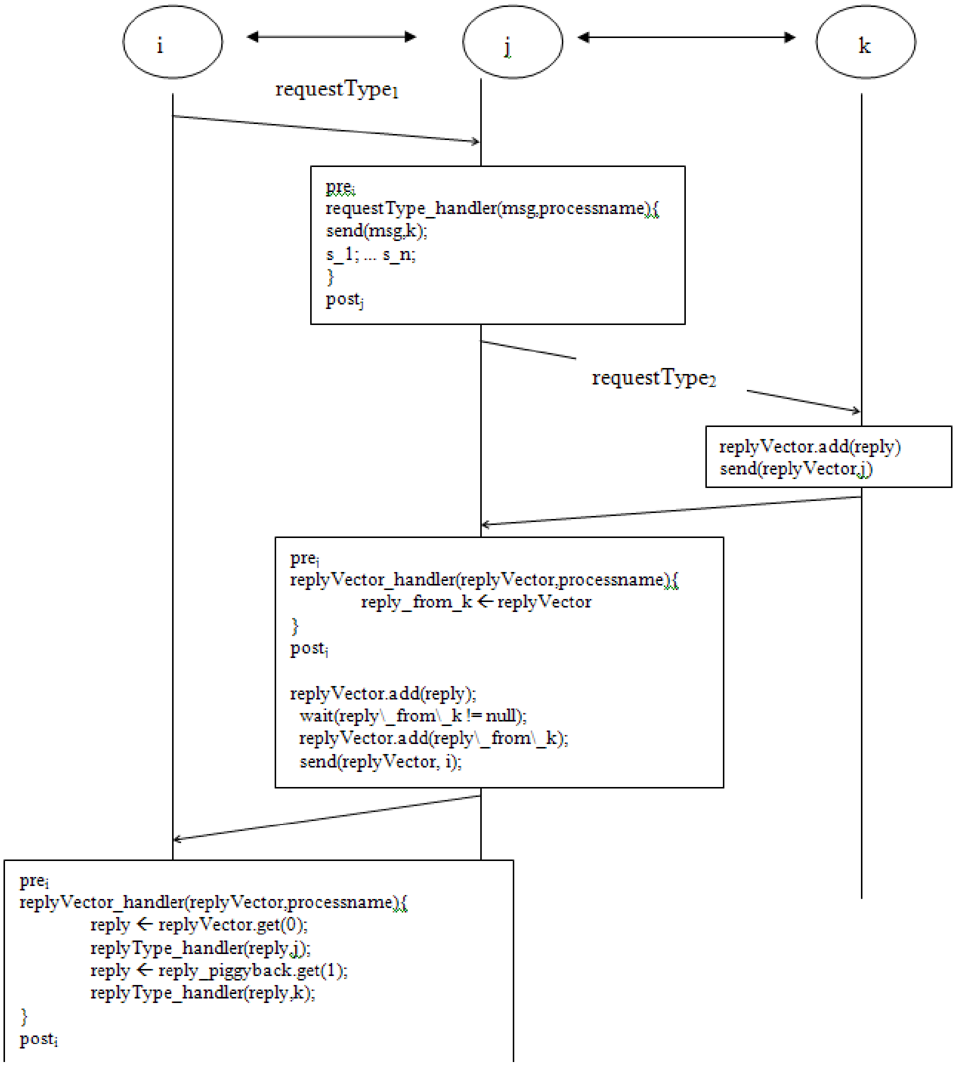
Figure 4.5: *Messages exchanged in Request-Reply pattern*

```
replyType_handler(reply,$k$)
}
```

$post_i$

4.4 Forwarding Post Conditions

As described in the previous sections, when we recognize a forwarding network pattern, we are able to reduce the number of messages being sent out by *Alg* by having a node forward a copy of its message to another node. In the original algorithm, the message sent from i to k would have been received by the network layer at j and then forwarded to k. In the optimized version, we remove the message from i to k and instead have the *algorithm* layer at j forward a copy of its message to k.



```
pre_i
requestType_handler(msg,processname){
send(msg,k);
s_1; ... s_n;
}
post_j
```

```
replyVector.add(reply)
send(replyVector,j)
```

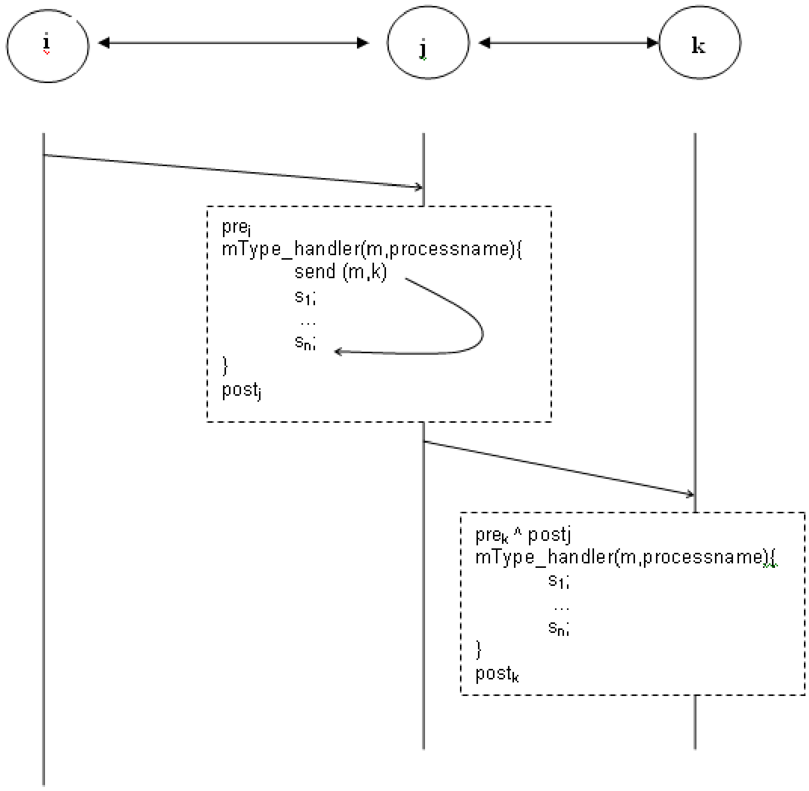
```
pre_j
replyVector_handler(replyVector,processname){
            reply_from_k ← replyVector
}
post_j

replyVector.add(reply);
  wait(reply\_from\_k != null);
  replyVector.add(reply\_from\_k);
  send(replyVector, i);
```

```
pre_i
replyVector_handler(replyVector,processname){
            reply ← replyVector.get(0);
            replyType_handler(reply,j);
            reply ← reply_piggyback.get(1);
            replyType_handler(reply,k);
}
post_i
```

Figure 4.6: *Messages exchanged and code changes made in the optimized Request-Reply pattern*

In the transformation described earlier, we incorporate the forwarding of message [section 4.1] by placing the send statement at the beginning of a message handler (4.1.2), making it the first action taken by the algorithm layer at j when it receives $mType$. This makes the forwarding the message independent of any action taken at j to process $mType$. However, given the format in which an algorithm Alg, we can also place the send statement at the end of the message handler at process j (Figure 4.3). By doing so, we can assert a stronger predicate as a pre-condition of the send statement, and carry over this predicate as the post



```
pre_j
mType_handler(m,processname){
          send (m,k)
          s1;
          ...
          sn;
}
post_j
```

```
pre_k ^ postj
mType_handler(m,processname){
          s1;
          ...
          sn;
}
post_k
```

Figure 4.7: *Forwarding Post Conditions*

conditions established at the message handler at j as shown in figure 4.7. This will result in more information being available for analysis at k which may result in better optimization.

However, if we forward a message at the end of the message handler, we might change the waiting pattern between processes in the system. For example, prior to optimization, assume that process k has a wait statement whose variables are modified by the message handler for mType. Assume that process j has a wait statement whose variables are modified by a message handler processing a message which is sent by k *after* it receives the message *mType*. Therefore, process k will wait for a message from i and process j will wait for a message from j [Figure 4.8, solid arrows]. Let j forward the message to k at the end of its message handler instead of at the beginning. Process k will now wait for message from j instead of i. Hence, we are changing the waiting pattern in the system and will introduce

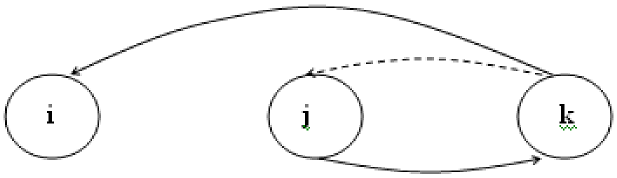


Figure 4.8: *Waiting patterns*

a deadlock because j is waiting for message from k and k is now waiting for a message from j. In this case, the set of messages in a network pattern form two or more message chains. For example, figure 4.3 consists of two message chains, where each chain contains one message. When we perform transformations for this pattern, we are combining the two message chains into one.

Let m_1 and m_2 be the first messages in two message chains. The causal set for sending these two messages causal(send m_1) and causal(send m_2) will contain all the message-send events of the chains except the send event of m_1 and m_2. Let chain(m_1) $= e_1 + e_2., e_k, ..e_n$ where e_1 is the message send event for m_1 and causal(m_1) $= e_2., e_k, ..e_n$
chain(m_2) $= g_1 + g_2...g_m$
e_i, g_i are events as describes in section 3.6.

Executions in the optimized algorithm:

Method 1 : *When send(m_2) is the* first *statement in the message handler of* m_1

With the optimization method described in the previous section, all executions of the the optimized algorithm will have send event of m_2 occur after the send event of m_1 (since it is the first action in the method handler of m_1). Therefore $e_1 \rightarrow g_1$ (where $\rightarrow$ is happens before relation).

Method 2 : *When send(m_2) is the* last *statement in the message handler of* m_1

In this case $e_i \rightarrow g_1$ where $1 < i \leq k$ and e_k is the last event of the message handler of m_1.

In both cases all execution of the optimized algorithm will have a similar execution in the original algorithm. The optimized algorithm will have fewer number of messages sent

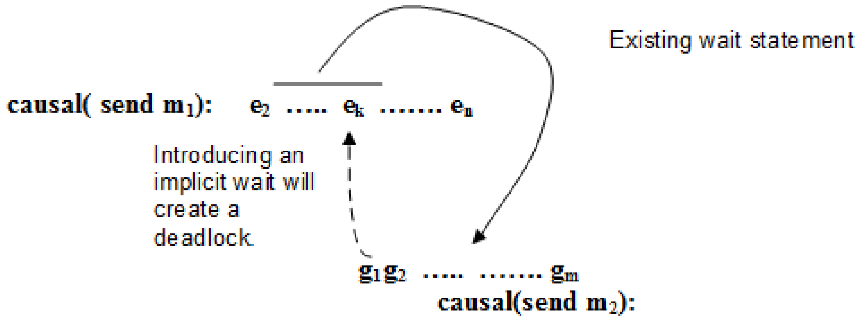


Figure 4.9: *Deadlock scenario*

and we introduce deterministic behavior between a few events as described above.

4.4.1 Preventing deadlocks:

Method 1: The only deterministic behavior introduced in this method is that the event g_1 happens exactly after event e_1. e_1 is the send event of m_1 and therefore there is no possibility of a wait event preceding g_1 in causal(send m_1).

Method 2: The deterministic behavior introduced by using this method is $e_i \rightarrow g_1$ where $1 \leq i \leq k$ and e_k is the last event of the message handler of m_1. This means that the receipt of m_2 can happen only after the last event e_k of the message handler of m_1. This deterministic behaviors can be seen as introducing an implicit wait. If any of the events in $e_1, ...e_k$ is a wait on a message send $\in$ causal(send m_2) then we would have introduced a deadlock because $e_i, 1 < i < k$ is waiting for an event in causal(send m_2) to happen and send m_2 can only happen after e_k.

Therefore when we combine two message chains in the different patterns introduced in this chapter we can use method 2 only if none of the events $e_i, 1 < i < k$ is a wait event on a message being sent in causal(send m_2).

4.5 Additional optimizations enabled by network transformations

As mentioned previously, identification of network patterns and the code transformations must be made before the analysis for optimizations based on application invariants. We find that the identification and optimization based on network patterns has a side effect of introducing deterministic behavior. For example, after optimization shown in figure 4.3(b), m is always received at process j before it is received at process k. This effect will reduce the state space of the algorithm which in turn will reduce the effort in analysis when we optimize the algorithm based on application properties.

Additionally if we are able to forward post condition (figure 4.7) we will be able to make stronger assertions. This will result in more information being available for analysis which may result in more optimizations being identified by the analysis described in chapter 3.

In this chapter we introduced 'network patterns' and showed that we can reduce the number of messages sent out by an algorithm based on the network topology of a system. We can obtain an optimized algorithm with respect to an application and the network by using these techniques in addition to the optimizations described in the previous chapter.

Chapter 5

Conclusion

The software executing on a node in a distributed system can be decomposed into many layers. Each layer is typically developed by a different group of developers and each may use the services provided by the other layers. The top-most layer, where applications execute, use the services of the lower layers, such as the network layer, to send and receive messages. The middleware algorithm layer provides the applications with enhanced services such as detecting termination of the system, access to shared resources using mutual exclusion and ordering of messages/events.

Application developers are typically isolated from the details of lower layers and use the services of the middleware algorithm through published APIs. To increase applicability, middleware developers, in turn, write generic algorithms which can work with any application. The flexibility of 'one algorithm handles all applications' often comes at the cost of performance.

To accommodate systems which may have strict Quality of Service requirements, middleware developers are often forced to write a version of the generic algorithm optimized for a particular application. Such an optimized version takes into consideration the context of the application and how the application uses the middleware algorithm. In this process, the algorithm developers may remove unnecessary computations and message exchanges to provide a more efficient version of the generic algorithm.

Re-writing a distributed algorithm can be a time consuming and error prone effort and the resulting algorithm will work only for one system. At most, it may address a class of systems which share the properties based on which the optimized algorithm was written. In this work, we have proposed an automated solution to this problem which provides considerable ease in terms of effort and exposure to errors.

5.1 Application Properties and Middleware Algorithms

To optimize a middleware algorithm for an application, we need to know the context in which the application will use the algorithm. The application context can be specified in terms of properties relevant to the services of the middleware algorithms. Furthermore, the middleware algorithm itself must be able to use these properties to perform its task more efficiently. Therefore, to be able to automate the optimization process, we need the following:

(a) Application properties

(b) A generic algorithm developed in a way that is amenable to optimizations.

In this thesis, we have proposed two ways to optimize a generic algorithm. In our first method, we analyze the application specification to derive properties relevant to the middleware. Specifically, we developed techniques to identify certain communication patterns which can be used to create input for the middleware algorithm. We designed the middleware algorithm which can configure/optimize its behavior with respect to the input.

In our second method, we do not analyze the application; rather, we require the developer to supply application properties. However, we require the algorithm developer to write generic algorithms in a format amenable to analysis. We provide tools to analyze a generic algorithm based on the application property and identify scenarios where we can reduce computation and messages exchanged by the middleware algorithm.

5.1.1 Optimizing Event ordering Algorithms

In Chapter 2, we have described a way to optimize ordering algorithms used to order events in an anonymous component based system. Such a distributed system consists of components which communicate with each other via events. Events are published and consumed on source and sink ports of a component respectively. This system may use ordering algorithms to enforce ordering requirements (such as causal ordering or total ordering) on events as they are delivered to the components.

General purpose ordering algorithms typically make no assumptions about the structure of an application - that is, they assume that events can be generated at any time. As a result, they behave conservatively and propagate all relevant dependency information. The event delivery system then uses this dependency information to deliver events to a component in the required order. This dependency information can result in large message sizes.

We have shown that we can analyze the application specification and infer pre-existing ordering between events which is introduced by the application. We provide this information to the generic algorithm which then can use it to reduce the dependency information flowing in the system. The steps in this methodology are as follows: We analyze the system specification and build a Port Topology Graph (PTG). This graph has two kinds of edges (a) inter-component edges that link a source port to a sink port and (b) intra-component edges that link a sink port to a source port within a component. An intra-component edge exists when the receipt of an event at a sink port of a component produces the publishing of another event by the component. The PTG is analyzed to produce two tables 'Generation rule table' and a 'Propagation rule table'. The entries to these tables are formed by identifying a 'causal cycle' pattern in the PTG. The causal cycle provides us with information regarding the possible causality between events in the system already enforced by the application. This information is then exploited to reduce the dependency information that is propagated. We demonstrate the use of this methodology for two algorithms: a causal ordering algorithm and a total ordering algorithm. We also show how properties of the underlying middleware can be exploited to optimize the behavior of the algorithms.

The causal order algorithm takes these two tables as input and uses these tables to propagate dependency information in cases when necessary. A general purpose ordering algorithm would propagate information in all cases.

5.1.2 Application Property based Optimizations

In our second approach, instead of analyzing the application specification to deduce properties, we provide rules and grammars for application and middleware algorithm developers to specify the application properties, the middleware algorithm and the topology of the system. We show how to optimize the middleware algorithm based on the application properties and the topology of the system on which it is running.

We have defined a middleware algorithm as a set of non interfering flows where each flow achieves a specific task. A flow is specified as a set of interacting process types which may execute on different nodes of the system. We assume that the algorithm writer annotates each function in a process type with pre and post conditions. We introduced 'interface variables' which can be used to specify application properties. These variable are used to pass information from the application to the middleware layer. They can also be used in the pre- and post-condition annotations by the algorithm developer.

Our analysis method creates an intermediate data structure called the 'call graph' which models the calling structure of functions in a process type. It then identifies redundancies between the application properties and the post conditions of the function calls. When we find a function fn whose post condition is satisfied by the application invariant, we analyze the affects of removing the function. We define a causal set of events which is caused by the function call to fn and remove the call to fn only if (a) no other events in the system is waiting on fn and its causal set, (b) the post conditions of all the function calls in the causal set is satisfied and (c) the affects of the events of the causal set on the global variables of the process type of fn can be determined. The Fig 5.1 shows the steps taken in this process.

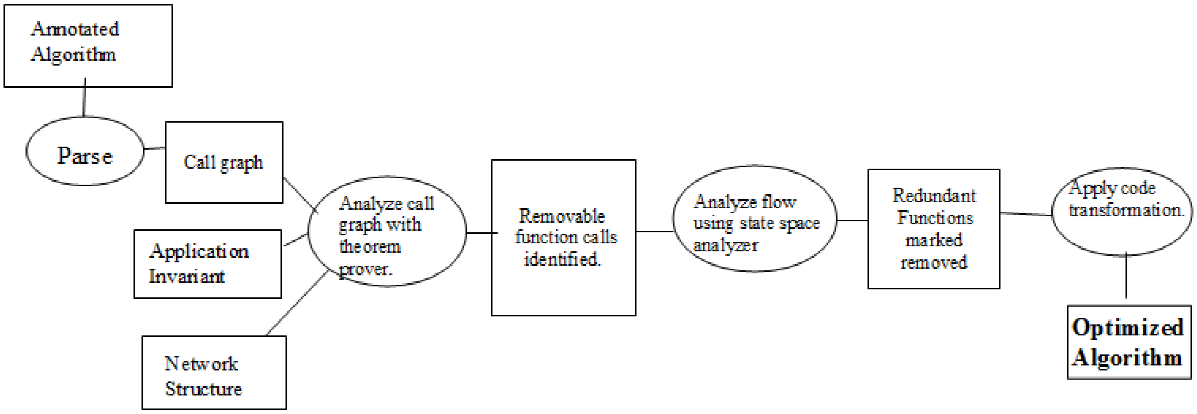

Annotated Algorithm
Parse
Call graph
Application Invariant
Network Structure
Analyze call graph with theorem prover.
Removable function calls identified.
Analyze flow using state space analyzer
Redundant Functions marked removed
Apply code transformation.
Optimized Algorithm

Figure 5.1: *Optimization Process*

We have demonstrated the above analysis process on a system with a ring topology using a bi-directional ring termination detection algorithm. We show that if the application properties specify that messages in the ring flow in only one direction, we can reduce the number of messages exchanged by the termination detection algorithm by half [Fig 3.10.1]. The number of messages exchanged by the optimized algorithm(Fig 3.10.1 (b)) is the same as a uni-directional ring termination detection algorithm.

We also show techniques to optimize the number of messages sent by an algorithm based on the network topology. We describe patterns of message exchanges and show ways to reduce the number of 'copies' of the same message being sent out by forwarding the message between the recipient nodes.

5.2 Limitation and Future work

To get better optimization results, *Alg* must be written in a modular fashion by the algorithm writer. An unstructured algorithm with functions having many 'wait' statements is not amenable to optimization. The annotations to a function are the main inputs to the analysis algorithm. Our analysis relies on the annotation of the functions in two ways :
(a) To determine if the post condition of a function is redundant with respect to the application properties which is the primary condition of removal of a function call. The more

assertions an algorithm writer can make on the values of the global and interface variables on the post conditions of functions, the better the chances of optimization.
(b) To determine the context of function call from its pre-condition and from the application specification. When we analyze a function call to determine the effects of its removal [analyzeNode() in section 3.4.1], we want to make sure that all events $e \in \text{causal}(fn)$ can be predicted.

The analysis process shown in Fig 5.1 has been implemented in three modules. Generation of the call graph, identifying a function which has the potential of being removed and then determining whether the function can be removed. We use Bogor to model and check whether the post conditions of all function calls in causal(fn) are satisfied by the application invariant.

We generate code in BIR grammar for Bogor from the algorithm specification and the network specification. However this code has required editing and modification. As Bogor does not allow global variables the signatures of all function calls have been modified to pass the global variables as parameters to the functions. Further implementation steps can be made to eliminate the manual intervention.

When performing a model check, we do not need to analyze the entire state space of a flow but only part it which begins with the function call to fn with the required parameters (Fig 5.2). We have manually modeled the Bogor code so that the model check begins with the call to fn and give it the context in which it is called. This process can be automated by extending Bogor[20] to find the context of a function call.

We have implemented the optimizations based on the application properties and have specified additional optimizations which can be made based on network topology in Chapter 4. This optimization can be incorporated in our process with extensions to the network topology grammar and adding more steps to our optimization process shown in Fig 5.1. Forwarding post conditions with a message increases the asserts which can be made at the recieving process which may result in increased optimizations.

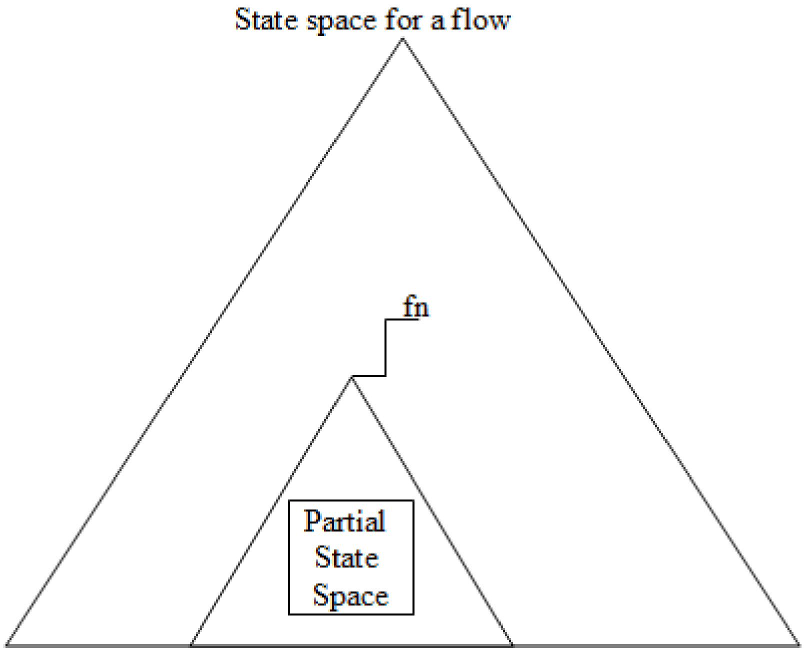


Figure 5.2: *State space of a flow*

Bibliography

[1] Antlr www.antlr.org/.

[2] Antlrworks www.antlr.org/works/.

[3] Berline A and Weise D. Compiling scientific code using partial evaluation. *IEEE Comput*, pages 25–37, 1990.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis a communication subsystem for high availability. *International Symposium on Fault Tolerant Computing*, 1992.

[5] Sandip Bapat and Anish Arora. Message efficient termination detection in wireless sensor networks. *IEEE INFOCOM Workshops*, 2008.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an eventbased infrastructure to develop complex distributed systems. *Proceedings of the 20th international conference on Software engineering*, pages 261–270, 1998.

[7] S. Debray and M. Hermenegildo. The mixtus approach to automatic partial evaluation of full prolog. in logic programming. *Proceedings of the 1990 North American Conference*, page 377398, 1990.

[8] B. Dutertre and L. de Moura. The yices smt solver. tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[9] A. Gokhale, B. Natarajan, D. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons. Cosmic: An mda generative tool for distributed real-time and embedded component middleware and applications. *Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2002.

[10] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. *12th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, 1997.

[11] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. *Proceedings of the 2003 International Conference on Software Engineering*, pages 160 – 173, 2003.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[13] Niel D Jones. An introduction to partial evaluation. *ACM Computing Surveys,Volume 28 Issue 3*, pages 480 – 503, 1996.

[14] Valeriy Kolesnikov and G Singh. Indigo: An infrastructure for optimization of distributed algorithms. In *International Symposium on Parallel and Distributed Computing*, 2008.

[15] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21:845–857, 1995.

[16] A. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Journal Distributed Computing, Volume 11 Issue 2*, pages 91 – 111, 1998.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM Volume 21 Issue 7*, pages 558 – 565, 1978.

[18] R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing. Volume 41 Issue 2*, pages 190 – 204, 1997.

[19] Robby, Matthew B Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. *ESEC/FSE*, 2003.

[20] Robby, Matthew B Dwyer, and John Hatcliff. A flexible framework for creating software model checkers. *Proceedings of the Testing: Academic and Industrial Conference on Practice And Research Techniques*, pages 3 – 22, 2006.

[21] D. Sharp. Avionics product line software architecture flow policies. *Digital Avionics Systems Conference*, 1999.

[22] Gurdip Singh, Valeriy Kolesnikov, and Sanghamitra Das. Methodologies for optimization of distributed algorithms and middleware. *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[23] A. Tanenbaum. *Distributed Systems*. Prentice-Hall, 2006.

Appendix A

Algorithm Grammar

```
grammar DistAlg;

options {output=AST;ASTLabelType=CommonTree;backtrack=true; memorize=true;}

@members {

    public static void main(String[] args) throws Exception {

        DistAlgLexer lex = new DistAlgLexer(new ANTLRFileStream(args[0]));

        CommonTokenStream tokens = new CommonTokenStream(lex);


        DistAlgParser parser = new DistAlgParser(tokens);


        try {

            parser.compilationUnit();

        } catch (RecognitionException e)  {

            e.printStackTrace();

        }

    }

}

compilationUnit

: processTypeWithTrigDecl processTypeDeclaration*

;
```

```
processTypeWithTrigDecl
: 'processType' Identifier  processWithTrigBody
;
processTypeDeclaration
: 'processType' Identifier  processBody
;
typeList
  :   type (',' type)*
  ;
processWithTrigBody
: '{' processWithTrigBodyDeclaration '}'
;
processBody
   : '{' processBodyDeclaration '}'
   ;
processWithTrigBodyDeclaration
: interfaceVarDecl globalVarDecl* annotatedMessageHandler annotatedTrig annotatedFunctio
;
processBodyDeclaration
: interfaceVarDecl globalVarDecl* annotatedMessageHandler annotatedFunction*
;
annotatedMessageHandler
: preCond messageHandlers postCond
;
messageHandlers
    : primaryMsgHandler (secondaryMsgHandler)*
;
```

```
primaryMsgHandler
: ’void’ ’receive’ ’(’ ’message’ Identifier ’,’ ’int’ Identifier ’)’ methodBody
;
secondaryMsgHandler
: ’void’ Identifier ’(’ ’message’ Identifier ’,’ ’int’ Identifier ’)’ methodBody
;
annotatedTrig
: preCond trigFunction postCond
;
trigFunction
: type ’trig’ Identifier methodDeclaratorRest
| ’void’ ’trig’ Identifier voidMethodDeclaratorRest
;
annotatedFunction
: preCond functionDeclaration postCond
;
preCond
   : ’pre:’ condition ’;’
   ;
postCond
: ’post:’ condition ’;’
;
condition
: condOrExp (’->’ condOrExp)*
;
condOrExp
: condAndExp ( ’||’ condAndExp )*
```

```
        ;
condAndExp
:   condEqualExp ( ’&&’ condEqualExp )*
;
condEqualExp
    :   condRelExp ( (’==’ | ’!=’) condRelExp )*
    ;
condRelExp
    :   condAdditiveExp ( relationalOp condAdditiveExp )*
    ;
condAdditiveExp
   :   condMultiplicativeExp ( (’+’ | ’-’) condMultiplicativeExp )*
    ;
condMultiplicativeExp
    :   condUnaryExp ( ( ’*’ | ’/’ | ’%’ ) condUnaryExp )*
    ;
condUnaryExp
    :   ’+’ condUnaryExp
    |   ’-’ condUnaryExp
    |   ’++’ condUnaryExp
    |   ’--’ condUnaryExp
    |   condUnaryExpNotPlusMinus
    ;
condUnaryExpNotPlusMinus
    :   ’˜’ condUnaryExp
    |   ’!’ condUnaryExp
    | primary
    ;
functionDeclaration
:  ’void’ Identifier voidMethodDeclaratorRest
|  type Identifier methodDeclaratorRest
;
interfaceVarDecl
      : ’interface:’ (variableDeclarators)*
;
globalVarDecl
: ’global:’ (variableDeclarators)*
;
methodDeclaratorRest
    :   formalParameters (’[’ ’]’)*
        //(’throws’ qualifiedNameList)?
        (   methodBody
        |   ’;’
        )
    ;
voidMethodDeclaratorRest
    :   formalParameters //(’throws’ qualifiedNameList)?
        (   methodBody
        |   ’;’
        )
    ;
variableDeclarators
    :   type variableDeclarator (’,’ variableDeclarator)* ’;’
    ;
```

```
variableDeclarator
    :   variableDeclaratorId ('=' variableInitializer)?
    ;
variableDeclaratorId
    :   Identifier ('[' ']')*
    ;
variableInitializer
    :   arrayInitializer
    |   expression
    ;
arrayInitializer
    :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '}'
    ;
type
: messageType '[' '3' ']'
| primitiveType ('[' ']')*
        ;
primitiveType
    :   'boolean'
    |   'char'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;
messageType
: 'message'
```

```
;

qualifiedNameList

    :   qualifiedName (',' qualifiedName)*

    ;

formalParameters

    :   '(' formalParameterDecls? ')'

    ;

formalParameterDecls

    :   type formalParameterDeclsRest

    ;

formalParameterDeclsRest

    :   variableDeclaratorId (',' formalParameterDecls)?

    |   '...' variableDeclaratorId

    ;

methodBody

    :   block

    ;

qualifiedName

    :   Identifier ('.' Identifier)*

    ;

literal

    :   integerLiteral

    |   CharacterLiteral

    |   StringLiteral

    |   booleanLiteral

    |   'null'

    ;
```

```
integerLiteral
    :    DecimalLiteral
    ;
booleanLiteral
    :   'true'
    |   'false'
    ;
// STATEMENTS / BLOCKS
block
    :   '{' blockStatement* '}'
    ;
blockStatement
    :   localVariableDeclarationStatement
    |   statement
    ;
localVariableDeclarationStatement
    :    localVariableDeclaration
    ;
localVariableDeclaration
    :   variableDeclarators
    ;
statement
    : block
    |   'if' parExpression statement (options {k=1;}:'else' statement)?
    |   'for' '(' forControl ')' statement
    |   'while' parExpression statement
    |   'do' statement 'while' parExpression ';'
    |   ’repeat’ statement ’until’ parExpression ’;’

    |   ’return’ expression? ’;’

    |   ’break’ Identifier? ’;’

    |   ’continue’ Identifier? ’;’

    |   ’;’

    |   statementExpression ’;’

    |   ’wait’ parExpression  ’;’

    ;

formalParameter

    :   type variableDeclaratorId

    ;

forControl

options {k=3;} // be efficient for common case: for (ID ID : ID) ...

    :   enhancedForControl

    |   forInit? ’;’ expression? ’;’ forUpdate?

    ;

forInit

    :   localVariableDeclaration

    |   expressionList

    ;

enhancedForControl

    :   type Identifier ’:’ expression

    ;

forUpdate

    :   expressionList

    ;
```

```
// EXPRESSIONS

parExpression
    :   '(' expression ')'
    ;
expressionList
    :   expression (',' expression)*
    ;
statementExpression
    :   expression
    ;
constantExpression
    :   expression
    ;
expression
    :   conditionalExpression (assignmentOperator expression)?
    ;
assignmentOperator
    :   '='
    |   '+='
    |   '-='
    |   '*='
    |   '/='
    |   '&='
    |   '|='
    |   '^='
    |   '%='
    ;
conditionalExpression
    :   conditionalOrExpression ( ’?’ expression ’:’ expression )?
    ;
conditionalOrExpression
    :   conditionalAndExpression ( ’||’ conditionalAndExpression )*
    ;
conditionalAndExpression
    :   inclusiveOrExpression ( ’&&’ inclusiveOrExpression )*
    ;
inclusiveOrExpression
    :   exclusiveOrExpression ( ’|’ exclusiveOrExpression )*
    ;
exclusiveOrExpression
    :   andExpression ( ’^’ andExpression )*
    ;
andExpression
    :   equalityExpression ( ’&’ equalityExpression )*
    ;
equalityExpression
    :   relationalExpression ( (’==’ | ’!=’) relationalExpression )*
    ;
relationalExpression
    :   additiveExpression ( relationalOp additiveExpression )*
    ;
relationalOp
    :   (’<’ ’=’)=> t1=’<’ t2=’=’
```

```
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    |   (’>’ ’=’)=> t1=’>’ t2=’=’
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    |   ’<’
    |   ’>’
    ;
additiveExpression
    :   multiplicativeExpression ( (’+’ | ’-’) multiplicativeExpression )*
    ;
multiplicativeExpression
    :   unaryExpression ( ( ’*’ | ’/’ | ’%’ ) unaryExpression )*
    ;
unaryExpression
    :   ’+’ unaryExpression
    |   ’-’ unaryExpression
    |   ’++’ unaryExpression
    |   ’--’ unaryExpression
    |   unaryExpressionNotPlusMinus
    ;
unaryExpressionNotPlusMinus
    :   ’~’ unaryExpression
    |   ’!’ unaryExpression
    | primary
    ;
primary
```

```
    :   parExpression
    |   literal
    |   Identifier ('.' Identifier)* identifierSuffix?
    ;
identifierSuffix
    :    ('[' expression ']')+ // can also be matched by selector, but do here
    |   arguments
    |   '.' explicitGenericInvocation
    ;
creator
     :    createdName (arrayCreatorRest)
     ;
createdName
     : primitiveType
    ;
arrayCreatorRest
    :   '['
        (   ']' ('[' ']')* arrayInitializer
        |   expression ']' ('[' expression ']')* ('[' ']')*
        )
    ;
explicitGenericInvocation
    :   nonWildcardTypeArguments Identifier arguments
    ;
nonWildcardTypeArguments
    :   '<' typeList '>'
    ;
```

```
superSuffix
    :   arguments
    |   ’.’ Identifier arguments?
    ;
arguments
    :   ’(’ expressionList? ’)’
    ;

// LEXER

DecimalLiteral : (’0’ | ’1’..’9’ ’0’..’9’*) IntegerTypeSuffix? ;
fragment
HexDigit : (’0’..’9’|’a’..’f’|’A’..’F’) ;
fragment
IntegerTypeSuffix : (’l’|’L’) ;
CharacterLiteral
    :   ’\’’ ( EscapeSequence | ~(’\’’|’\\’) ) ’\’’
    ;
StringLiteral
    :  ’"’ ( EscapeSequence | ~(’\\’|’"’) )* ’"’
   ;
fragment
EscapeSequence
    :   ’\\’ (’b’|’t’|’n’|’f’|’r’|’\"’|’\’’|’\\’)
    |   UnicodeEscape
    |   OctalEscape
    ;
```

```
fragment
OctalEscape
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;
fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;
Identifier
    :   Letter (Letter|JavaIDDigit)*
    ;
fragment
Letter
    :  '\u0024' |
       '\u0041'..'\u005a' |
       '\u005f' |
       '\u0061'..'\u007a' |
       '\u00c0'..'\u00d6' |
       '\u00d8'..'\u00f6' |
       '\u00f8'..'\u00ff' |
       '\u0100'..'\u1fff' |
       '\u3040'..'\u318f' |
       '\u3300'..'\u337f' |
       '\u3400'..'\u3d2d' |
       '\u4e00'..'\u9fff' |
       '\uf900'..'\ufaff'
    ;
fragment
JavaIDDigit
    :  '\u0030'..'\u0039' |
       '\u0660'..'\u0669' |
       '\u06f0'..'\u06f9' |
       '\u0966'..'\u096f' |
       '\u09e6'..'\u09ef' |
       '\u0a66'..'\u0a6f' |
       '\u0ae6'..'\u0aef' |
       '\u0b66'..'\u0b6f' |
       '\u0be7'..'\u0bef' |
       '\u0c66'..'\u0c6f' |
       '\u0ce6'..'\u0cef' |
       '\u0d66'..'\u0d6f' |
       '\u0e50'..'\u0e59' |
       '\u0ed0'..'\u0ed9' |
       '\u1040'..'\u1049'
   ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
    ;
COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;
LINE_COMMENT
```

```
: '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
;
```

Appendix B

Network Topology Grammar

```
grammar Network;
options {output=AST;ASTLabelType=CommonTree;backtrack=true; memoize=true;}

@members {
    public static void main(String[] args) throws Exception {
        NetworkLexer lex = new NetworkLexer(new ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        NetworkParser parser = new NetworkParser(tokens);

        try {
            parser.system();
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }
}

system
```

```
: 'System' STRING '{' nodeList channelList mappingList '}'
;

nodeList
: 'nodelist' ':' integerLiteral (',' integerLiteral)* ';'
;

channelList
: 'channelList' ':' channel (',' channel)* ';'
;

mappingList
: 'mapping' ':' mapping (',' mapping)* ';'
;

mapping
: integerLiteral ':' STRING
;

channel
: integerLiteral '->' integerLiteral
;

integerLiteral
    :   DecimalLiteral
    ;
```

```
//Lexer
DecimalLiteral : (’0’ | ’1’..’9’ ’0’..’9’*) ;


ID  : (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’0’..’9’|’_’)*
    ;


INT : ’0’..’9’+
    ;


WS  :   ( ’ ’
        | ’\t’
        | ’\r’
        | ’\n’
        ) {$channel=HIDDEN;}
    ;


STRING
    :  ’"’ ( ESC_SEQ | ˜(’\\’|’"’) )* ’"’
    ;


CHAR:  ’\’’ ( ESC_SEQ | ˜(’\’’|’\\’) ) ’\’’
    ;


fragment
HEX_DIGIT : (’0’..’9’|’a’..’f’|’A’..’F’) ;


fragment
```

```
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

Appendix C

Application Properties Grammar

$prop : exp$

exp : $arith_exp$

$|rel_exp$

$|logical_exp;$

$logical\_exp : rel\_exp$

$|\neg logical\_exp$

$|logical\_expANDlogical\_exp$

$|logical\_expORlogical\_exp;$

$rel\_exp : arith\_exp$

$|rel\_exp < rel\_exp$

$|rel\_exp \leq rel\_exp$

$|rel\_exp > rel\_exp$

$|rel\_exp \geq rel\_exp$

$|rel\_exp = rel\_exp$

$|rel\_exp \neq rel\_exp;$

$arith\_exp : var$

$|arith\_exp * arith\_exp$

$|arith\_exp/arith\_exp$

$|arith\_exp + arith\_exp$

$|arith\_exp - arith\_exp;$

$var : IDENTIFIER;$

Appendix D

Termination Detection Algorithm

Termination Detection Algorithm: Initator

```
INTERFACE:
processName $\in$ 1..n
processName.state $\in$ ”active”,”passive”

GLOBAL:
// array of size n where n is the number of nodes in the system
state[n]

trig_detectTermination(n):
terminated = false
repeat
    resetStateInfo()
    getStateofNeighbors()
    terminated $\leftarrow$ checkForTermination()
until terminated = false
return
```

Post: $\forall$ i: *i*.state = ”passive”

```
  getStateofNeighbors(): // n = number of neighbors
  CO j $\leftarrow$ $\{processNamesofneighbors\}$
  getState(j)
  return
Post: $\forall$ i: state[i] = ”passive” $\rightarrow$ i.state = ”passive”

  getState(processName):
  send(marker, processName)
  wait(state[processname] != null)
  return
Post: state[processName] = ”passive” $\rightarrow$ processName.state = ”passive”

  state_handler(state,processName):
  if state = active then
    state[processName] $\leftarrow$ ”active”
  else
    state[processName] $\leftarrow$ ”passive”
  end if
Post: state[processName] = ”passive” $\rightarrow$ processName.state = ”passive”

  boolean checkForTermination(states,n):
  term = true
  for i = $\{processNamesofneighbors\}$ do
    if state[i] = ”active” then
      term $leftarrow$ false
    end if
```

```
 end for
 return term
Post: :

 receive(msg,processname):
 return

 resetStateInfo():
 for i = 0 to $n$ do
   state[i] ← null
 end for
 return
```

Termination Detection Algorithm: Responder

```
 INTERFACE:
 processName ∈ 1..n
 processName.state ∈ "active","passive"

 GLOBAL:
 reply

 trig_marker_handler(marker,processName): // secondary message handler
 if self.state = "passive" then
   reply ← "passive"
 else
   reply ← "active"
 end if
```

```
  send(state(reply),processName)
  return
Post: self.state = passive → reply = passive
```

```
  receive(msg,processName):
  return
```