Trace-based Load Characterization for the Automated Development of Software Performance Models

by

Curtis E. Hrischuk, B. Sc., B. Eng., M. Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

22 May 1998

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis:

"Trace-based Load Characterization for the Automated Development of Software Performance Models"

submitted by Curtis Hrischuk, B. Sci., B.Eng., M. Eng.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Chair, Department of Computer and Systems Engineering

Thesis Supervisor
Professor C. M. Woodside

External Examiner
Professor Karsten Schwan
College of Computing
Georgia Institute of Technology

Carleton University
22 May 1998

Abstract

Performance models of software designs can give early warnings of problems such as resource saturation or excessive delays. However models are seldom used because of the considerable effort needed to construct them. Trace-based Load Characterization (TLC) is a model building technique that gathers the necessary information from an executable design to develop a Layered Queueing Network model in an automated fashion. It uses two formal models for this purpose: the angio trace and proper time. *Proper time* characterizes the execution of a distributed operation by cause-and-effect event relationships, as well as identifying blocking, non-blocking, and synchronization interactions between tasks. A proper time model is a Task and Operation Event Graph (TOEG) which is a new graph grammar. An *angio trace* is an instrumentation specification for generating a TOEG and it is a new, causal logical clock. The LQN *sub-model* is a third model type that is used to assemble several scenarios into an LQN. TLC is appropriate for a message passing distributed system where tasks interact by point-to-point communication. The TLC technique can be applied throughout the software life-cycle, even after deployment. With TLC, the performance analyst can focus on the principles of software performance analysis rather than model building.

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1.0 Introduction

1.1 The Need for Software Performance Engineering and Performance Models

The performance of a computer system is crucial to its use. As computer systems have become larger, geographically distributed, feature rich, and more complex, the "fix the performance issues later" attitude has proven to be too costly (Smith [150] provides several examples). A better approach is to use *Software Performance Engineering* (SPE) to ensure the performance expectations of a computer system are met as it is being designed. SPE is a method for constructing software systems to meet performance objectives.

An SPE analysis can prevent, identify, and solve performance problems by suggesting corrective measures when necessary. It may utilize a combination of experience, performance measurements, or performance models.

A first-cut SPE analysis may use experience or intuition [88]. This approach may be appropriate if something similar to the proposed system already exists and an "expert" is available. However, it is difficult to apply to large systems because of the complexity.

Another analysis approach is to measure performance metrics during an experiment or under normal use [88]. Measurements are considered to be trustworthy since they are taken from the final (or a similar) system. However, even with a small system it is difficult to isolate system variables and application behavior to allow generalization of the measurement data [71]. Extrapolation of measurements is difficult, such as estimating how software will scale. This approach may be impractical due to the system size, expense, or lack of availability. In particular, metrics of a distributed system are difficult to measure because the control and data-flow is spread amongst computing nodes.

The approach that lies behind this research involves constructing a model to characterize the key performance aspects of the system by combining structural information with behavior descriptions. A performance model is evaluated to produce performance predictions that are compared with the performance requirements (e.g., utilization or delay values). Performance models are flexible, providing predictions for varying environmental conditions or design alternatives. So, if the performance requirements are not met then the predictions can help identify the source of the problem and the model can be used to explore potential corrections. If it is a new concept being analyzed, modeling is the only analysis approach available [155].

Performance modeling can be used with the other analysis approaches. An expert may use a model to document the knowledge and experiences they are drawing upon, providing an opportunity for feedback. A model can identify the key parameters to measure. If measurements do detect a problem the model can help to understand the problem. Measurements can also provide feedback for a documented model by validating its predictions.

Performance "modelling, provides a basis for gathering, organizing, evaluation, and understanding information about a computer system [88]."

1.2 Obstacles to Developing a Performance Model

Software performance models are heavily used in SPE but the development of a model is a large undertaking. To develop a model, the analyst must understand the system's behavior by meeting with the developers and examining detailed software descriptions, such as a CASE model, design documents, source code design descriptions, or the implementation source code. The analyst develops a model with this information by identifying: the important end-to-end system behavior, the involved software components, the interactions between these components, the device usage of each component, and then

converting the information into the appropriate model format, such as a simulation model [78], queueing network model [88], or a state-based model like a Petri-Net [149]. For even a moderately sized system there is a large quantity of information to be gathered, examined, comprehended, and ranked for inclusion in the model. Object-based designs are further complicated by polymorphism, inheritance, and class structures. Clearly, the effort needed to construct a single model is substantial.

For SPE to be most successful it begins early in the design process, continuing throughout the software design process, repeating the model development exercise at key milestones. This again multiplies the analyst's effort. For small systems it is possible to maintain the synchronization between the model and design but this is almost impossible for large, industrial software projects which benefit the most from performance analysis.

Another technique is needed for building models, one which relies heavily on automation to ease the effort and provide for better accuracy. The new technique should be able to be used throughout the software life-cycle, tracking the performance predictions of an early design through to the deployed system. Where possible, the model development is automated to best leverage the analyst's effort. That is the goal of this research.

1.3 An Introduction to Trace-based Load Characterization

This research presents a technique suitable for automatically generating a performance model of most message passing distributed systems. The technique is called *Trace-based Load Characterization* (*TLC*) because it uses an execution trace as its input. TLC generates a Layered Queueing Network performance model (LQN) that is well suited to modeling a message passing distributed system. Traces are a low cost source of behavior information because they can be automatically (re)generated from instrumentation. Traces reveal the dynamic details of a design that are difficult to determine from source code or documentation but are important to performance analysis, such as: data dependent

branching, the identity of tasks involved in anonymous or dynamically bound interactions, and the involvement of the polymorphism and inheritance hierarchy of an object-based system.

TLC is intended for a distributed system that can simultaneously execute several distributed operations. A *distributed system* is composed of geographically dispersed, heterogeneous hardware with scheduled, concurrent software objects. We refer to these software components as *tasks*.[1] A *distributed operation* is a set of coordinated interactions between shared system server tasks and user specific tasks. The execution of a distributed operation is called a *scenario* [153]. Tasks communicate solely by messages. Communication channels are reliable, point-to-point, dynamically established at execution, having finite but unpredictable delay. No assumptions are made about the order in which messages are delivered. The communication protocol may be blocking (i.e., Remote Procedure Call) as well as non-blocking (i.e., asynchronous).

A system that executes distributed operations differs somewhat from the classical parallel or concurrent system model, such as described in [143]. First, tasks are resources because they are shared by simultaneous distributed operations. Secondly, a task's lifetime can extend beyond that of a distributed operation. The set of tasks is also dynamic, where tasks may be added or removed. Third, task execution follows a cycle, beginning when a service request message is accepted and ending when the service request is satisfied. DCE RPC [117], CORBA[114], Java [92, 49], and mobile agents [123] are technologies which are being used to build these types of distributed operations.

To apply TLC it must also be assumed that a finite set of scenarios can capture all the important behavioral characteristics of the distributed operation. If TLC is to be used early in the design process, initial resource usage estimates are needed for the individual activities a task may perform.

1. If a process is the unit of concurrency of the operating system, a task is a single-threaded operating system process. If processes can be multi-threaded, the task is a single thread.

TLC uses an executable software design as its primary data source, supplemented by environmental and configuration information.

TLC is a well-formed, traceable model building process. It is well-formed because a chain of formal transformations is defined from the input model domain (an executable design) through to the output model domain (an LQN performance model type). The chain of transformations is developed by working backwards from the output model domain, through one, or more intermediate modeling domains, until the input model domain is reached. The development occurs from the output model domain to the input model domain to identify the information each preceding model domain must preserve. So, the first step in defining the model building process is to formally describe the output model type's properties that must be preserved. The second step is to select a model domain that is close to the output model type, identifying the properties that are preserved. Then, if the selected formal model is not the input model type, step two is repeated. It should be emphasized that the operation of the model building process is the inverse of its development: beginning with an executable design and ending with the performance model.

TLC uses the model types of Figure 1. They are listed in the order in which the model building process operates:

1) *Language statement*: the source code statements of the task's behavior.

2) *Angio trace instrumentation*: It atomically generates and records angio trace event records when language statements are executed. It is an implementation of an angio trace specification.

3) *Angio trace events*: Events that have time stamp values to order the events of a distributed operation by their cause-and-effect relationships.

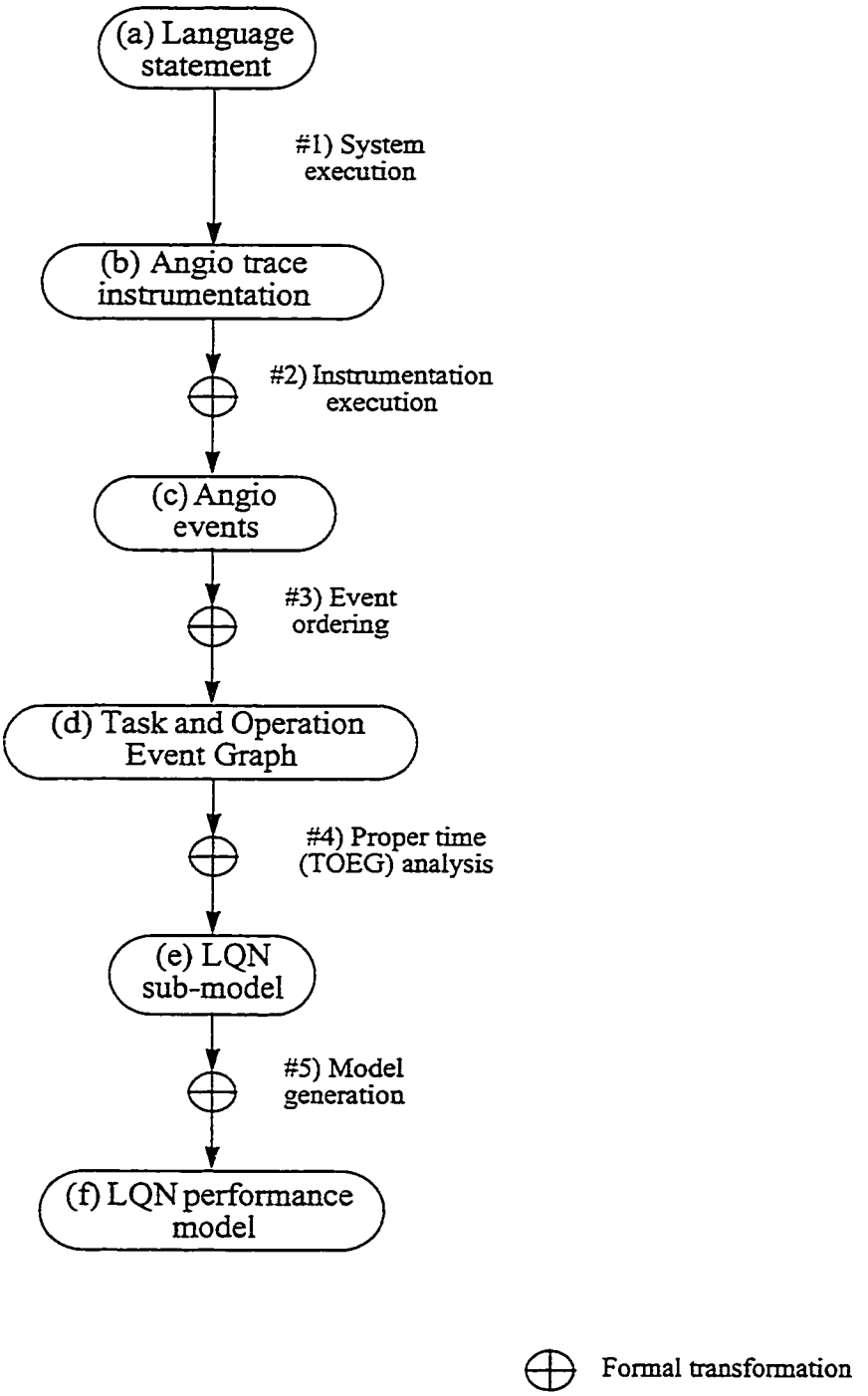


Figure 1: Automated Model Development Strategy

4) *Task and Operation Event Graph*: A graph grammar that can characterize all of the possible distributed operation executions.

5) *LQN sub-model*: A parameterized performance model building block that characterizes the workload of a scenario. This includes the involved tasks, their individual activities, task interactions, and the communication protocol elements.

6) *LQN performance model*: It is the target model type that is being generated. It is a type of queueing network model that is adapted for modelling the interactions of software tasks, including a queue for each task, as well as for each device.

A formal transformation engine is also needed to transform one model type into its successor model type.

Each of these model types and the transformation engine is briefly discussed below, identifying the properties which it characterizes. The order of description is from the output model type to the input model type.

1.4.1 LQN Performance Model

Software performance models of distributed operations describe tasks and their interactions because they affect task blocking, queueing delays at devices and message queues, parallelism, and resource contention. For example, a heavily used task can queue arriving requests and can even become a bottleneck [111]. Blocking task interactions accumulate the nested queueing and service time delays of lower level servers. The Layered Queueing Network (LQN) model has been proposed to study the average performance of these systems [167, 126]. The LQN extends queueing network models to include contention effects for software resources and for hardware devices. TLC incorporates enough detail to construct an LQN.

An LQN predicts the average performance of a distributed system for a given workload. The average predictions include the throughput of devices and software resources, the waiting time at message queues, the waiting time at device queues, the utilization of devices and software resources, and the end-to-end delay for completion of an operation. An LQN can be evaluated analytically or by simulation to produce performance predictions.

This research separates a performance model into a *model template* and a *model experiment*. The model template is a performance model with unassigned parameter and it is the chief product of TLC. A model experiment is a model template with values, or ranges of values, assigned to its parameters. This distinction is important because simulation developers often ignore it since a simulation model is the seamless merging of the model template and model experiment.

An LQN performance model describes the system as the execution of several distributed operations. To model hardware and software resource contention and synchronization delays there is a lot of information to capture for each scenario. A *model template* for each scenario provides this information: the scenario's task architecture, its interactions between tasks, the communication protocol elements of each interaction, the local activities each task performs, and the device demands of each activity. It is possible to merge the model templates for several scenarios into a single model template which assigns probability values to task interactions and averages the device demand values.

A *model experiment* adds information to fill in the model template's parameters. The parameters that can be easily varied include the workload mix, device allocation, device properties (e.g., scheduling), task properties (e.g., priority, replication), and resource management policies (e.g., buffer levels, time-out values). The *workload mix describes*, for each scenario, the initial population of customers and their delay between receiving a reply to a request and submitting another request. Alternatively, a rate of customer arrivals can be defined.

1.4.2 Layered Queueing Network Sub-model

A *Layered Queueing Network sub-model (LQN sub-model)* is a model template that describes the workload for a single scenario so it is the building block of an LQN. It does differ from the model template description given above in one important aspect: it is causal and not probabilistic. This means that an event (activity or task interaction) is assumed to always occur. So, the terms scenario, model template, and LQN sub-model are synonymous. The LQN sub-model can be described as a node-labelled, arc-labelled, directed, acyclic graph.

1.4.3 Proper Time and the Task and Operation Event Graph (TOEG)

The two approaches that have been used to formally characterize the execution of a distributed operation are: a partial order language or a regular expression language. A partial order characterizes concurrency but it is difficult to describe blocking interactions or synchronization between tasks. A regular expression language characterizes blocking and synchronization but it loses information about software structure and concurrency because operations are described by event interleaving. Two regular expression languages are path expressions [26, 8, 21] and flow expressions [146]. This research considers a third formal approach, a graph grammar for characterizing a distributed operation that overcomes the limitations of the other approaches. It is called *proper time*.

Graph grammars are the most general characterization approach, and a wide variety of specification techniques could be described as a graph grammar (e.g. MASCOT, ADA, StateCharts, and actor specifications [120, 70, 48, 72]). They are used to explore the operational characteristics of a design specification. Proper time is different because it is the first graph grammar designed to characterize the execution of distributed operations.

Proper time has labelled nodes that represent event types and labelled, directed edges that are different types of causal relationships between events. A proper time model is a graph called a Task and Operation Event Graph (TOEG). Its semantics characterize the communication protocol elements and task concurrency. Proper time and the TOEG

provide a common format for the trace analysis, at a higher level of abstraction than the angio trace's partial event order. This allows the trace analysis to consist of formal, modular graph rewriting operations.

1.4.4 Angio Tracing and Its Instrumentation

An angio trace overcomes many of the difficulties associated with tracing a distributed system because it takes a different approach. The name "angio trace" is derived by analogy from an angiogram. An angiogram is a visualization of an individual's blood flow that is produced by injecting a radio-opaque dye into the blood stream and taking an X ray of the dye dispersion. Similarly, an angio trace assigns a different dye to each distributed operation so that they can be distinguished. An angio trace event records special time stamp values so that a set of partial ordering relations can be used to reconstruct the cause-and-effect relationships between them. It uses an embedded, software implementation of a global, causal clock to order the recorded events. Angio tracing was introduced in [65] by the author and has been further described in [66].

Angio traces could be extracted from many sources at various steps of the development functional prototypes, executable designs, detailed simulations, or the production system. It has been prototyped in several environments: a functional prototyping environment (MLOG [77]), a commercial prototyping environment (ObjecTime [115]), a distributed software system simulator called Parasol [109], coarse-grained UNIX tasks [67], and in the DCE RPC environment [117] using data collected by the POET debugger [160].

1.4.5 The Transformation Engine

Selecting a transformation formalism was based on the observation that all of the model types can be described as graphs with typed nodes, typed arcs, and rules for connecting nodes with arcs. Each model type is a *graph grammar*, with a syntax and a semantics. The syntax identifies a properly constructed graph model by specifying the structural components (i.e., labelled nodes) and the relationships between those components (i.e.,

labelled arcs). The semantics identify if a graph has a valid meaning in the model domain. A graph grammar may have optional rules for rewriting portions of a graph which can be used to define semantic equivalences of sub-graphs. Using a graph grammar to construct automated transformation engines is a new application of graph grammars.

By describing each of the model types as a graph grammar, the translation of one model type to another can be formally described. The *algorithmic attributed graph grammar* approach was selected as the type of graph grammar to use because it operated at the appropriate level of abstraction and tools were available [136, 105, 106, 36].

1.4.6 Overview Summary

To summarize, the goal of this research is a technique to develop a performance model through a series of formal transformations. The technique is called TLC. It uses three important intermediate models in the development of a Layered Queueing Network model. They are: the angio trace, the TOEG, and the LQN sub-model. In this work, the transformation from one model domain to another is formal and based on graph rewrite operations. TLC addresses several of the concerns and issues with developing a distributed software performance model, which have been highlighted in this chapter. They are summarized in Table 1.

Concern or Issue	TLC's Approach
Early availability of a model.	An angio trace or a TOEG can be generated by the emulation, simulation, or abstract execution of an executable design.
Can be applied to a distributed operation.	Angio tracing and proper time are designed for characterizing the execution of a distributed system.
Accuracy of model development.	Formal transformations are used which reduces errors and enables repeatability.

Table 1: List of Open Issues and Concerns in Performance Model Development

Concern or Issue	TLC's Approach
Defining the structural components of a model.	Captured by an angio trace (section 7.2).
Characterizing the interactions between tasks.	Captured by an angio trace (section 7.2).
Assign service times to the components.	Service times are logically represented if actual values are not available in the trace. The logical service times are converted to actual demands in a later step (section 5.6).
Keeping in step with the design and documentation.	Traces can be regenerated and models rebuilt when appropriate.
Identifying unintended, undocumented interactions.	Captured by an angio trace.
Accurately characterizing the software execution paths even though they depend on system data values.	The effects of data dependencies are recorded by the trace and reflected in the corresponding model parameters.
Object-based complications of polymorphism, inheritance, and class structures.	Captured by an angio trace (section 7.2.1).

Table 1: List of Open Issues and Concerns in Performance Model Development

1.5 The Evolution of TLC

TLC has progressed through three generations of development. The second and third generations constitute the body of this research.

The first generation was an ad-hoc model building technique originally reported by the author in [65, 63]. It characterized the tasks, their interactions, and the concurrency of the distributed operation. However it was limited to systems which solely used an RPC (request-reply) protocol, and it assumed the availability of a global system monitor to record events in the order of occurrence. A partially synchronous interaction (called a *forwarded RPC*) was modeled by the trace analysis. It was sufficient for the types of LQNs that could be evaluated at that time. Angio tracing was introduced but the instrumentation was developed by trial and error. The angio traces were generated from an object-oriented,

functional prototyping environment called MLOG [77]. Trace processing was done in an ad-hoc fashion, using the text processing language AWK [5].

The second generation of TLC overcame the limitations of pure RPC communication and the need for a global monitor [66]. It characterized asynchronous (send-and-continue) interactions, capturing every message and its subsequent workload. The angio trace instrumentation was better specified using a partial order notation. It was based on a logical clock so that it no longer depended on a global monitor to order the events [143, 41, 97]. The trace analysis was attempted using AWK scripts but it proved too unwieldy, so it was rewritten in Prolog [156] using a rule-based approach with task interactions detected using a partial order notation.

The third generation of TLC is reported here. TLC now characterizes task synchronization, completing the LQN task interactions which can be modelled. Characterizing task synchronization also overcomes several of the deficiencies of the second generation of TLC (discussed in Chapter 6.0). Adding synchronization required formalizing the technique to properly ensure an accurate model was being generated, which required development of proper time. Proper time characterizes the cause-and-effect relationships between a distributed operation's events (completeness) so that this characterization is not ambiguous (consistency). The angio trace is now based on proper time, thereby inheriting the completeness and consistency attributes. TLC now uses a graph grammar formalism for the model transformations. The benefits of using a graph grammar are that it provides a single formalism for specifying modular transformations as well as tools to implement and test the transformations. Before a commitment to the graph rewriting approach was made it was explored in [64] and found to be appropriate for the analysis.

1.6 Contributions

The main contribution of this research is the overall development of a formal technique to automatically generate an LQN performance model of a distributed operation provided the assumptions of section 1.3 are met.

There are several other original contributions. They are:

- A formal model to characterize the execution of a distributed operation by cause-and-effect event relationships, as well as identifying blocking, non-blocking, and synchronization interactions between tasks. It is called *proper time*.

- A proof that proper time characterizes all possible execution behavior of a distributed operation (completeness) and that this characterization is not ambiguous (consistency). This proof is by enumeration.

- The development of an instrumentation specification to construct proper time models. This involved the invention of a new, causal logical clock. The instrumentation specification is called *angio tracing*.

- Two strategies to raise the level of abstraction of the execution history of a distributed operation.

A minor contribution is the implementation of the model transformations using the graph grammar research tool PROGRES, which is the one of the largest existing applications of the PROGRES tool.

There are two general questions addressed by this research which have broad implications for the field of distributed computing. They are:

- What formalism captures the causality of a distributed operation's execution for automated analysis?

- How can a distributed operation's execution history be empirically characterized into the representation identified above?

Proper time and angio tracing, respectively, are answers to these questions.

1.7 Dissertation Overview

This dissertation is organized as follows. Chapter 2.0 reviews the state of the current model development research, including a review of research that is associated with the implementation of TLC. Chapter 3.0 and Chapter 4.0 present background material about graph grammars and the LQN performance model. The four intermediate model types that are used in the development of the performance model are described in Chapter 5.0 (LQN sub-model), Chapter 6.0 (Proper Time and the TOEG), and Chapter 7.0 (angio trace specification), and Chapter 8.0 (angio trace instrumentation). Chapter 9.0 is a case study.

Chapter 2.0 Performance Model Development and Associated Research

To provide a context for the discussion, the first part of this chapter is an example performance analysis. Then the second part discusses the different techniques for developing a performance model. The last part describes associated research, namely the monitoring, event ordering, and abstraction of a distributed operation's execution. Many other researchers have investigated these topics and a comparison is given here.

2.1 An SPE Analysis Example

The example performance analysis illustrates the information needed to build a model and the manner in which a model is used. It considers an On-Line Transaction Processing (*OLTP*) system that is similar to Figure 2. The most important performance prediction is the average response time the user sees. Alternative configurations of the software are examined to see how the response time can be lowered.

The example OLTP is based on an industrial transaction processing system developed by a local company, as described by its designers. The OLTP system can have many outstanding user requests which are being serviced simultaneously (e.g. an airline reservation system). It is configured as a distributed system with several multi-processor nodes which may have a local set of disks that are serviced. This description is adapted from [65, 121].

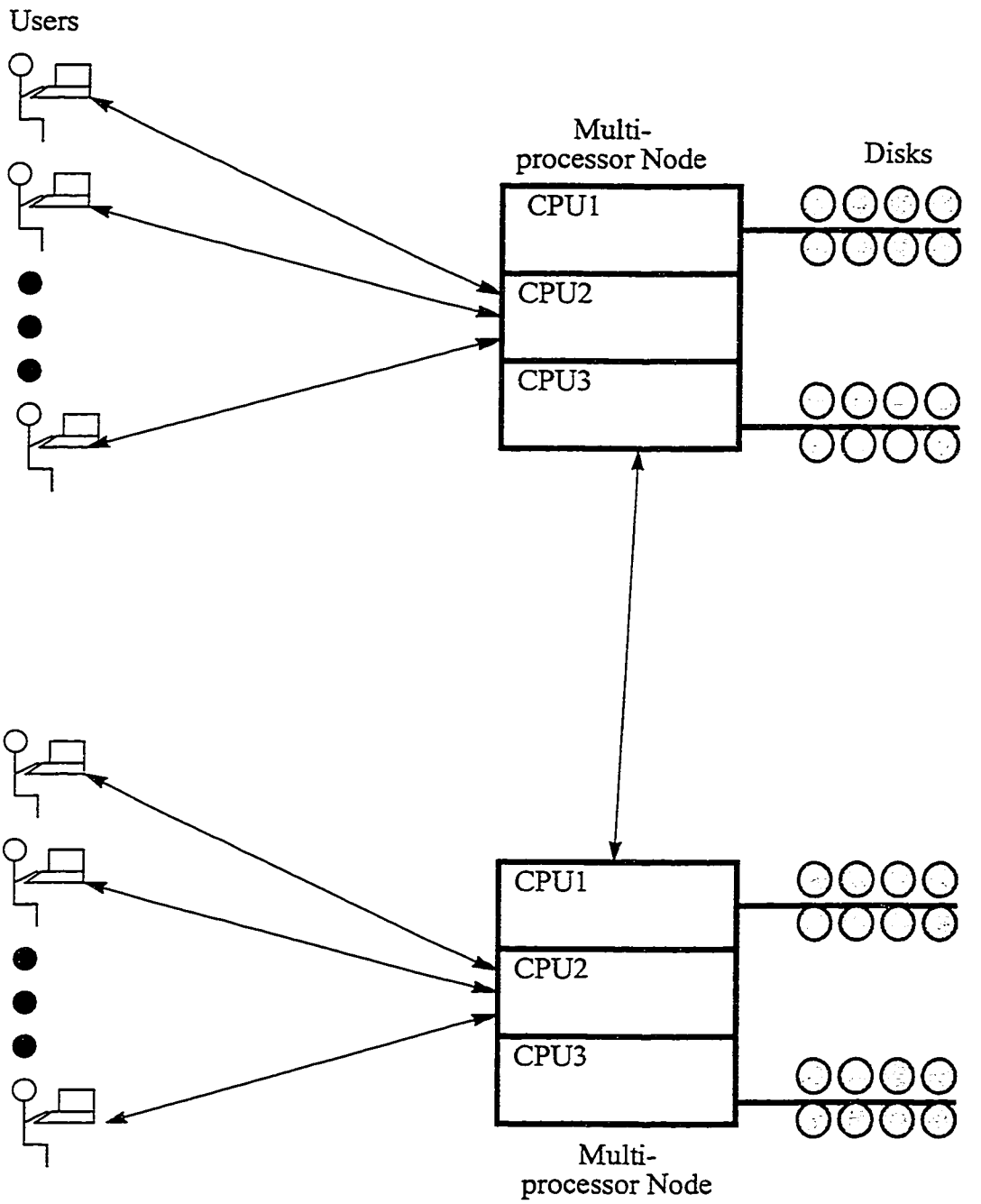


Figure 2: Example OLTP System

2.1.1 Hardware and Software Description

The first place to start when building a model is to identify the hardware and software elements of the system. The example OLTP (Figure 3) hardware consists of two processors, each with a disk array, that are connected by a bus.

The software architecture consists of several types of tasks that each provide a specific service. The tasks communicate solely by message passing and they do not retain state information about a transaction to support fault-tolerance. A pool of software tasks, all of the same type, can be allocated, so that a free server task will accept the first message arriving at the pool and will serve it.

The different task types are shown as parallelograms in Figure 3. They are:

- *Communication Task (CM):* The software driver interface for terminal input and output.

- *Dispatcher (DP):* The Dispatcher is responsible for the validation of user input and the sending of application commands to Application Tasks.

- *Name Server (NS):* Maps a task type identifier to the message queue of an available server task.

- *Application Task (AP):* Takes a user's command and decomposes it into primitive SQL operations on a database. The primitive SQL database operations are sent to an SQL Disk Task for execution.

- *SQL Disk Task (DT):* Translates low level SQL database operation into read and write operations on a particular disk volume.

Unless otherwise noted the tasks use RPC as the communication protocol.

The high-level activities that the tasks use to perform their services are listed in Table 2. This information is useful for estimating and managing the service demands.

Activity Identifier	Description	Estimated CPU Instructions
1) Send	Synchronous message send to another task.	5000
2) Reply	Reply to a received synchronous message.	5000
3) In	Input from user's terminal.	5000
4) Out	Output of formatted data to user's terminal.	5000
5) Async	Asynchronous message send to another task.	5000
6) Rcv	Block until reception of a message from another task.	5000
7) Cvt	Convert user command to an internal message	5000
8) Fmt	Format a screen of data for output to the user.	1000
9) Lkup	Lookup a task's message port address.	2500
10) SqlAdd	Perform an SQL add operation on a table.	12000
11) Sql-Prepare	Preparation for servicing an SQL command.	5000
12) Acc	Access a disk.	1500

Table 2: Example Activities and Resource Functions for an OLTP System

2.1.2 Scenario Behavior

Once the tasks are identified, the way in which they interact in a scenario needs to be understood. In this example, a user can issue two types of commands: 1) a validation command (*VC*), which processes entered data and does not need a database access and 2) an application command (*AC*) which always adds a record to an SQL table that is partitioned across two disks.

The steps in servicing a validation command scenario are listed below, where the sequence numbers in Figure 4 associate the flow of execution with the high-level activities:

VC1: The User issues the validation command ("VC") and waits for a reply.

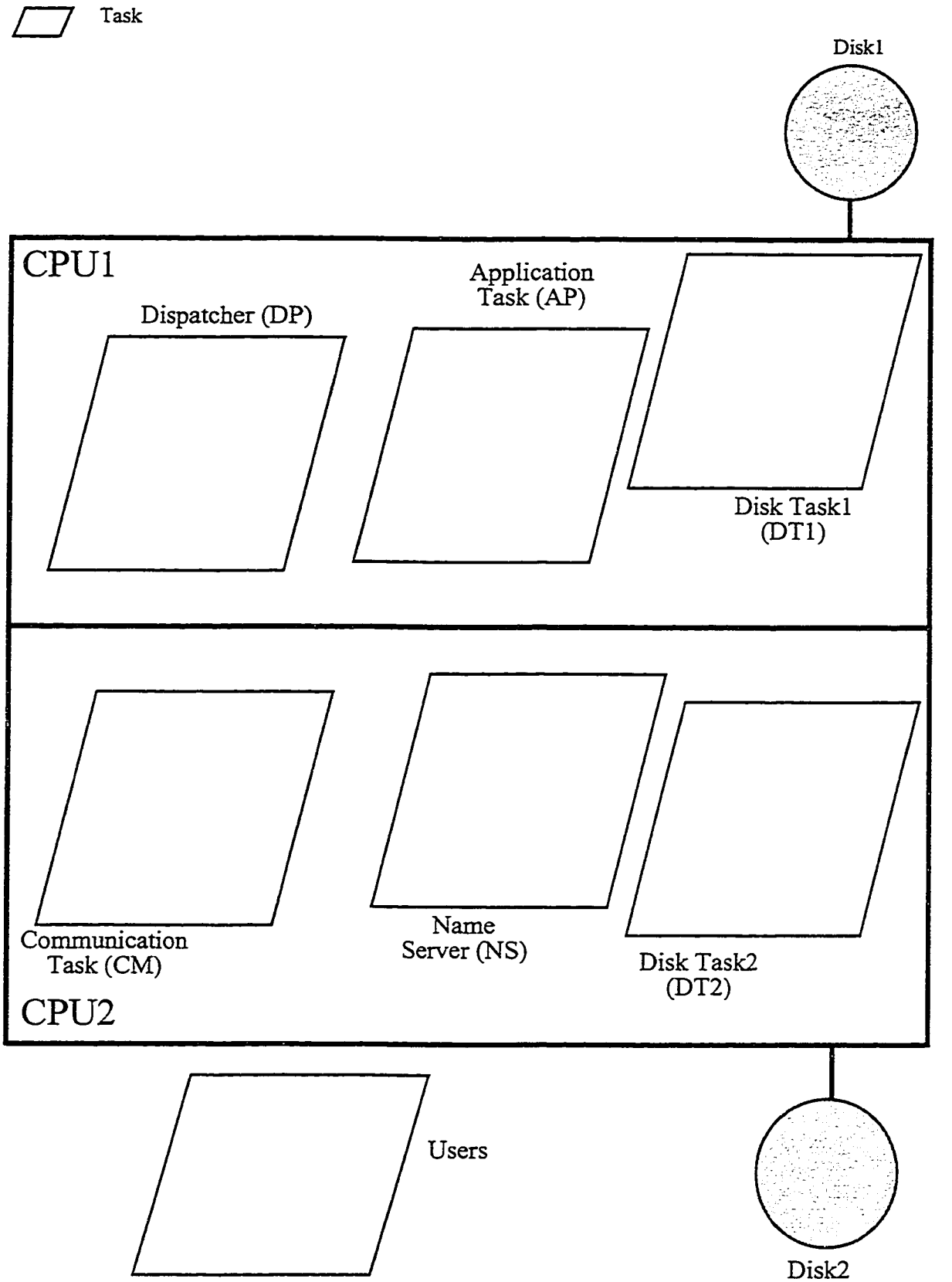


Figure 3: OLTP Configuration

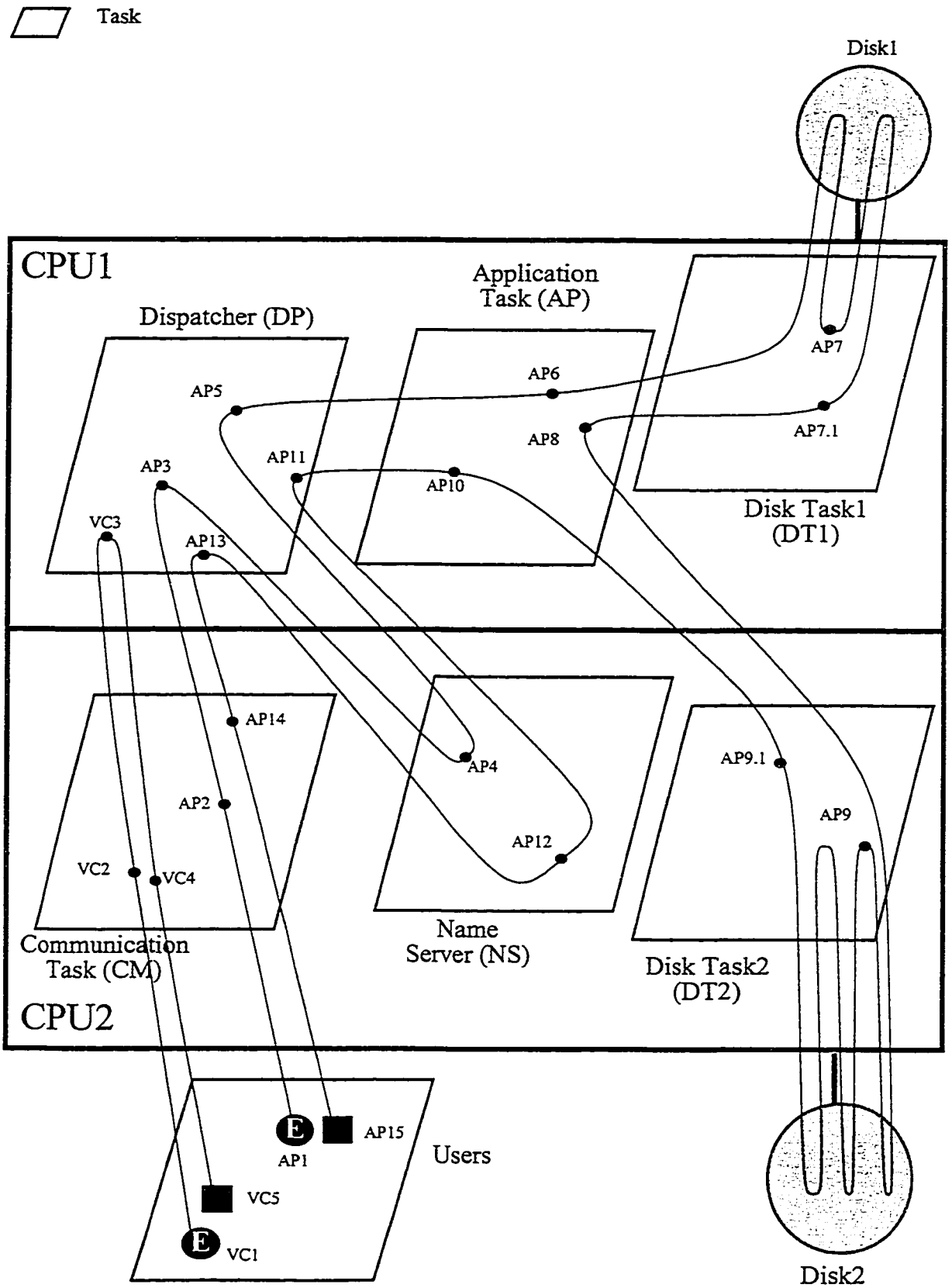


Figure 4: OLTP Scenarios

VC2: The Communication Task receives the validation command from the user's terminal, converts it to a message, sends the message to the Dispatcher, and waits for a reply. The activities (from Table 2) are {3, 7, 1}.

VC3: The Dispatcher receives the message, performs the validation, formats a screen for output to the terminal, and then replies to the Communication Task. The activities are {6, 8, 2}.

VC4: The Communication Task receives the response, and outputs it to the User's terminal. The activities are {6, 4}.

VC5: The User receives the reply which ends the validation command and the scenario. The User would then wait a certain amount of time (its think time) before submitting another request.

The validation scenario has several Remote Procedure Calls. For example, the client task (i.e. User) makes a request to a server task (i.e. CM) and then waits for the reply. Another RPC occurs between the CM and DP which introduces additional delay.

The steps in servicing an application command scenario are:

AP1: The User issues the application command ("AP") and waits for a reply.

AP2: The Communication Task (CM) receives the application command, sends it to the Dispatcher (DP), and waits for a reply. The activities are {3, 7, 1}.

AP3: Using an RPC, the DP requests an Application Task's address from the Name Server (NS). The activities are {6, 1}.

AP4: The NS replies to the DP with the address of an Application Task (AP). The activities are {6, 9, 2}.

AP5: The DP asynchronously sends the command to the AP. The activities are {6, 5}.

AP6: The AP receives the command, sends an SQL operation to SQL Disk Task 1 (DT1), and waits for a result. The activities are {6, 1}.

AP7: DT1 executes the SQL command and replies to the AP with the result. In this case, there are two consecutive disk accesses to "Disk1" recorded in the trace. After replying, DT1 continues execution in preparation for servicing another SQL command (e.g. returns memory to the free pool). This post-reply execution is important for performance and it is represented as another activity *AP7.1*. The activities are: AP7 = {6, 12, 12, 10, 2} and AP7.1 = {11}.

AP8: The AP receives the results from DT1, sends another SQL command to SQL Disk Task 2 (DT2), and then waits for the result. The activities are {6, 1}.

AP9: DT2 makes three consecutive accesses to "Disk2" using the SQL optimizer heuristics, and replies to the Application Task. After replying, DT2 prepares for its next SQL command. This post-reply execution is represented as another activity *AP9.1*. The activities are: AP9 = {6, 12, 12, 12, 10, 2} and AP9.1 = {11}.

AP10: The application command has been satisfied and the AP sends a response back to the DP using asynchronous communication. The activities are {6, 5}.

AP11: The DP requests the address of the CM from the NS. The activities are {6, 1}.

AP12: The NS replies with the CM's address. The activities are {6, 9, 2}.

AP13: The DP sends the response to the CM. The activities are {6, 8, 2}.

AP14: The CM sends the reply to the User. The activities are {6, 4}.

AP15: The User receives its response to the command and the scenario ends.

The descriptions of each scenario describe the workload to the system.

2.1.3 Configuring the Model Parameters

A model template can be constructed using the information that has been gathered and the next step is to assign values to the remaining parameters.

The workload needs to be defined. The effect of the workload is investigated by varying the total number of users from 1 to 200. The workload parameters that are fixed include the think time and ratio of the workload distributed to each scenario. In this experiment, the user has an average delay of a half-second between receiving the response to a request and issuing a request. It is also estimated that, on average, 40% of a user's requests will be an application command while the remaining 60% of the requests are validation commands.

A parameter that can be adjusted for the configuration and is varied in the performance experiments are the number of tasks in the application task pool and disk task pool. These values affect the maximum number of client requests that can be serviced concurrently.

The scenario descriptions identify which activities are performed but the activities must be converted into service times to be part of a performance model. To convert the activity resource demands of Table 2 into a service time, the properties of the devices need to be known. Table 4 shows the resource definitions of the devices used in the model. The resulting resource demands for each activity are shown in Table 3. A CPU service time is calculated by dividing the number of instructions by the processing rate of the CPU (in

2.2 Performance Predictions of the Example OLTP System

The model was solved for a range of parameter values. The parameters that were varied were: the aggregate workload (from 1 to 200 users), the size of a disk task pool (a size of one or four), and the size of an application task pool (a size of one, four, or eight). Some of the performance predictions were graphed and are shown in Figure 5 and Figure 6.

The conclusions that can be drawn from the performance predictions for the user and device are:

- *Average user response time*: It shows that the best configuration is the one with an application task pool size of eight and the disk task pool size of four. It may appear obvious that more is better, but an application task pool size of four only increases the user's delay by 15%, still being under three seconds for the workload of 200 users.

- *Utilization of CPU1*: It varies from 15% (only one application task) to a maximum of 35% (eight application tasks and four disk tasks). The horizontal lines indicate that there is a bottleneck present in the system since the additional workload has no effect.

- *Disk 1 Utilization*: It does not approach full utilization except with the last configuration, where it approaches 90% utilization. This indicates that it will saturate once the current bottleneck is relieved.

These performance predictions would be available from conventional types of queueing network models. An LQN produces performance estimates for the tasks as well as the devices.

The task architecture appears to have a large impact on the performance of this example which is shown in the task performance predictions (Figure 6). The conclusions that can be drawn from the performance predictions for the tasks are:

- *Application task throughput:* The rate of clients serviced by the application task pool increase as the application task pool size increases which means that it is servicing more requests concurrently. The service rate also increases as the disk task pool size increases, which can be attributed to more tasks being available to the application task, allowing more concurrent activities.

- *Application Task Utilization:* The application task pool is a bottleneck because all of the tasks in a pool are always busy, even with a low number of users. The only configuration which is not a bottleneck is the one with a pool size of eight tasks.

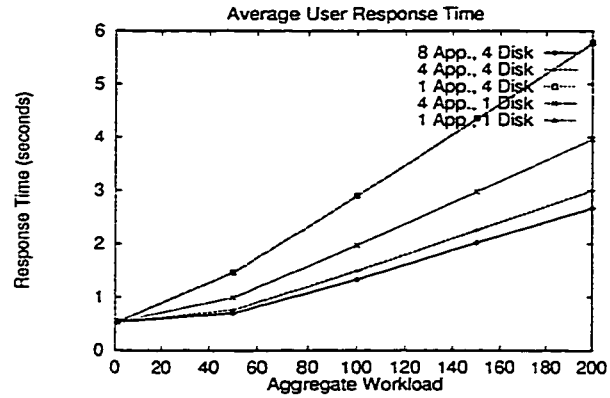
- *Disk Task 1 (DT1) Utilization:* The average number of disk tasks being used is always lower than the number available, so it is not a bottleneck.

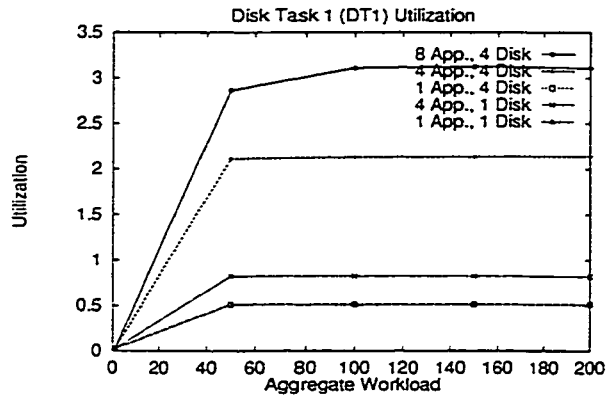
Other performance experiments could be conducted by varying the CPU rate, the service time of an activity, or reallocating tasks to processors.

This concludes the performance analysis example.

2.3 Techniques for Developing a Performance Model

Software performance engineering and performance model development are closely linked. An SPE methodology usually includes at least one model development technique because models are heavily used during an analysis. Models can be categorized by their solution technique which may be simulation or analytical. However, this is misleading because the performance predictions of an analytical model can be produced by simulation.





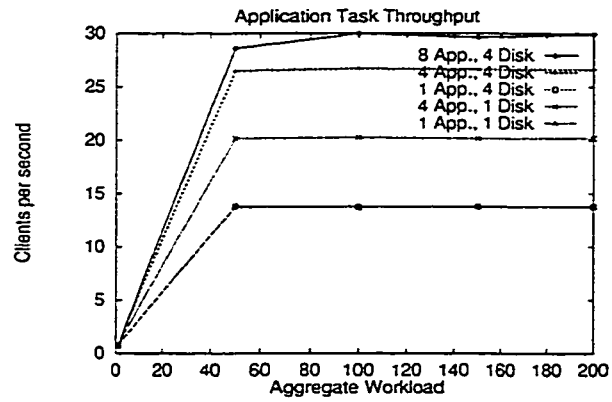
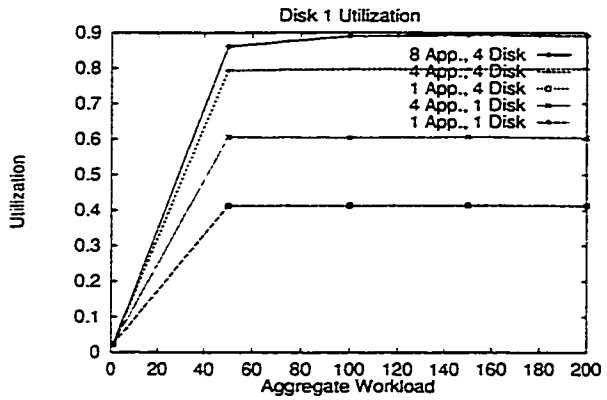
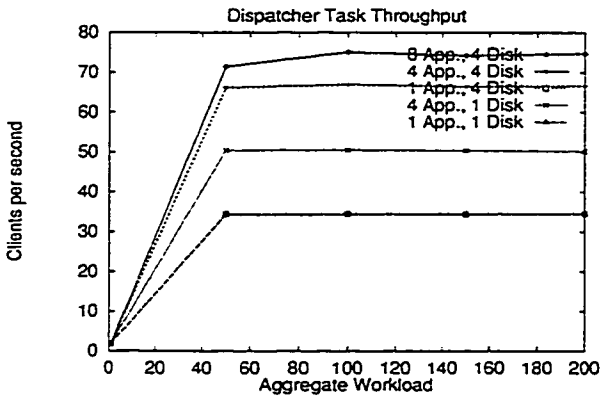


Figure 5: User and Hardware Performance Predictions





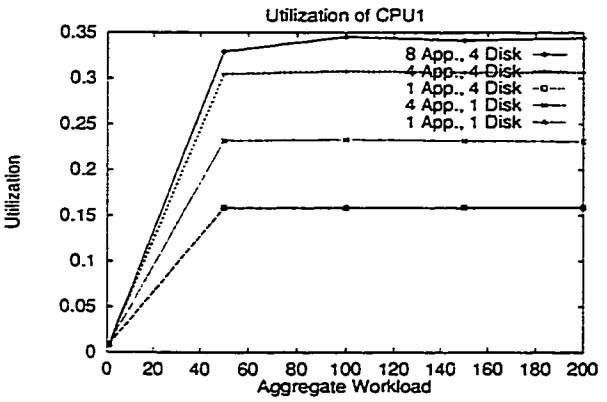


Figure 6: Performance Predictions of the Tasks

Instead, the classification used here considers a model to be executable or abstract. An *executable performance model* includes algorithms which closely mimic an implementation's and it is emulated to produce performance predictions (e.g., a detailed simulation). An *abstract performance model* captures the essence of an implementation sufficient for a mathematical analysis to predict performance.

Some performance model development techniques are similar to a software development process, which is natural since they are modeling software - in fact, executable performance models are themselves software. To distinguish the two some terms are defined here. To maintain the distinction between a performance model and a software design the word *technique* will apply to developing a performance model and the word *process* will apply to software development. The term *design* will be used to refer to software and the term *model* will refer to a performance model.

The several techniques that are discussed here are:

- *Simplistic executable model development* which constructs an executable model that mimics the software's behavior.

- *Evolutionary executable model development* develops an early executable design to use as an early performance model and the executable design is evolved into an implementation.

- *Investigative abstract model development* constructs an abstract model from the analyst's investigation.

- *Hybrid abstract model development* uses execution information as model parameters.

- *Transformational abstract model development* constructs an abstract model using a series of formal transformations.

The last three techniques differ in the manner in which model information is gathered and the amount of effort expended by the analyst.

Each of these modeling technique is discussed in turn.

2.3.1 Executable Performance Models (e.g. Simulation)

An executable model development technique provides a feeling of security because it develops a model that very closely resembles the system being analyzed. An executable model is constructed as a scaled-down version of the implementation, with similar software structure and algorithms. The executable model can be very detailed, even using portions of the implementation's source code [23, 109]. The performance predictions are generated by assigning resource usage to the software components, emulating the model, and measuring the performance as predicted by the emulation. Usually, the large amount of detail in an executable model restricts the evaluation to a simulation technique. If a performance problem is detected and design modifications are identified to improve performance, these modifications can be directly related back to the implementation because of the fine-grain nature of the executable model. The executable model development strategies that are discussed include general simulation modeling, SREM/ SYSREM, functional prototype environments, and MIDAS.

The simplest executable model development technique constructs a simulation that mimics the key executable aspects of the design, including important algorithms. There are many books devoted to constructing a simulation model with a variety of simulation toolkits (e.g., Ptolemy [23], Parasol [109]) or simulation languages (e.g., Simula [82]). A difficulty with this simple approach is that the models are expensive to develop, involving programming, debugging, validation, and maintenance.

Executable performance model development can occur early in the software design process. SYSREM/SREM [6] is one of the oldest software specification methodologies and it uses a simple executable model for performance prediction. It constructs a simulation

model from a library of sample algorithms. The simulation model is used for functional and performance testing of a specification. Each item in the library has a performance budget allocated so that the response time for a scenario can be predicted. Tools reduce the effort needed to construct the simulation model and manage the specification information.

Other executable model development techniques have an evolutionary approach because the software development process is evolutionary. Evolutionary functional prototyping uses an executable design early in the development effort to validate a specification or design approach. Then the executable design is refined into a production system. The executable design can also serve as an executable model by allocating budgeted execution times to the software components and then emulating it to measure its predicted performance. The benefits of this approach are: (1) the performance prediction accuracy increases as more of the design is implemented, (2) a separate model is not developed or maintained, and (3) the model is consistent with the evolving system. Two well known prototyping environments with this capability are: Statemate [54] and the Computer Aided Prototyping System [94].

MIDAS (Methodology for Integrated Design and Simulation) adds a twist to the evolutionary technique by evolving a discrete-event simulation into a deployed system [9]. A MIDAS system combines simulation components with implementation sub-systems so that measurements can be made and used as performance predictions. MIDAS uses an advanced simulation engine to coordinate simulated time and real (wall-clock) time. Checkpointing and rollback of task state information is used to unravel system execution when cause-and-effect relationships are violated, similar to the *time warp* method of discrete-event simulation [73]. However, applying MIDAS to a distributed system is difficult because a global clock, or perfectly synchronized clocks local to each task, are not available [143]. Also, it is not clear how checkpointing and rollback can be applied in a distributed system where several operations execute simultaneously and share server tasks.

The executable model development technique has several drawbacks. Their fine-grain detail can make them costly to evaluate. Performance predictions occur late in the software development process because a nearly complete system definition is needed. When a performance problem is detected it can be difficult to isolate the factors that contribute to the problem or to (efficiently) consider alternative designs or configurations. It can be difficult to gain insight because of the lack of separation between the software design and the performance model.

2.3.2 Abstract Performance Models

An abstract model characterizes the system so that performance predictions are produced by mathematical techniques. To develop an abstract performance model an intermediate *software execution model* is needed. The software execution model characterizes the execution behavior, including model parameters that are assigned values. Once the model is constructed and parameter values assigned, it is evaluated by a toolset to produce performance predictions.

An abstract model has less detail than an executable model, so the performance attributes are not lost in the details. Another benefit is the lower modeling development and execution costs, so that several variant models can be developed for an (in)complete software design or for alternative configurations. However, the accuracy of an abstract model's performance predictions are difficult to assess because the mathematical technique may ignore some aspects of the system, make wide assumptions about the system execution, or trade off accuracy for solution speed. Frequently, the accuracy of the performance predictions are sufficient for analysis purposes. An abstract model can provide performance predictions for very-large systems.

The simplest abstract models ignore software structure, such as queueing network models [88], Petri nets [68], and asymptotic bounds analysis [96]. For example, queueing network models only need the raw, hardware service times for each scenario. These simple models are appropriate for characterizing a centralized computer system with only a couple

of classes of operations. However, a distributed operation's performance model needs to include software structure (see section 1.4.1).

An obstacle to developing an abstract model is the characterization of the software structure and resource usage parameters. The difficulties include:

- *Predicting or measuring the resource parameters can be difficult.* Software execution may depend on system data values which are difficult to predict and are resolved only during execution (e.g., data which affect choices and loops). Typical design documentation does not give information about data, which complicates this step.

- *Abstracting and describing a model's structure can be difficult.* A model's structure utilizes knowledge that is obscured by information hiding. Identifying interacting software components is very difficult if interactions are not predetermined in the source code.

- *Relating the workload to the software execution can be difficult.* This involves selecting a few scenario executions from a huge number that a design is (commonly) capable of.

- *It can be difficult to relate a model optimization to a design modification.* Performance enhancements that are made to the abstract model may be difficult to trace back to the software component and its corresponding modifications.

- *Tracking of models and resource estimates is difficult.* If models are developed at design milestones, it can be difficult to track the changes in the design documentation or between different model versions.

Experience and guidelines for assigning values [150] can be a help in overcoming these difficulties, but this has limitations.

There are three abstract model development techniques that are discussed: investigative, hybrid, and transformational.

2.3.2.1 Investigative Abstract Model Development

Investigative abstract model development has the analyst investigate a software design to produce a software execution model and a performance model. The techniques that are discussed here are: Smith's SPE methodology, the Structure and Performance Specification Method *(sp)*, and the software development methodology of Baker, Chester, and Yeh.

Smith's SPE methodology [150, 151] can be applied throughout the development and life-cycle of a system. It includes a methodology for gathering the necessary data to construct a software execution model, emphasizing meetings with software developers to gather the information. This methodology constructs a parameterized software execution model called an *execution graph*. An execution graph has nodes which are software components and the arcs represent a transfer of control. The variability of service time estimates is captured by nodes having data-dependent parameters and edges having values to represent the probability of selection or loop counts. The data-dependent parameters are assigned values for different execution scenarios which are used to calculate a service time. Graph reduction rules are applied iteratively to aggregate service time values for the model. The type of model that is developed is an extended queueing network model called an *Information Processing Graph* [145]. The Information Processing Graph is then applied to an extended queueing network modeling tool for evaluation. It is not clear how well SPE can be adapted for developing models of distributed operations because it emphasizes execution behavior at the loss of software structure information.

The *Structure and Performance Specification Method (sp)* [163] develops a software execution model that can be used to develop different types of performance models. It characterizes software execution as hierarchically structured operations on abstract data types (i.e. a call-graph). Higher level operations are characterized as a number of invocations of primitive, bottom-level software components. As a result, only the bottom-

level components need to have service time values characterized. Using graph reduction techniques, *complexity functions* are developed that specify hardware demands independent of the environment and workload. The additional information needed to generate a model from an *sp* description are: a workload model to map component service times to actual resources and the hardware device characteristics (e.g., CPU speed). The resulting workload description is then mapped onto a corresponding model type, including some of the software structure information.

Lastly, Baker, Chester, and Yeh [10] have proposed a software development methodology where performance evaluation is a design aid, using performance metrics to choose between alternative designs. The methodology uses both a functional prototype for functional verification and a separate abstract model for performance estimates. The prototype is not used in developing the model. The software execution model is hierarchically structured, describing resource usage in a fashion similar to complexity functions. An extended queueing network model is developed with this information.

Investigative abstract model development requires a large effort to characterize the software execution model. Other abstract model development techniques reduce this effort through automation.

2.3.2.2 Hybrid Abstract Model Development

A *hybrid abstract model development* technique uses detailed implementation execution information to provide model parameter values. It has been used to fine-tune a fully developed, static message-passing scientific computation that follow the SPMD (Single Program Multiple Data) paradigm, where each processor executes the same program with different data sets. Typically, the source code, compiler information, hardware platform description, and trace files are used to produce simulation models for tuning communication or caching performance. The hybrid model development strategies that are discussed are the Modeling Kernel and the Parameter based Performance Prediction Tool.

The Modeling Kernel predicts the speedup of a parallel program's execution time as processors are added [99, 131, 174]. Its purpose is to minimize data transfer cost during the execution of a parallel program. The Modeling Kernel extracts software structure and communication pattern information using parse trees of the source code. Trace files of program execution are analyzed to find iteration counts, branch frequencies, message sizes, and basic-block execution times. The types of performance analysis it performs are a complexity analysis and abstract interpretation. *Complexity analysis* provides asymptotic predictions of performance for an architecture and program size. *Abstract interpretation* provides communication delay measures, including synchronization time.

The Parameter based Performance Prediction Tool (P^3T) is used to understand the impact of a specified data distribution on a scientific parallel program for tuning caching performance. It produces performance predictions for a known, instrumented parallel program and hardware architecture [39]. Execution traces are used to derive loop iteration parameters and frequency information that is fed into a parallelizing compiler to produce performance predictions for the work distribution, the number of message transfers, the transfer times, the amount of data transferred, network contention, and the number of cache misses.

The range of systems that a hybrid model development technique can address is narrow because it assumes source code availability, a static software structure, and homogenous execution platforms which only execute a single program. For this reason, the traces are used to determine resource demands but they pay little attention to software architecture or software resource constraints. These assumptions are reasonable for a SPMD environment but cannot be extended to a heterogenous, distributed system (Multiple Program Multiple Data paradigm).

2.3.2.3 Transformational Abstract Model Development

A *transformational abstract model development* technique uses a series of (semi-) formal transformations to develop an abstract model.

This technique has been used for centralized computer architectures which have logging facilities to record resource usage. An example is reported in [88] where IBM's MVS Resource Measurement Facility uses a software monitor to record and associate service times with major subsystems and monitoring periods. This information can then be manipulated into a model. However, little information (if any) is recorded about software structure which makes it inappropriate for models of distributed operations.

It should be noted, that a transformational model development technique can complement the other model development strategies. For example, it can be used in conjunction with MIDAS to develop abstract models for examining "what if" situations, without modifying the simulation/implementation code.

This research has many points of contact with other research programs which are discussed next.

2.4 Approaches to Monitoring a Distributed Operation

TLC uses traces which are recorded by a monitor. Previous research about monitoring a parallel or distributed system is reviewed here.

A monitor is a tool that is used to observe the activities on a system [71]. Although, there is little research about monitoring distributed operations, there is an expectation that much of the parallel program monitoring research is applicable to distributed operations. However, this sub-section and the next section will show that a distributed operation monitor has a different set of requirements than that of a parallel program monitor. Angio tracing satisfies the requirements to monitor a distributed operation. An introduction to monitoring terminology is provided first, which is taken from [134], [59], and [116].

Monitoring gathers information by recording events about an operation as it executes. An *event* can be defined in two fashions, either as an atomic, instantaneous action or as a

description of an task's activity. The latter definition is used here, since the first definition is more a characteristic of how an event record is atomically stored.

An event record contains information about an task's activity and it consists of at least an event token, a time stamp, and optional event data. The *time stamp* is generated by a monitor and represents the acquisition time of the event record. The set of events is stored as an *event trace*.

Some approaches to specifying the data in an event record are:

- Recording header data in the trace file to describe the fields [46].
- A self-describing trace format [3].
- An entity-relationship like specification [152, 81, 116].
- A trace description language [102, 51, 103, 32].

The approach adopted by angio tracing is to provide an event format which includes the time stamp information while the data is user-defined. The particular data that is of primary concern is a resource usage identifier or service time value.

A monitor can be characterized by the level of interference it imposes. If the monitor requires the use of resources it is said to be *intrusive*. Intrusive monitoring raises the possibility that collecting information to analyze an operation's behavior, can alter that very behavior. If no resources are consumed, the monitor is *nonintrusive*. A nonintrusive monitor has no effect on the order and timing of events.

There are three approaches to implementing a monitor: hardware, software, and hybrid approaches. *Hardware monitoring* is minimally intrusive because it uses dedicated hardware for instrumentation, storing, and processing of the events. Hardware monitoring requires instrumenting the hardware platform on which the software runs; it does not

instrument the software. Relating captured data to software-level execution is difficult because the recorded events are at the hardware level.

Software monitoring is intrusive because it requires instrumenting the source code, system libraries, or compiler. Events are stored in a reserved memory area of each task. The advantages it has over hardware monitoring is that it does not need special-purpose hardware so it is more portable and it presents information at an abstraction level closer to the software.

Hybrid monitoring is a combination of the software and hardware approaches. Typically, hybrid monitors apply a high-resolution, hardware derived global time stamp to event data and pass it along to external sub-systems for processing and storage. The software is instrumented to record software level events.

A monitor can collect information either by sampling or tracing. *Tracing* consists of the reporting of all occurrences of an event within a certain interval of time. Tracing is synchronous with the occurrence of an event; it is performed when all occurrences of an event must be known or when each occurrence of an event must be followed by a certain action. With tracing, the dynamic behavior of the program is abstracted to a sequence of events. *Sampling* is the collection of information at the request of the monitor. Sampling may be asynchronous with the occurrence of an event; it is useful when an immediate reaction to an event is not necessary. Sampling allows only statistical statements about the program behavior. *Profiling* involves collecting execution counts or performing timing at the procedure, statement, or instruction level, using sampling or tracing.

Tracing is performed using a collection of sensors. A *sensor* is a small piece of instrumentation code residing within the program being monitored. Upon being triggered, a sensor generates an event. A sensor is triggered either by a change to the entity it observes or by a request from the monitoring system. If a sensor also contains analysis code, it is

termed an *extended sensor*. Sensors may be individually or collectively enabled or disabled.

Probes are the collection mechanism for sampling. *Probes* sample attribute values and execute asynchronously. A monitor can probe the executing program at any time for an attribute value, which results in the return of the most recent value of that attribute.

2.4.1 Monitoring Programs for Performance

There is a large body of work about performance tuning of parallel programs by monitoring performance. *Metrics* are time-varying functions that characterize some aspect of a program's performance. However, there is little research that addresses performance debugging and tuning of distributed operations, other than the expectation that the lessons learned from parallel programs can be transferred to a distributed operation. For this reason, the material reviewed focuses on the monitoring of parallel programs.

There are many different monitor attributes, such as:

- Instrumentation portability (e.g., PICL [171], ChaosMON [76]),
- Using shadow processors for data collection (e.g., [119]),
- Tools for automated instrumentation before execution (e.g., AIMS [174]) and during execution (e.g. Paradyn [102], Pablo [3]),
- Run-time enabling of data gathering by software switches (e.g., Issos [116]) or by program steering (e.g., Falcon [51]),
- Trace event ordering using precision hardware time stamp (e.g., ZM4/Simple [32], VIZIR [52], MultiKron [53]),
- time stamp compensation for probe intrusion (e.g., AIMS [174]).

Most monitors sample and aggregate measurements using a specified criteria, and then present the resulting metrics visually for analysis (e.g., Paragraph [56]) or to an expert system for evaluation (e.g. W^3 [60]).

There are four requirements that limit the use of parallel program monitoring research to distributed operations. First, hardware or hybrid monitoring of a distributed operation is not possible because of the geographically dispersed environment. The only alternative is to use a software monitor approach.

Secondly, a strategy for minimizing tracing overhead needs to be selected. Parallel program monitors have used several strategies. The simplest strategy is to enable trace sensors at run-time [116]. A more elaborate strategy is the on-line control of the program and monitoring overhead as it executes (e.g., Falcon [51], Paradyn [102], Pablo [3]). This is difficult to apply to a distributed operation because the software components may not be known in advance, making instrumentation adjustments *a priori* impossible. Angio tracing uses a different strategy, where event recording is enabled at execution time by triggering on operations that are identified to be instrumented; other operations which are executing simultaneously do not need to have events recorded.

If distributed operations are to be considered in isolation it is required that tracing be used. However, most parallel program monitors use sampling. Sampling is justified in a parallel programming environment because parallel operations have a static structure and they run in isolation. However, this is not true of distributed operations, where the sampled data values can be attributed to the wrong operation since operations execute simultaneously and share resources.

The last requirement is the need to order the trace events. This last requirement is of special importance and covered in the next section.

2.5 Ordering the Events of a Distributed Operation

There are two reasons why ordering recorded events in a distributed system is difficult. First, a global clock is not available because the processor nodes are geographically dispersed. The second reason is that a perfectly synchronized clock reference is not possible because of poor clock granularity, poor clock synchronization, clock drift, or unpredictable communication delay.

For small distributed systems or certain types of operations these difficulties can be neglected or overcome with special hardware. The simplest approach uses wall-clock time, ignoring clock drift and skew [116]. Another approach supplies a precision hardware clock to all of the nodes [32, 53]. It is also possible to periodically synchronize the real-time clocks of each node and post-process events to compensate for clock drift and skew [46, 173, 174]. However, the only reliable technique to order events in a distributed system is to use a logical clock [143, 41, 97].

A *logical clock* establishes a partial or total ordering on the system events that is the same ordering that occurred at execution. A logical clock has two components: a methodology to time stamp events and an ordering relation that uses the time stamp values to order the events. The most widely used ordering relation is called "happened before" because it orders the events temporally. However, it will be shown in section 6.6.1 that a "happened before" ordering is not very useful for characterizing the execution of a distributed operation.

The next sub-section defines the *happened before* ordering relation and several of its implementations by different types of logical clocks. For a discussion of the implementation mechanics the reader is referred to the referenced articles. For an overview of logical clock types, the survey papers [124] and [143] provide a good starting point.

2.5.1 The Happened Before Temporal Relation

Before describing the happened before ordering relation it is useful to define a *partial order* and introduce some notation.

Definition 2.1 A relation R is a partial ordering if it is reflexive (xRx), transitive ($xRy \wedge yRz \rightarrow xRz$), and antisymmetric ($xRy \wedge yRx \rightarrow x = y$). The ordering is partial, rather than total, because there may exist elements x and y for which neither xRy nor yRx exist.

Any partial ordering relation can be used to order the events in a system and the *happened before* relation is the one that is most frequently applied.

In the classical distributed system paradigm, according to the survey paper of Schwarz and Mattern [143], a distributed system consists of N tasks: $P_1 \ldots P_N$. Let E_i denotes the sequence of events occurring in task P_i (i.e. its history) and $E = E_1 \cup E_2 \ldots E_N$ denotes the set of all events of the distributed system. Since each P_i is strictly sequential, its sequence of events (E_i) are totally ordered by their occurrence and written as $E_i = \{e_{i1}, e_{i2}, e_{i3}, \ldots\}$. Then the *happened before* relation is defined as:

Definition 2.1 The *happened before* relation ($\rightarrow$) is the smallest transitive relation satisfying:

2.1.1) If $e_{ij}, e_{ik} \in E_i$ and $j < k$ then $e_{ij} \rightarrow e_{ik}$ (i.e., it orders events in the task).

2.1.2) If $e_{in} \in E_i$ is a send event and $e_{i'n'} \in E_{i'}$ is the corresponding receive event then $e_{in} \rightarrow e_{i'n'}$ (i.e., it orders the send and receive communication events).

Associated with the happened before relation is the *concurrency relation* (Definition 2.2). The concurrency relation indicates when the happened before relation cannot determine which of two events happened first.

Definition 2.2 The concurrency relation (||) is defined as ($e_{in} \parallel e_{i'n'}$) iff

$\neg(e_{i'n'} \rightarrow e_{in})$ and $\neg(e_{in} \rightarrow e_{i'n'})$.

If ($e_{in} \parallel e_{i'n'}$) holds then e_{in} and $e_{i'n'}$ are said to be *concurrent* which means the events *may* have occurred simultaneously.

A logical clock is constructed by assigning time stamps to every event and using an ordering relation to order them. If event e_{in} happened before event $e_{i'n'}$, then the time stamp of e_{in} (T_{in}) is smaller than the time stamp of $e_{i'n'}$ ($T_{i'n'}$) : that is, $T_{in} < T_{i'n'}$. To generate the time stamp, every task maintains its own *local* logical clock that is advanced using a set of prescribed rules. A task's local clock represents its best approximation to a global logical clock. When tasks send messages, they include information about their local clock. A receiving task uses the included time stamp information to update its local clock. Internal, send, and receive events advance a task's local clock.

The two most common logical clock types are the scalar or vector logical clock, which are both based on the "happened before" ordering relation. A *scalar logical clock* is constructed by each task having a counter [85]. It does not properly characterize concurrency but it can order events. A *vector logical clock* properly characterizes concurrency because each task's logical clock stores information about the other task's logical clocks [41, 97, 124].

To implement a *scalar logical clock* each task has a counter that is incremented with each recorded event [85]. If a message is received that has a larger time stamp than the

receiving task's current counter, the received time stamp replaces the current clock value. The happened before relation is defined as:

Definition 2.3 Scalar logical clock implementation of the happened before relation:

$$e_{in} \rightarrow e_{i'n'} \Leftrightarrow (T_{in} < T_{i'n'})$$

A total ordering of events can be constructed by appending a task's identifier to a time stamp value. Concurrency can only be correctly inferred when $(T_{in} = T_{i'n'})$. This type of logical clock is antisymmetric since $(T_{i'n'} > T_{in})$ can be interpreted as either $(e_{in} \rightarrow e_{i'n'})$ or $(e_{in} \parallel e_{i'n'})$ [157].

A *vector logical clock* characterizes concurrency more precisely than a scalar logical clock [41, 97, 124]. It is constructed by each task maintaining a vector of integers that constitutes its local clock. More formally, each task P_i maintains a vector $vt_i[1...N]$ where: $vt_i[i]$ is the local clock of P_i and $vt_i[i']$ represents task P_i's latest knowledge of task $P_{i'}$'s local time. A time stamp consists of the entire vector vt_i and each message sent from P_i includes the entire vector vt_i.

A full determination of the temporal order of two events can be made by comparing two vector time stamps. If two events e_{in} and $e_{i'n'}$ have time stamp vectors vt_i and $vt_{i'}$, respectively, then the happened before relation is characterized by Definition 2.4. If the tasks and the position of their clock values in the vector are known then Definition 2.4 can be simplified to that of Definition 2.5. Concurrency can be properly determined in both cases using Definition 2.2.

Definition 2.4 The general vector time stamp happened before relation:

$$e_{in} \rightarrow e_{i'n'} \Leftrightarrow \{\forall x; vt_i[x] \leq vt_{i'}[x]\} \wedge \{\exists y; vt_i[y] < vt_{i'}[y]\}$$

Definition 2.5 An efficient vector time stamp happened before relation where x is the vector position of the clock value of task i: $e_{in} \rightarrow e_{i'n'} \Leftrightarrow \{vt_i[x] < vt_{i'}[x]\}$

An implementation difficulty of a vector clock is the size and communication overhead of the time stamp. The vectors must be at least of size N if nothing is known about the computation except the number of tasks [29]. If N is large, the amount of time stamp data associated with each message and event becomes unacceptable. There have been several approaches to reducing this cost.

Singhal and Kshemkalyani [147] reduce the communication bandwidth by only sending vector clock entries that have changed from a message last sent to a receiver. Each task maintains two additional vectors to store the information between interactions. However, the communication channels must be FIFO. In this approach, post-execution analysis is needed to recover the happened-before relation between different messages sent to the same receiver [100].

Fowler and Zwaenepoel give a direct-dependency technique [42] which reduces communication cost by only maintaining the happened before relation for direct interactions. The transitive component of the happened before relation is constructed by post-execution analysis. This allows a task's local clock to be an event counter, and each task also maintains information about the task it directly communicates with for each event. Each message carries with it the sending task's event counter when the message was sent. The information that is recorded for each communication event is the sending task, receiving task, and the appropriate event counters.

Valot [161] suggests that there is a trade-off between space (memory) and time stamp accuracy for the happened before relation. She describes a family of time stamps (called *k-vectors*) that can be tailored for the particular analysis needs. Instead of allocating a position in the vector to a single task, a subset of the available tasks are each assigned a

single position in the vector. The size of the k-vector is the number of subsets chosen. The appropriate selection of vector clock subsets provides better time stamp accuracy for a given vector size. However, *a priori* knowledge of the simultaneous concurrency during the computation's execution is needed for the optimum assignment of a task to a position in the k-vector.

By restricting the class of system or operation that uses a logical clock, the size of the time stamp can be reduced. Meldal, *et. al.* [100] used knowledge of the fixed communication links between tasks to provide a temporal ordering between messages arriving at the same task. This approach is used to determine the temporal ordering of messages arriving at the same process. The cost associated with this technique depends upon the network topology.

Alternatively, if a time stamp's memory size is constrained then the happened before relation can be adjusted. Interval clocks were introduced in [34] to better approximate the happened before relation with a constant time stamp size. An interval clock gives a better result than a scalar clock for the same communication cost. By using a bit array vector value instead of a counter, the happened before relation can be established by post-execution analysis. If only blocking RPC style communication is used then interval clocks describe the happened before relation with no additional post-execution analysis [34].

It is shown in section 6.6.1 that there are limitations when using the happened before relation to analyze the execution of a distributed execution.

2.6 Abstraction of a Distributed System's Event History

A large component of this research is the development of an abstract model from the execution history of a distributed system. There have been other approaches to raising the abstraction level of an execution history. For example, *behavioral abstraction* [14] clusters one or more primitive events into a single abstract event, and filters other primitive events

As discussed in the next chapter, using a graph language overcomes several other shortcomings of both the partial order characterization and a regular expression language. First, they retain little information about software structure because they focus on the system's behavior. Secondly, they do not identify the communication protocol elements which are needed for performance analysis (this is discussed further in section 4.3 and section 6.1). Third, they can only characterize a temporal ordering of execution and not a causal ordering - a difference that is discussed in section 6.6.1. Lastly, although each provides a means to specify abstract events, neither is a formalism for manipulating a complete set of events.

Chapter 3.0 Introduction to Algorithmic Graph Grammars and Graph Rewriting

Graph grammars and graph rewrite systems were introduced over 25 years ago [122, 133]. TLC uses a graph grammar to implement the two main transformations of Figure 1, based on an algorithmic attributed graph grammar approach.[1] This chapter describes those aspects of this approach which are important for this research, including a graph rewriting tool which supports it. The reader is referred to [136, 105, 106, 36] for more details about the theory. Tutorials and overviews of other types of graph grammars are found in [37, 35, 38, 128, 129]. There are many applications of graph grammars and they are not covered here for reasons of space.

3.1 Introduction to the Algorithmic Attributed Graph Grammar

Three items are used to exploit a graph grammar: the graph grammar itself, a graph rewrite system which implements the graph grammar, and a methodology for the use of both. The *graph grammar* is a specification of the family of graphs that can be constructed from a given graph instance (called the *host graph*). A *graph rewrite system* is a tool that implements a graph grammar as a collection of rules to rewrite an input graph, including an algorithm for selecting the rules to be applied. The *methodology* guides the development effort. These three components are available for the algorithmic attributed graph grammar approach which is implemented by the PROgrammed Graph REwriting System language (*PROGRES*) and its corresponding toolset [136, 138, 137, 141, 107].

TLC uses the algorithmic attributed graph grammar because it is the approach best suited to model transformation. Using this approach, a complex query is formulated as a

1. Another name for this approach is the set-theoretic graph grammar approach.

host graph traversal and a complex graph update operation as a *graph rewrite rule*. An important feature of the algorithmic approach that other graph grammar approaches do not have is that the selection of graph rewrite rules can be made by a complex algorithm, which itself may have nested algorithms and graph rewrite rules. This enables a model transformation to be constructed as a set of independent, declarative graph rewrite rules.

The components of the algorithmic attributed graph grammar are typed nodes, labelled directed edges, a graph schema, the graph rewrite rules, and the algorithms that are used to select and apply graph rewrite rules.

The primitive elements of a graph are the node and the edge. A node is considered to be a data structure that stores application information as attributes. A directed, labelled edge represents a binary relationship between the source and target nodes, with the edge label identifying the type of the relationship. Edges do not have attributes.

Application domain knowledge is captured as a specification that is also a graph, called a *graph schema*. The graph schema describes how a host graph can be constructed because it identifies what node-types and edge types can be connected and the direction of the edge relationships between the nodes. It is very similar to a database schema defined in a binary Entity Relationship notation or object-oriented analysis class diagram.

The two main graph rewriting mechanisms are the production and the transaction. The *production* is the most primitive graph rewrite operation. It modifies a fragment of the host graph. A *transaction* selects and applies one or more productions to rewrite a host graph. A transaction is atomic, either succeeding in entirety or leaving the host graph in its initial state. A transaction is made up of imperative control statements and declarative rules, to specify, select, and apply a set of graph rewrite rules.

Each of these items is discussed in the following sub-sections, as well as a methodology for building applications using this graph grammar approach.

3.2 A PROGRES Graph Schema

A graph schema is the basis of a PROGRES specification because it defines the node types that may be connected by each labelled edge, as well as node attributes. The attributes of a node type or class are not shown in the graph schema figures for clarity.

A node's attributes serve as local variables whose values are stored intrinsically or computed at run-time. The built-in PROGRES attribute types are boolean, integer, and string; other attribute types can be imported from an external source library. An *intrinsic attribute* can be read and set during a graph modification. A *derived attribute's* value is specified as a function which is calculated at run-time, whose parameters are the structure of the graph and the attribute value(s) of any node(s) in the graph. Derived attribute evaluation follows a lazy strategy, where read access of a derived attribute forces all attributes they depend on to be evaluated [80]. Once a derived attribute is evaluated its value is stored. As long as the attribute is neither directly nor indirectly affected by modifications of the graph, the value remains valid.

In PROGRES, every node belongs to exactly one *node type* and each node type itself has a type, a so-called *node class*. The node type or node class attributes are declared in its type or class definition.

A *node type* declaration identifies an attribute's initialization value, as well as the manner in which derived attributes are (re)computed. A node type is a first order value, so it can be a used as a parameter (e.g, in a function or a production) and it can be assigned as an attribute.

A *node class* declaration is used for node type coercion. Node types with common properties have a common superclass. A node type can inherit the attributes of several node classes using multiple inheritance.

Using this example graph schema, a distributed system with two tasks and two processors is shown in Figure 8. The processors communicate bidirectionally. Each task executes on its own processor. The tasks interact using a client-server paradigm, with task A initiating an RPC to server task B. A more complex system could be described by adding more tasks with other interactions, adding more processors, or altering the communication network structure (e.g., a ring, hypercube).

The discussion now turns to how graph manipulations are specified by rewrite rules.

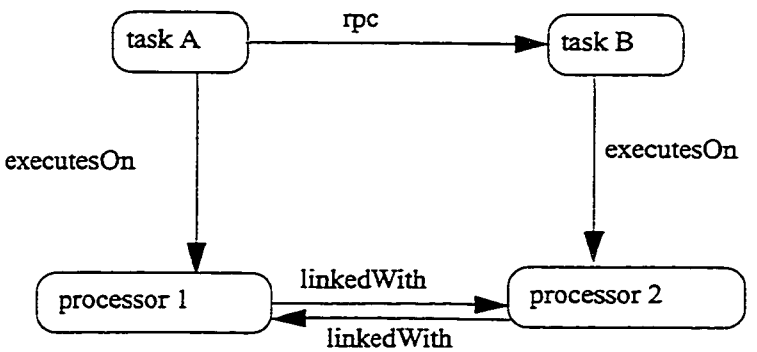


Figure 8: Example Distributed System Task Call Graph

3.3 PROGRES Graph Rewriting Rules

A graph rewrite rule serves two purposes. First, it can describe application constraints that limit the set of graphs that can be constructed but which are not captured by the graph schema. For instance, a graph rewrite rule can ensure that a processor should not be linkedWith itself or that a task should not perform an rpc with itself. Secondly, a graph query or transformation is described as a rewrite rule that obeys the type restrictions of the graph schema. In PROGRES, a graph rewrite rule is composed of transactions, tests, productions, restrictions, and path declarations. Each one of these items is discussed in turn.

A *transaction* is a graph transformation which either completes in entirety (i.e., atomically) or aborts with no modifications. A transaction has deterministic and nondeterministic control elements to govern the application of productions and tests [80].

Productions and tests are the parameterized, basic operations of the PROGRES language. A *test* is a parameterized graph query operation that is used to search for the existence (or nonexistence) of a subgraph pattern in a host graph. The host graph may contain additional edges not in the subgraph pattern. A test may have input and output parameters, such as attribute values, a node type, a node, or a set of nodes with common properties.

A *production* can be used like a test but it has a second component for rewriting the graph. The left-hand side (LHS) of a production represents the pattern that is searched for and is described like a test. But, the right-hand side (RHS) of the production describes a replacement for the found subgraph pattern. A copy of the right-hand side graph is substituted for an occurrence of the left-hand side of the production. A production can alter the attributes of a node, add edges, delete edges, add nodes, delete nodes, and change the type of a node or edge. A single production can be used as a simple transaction.

A complex production or test can use restrictions or path expressions to define sets of nodes with specified attributes or complex relationships. A *restriction* is a negative (or positive) constraint on the attribute(s) or type of a single node. A *path expression* navigates from a given set of source nodes, along certain edges in a host graph, to return a set of target nodes. It can be viewed as, either, a derived binary relationship or a function that describes a graph traversal in terms of edges and other path expressions. A restriction or path expression succeeds if all positive node and edge patterns are found, as well as no match for the negative node pattern exists.

A simple example production is given in Figure 9 which converts a multi-processor system to a uniprocessor system and replaces all RPC interactions with a pair of asynchronous interactions. In the left hand side of the production:

- Each node is identified by a number preceded by an inverted apostrophe.
- The type of each node follows the colon.
- The relationships between the nodes is identified by the typed edges.

The right hand side of the production deletes the processor node '4 because it appeared on the left hand side but it does not appear on the right hand side; the other nodes are mapped by the assignment of their node numbers. The linkedWith edges are deleted since only one processor node remains. The rpc edge of the left hand side is also replaced by two asynchronous messages that are used to emulate a blocking RPC. This production rule can be applied repeatedly until all processors are removed except one. If more than one task executes on a processor, this could be accommodated by defining a set of nodes for '2 instead of the single node.

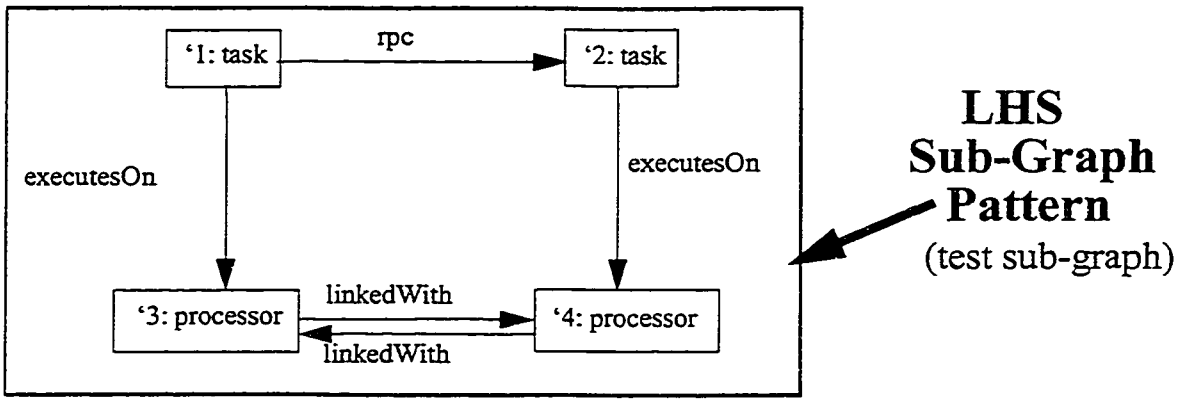
3.4 The Execution of a Graph Rewriting Operation

Graph rewrite rules can be programmed in a declarative fashion but it can be useful to understand how they are executed. That is the topic of this sub-section.

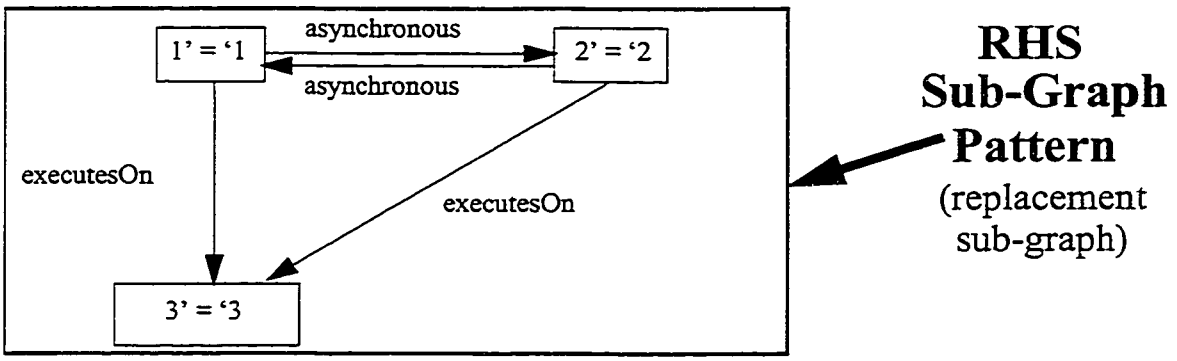
In PROGRES, the execution of a production is divided into several sequential steps [138, 140]. The steps involved in evaluating a production are:

1) *Subgraph test*: Select any subgraph within the host graph complying with the left hand side of the rewrite rule, including the identified edges.

production uniprocessor()
=



:: =



end;

Figure 9: Production Rule to Convert to a Uniprocessor System

2) *Check precondition*: Application conditions (e.g., structural and attribute properties) of the left-hand side node(s) are checked. Finally, path conditions and restrictions are checked.

3) *Subgraph replacement*: Remove the selected nodes from the host graph, including all of their incoming and outgoing edges. Then insert the nodes and edges of the right-hand side subgraph.

4) *Embedding transformation*: Connect the new subgraph with the remainder of the host graph. Nodes from the left-hand side which occur in the right hand side anchor the new subgraph. By default, edges not mentioned in the left-hand side and which are adjacent to a node that is both part of the left- and right-hand side, are not affected by the application of a production. New edges may be added by the right-hand-side.

5) *Attribute transfer*: Assign values to the attributes of the nodes of the new subgraph.

6) *Attribute reevaluation*: Compute the new values of all derived attributes within the new graph, or at least those derived attributes which need to have new values assigned.

3.5 Applying the Algorithmic Graph Grammar Approach using PROGRES

PROGRES is a specification language for algorithmic attributed graph grammar applications. It is supported by an integrated graph rewriting environment (also called PROGRES) that consists of language-specific tools for editing, analyzing, and debugging graph grammars [108, 135]. For example, a PROGRES graph grammar specification is entered using a syntax-directed editor [139], analyzed by a type-checker, and tested using the integrated interpreter and compiler.

The methodology for developing an application using PROGRES is described in [141] and reviewed here:

1) The analyst gathers informal requirements from documentation, interviews, etc.

2) The graph schema is developed as a graph model that reflects the complex data structures and relationships.

3) The graph analysis and modifying operations are programmed as sequences of subgraph tests and graph rewrite rules.

4) The specification's correctness is operationally tested using the PROGRES interpreter.

5) Aside from explicitly programming aspects of the graph matching process, the execution efficiency is improved by using attributes as node indexes or by precomputing (caching) heavily used, complex graph traversal operations.

6) A prototype implementation is generated with a graphical user interface. For increased analysis speed, C source code can be generated that is semantically equivalent to the PROGRES specification.

The two aspects of the methodology which are the essence of this research are: (1) the development of the graph schema for each model type and (2) specifying the model transformations using the graph rewrite rules. These two components constitute an operational specification of a graph grammar.

A limitation of the PROGRES environment is that only 64,000 nodes can be created during the transformation of a host graph. This not only includes the number of nodes in the host graph, but the intermediate nodes that are created during the execution of the transactions and productions. For example, if a transaction consists of ten productions that each create ten intermediate nodes that are non-terminal symbols, then the transaction can only be executed 640 times (64,000 / (10 * 10)). It is expected that this limitation will be lifted in future releases of PROGRES.

3.6 Using the Algorithmic Attributed Graph Grammar for Model Transformations

Two new approaches are used for transforming a model type (described as a graph) from its initial domain into a resulting graph model of the target domain. The first approach simplifies the resulting model type by removing terminal symbols (i.e., a node type) from the input model domain. This approach is used for the LQN sub-model to LQN translation.

The second approach is the rewriting of a host graph model, in small steps, until the rewritten graph complies with the target domain's graph grammar. This type of translation is rule-based graph rewriting [55, 83, 58], with a set of graph rewrite rules and a control algorithm for selecting the rule(s) to be applied [127]. To accomplish this a synthesized graph syntax is developed that combines the input domain's graph syntax and the target domain's graph syntax. A set of graph rewrite rules, using the synthesized graph syntax, are specified to convert the terminal symbols (i.e., sub-graphs of node types) from the initial domain into terminal symbols of the target domain (i.e., a replacement sub-graph). A control algorithm governs the application of these rewrite rules to prevent or manage conflicts for the rewrite rule selection. This strategy is shown in Figure 10. The model transformations from angio traces to a proper time, and from proper time to an LQN sub-model, both use this second approach.

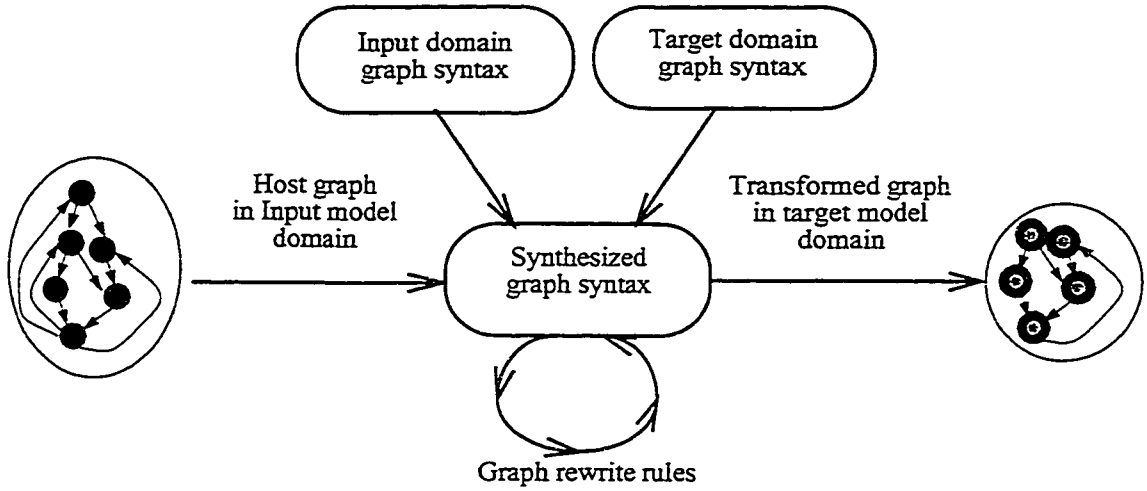


Figure 10: Combining Graph Grammars for Model Transformation

Chapter 4.0 LQN Graph Grammar

This chapter introduces the Layered Queueing Network (LQN) model because it is the target model type of TLC (Figure 1). A graph schema for constructing an LQN file for evaluation is developed. This graph schema is the basis for the automatic model generation; TLC constructs models which are compatible with this schema. An example LQN is presented using the graph grammar.

4.1 Introduction to the Layered Queueing Network Model with Activities

The Layered Queueing Network (LQN) model is useful for analyzing the performance of distributed operations because it captures the queues that form at hardware and shared software server tasks, as well as the synchronization time incurred by tasks when they interact. The LQN is appropriate for predicting the performance of many kinds of distributed systems, including client-server applications [2], peer-to-peer applications [90], communications switching software [15], transaction processing systems (e.g. ENCINA [62]), and systems based on midware software technologies [16] such as CORBA [113] or DCE [33]. The LQN is being extended from that described in [126, 125, 166, 167, 111, 43] and this research accommodates the extensions of [43]. These extensions provide for modeling concurrent threads of execution internal to a task and the synchronization between threads of execution. The analytic solution techniques for LQN models are the Stochastic Rendezvous Networks [168], the Method Of Layers [56], and the Layered Queueing Network Solver [43]. The last two techniques are based on approximate mean value analysis [144, 75]. Simulation tools for evaluating an LQN are also available. Before describing the LQN a few definitions are presented. The terminology and figure icons for the LQN are taken from [43].

An LQN represents the workload of one or more distributed operations. For each distributed operation's execution (a *scenario*) the workload is characterized by the tasks in the scenario, the service(s) each task provides, the task interactions, and the communication protocols. A *task* represents a concurrent entity, which may be a user, device, or software component. In an LQN, a task can act as a client task, a server task, or alternate, where it first acts as a server by accepting another task's service request message and then acts as a client by sending service request messages to other server tasks. To avoid confusion, when a task acts like a client it is called an *initiator* and it is called a *responder* when it is acting as a server.

The LQN resource usage parameters are a statistical representation of a task's execution, including the demand made at hardware devices and the service requests to responding tasks. The output of an LQN is the predicted average throughput and response time of each workload type and task.

The unit of modelling in a layered queueing networks is the *activity*. Activities are components in the performance model that represent the lowest level of detail necessary. Activities are connected together to describe the probabilistic behavior of a task's execution and service demand(s).

An activity can describe the change in the concurrency level due to fork and join behavior within a task or between tasks. *Forking* occurs when a thread of execution splits into two or more concurrent threads of execution. Examples of forking are: sending an asynchronous message so that the sending and receiving tasks execute concurrently; a server replies to a blocked client task and the server continues execution concurrently with the client; or a new task is created. A *join* occurs when two or more threads of execution are merged into a single thread of execution. Some examples of joining are barrier synchronization, nested accepts, and guarded accepts. The semantics of the fork and join operation are that a task will accept a message when its internal threads have ended or are blocked.

A task accepts a service request message at an *entry*. Requests for service are characterized as entry-to-entry interactions using a communication protocol. An entry may correspond to an actual task communication port or a message type that identifies a particular service provided by a task. An entry that is involved in a join will accept one message.

The communication protocol types that the LQN represents are:

- *Asynchronous* (asy): an initiating task makes a service request to another task but it does not block to wait for a reply. No reply message is needed.

- *RPC or rendezvous* (rpc): the initiating task waits for a reply to its service request and it becomes ready to run once it receives a reply message.

- *Forwarding* (fwd): The previous interaction types have involved two tasks but a forwarding interaction can involve several tasks. It is a system level interaction that occurs when the initiating task blocks on its request but the responding task asynchronously sends the request to another responding task. Each responding task can continue to forward the request further to other responding tasks. The last responding task in the series sends a reply directly to the blocked task. This type of behavior occurs when a task acts as an administrator manager by dispatching service requests to a pool of worker tasks [47] or as a form of rate control for a task pipeline.

A responding task's execution of an RPC-based interaction can be described as two phases of execution. The first phase consists of the demands between the RPC request acceptance and the subsequent reply (or forwarding) of the request. Any activities that occur after the reply (or request forwarding), but before the acceptance of the next request, are labelled as a *second phase* of execution. The second phase identifies the resource contention that occurs between the initiating task and the continuing responding task.

Icon	Name	Explanation
e1 t1	Task (with entry)	A sequential thread of execution which can initiate or accept service requests.
e1 t1	Task pool (multi-server)	A set of tasks that share a common input queue.
p1	Device	A device consumes time (i.e., executes) on behalf of a request from a task.
→	Activity flow	Flow of execution to the next activity in a task.
a	Activity	Basic unit of modeling.
a[e]	Activity with a reply.	The activity generates a reply to entry *e* once it finishes making demands.
c &	And-fork	A single thread of execution forks into two threads of execution.
& c	And-join	A synchronization point where two threads of execution become merged.
c p + 1-p	Or-fork	One branch is chosen with probability *p*.
+ c	Or-Join	Two alternative threads merge into a common thread.
* n c	Repetition	Repeat the activity an average of *n* times.

Table 5: LQN Icons and their Explanation

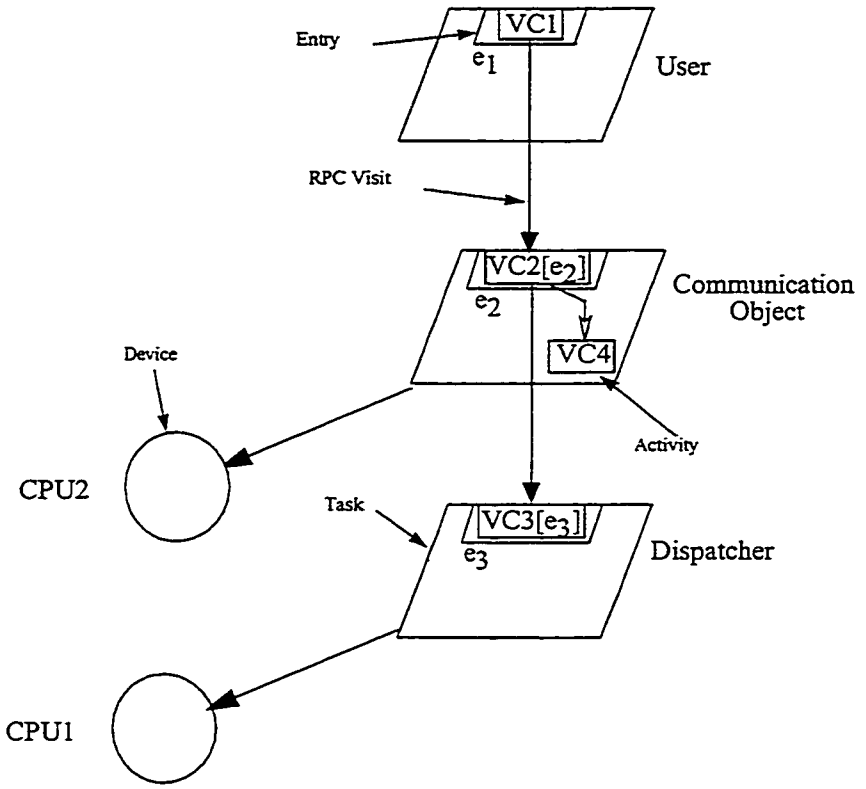


Figure 11: LQN Example of the Validation Scenario

4.2 Components of an LQN Graph Grammar

An LQN graph grammar can be divided into the following components that can be described as fragments of a graph schema.

- The *scenario structure* considers a task as an opaque object that interacts with other tasks.

- The *task behavior* is the set of directed sub-graphs of activities which describe the internal, probabilistic execution of a task.

- The *environment configuration* describes key aspects of the target execution environment.

4.4 LQN Task Behavior Graph Schema Fragment

The task behavior is described as a graph that begins with the first node after the acceptance of a request at an entry. A successor node may be a resource demand, an interaction with another task, the acceptance of another request at an entry, or a *phase node* that identifies a new phase of service. Task behavior can also include the forking of an internal thread of execution, the joining of several concurrent threads of execution, or the selection between alternative threads of execution using a probability weighting. The task behavior graph schema is shown in Figure 13.

The node types that have the same behavior as described in Table 5 are the and-join, or-join, and-fork types. The phase and reply nodes have been added although they were implicit in the LQN components of Table 5.

The or-fork node type is different then that described in Table 5. It indicates that the next activity of the task is selected using a probability weighting. The or-fork node type is not part of the taskEvent node class because its successor node is a probability node (prob). The definition of the TaskEvent node class, TaskEventTerm node class, and nextTask edge type ensure that the or-fork node type can only be the target of a nextTask edge and not a source node.

The probability node type (prob) has an attribute to store the probability of selecting a branch of an or-fork node since edges cannot store values.

The task behavior graph schema has several edge types:

- The target of a nextTask edge is the succeeding activity performed by the task.

- The target of a selectEvent edge type is a set of probability nodes that are used to select the next node of the task. The source of a selectEvent edge type must be an or-fork node.

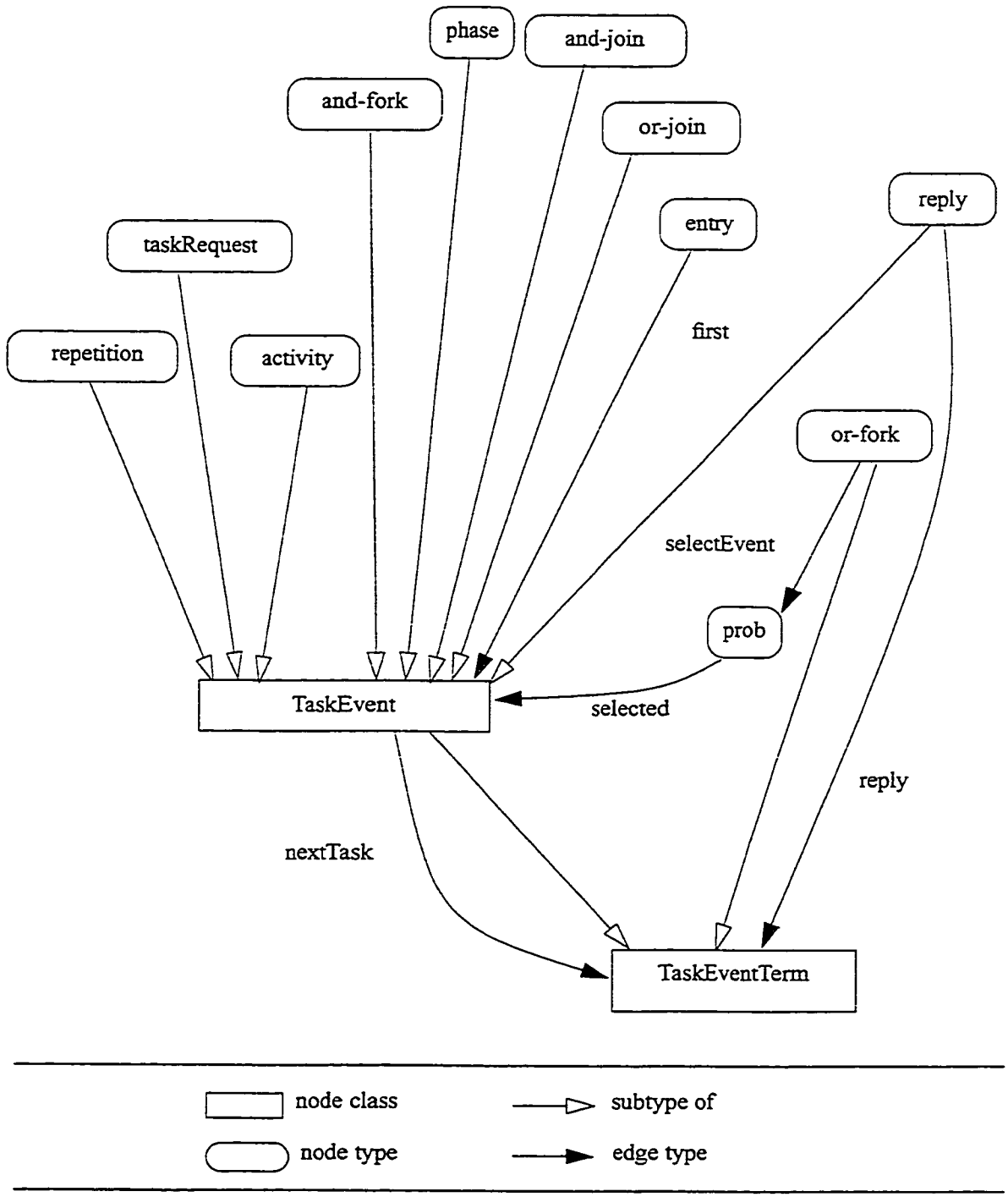


Figure 13: LQN Internal Task Graph Schema Fragment

- The target of a selected edge type is the next node for that execution thread.

- The target of the reply edge type is the node that occurs when the initiating task become unblocked.

- The source node of a first edge is the entry which begins the task behavior upon acceptance of a message, and the target node is the first node that occurs after accepting the message.

The phase node, reply node, and reply edge are components of an RPC or forwarding interaction. A task's behavior sub-graph may have several phase nodes. For example, a deterministic third phase of service is useful for modelling task pipelines.

The activity and entry node types have attributes. An entry node has an attribute to store its name. An activity node type has attributes to store the mean service time of the activity and its distribution. By default, a service time demand at a processor is exponentially distributed. However purely deterministic, hyper- or hypo-exponential service time distributions are also allowed. Activities can also have zero service time in which case no request is made to the processor.

4.5 LQN Environment Configuration Graph Schema Fragment

An LQN performance model incorporates environment configuration details of the target system. The environment configuration information is stored as attributes of the node classes and types. These include the components that characterize the task behavior (activities, entries, and demands), the workload mix, and the hardware devices. The associated graph schema is in Figure 14.

A customer node represents the source of a request which originated from outside the system (e.g., a button push). It has the workload mix as an attribute.

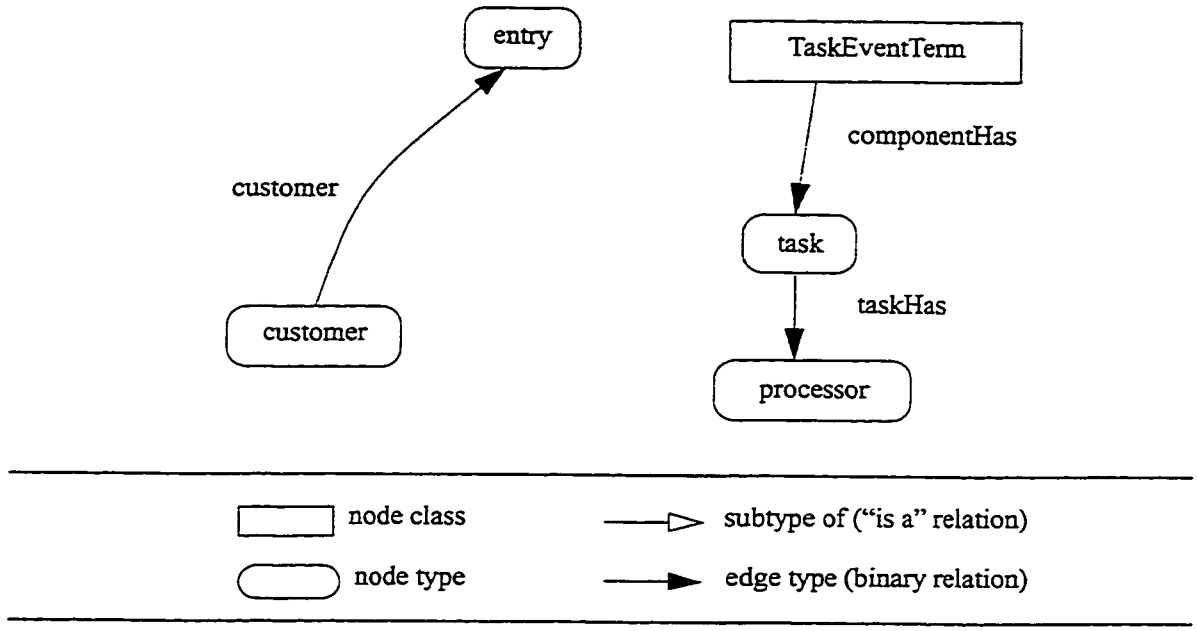


Figure 14: LQN Environment Configuration Graph Schema Fragment

The attributes of the task node type include the a task priority, a task identifier, task scheduling, and the type of server that characterizes this task. Task scheduling refers to the order in which messages are accepted by tasks and may be first-in-first-out or head-of-line priority. The alternatives for the type of server are: it provides a service to a single customer at a time (single server), it provides service simultaneously to multiple customers (multi-server),[1] and its clients do not queue but have a specified service time (infinite server).

The processor node type has an attribute to specify its scheduling policy, which may be processor sharing, first-come-first-served, head-of-line priority, or random. It also has an attribute for the relative speed of the processor.

This completes the discussion about the individual components.

1. A multi-servers is a set of identical tasks that share a common input queue. At present, second phases are not handled, nor are phases with non-exponential service time distributions.

4.6 An Example LQN Using the Graph Schema

The complete LQN graph schema that is used to construct an LQN input file is found in Figure 15. The icons that are used to represent an LQN using the graph schema are shown in Figure 17 and they borrow as much as possible from the LQN icons of Table 5. Using the LQN graph schema icons, the validation request LQN example of Figure 11 is shown in Figure 17 without the environment configuration (for clarity).

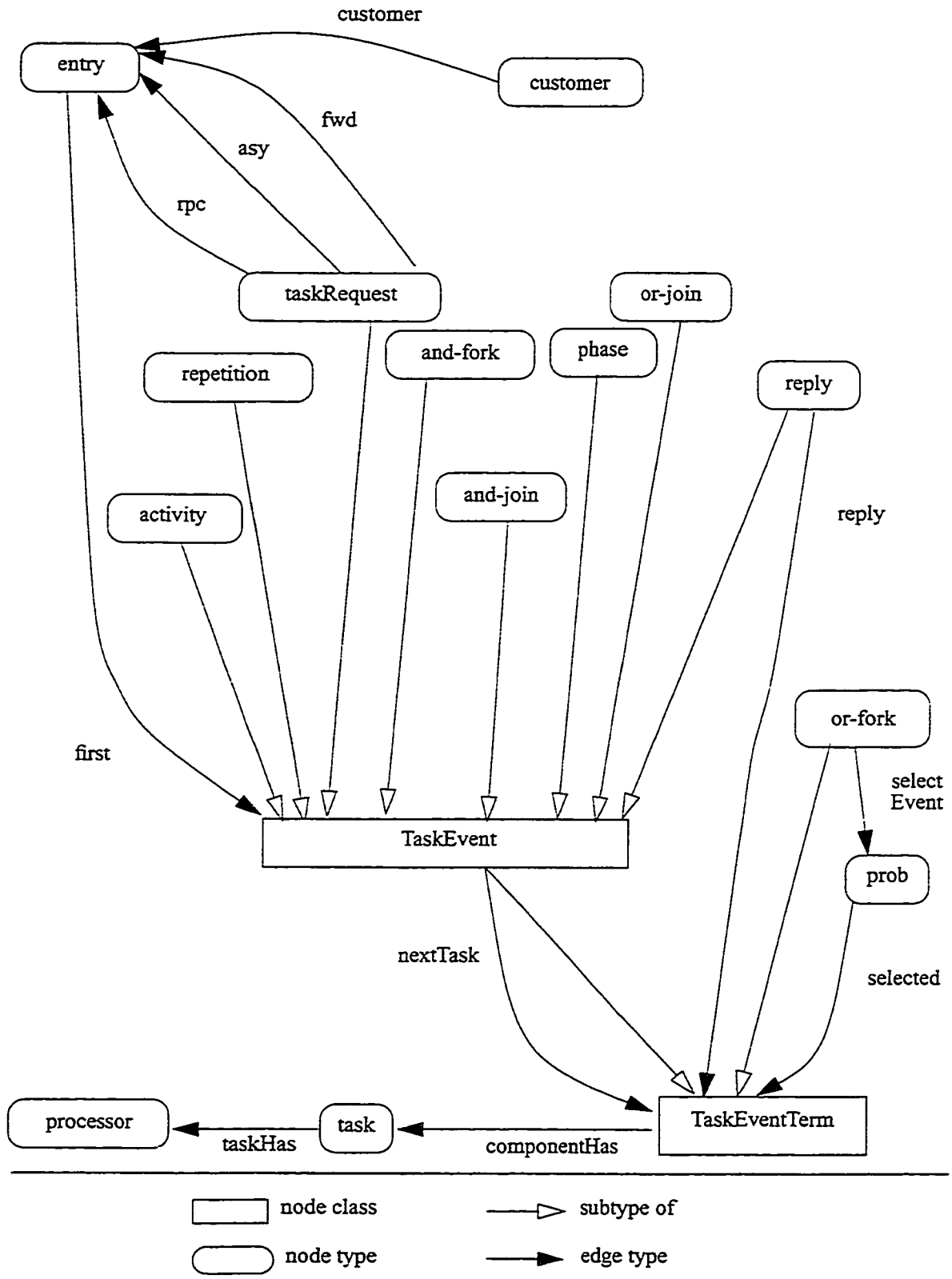


Figure 15: Complete LQN Graph Schema

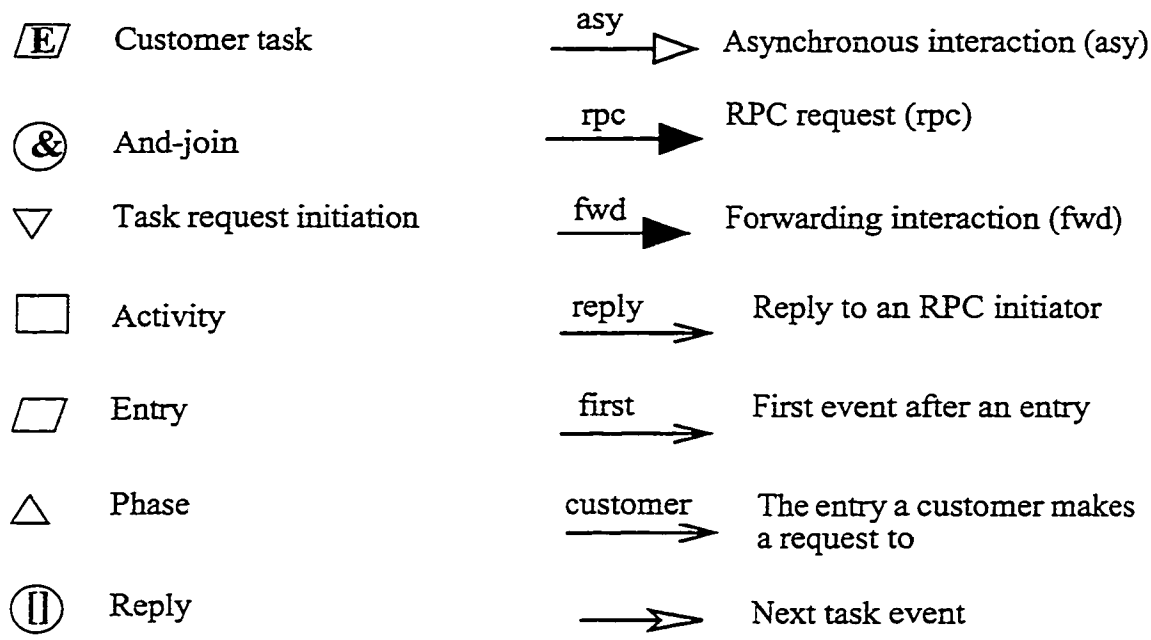


Figure 16: LQN and LQN Sub-Model Icons

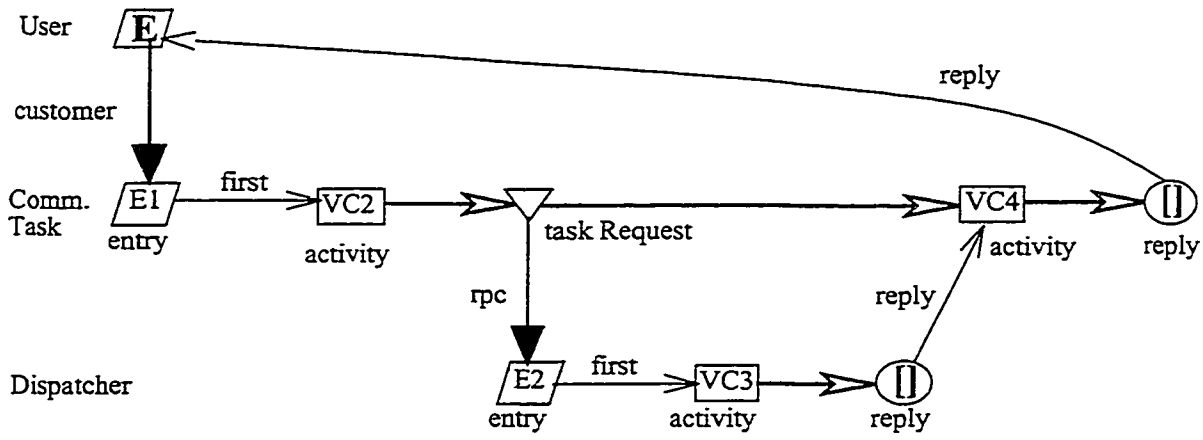


Figure 17: Example LQN Graph of the Validation Request

Chapter 5.0 LQN Sub-Model to LQN Transformation

This chapter describes what an LQN sub-model is and how several sub-models are combined into an LQN (step #5 of Figure 1). This transformation has a substantial user input and data management component so the emphasis is on the usability and data management issues. A description is provided of the Performance Analysis by Model Building (PAMB) tool that has been developed to aid the analyst in constructing an LQN from sub-models, evaluating the LQN, and reviewing the predictions. A restriction of the tool is that it cannot construct a model that includes tasks with internal concurrency.

5.1 The LQN Sub-model Requirements and Assumptions

An LQN sub-model characterizes the execution of a single scenario using the components of the LQN. The components of a sub-model are: tasks, their entries, entry-to-entry interactions, and task activities. A sub-model activity represents a service time at a device. A sub-model does not include the environment information which is part of a performance experiment definition, including the aggregate workload, device descriptions, or task properties. For example, Figure 18 is a sub-model of the validation scenario LQN of Figure 11. Another example is Figure 19 which is a sub-model of the application command scenario described in section 2.1. The two most notable differences from a complete LQN are that the devices are removed and the customer task identifies the first entry of the sub-model.

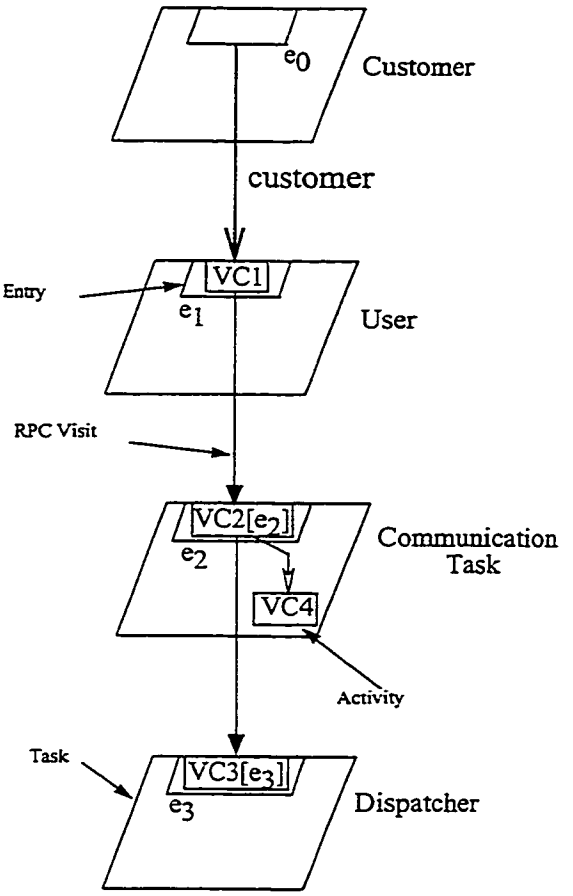


Figure 18: LQN Sub-model Example of the Validation Scenario

The requirements a sub-model satisfies are:

- It is developed from an execution trace of a scenario because that is the initial input of TLC.

- It can be described as a node-labelled, arc-labelled, directed, acyclic graph so that it is compatible with a graph grammar transformation engine.

- It characterizes the relatively static key performance aspects of a scenario, allowing the easily modified parameter values to be supplied by the analyst.

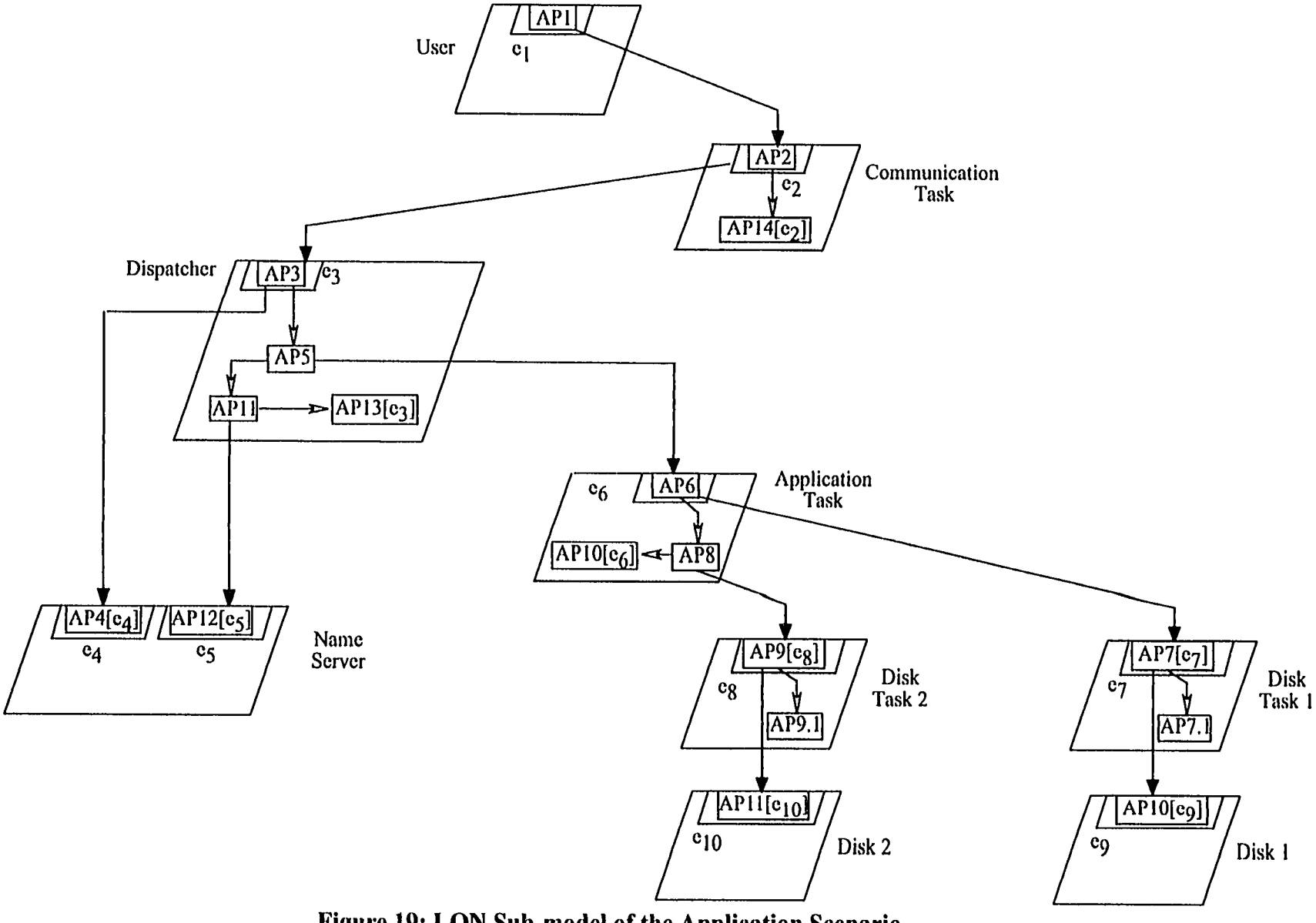


Figure 19: LQN Sub-model of the Application Scenario

- Device demands are represented as logical units that are converted into service times, allowing environments with no notion of real time to have models built.

- It is initiated by a single external request.

A sub-model is derived from a single execution trace so it is more restricted than an LQN in three ways. First, a trace is deterministic so the sub-model does not include probability values or selection. Secondly, a sub-model task does not have internal concurrency whereas an LQN task can have internal concurrency (such as parallel activities within the task). This is a result of the sub-model task characterizing the smallest unit of concurrency within a distributed operation. If a process is the smallest unit of concurrency of the operating system then a sub-model task is a single-threaded process. If an operating system process can be multi-threaded then the sub-model task is a single thread. Lastly, the initiation of concurrency is represented in a sub-model as an asynchronous interaction (asy), whereas an LQN can also initiate concurrent activity using the and-fork activity.

5.2 LQN Sub-model Graph Schema

The LQN sub-model graph schema is a subset of the LQN graph schema. It can be derived from the LQN graph schema by:

- Removing the task node type and processor node type because they are environment parameters that are added for each experiment.

- The or-fork, prob, or-join node types, as well as the TaskNdTerm node class, can be removed because a sub-model is deterministic.

- The and-fork node can be removed because the initiation of concurrency is represented using an asynchronous interaction.

In summary, the following nodes can be removed from the LQN graph schema as well as the edge types which would be dangling: workload mix class, task node type, processor node type, or-fork node type, prob node type, or-join node type, and-fork node type, and the TaskNdTerm class. The resulting graph schema for the sub-model is found in Figure 20.

5.3 Constructing an LQN from LQN Sub-models

The Performance Analysis by Model Building (PAMB) tool provides an interface through which the analyst combines sub-models and assigns values to parameters to build an LQN. The components that make up PAMB are shown in Figure 21. The steps for constructing the model are:

(1) Using the configuration interface, the scenario(s) to be included in an LQN are selected by choosing the corresponding sub-model(s). The aggregate workload and the distribution of the workload for each scenario is also assigned.

(2) Using the environment interface, parameter values are assigned to configure the model for its target environment.

(3) A model is constructed by combining the information and introducing service times for each task's activities using a set of resource functions.

(4) Using the experiment interface, any remaining parameter variables are assigned a value, or a list of values to be considered.

(5) and (6), For each variable assignment a model is generated and evaluated by analytic or simulation means.

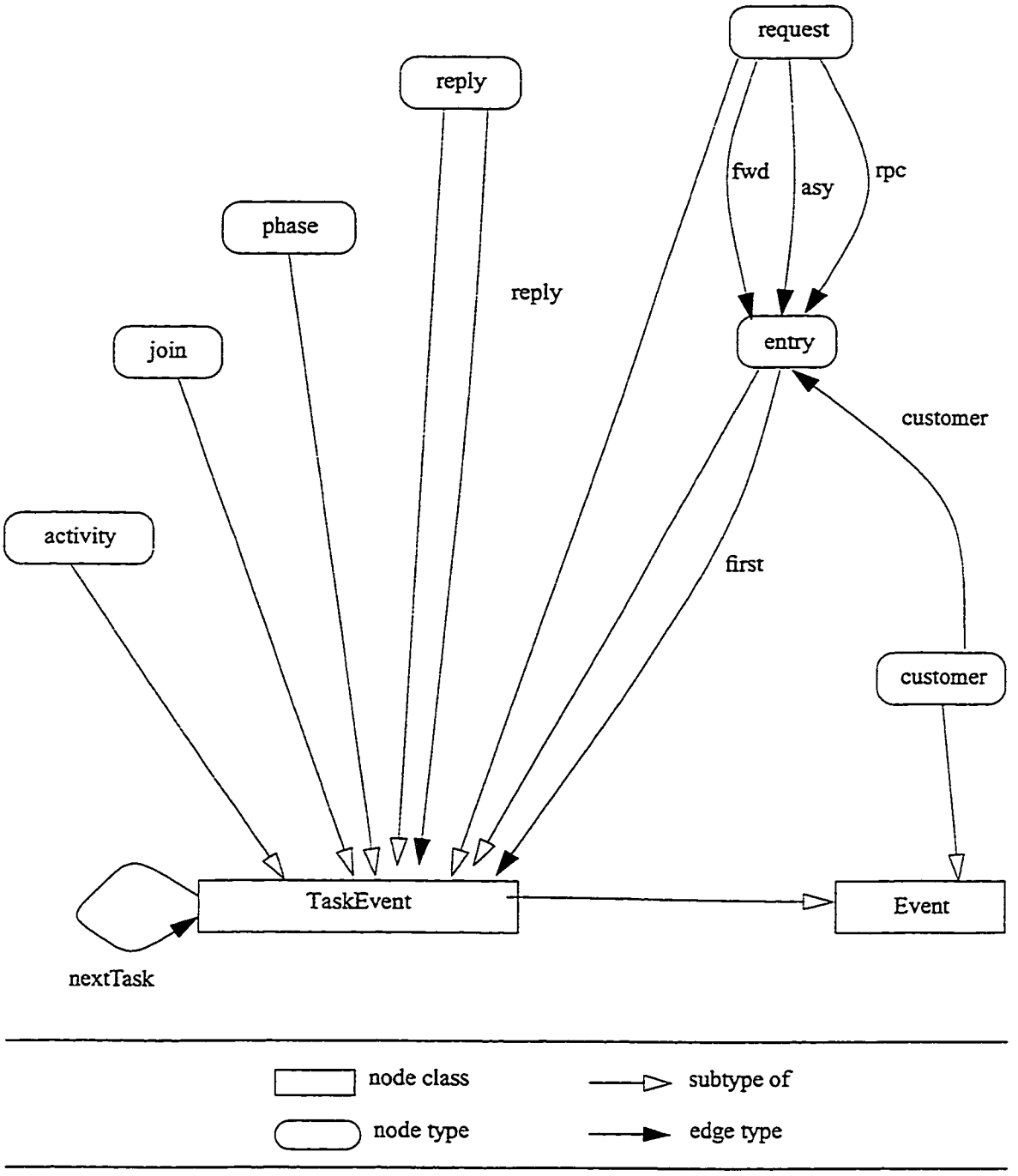


Figure 20: LQN Sub-Model Graph Schema

(7) Once the suite of models are evaluated the performance predictions are gathered, filtered, and sorted by scenario, task, and device. They are presented to the software developer as a set of graphs.

The experiment interface also allows parameter variables to be systematically assigned values for conducting a what-if analysis, sensitivity analysis, or trend extrapolation. It is not discussed further.

The steps that are important for this transformation are (1), (2), and (3). Each step is now presented.

5.4 Constructing a Skeleton LQN

The analyst uses the configuration interface to construct a skeleton LQN by selecting the sub-models to include in it and assigning an average percentage workload, from the aggregate workload, to each sub-model.

Constructing a skeleton LQN has various degrees of complexity. The simplest case is building an LQN from a single sub-model. An example of this is constructing an LQN just from the validation command sub-model (Figure 11). A more complex LQN is constructed by merging several sub-models, such as combining the validation command sub-model with the application command sub-model of Figure 19. The last approach to constructing an LQN is to use an existing configuration as a template and to override specified parameter values.

Once the sub-models have been selected the *aggregate workload* is divided amongst the different sub-models using a *workload ratio* for each sub-model, which is the probability of the customer initiating that scenario. Once a "customer" of the aggregate workload selects a target sub-model it makes an RPC interaction with the first entry of the selected sub-model. For example, in Figure 22 an aggregate workload customer is represented by a

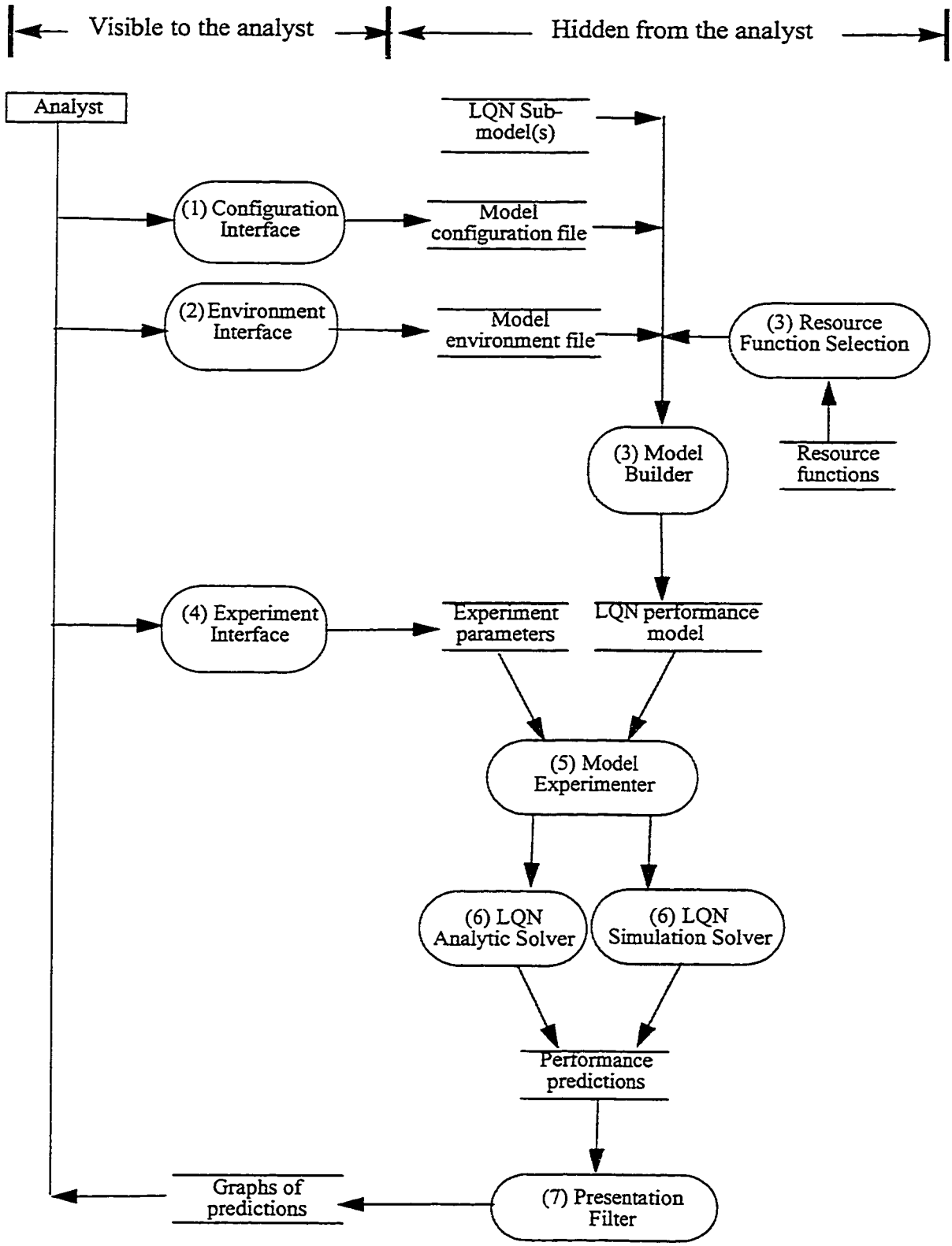


Figure 21: Building a Model with PAMB

task that visits the validation request sub-model 60% of the time (a workload ratio of 60%) and the application command sub-model 40% of the time. The aggregate workload of entry e_0 could be a closed or open customer class.

The entered configuration information is stored in a *model configuration file*. Although it is not yet implemented, the analyst will be allowed to include other model configuration files just like a sub-model. This will allow hierarchical composition of an LQN model by combining sub-models and configuration files into a single, flattened LQN.

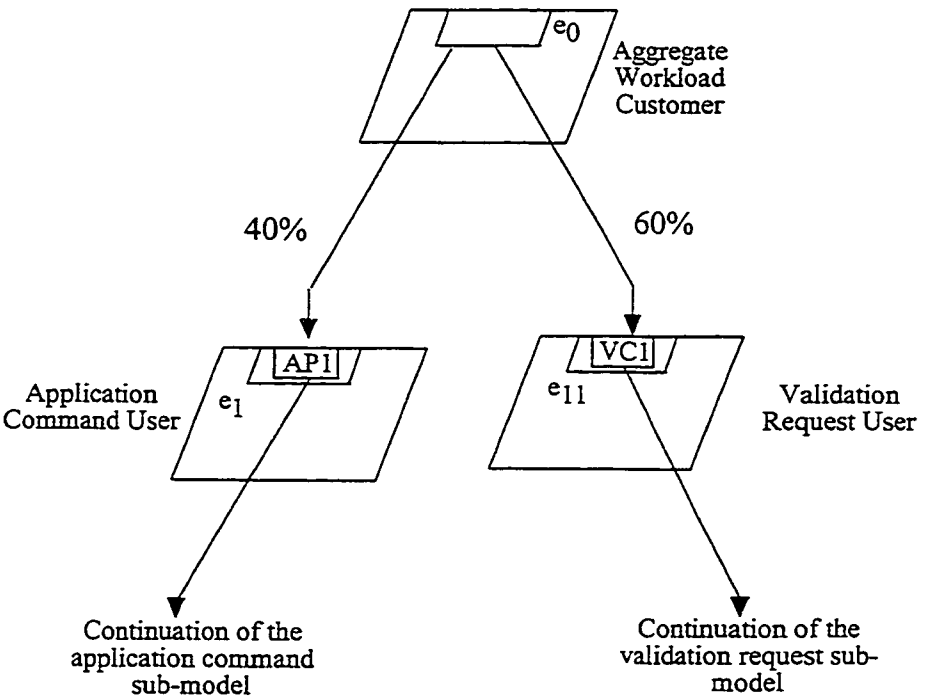


Figure 22: Workload Distribution across Sub-models

5.5 Defining the Environment

Once the skeleton LQN is constructed, the next step is to add details about the execution environment to describe the devices' attributes, tasks' attributes, and information used to

select resource functions. Default values are provided for all environment attributes so that a sub-model can be used as an LQN without having to generate the environment description.

The attribute information supplied for each device is:

- A processor name.
- The scheduling discipline.
- The number of multi-server copies.
- A service rate.

The attribute information provided for each task is:

- A task name.
- The number of identical server threads.
- A priority.
- Its time-slice.
- The name of the processor it executes on.

The environment information can also record low-level, manual edits to an entry's attributes. This can be used to change the default values: entry service times can be changed from the default of an exponential distribution to a deterministic distribution, or the visit

distribution can be changed from the default of a deterministic distribution to a geometric distribution.

The environment information can include information that is important for selecting a resource function. This information can include the operating system, compiler vendor, compiler directives, the type of processor, etc. This is discussed next.

5.6 Estimating Resource Usage for Task Activities

The LQN requires service times for activities and they are introduced using resource functions. A *service time* is the amount of time an activity consumes at a device. A *resource function* converts a sub-model's activity, using its activity identifier, into a representative service time. An activity's identifier may also have values which are used as resource function parameters, to provide a better estimation of the service time. A resource function may return a list of service times, such as estimating network or disk usage. It is similar to the complexity function of Vetland [163] (e.g. parameterized execution time [74]). This research assumes that resource functions are available.

The main issues that are briefly considered about resource functions are: how are resource functions constructed, how are resource function parameters identified, and how the resource function selection is automated. The issues not considered here are general database and site management concerns, such as storage and accessibility.

5.6.1 Constructing a Resource Function for an Activity

A resource function may be constructed early in the development process without target hardware, using a rule of thumb or a device budget estimate. The initial estimates for a resource function can be tracked and improved (if necessary) as the development progresses.

If the target hardware or a similar system is available then a resource function may be constructed from measurements. For example, the Modeling Kernel (discussed in section 2.3.2.2) estimates resource functions by applying regression analysis to a small number of execution runs. When measurements are being made, care must be taken to minimize the monitoring cost and probe effect. Patil [119] describes two techniques to minimize the probe effect by dedicating one processor, in a dual-processor configuration, for monitoring. Larus [86, 87] reduces the amount of tracing information by only recording *significant events* (e.g., basic block boundaries). The significant events are then used as markers to regenerate a full trace by post-execution analysis of the trace and the original program.

Workload characterization principles [25, 71] are useful for constructing a resource function but flexibility is the chief guiding principle. Flexibility is needed to allow the analyst to use various sources of information to construct a resource function, such as using a local function call, an external function, a database operation, a shell script, a scalar value, or an object (such as another sub-model).

Flexibility is also needed for defining the parameters to a resource function. The adopted approach considers a parameter as a name-value pair. This is flexible because the ordering of parameters is not important so they do not always have to be provided, and a resource function can be transparently refined by adding more name-value pairs. Resource functions can have common predefined parameters such as the processor type and target software component but this is not enforced.

5.6.2 Retrieving Resource Functions

The selection and use of a resource function must be automated otherwise it will be too labor intensive for general use. However, the use of automation implies that the selection process is well defined and standardized, which is in conflict with the need for flexibility. A compromise was to use a multi-step selection process.

Resource functions for a particular target environment are grouped into a *resource function table.* Just like a resource function, the resource function table has name-value parameters. Example target environment attributes are an operating system identifier or compiler identifier.

But what if there is no identical match for the target environment? What if more than one target environment matches the provided information? A selection policy object is an abstraction that completes the selection decisions in the event of a conflict. A *selection policy object* embodies the policy for selecting the most appropriate item from a set of candidate items. It is intended to be general and extensible, allowing the user's selection preferences to be captured and reused. It can be composed of a chain of subordinate selection policy objects.

Currently, a simple selection policy object for selecting a target environment has been adopted. Its implementation assigns an importance weighting (between 0.0 and 1.0) to each parameter and the target environment with the largest importance weighting sum is selected from the candidates.

The questions regarding selecting a target environment also apply to selecting a single resource function from a resource table, so a resource function selection policy is needed as well. It also uses the simple importance weighting summation approach.

The selection policy object may need additional information about the sub-model and this is provided by a *sub-model resource bridge.* A sub-model resource bridge can provide context information about the activity so that the appropriate resource function is selected. For example, a sub-model resource bridge for the ObjecTime tool can determine whether an RPC between two logically concurrent entities is truly an RPC or it is to be treated as a procedure call.

This leads to a four step process for automating the resource function selection. First a resource table selection policy is provided. Then a resource function table is selected given

the attributes of the target environment. In the third step a resource function selection policy for the selected resource table is identified. The last step retrieves the resource estimates from the resource table once the activity identifier and parameter(s) are passed to the resource table.

This approach appears to work but it is not certain how well it will scale as the number of resource functions or target environments increases.

5.6.3 The ObjecTime Example

An approach to generating and managing resource functions for the ObjecTime toolset is described in [170] and reproduced here. This information was accessed by PAMB to generate an LQN from an ObjecTime design's angio trace.

The ObjecTime Micro-Run Time System (in Figure 23) insulates an ObjecTime design from the operating system. It provides a small number of object classes and primitive operations used by an ObjecTime design. These basic services are standardized across target platforms so they can be measured and serve as resource functions, being available early in the design process.

A *Resource Function Management Utility* (RFMU) designed by Bayarov [170] manages the resource functions, as well as providing tools to systematically make measurements. It communicates with the model builder of Figure 21 to provide service time parameters for a model, passing a resource function identifier (with optional parameter values) and returning a service time value.

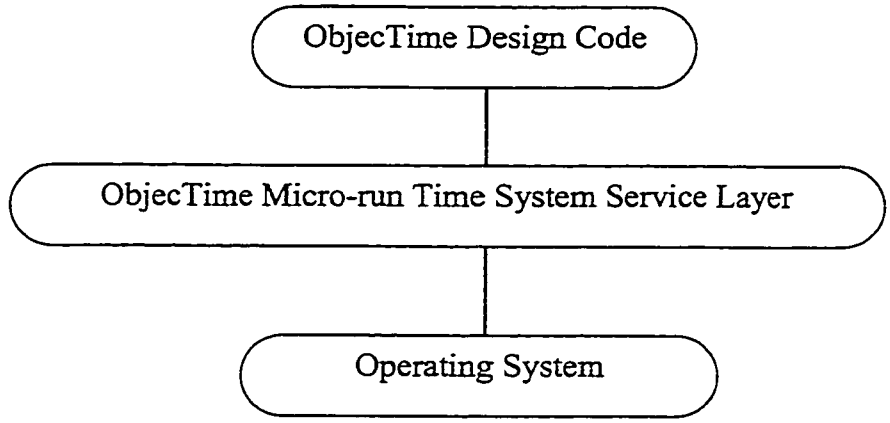


Figure 23: High-level Block Diagram of an ObjecTime Target Environment

Resource function development is automated by a *test harness* which runs a measurement *test suite,* to measure, collect, and store the service time data about the run-time system primitives. This approach was taken because there are a large number of standard target platforms to test (over 60), where a *target* was considered to be a particular operating system, the version of processor, and the specific compiler. The test harness systematically runs the test suite on each target by automating the compilation, target loading, running, and output result collection. It is developed in Expect/Tcl scripts [91, 118].

The test suite measures the service times of all of the micro-run time system primitives. The measurements are made using the operating system's standard timer utility.

5.7 Building the Model

The model builder uses all of the information of the previous sections to construct an LQN.

The following assumption is used to combine the entries of a task from different sub-models: *tasks (devices) with the same identifier in different sub-models are merged into a single item in the LQN model.* For example, in Figure 24, the Communication Task merges two entries, one from the validation command sub-model (Figure 18) and one from the application command sub-model (Figure 19).

When combining sub-models, if duplicate entries exist in different sub-model components they are given unique identifiers. For example, the Validation Request User's entry is renamed from e_2 in Figure 18 to e_{12} in Figure 24 to avoid an entry name conflict.

It is important to describe the scope in which names must be unique to avoid name clashes. The name space scoping is as follows: workload and device names must be globally unique; task names are unique within a sub-model scope; and entry names are unique within a task name scope. The task and device name spaces are distinct.

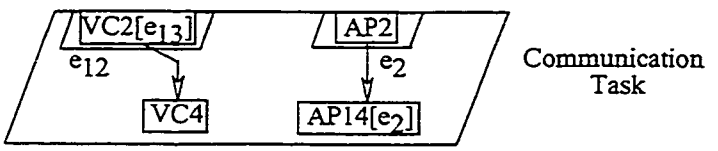


Figure 24: Combining Task Entries across Sub-models

5.7.1 Using an Existing Configuration File as a Template

If only a few parameters of an existing configuration file need to be altered there is the capability to reassign new values to these parameters. A configuration override is a value which alters a sub-model's attribute(s). A configuration override is treated as a text substitution that applies to a single sub-model. The attributes of a configuration file that can be overridden are

- Rename a task in a sub-model, which can be used to merge several tasks.

- Rename a device in a sub-model, which can be used for reallocating device demands.

- Override environment parameters for a device or task.

- Override an entry's attribute.

- Replace the service time of a task's entry with the specified value. The specified value may be a string label that later processing will resolve to a real-number value.

- Add a specified number of interactions from a source entry to a target entry (not yet implemented). The specified number of interactions may be a string label that later processing will resolve to a real-number value.

- Delete an interaction from a source entry to a target entry (not yet implemented). A warning is produced if the specified interaction does not exist.

If a configuration override identifies a task or device that cannot be found, a warning is issued.

5.8 Restrictions on Transforming Sub-models into an LQN

There are restrictions on the range of models that the current model builder can generate. The restrictions can be lifted by extending the tool's capabilities.

The first restriction was mentioned in the introduction and it is: an LQN model with tasks that have internal concurrency cannot be constructed.

Another restriction is that multi-server (multi-threaded) tasks can become confused if they need to reply to an RPC after a synchronization has occurred with another task. The

problem is that there is no way to associate the multi-server's reply with the blocked client it should reply to. Figure 25 illustrates this point when the Client and Server tasks are both multi-servers. If there is more than one Client task blocked on the Server task's entry e_2, it is not known which blocked client the reply of activity A8 (A8[e_2]) is associated with. In the current LQN simulation solver this quickly leads to deadlock because the RPC replies are sent to the wrong Client task, leaving the real initiating task blocked. Currently, PAMB detects these type of situations and halts. A further release of PAMB will alter the sub-model to avoid the deadlock.

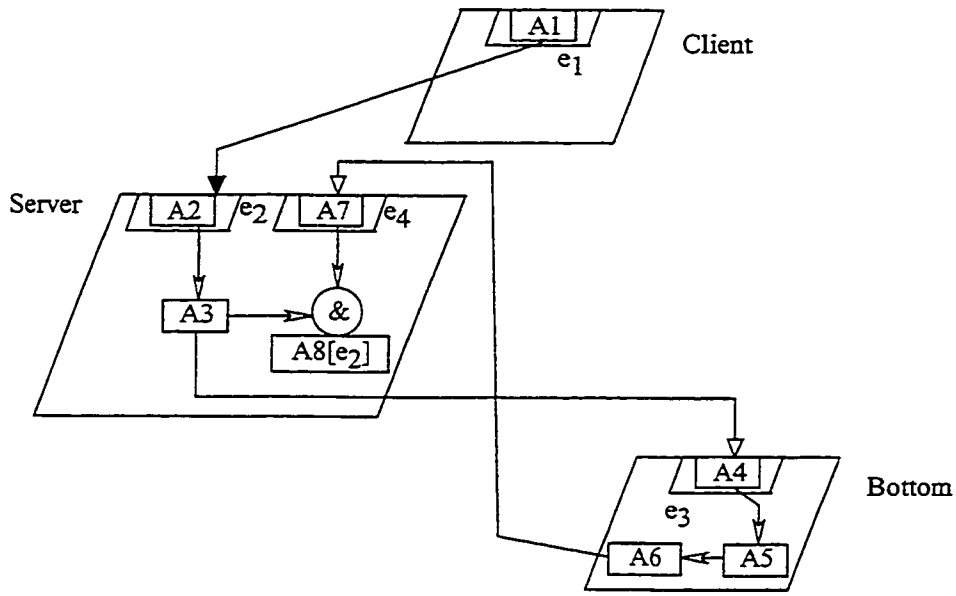


Figure 25: Multi-Servers and And-join Interactions

Chapter 6.0 Proper Time and the Transformation to a Sub-model

Proper time is the analysis format which overcomes deficiencies of the second generation angio trace and trace analyzer [66].

The second generation angio trace satisfied its goal to capture the total workload of a distributed operation but it had a weak characterization of task execution. The second generation angio trace assumed that a task's execution could be broken into independent segments with each segment beginning when a task accepted a message. For example, in Figure 26, the acceptance of three messages break the execution of Task A into three graphs that are separated by gaps. This also implied that the order in which a task received its messages was not significant. However, this assumption is an obstacle to developing more precise models for two reasons. The first reason is that the order in which a task accepts a message can be significant because of synchronization points that introduce delays. For example, Figure 27 shows how a synchronization can delay the servicing of Scenario (3)'s message. The second reason is that an incorrect trace analysis could result when the ordering of the messages was significant (discussed in section 6.4.3).

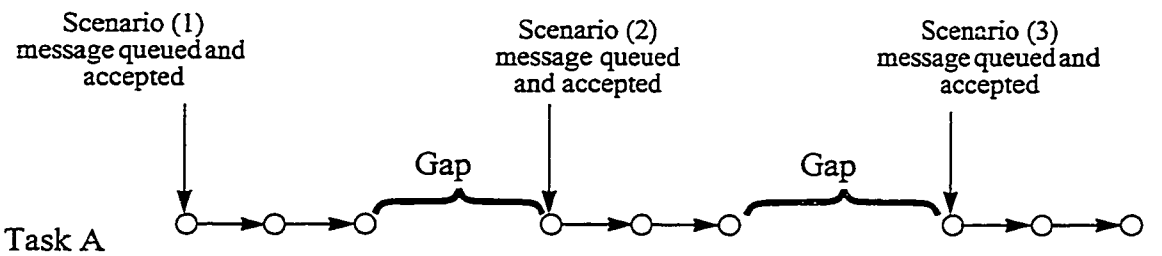


Figure 26: Second Generation Interpretations of Three Messages

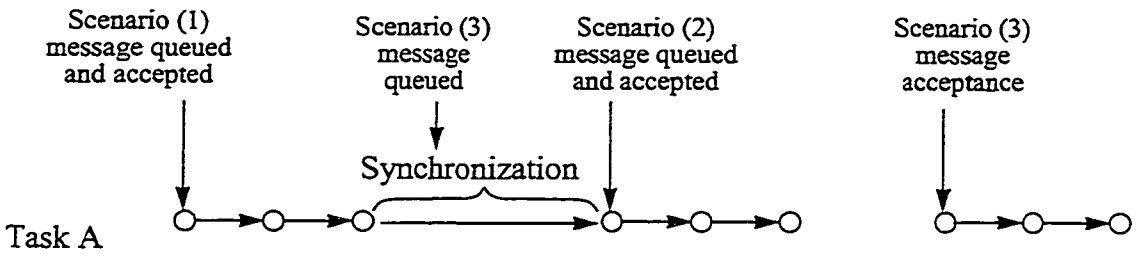


Figure 27: Three Messages with a Synchronization Delay

Relaxing this assumption required characterizing task synchronization and including it in the trace analysis. However, this greatly increases the trace analysis complexity because the shape of the distributed operation's execution graph became a directed acyclic graph, whereas it was previously a tree.

The increased complexity required a more formal approach for the angio trace characterization and trace analysis, otherwise the precision and validity of the generated model was questionable. Proper time was developed to satisfy these needs.

Proper time characterizes the distributed operation's and the tasks' frames of reference as three types of graphs. A *task event graph* characterizes the execution of a single task as a linear graph. An *operation event graph* characterizes an operation as a set of execution threads that share a set of interacting tasks. These two graph types are overlaid to form a *Task and Operation Event Graph* (*TOEG*) where each event has two points of reference. The TOEG is analyzed to produce a sub-model.

"Proper time" is a concept in relativity theory [142] which allows the same event to be interpreted differently for each frame of reference. The TOEG is a proper time model where a proper time event type is represented as a TOEG node type, and the relationship between two proper time events is a directed arc in the TOEG. Proper time is the framework in which the TOEG is interpreted.

An example of the way in which proper time characterizes an RPC interaction will help to illustrate the need for the three graph types. Figure 28 shows two task event graphs that make up an RPC interaction, one for the Client task and one for the Server task. In the task event graph figures, a square box is the task information recorded by an event and the diamond is where the Server task accepts a service request. Figure 29 is the operation event graph which views the RPC from the scenario perspective without any structural considerations. The circle represents the activity information recorded by an event. Since an RPC is a single thread of control, without any concurrency, the operation event graph is a linear sequence of events.

By overlaying the two graph types of Figure 28 and Figure 29, the Task and Operation Event Graph (TOEG) of Figure 30 is formed. The sub-graph of events $\{e_2, e_3, e_6, e_7\}$ form a distinct pattern which always identifies an RPC interaction, information that was not available when the other event types were examined in isolation. This example illustrates the usefulness of combining the two orthogonal graph descriptions.

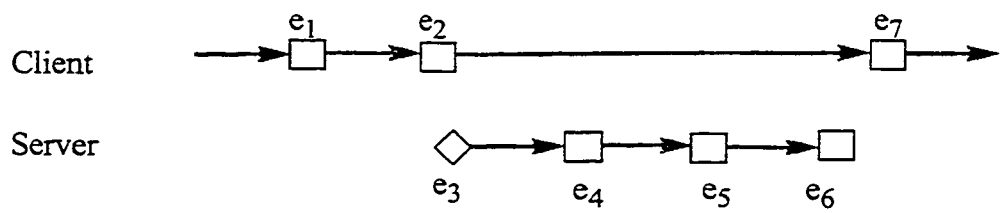


Figure 28: Task Execution Graph of an RPC Interaction

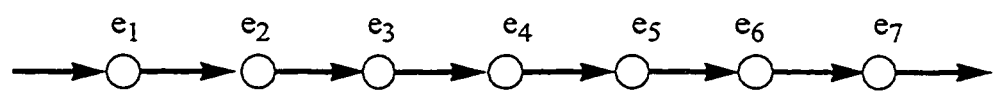


Figure 29: Operation Event Graph of an RPC Interaction

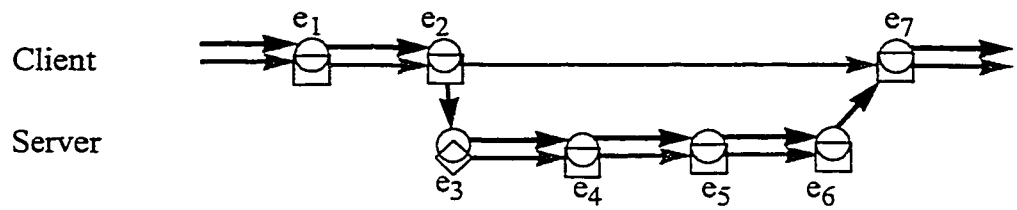


Figure 30: Task and Operation Event Graph of an RPC Interaction

This chapter is arranged in three parts. The first part introduces the requirements, syntax, and assumptions of proper time. The second part describes the conversion from a TOEG to a sub-model (step #4 of Figure 1). The last part provides the proof of proper time's completeness and consistency which also define the requirements for the angio trace specification.

6.1 Proper Time Requirements and Assumptions

Proper time characterizes concurrency using a causal, linear time, non-atomic, non-interleaving model that is a graph grammar. The non-interleaving property properly characterizes concurrency within an operation and captures the task architecture. The causal property constrains the scope of a proper time model to a distributed operation, it segments the task behavior into individual service periods, and it helps to identify task blocking or synchronization. The linear-time property allows proper time models to be derived from execution traces.

The graph grammar of proper time is a node-labelled, edge-labeled, directed, acyclic, finite graph specification. The nodes are recorded events and an edge identifies a cause-and-effect relationship between two nodes.

The different node and edge types characterize the following communication protocol elements:

- *External request initiation*: A request originates from outside the system being monitored.

- *Blocking request initiation*: The initiating task cannot proceed until it receives a reply to a request it has just made.

- *Non-blocking request initiation*: The initiating task makes a service request to another task and the initiating task does not block.

- *Request acceptance*: A blocked responding task accepts a new service request.

- *Synchronization acceptance*: A responding task is already processing a service request but it must accept another message to continue the service.

- *Reply to a blocking request initiation*: A responding task sends the reply to the blocked initiating task. The LQN needs this information to identify the second phase of execution.

- *Reply acceptance*: A blocked initiating task receives a reply, completing its RPC interaction.

This identifies all of the important LQN communication protocol elements.

The assumptions of proper time identify the class of system that it can describe. The assumptions are:

- A task only executes on behalf of an operation.

- For an operation to proceed it needs a task to execute on its behalf.

- Tasks interact solely by point-to-point message communication.

- An operation always successfully completes (i.e., it does not deadlock).

- A task cannot atomically accept a service request and send a service request in the same event.

- The TOEG captures all of the relevant information for analysis purposes (i.e., the event instrumentation did not miss important data). This is a *closed world assumption.*

- Tasks complete processing their current service request before starting another service request, preventing the interleaving of service requests.

Many message passing distributed systems satisfy these assumptions.

6.2 Task and Operation Event Graph Overview

In the following sub-sections it will be shown that a TOEG is constrained in the way its node types can be connected by its typed edges. The constraints are based on the definition of a TOEG node.

In the TOEG a node is a six-port building block (Figure 31), where a port is the source or target of a single edge. There are six ports because a TOEG node has at most, three incoming and three outgoing edges; it is the Cartesian product of a binary graph (i.e., the operation event graph) and a linear graph (i.e., the task event graph).

The six port types are:

- *InTask* is the target of an edge connected to the preceding node in the same task event graph.

- *InOpPd* is the target of an edge connected to the preceding node in the same operation thread and in the same task event graph.

- *InOpExt* is the target of an edge connected to the preceding node that is part of the same operation but not in the same task event graph. When an external event occurs or a message is received by a task this port is the target of an edge.

- *OutTask* is the source of an edge connected to the succeeding node in the same task event graph.

- *OutOpPd* is the source of an edge connected to the succeeding node that is in the same operation thread and in the same task event graph.

- *OutOpExt* is the source of an edge connected to a succeeding node that is part of the same operation but in another task event graph. When an external event occurs or a message is sent by a task this port is the source of an edge.

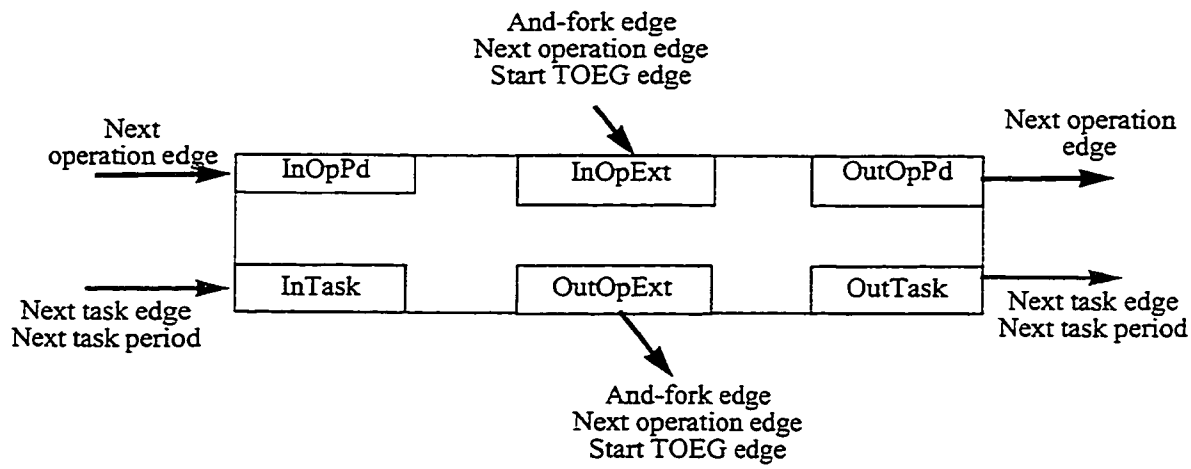


Figure 31: A TOEG Node as a Six-Port Device

A minimal number of node and edge types are provided so that the semantics are clear, consistent, and non-overlapping. For example, each node type has unique properties to allow it (or prevent it) them from connecting to other node types. The justification for selecting the node types is given in section 6.6.2.

Just as nodes are given a type to reflect their properties, there are different types of edges, each of which denotes a type of cause-and-effect relationship between two nodes. In Figure 31, the position of a node's port identifies the edge type that connects with it, and whether it has an incoming or outgoing edge attached. The justification for selecting the edge types is given in section 6.6.1.

The next sub-sections introduce the different node and edge types for each graph type.

6.2.1 Task Event Graph

The task event graph describes the cyclical execution of a task as a sequence of linear subgraphs, one for each service period. A task satisfies the service requests[1] of other tasks one at a time, with the processing in a *service period* being independent of other service periods. This is a run-to completion view of execution. Each service period is also a linear sub-graph of *task events*.

The *task event graph* is an attributed, edge labelled, directed, linear graph. It has two types of nodes and two types of edges:

◇ *"Period start" node*: this is the first node of each service period.

□ *"Task activity" node*: represents an activity that the task performed. It is the default task node type so it is usually not shown.

⟶ *"Task's next node" edge*: its target is the next node in the same task period.

—P➤ *"Task's next period" edge*: its source is the last node of a task's period and its target is the period start node of the task's next service period.

1. External events which initiate an operation are also considered to be service requests.

A service period ends when a node is not the source of a *task's next node edge*. The default edge type is the task's next node edge, which will be abbreviated to *next task edge*.

The task event graph of Figure 32 is an example with all of the task event graph's node and edge types. There are two linear graphs, one for a Client task and one for the Server task. It should be pointed out that the Server task starts a new service period at the diamond but the cause of the new service period is not identified in this figure.

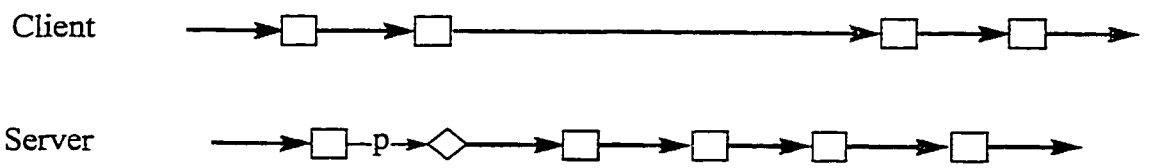


Figure 32: Example Task Event Graph

6.2.2 Operation Event Graph

An *operation event graph* characterizes the concurrency and sequence of events that occur in a scenario. It is made of concurrent threads of execution which are represented as linear sub-graphs and each linear sub-graph is called an *operation thread*. Each operation thread is a linear sub-graph of *operation events*. An operation event graph with concurrent operation threads uses special node and edge types to characterize the causal relationships of the operation threads.

An operation event graph is a node labeled, edge labelled, binary, finite, directed, acyclic graph. The operation event graph node types and edge types are:

- E *"External" node*: it is a marker for an external request.

- ● *"Thread begin" node*: it begins an operation thread.

"Operation activity" node: has an attribute to store operation information, such as an activity identifier.[2]

"And-fork" node: it forks a new operation thread to characterize the introduction of logical concurrency within the operation. It is the only node type with outdegree greater than one.

"And-join" node: it joins two operation threads into a single operation thread. The and-join node is the only node type with indegree greater than one.

"Thread end" node: it finishes an operation thread. It distinguishes between a terminated or incomplete (i.e. deadlocked) operation thread.

"Start the operation" edge (st): its source is an external node and its target is a thread begin node of the thread caused by the external node.

"Operation thread's next node" edge: its target is the next node in the same operation thread that succeeds its source.

"Operation thread's fork" edge (f): its source is an and-fork node and its target is the thread begin node of the forked thread.

An operation may have more than one external node if it has additional separate initiating events. If so, they usually join later. The default edge type is the "Operation thread's next node" edge which is abbreviated to *next operation edge*.

2. A process sending a message to itself is considered to be an activity node.

There are two operation event graphs shown in Figure 33 because there are two unconnected sub-graphs. This means that two different scenarios are shown in the figure. The larger operation event graph also has two operation threads, with the top-most operation thread forking the child operation thread. Both of the operation threads synchronize and then continue execution.

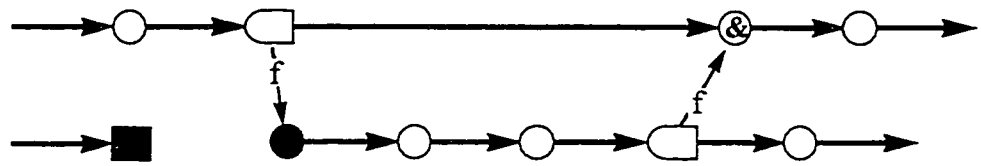


Figure 33: Example Operation Event Graph

The TOEG of Figure 34 combines the task event graph of Figure 32 and the operation event graph of Figure 33. By overlaying the two graphs it is seen that the two scenarios of the operation event graph are ordered by the execution of the Server task (i.e., the task's next period edge). A close examination of the Client task's execution shows that it sends a message to the Server task and then immediately waits for a reply, which is an equivalent RPC interaction using two asynchronous messages. This illustrates the manner in which the TOEG semantics characterize the different type of LQN sub-model interactions.

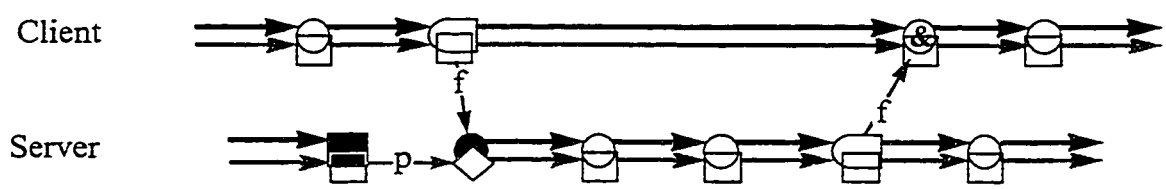


Figure 34: Example Task and Operation Event Graph

The TOEG's semantics are examined in detail next. The TOEG figures will no longer show the icon for the task activity node because it is the default node type for the task event graph.

6.3 TOEG Graph Grammar Semantics

In a valid TOEG, all of the nodes and their preceding or succeeding edges are involved in a node connection found in Table 6. The cause and effect relationship between any two events constrain the set of possible node connections to the fifteen cases found in Table 6 (discussed in sub-section 6.6.2). It should be mentioned that the task period start edges in Table 6 are optional because the task may not continue to execute once its period ends, so there would be no succeeding edges or nodes. This section describes the syntax (structure) and semantics (interpretation) of the TOEG. To clarify the semantics, example graphs are provided in the next section.

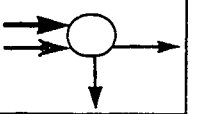
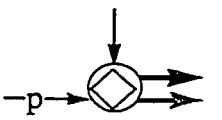
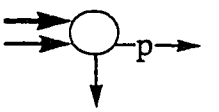
Node Connection Interpretation	Allowed Task Role(s)	Node Connection Figure
(A) External system request.	No task	A 
(B) End of the task period and operation thread.	Any role	B 
(C) An operation activity event.	Any role	C 
(D) Initiation of an RPC interaction.	Initiator or Forwarder	D 
(E) Acceptance of a service request sent as an RPC interaction.	Responder or Forwarder or Replier	E 
(F) Sending the reply to a service request sent by an RPC interaction. The responding task's service period ends.	Replier	F 

Table 6: TOEG Node Connection Figures and Interpretation

Node Connection Interpretation	Allowed Task Role(s)	Node Connection Figure
(G) A blocked initiating task in an RPC interaction receives the reply to its service request. The replying task ended its service period after it sent the reply. An and-join node is not used because there is only one operation thread.	Initiator	G
(H) (1) An initiating task initiates an asynchronous interaction. (2) A replier task sends the reply to a service request sent as an RPC interaction and it does not end it service period but continues executing after sending the reply. (3) A forwarding task forwards the service request to another responding task.	(1) Initiator or (2) Replier or (3) Forwarder	H f
(I) A blocked task that is **not** processing a previously accepted service request now accepts a service request that was sent asynchronously.	Responder or Forwarder or Replier	I f p
(J) A blocked task that is processing a previously accepted service request completes a synchronization by accepting a message. The message was sent as an RPC interaction.	Responder or Forwarder or Replier	J &
(K) A blocked initiating task in an RPC interaction receives the reply to its service request. The replying task did not end its service period after it sent the reply.	Initiator	K f &
(L) A blocked task that is processing a previously accepted service request completes a synchronization by accepting a message. The message was sent as an asynchronous interaction.	Responder or Forwarder or Replier	L f &
(M) A blocked task that is **not** processing a previously accepted service request now begins an operation thread because of an external request.	Initiator	M st p
(N) A blocked task that is processing a previously accepted service request completes a synchronization by accepting an external request.	Responder or Forwarder or Replier	N st &
(O) A blocked initiating task in an RPC interaction is interrupted by an external event.	Initiator	O st &

Table 6: TOEG Node Connection Figures and Interpretation

An observation that is important for characterizing a distributed operation is that when a task is involved in an interaction its future execution is restricted so that it does not violate the semantics of the interaction or communication protocol. These restrictions are identified by the *role* a task has in an interaction. Because a role limits the future execution of a task it also restricts the future events recorded for the task and the node types that correspond to the events, as well as the edges that can connect the nodes. A task's role is important in defining the semantics of the TOEG.

By inspection, in any message exchange there are four roles a task may take on. The first role type is an *initiating task* which requests services from other tasks. The second role type is a *responding task*, where it accepts a service request from an initiating task and satisfies the service request. The third role type is a *forwarding task*, where a task accepts a service request, does some processing, and then forwards the service request to another task for further processing. The last role type is as a *replier task*, where a task sends a reply back to an initiating task to indicate that its RPC has been completed and it can continue executing. The roles that are supported by each node connection and the corresponding interpretation are identified by the column in Table 6 called "Allowed Task(s) Role".

The different roles and node connections characterize the following elements of an external interaction, RPC interaction, synchronization, and asynchronous interaction:

- *External request initiation*: A request was externally generated and it begins an operation. This is item {A} in Table 6.

- *Blocking request initiation*: A task cannot proceed until it receives a reply to a request it has just made. This is item {D} in Table 6.

- *Non-blocking request initiation*: A task makes a service request to another task and it does not block to wait for a reply. This is item {H} in Table 6.

- *Request acceptance*: A blocked responding task accepts a new service request and begins a new period, which are items {E, I, M} in Table 6.

- *Synchronization acceptance*: A responding task is already processing a service request but it is blocked, waiting to accept another message to continue the service. These are items {J, L, N} in Table 6.

- *Sending a reply to a blocking request*: A replier task sends the reply to the blocked initiating task. These are items {F, H} in Table 6.

- *Acceptance of a reply:* A blocked initiating task receives the reply to its blocking request. These are items {G, K} in Table 6.

This characterization satisfies the requirements for identifying the types of interactions in a sub-model (section 6.1).

6.4 Examples of TOEG Interactions and the Task Role Types

To clarify the semantics of the TOEG and the task role types, example sub-graphs are presented for RPC, asynchronous, synchronization, forwarding interactions, as well as the RPC interruption.

6.4.1 Example Interactions

The TOEG figures follow several conventions. Time proceeds from left to right. The consecutive nodes of a task are at the same vertical level. The consecutive nodes of an operation thread can crossover to different tasks (e.g., blocking RPC).

A TOEG of an RPC is shown in Figure 35. The operation event graph resembles a procedure call graph if the tasks were procedures.

A TOEG of an asynchronous interaction is shown in Figure 36.

A synchronization interaction occurs when the synchronizing task has started a service period and it must accept another message to continue execution. There are four possible ways a synchronization occurs. The first case is where the message was sent using a blocking communication protocol (Figure 37). The second case is where the initiating task uses an asynchronous communication protocol to send the message (Figure 38). The third case occurs where a blocked initiating task receives its reply to a service request that used an RPC communication protocol (Figure 39). The procedure call graph analogy breaks down in this case because there are two concurrent operation threads, since the responding task continues execution after sending the reply. This third case is characterized as a new operation thread being forked for the reply. The last synchronization case involves an external event being accepted (Figure 40).

A forwarding interaction involves an initiating task, a forwarding task that receives the initiating task's request, other forwarding tasks that forward the request in a task pipeline, and a replier task. An example is shown in Figure 41, where: the initiating task (Task A) sends the RPC request and blocks, the first forwarding task (Task B) processes the request, and forwards it in an asynchronous fashion to another forwarding task (Task C), Task C processes the request further and forwards it to Task D which replies to the initiating task.

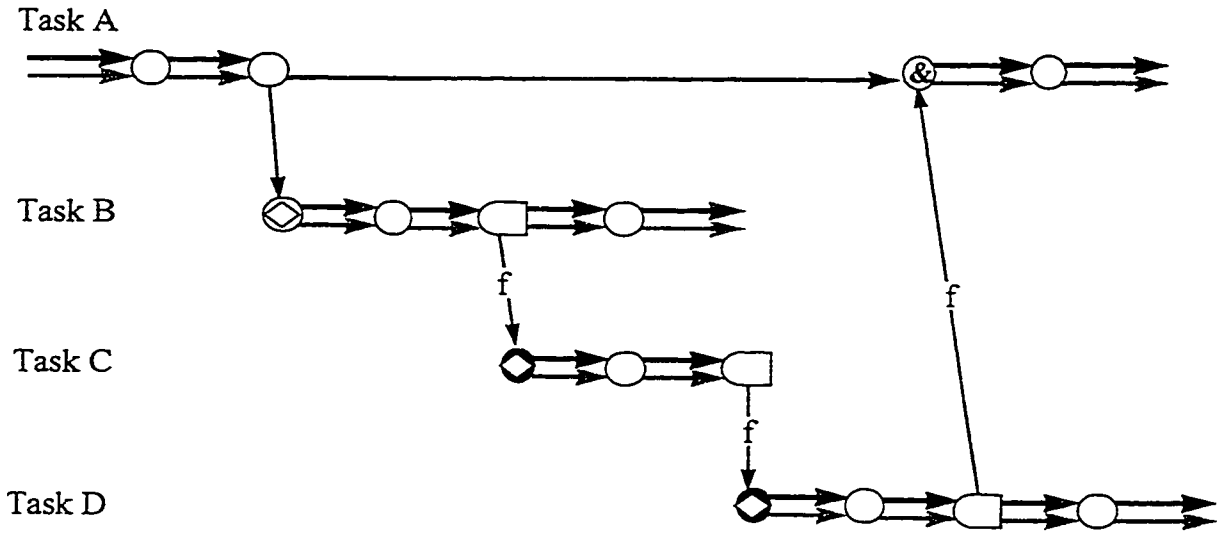


Figure 41: An Example of a Forwarding Interaction with Two Levels of Forwarding

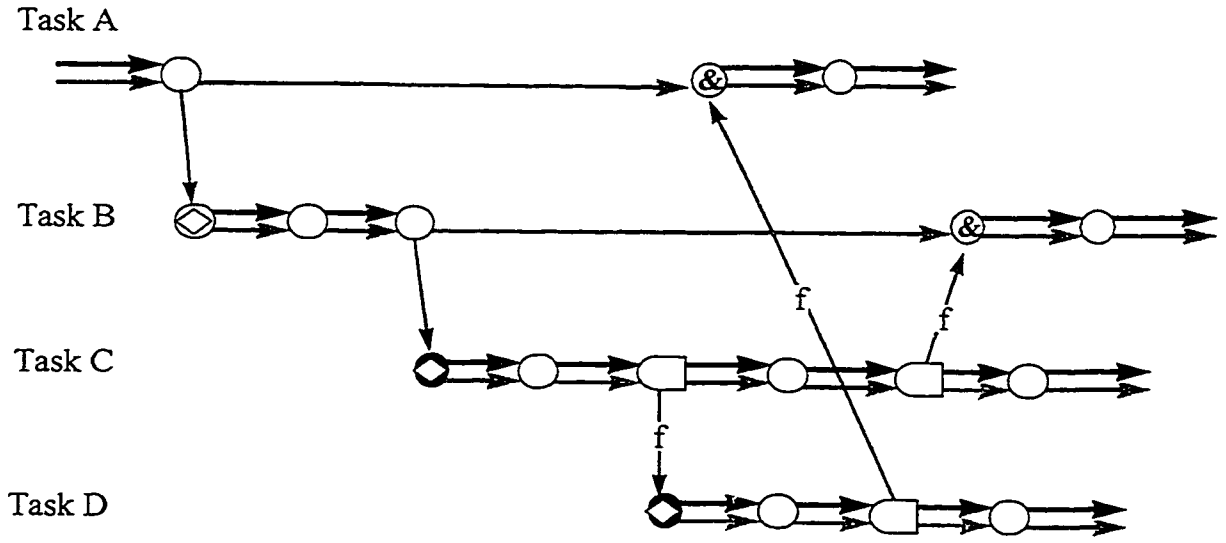


Figure 42: An Example of a Forwarding Interaction with Blocking Communication

More complex variations of forwarding are identified by the sub-model transformation described in the next section. One variation occurs when the request is forwarded using an RPC protocol. An example is shown in Figure 42, where the forwarding chain proceeds as follows: initiating Task A sends its request to Task B which forwards the request to Task C, Task C does some processing and then forwards the request to Task D, and, finally, Task D replies to Task A. In this example, Task B used a blocking communication protocol to forward the request to Task C and Task B receives a reply to its forward communication from Task C.

There are other possibilities in which forwarding can occur. Just as an RPC interaction can be constructed using asynchronous messages, the task communication by forwarding tasks can be blocking or asynchronous. For example, just like Figure 45, the initiating task may use an asynchronous communication protocol to begin the forwarding interaction. The forwarding task (Task C) may also use an asynchronous communication protocol (Figure 41) or a synchronous communication protocol (Figure 42).

Two examples of an interrupted RPC are shown. The first type of interrupted RPC is shown in Figure 43 which is a modified Figure 35. In Figure 43 the RPC is interrupted by an external event such as a software interrupt, so the initiating task resumes execution before receiving a reply to its request. It does receive the reply to its RPC interaction from Task B later on. The second type of RPC interruption occurs when a blocked initiating task accepts a message in place of a reply but the message is not causally connected to its request. Figure 44 is an example of this situation and it is a modified Figure 39. In this example, initiating Task A receives a message from Task C that is interpreted as a reply, but Task C has no causal connection to the RPC. Instead, the reply message from Task B is received later.

The model transformation of the next section handles the RPC interruption by using equivalent semantics that treat the initiating task as using an asynchronous interaction rather than the RPC. This is semantically equivalent because the initiating task, once it

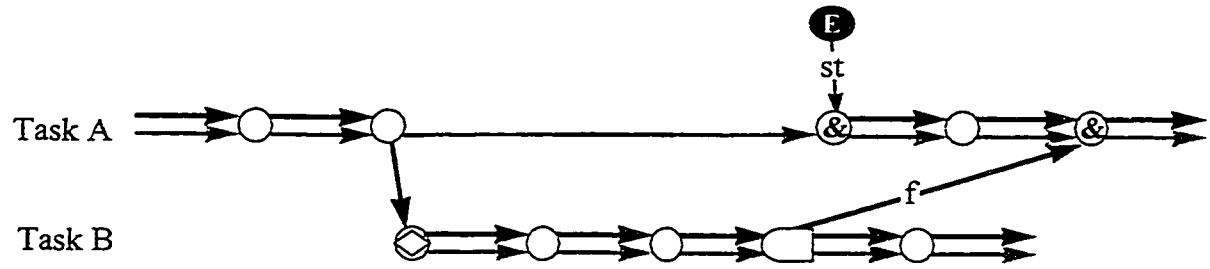


Figure 43: An RPC Interrupted by an External Event

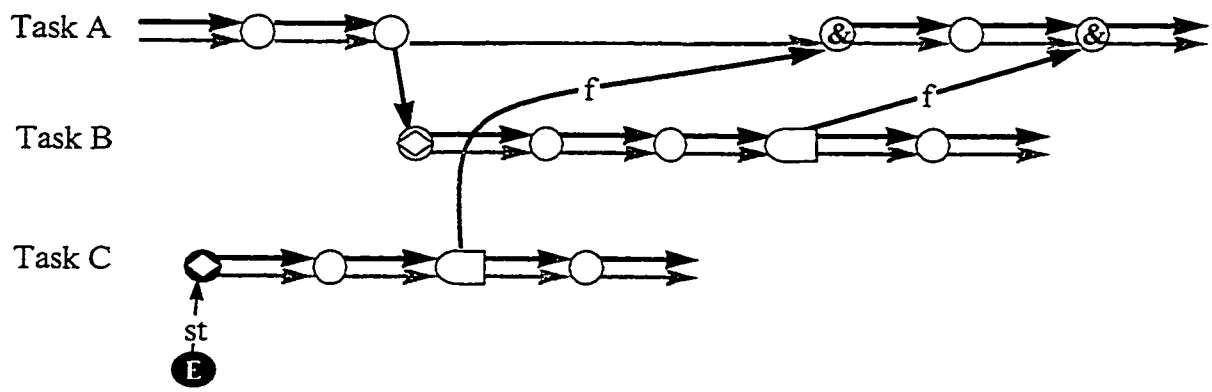


Figure 44: An RPC Interrupted from a Causally Unconnected Event

sends the message, is blocked in a synchronization, waiting for the external event (Figure 43) for the message reception (Figure 44).

Variations of the presented interactions occur when a responding task receives a message in a synchronization.

6.4.2 Task Role Examples

The task roles are described here for the various interactions. A task's role is intimately tied to the interaction that the task is in and the possible roles associated with each interaction are listed in Table 7. They are important for determining the graph rewrite operations for the TOEG to sub-model transformation which is the topic of the next sub-section. The reader will notice that what initially seems to be a responding task may become a forwarding or replier task due to its behavior after receiving the message.

Interaction Type	Initiator Role	Responder Role	Replier Role	Forwarder Role
Asynchronous	Yes	Yes	No	No
RPC	Yes	No	Yes	No
Forwarding	Yes	No	Yes	Yes
External	No	Yes	No	No

Table 7: Communication Protocols and the Involved Task Role Types

The asynchronous interaction has only two task roles, the initiator and the responder. For example, in Figure 36, Task A is the initiating task and Task B is the responding task.

An RPC interaction has two roles as well, an initiator and the responding task that becomes a replier task. For example, in Figure 35, Task A is the initiating task whereas Task B is the replier task.

An RPC interaction can also be constructed using an asynchronous communication protocol and this is shown in Figure 45. This consists of a paired set of asynchronous

messages with the first one being the initiating RPC message and the second being the reply. It is deduced that this is an RPC because the initiating task (Task A) does not record further events so it is assumed to be blocked until it receive the reply message from Task B. In this example, Task B continues executing once the reply message is sent to Task A. This deduction relies on the closed world assumption.

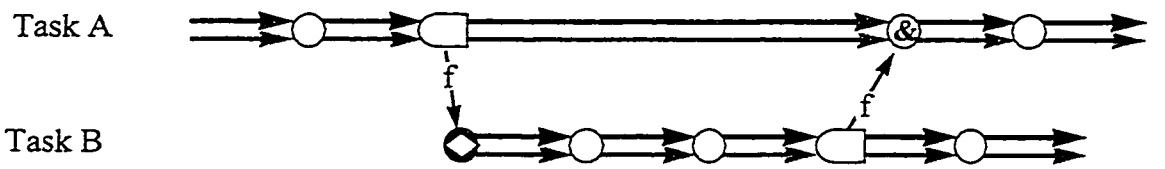


Figure 45: An RPC Interaction using Asynchronous Communication

A forwarding interaction has three roles. Using Figure 41 as the example, the initiating task (Task A) blocks after sending its request. The first forwarding task (Task B) receives the request from the initiating task, does some processing and then forwards it to another task (Task C) for more processing. Task C receives the message does some more processing and then forwards it again. The last task in the forwarding chain acts as a replier, sending the reply message back to the blocked client.

The last type of interaction in the table is the externally initiated interaction. In this case there is no initiating task because the source of the interaction is outside of the system being monitored. However, there is a responding task which receives the external request, which is Task A in Figure 40.

A requirement for constructing performance models is to identify when a replier task or forwarding task continues executing after sending the reply. This is needed to identify the LQN second phase of execution (section 4.4).

6.4.3 Overcoming an Interpretation Conflict of the Second Generation Angio Trace

The ability to characterize synchronization solves a trace interpretation difficulty of the second generation angio trace. The second generation assumed that the order in which a

task accepted a message was not significant but this led to an interpretation conflict: should the two asynchronous messages of Figure 46 be considered an equivalent RPC or should they be left as two asynchronous messages.[3] It looks as if Task A initiates an RPC by forking the operation thread in Task B and that Task B replies to Task A. The second generation angio trace analysis made its best attempt and always assumed it was an equivalent RPC interaction, but this was not always the case. This shortcoming was identified in a large model, where the analysis improperly interpreted two RPCs using asynchronous communication from four asynchronous interactions and this led to a model with a potential deadlock condition (RPCs in both directions).

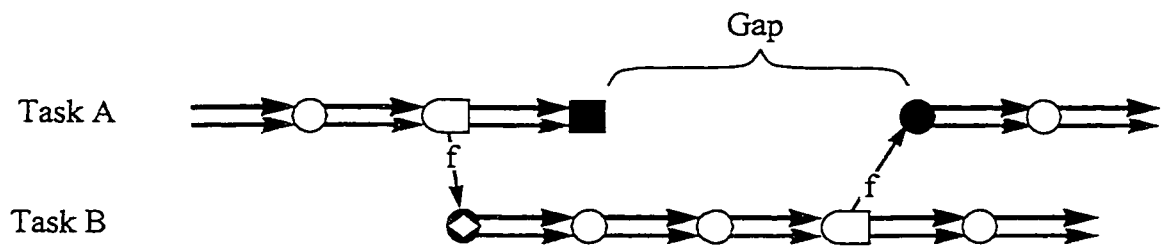


Figure 46: Second Generation Identification of an RPC using Asynchronous Communication

This difficulty is resolved here by characterizing the task service periods and synchronization points. As shown in Figure 45, the interpretation conflict no longer occurs because the next operation and next task edge types fill the gap of Figure 46; if it were not an RPC interaction then a next period edge would fill the gap.

6.5 TOEG to Sub-model Transformation

This sub-section describes the graph rewriting transformation from a TOEG to an LQN sub-model which was implemented by the tool PROGRES. This is the largest graph rewriting transformation. Once the necessary graph rewrite operations are discussed the

3. The second generation angio trace is shown using the TOEG notation for convenience sake.

graph schema that is used to construct them is presented. An overview of how the graph rewrite rules were constructed is provided. The order in which the graph rewriting rules are applied are important and governed by a control algorithm, which is then explained.

6.5.1 Selection of Graph Rewrite Rules

The main focus of the graph rewrite rules are to identify when an interaction occurs and the type of the interaction. There are four types of interactions that are integral to the sub-model: the RPC, asynchronous, forwarding, and external interactions. As alluded to in section 6.4.2 identifying an interaction is complicated, being closely tied to the roles that a task takes on in an interaction.

Coverage of the different ways in which interactions can occur is ensured by enumerating the ways in which the interactions can occur. There are several questions that guide the enumeration:

- When a message is sent, is it sent using a synchronous or asynchronous protocol? If an asynchronous protocol, is it part of an RPC constructed from asynchronous messages?

- When a message is received, is it the start of a new service period, is it a synchronization, or is it a reply?

- Does the replier task continue executing after sending the reply or does its service period end?

- When a forwarding task forwards a message, does the forwarding task continue executing after the sending the message or does its service period end?

Clearly, some of these questions do not apply to all interaction types (e.g., there is no replier in an asynchronous interaction). The forwarding interaction is particularly difficult because the number of intermediate forwarding tasks is not fixed but determined at run time.

The graph rewrite rules that are used in the TOEG to sub-model transformation are derived from enumerating the possibilities listed in Table 8.

There are three graph rewrite rules that are needed in addition to those dedicated to task interactions. A graph rewrite rule is needed to detect the start of a distributed operation and another to detect the end of an operation thread. Finally, a graph rewrite rule simply marks an activity node as having been processed and not being part of an interaction. There are thirty four graph rewrite rules in total.

Interaction Type	Initiator Role	Responder Role	Replier Role	Forwarder Role
Asynchronous	Request is sent asynchronously.	Message reception is (not) a synchronization?		
RPC	Request is sent asynchronously or by a blocking protocol?		Message reception is (not) a synchronization? Replier does (not) continue after sending the reply?	
Forwarding	Request is sent asynchronously or blocking protocol?		Message reception is (not) a synchronization? Responder does (not) continue after sending the reply?	Message reception is (not) a synchronization? Responder does (not) continue after sending the reply? Request is forwarded asynchronously or blocking protocol?
External		Message reception is (not) a synchronization?		

Table 8: Categorizing Graph Rewrite Rules for the LQN Sub-model Transformation

6.5.2 Synthesized Graph Schema and Transformation Conventions

The graph rewriting operation involves developing a synthesized graph schema, that includes the sub-model graph schema and the TOEG graph schema.

The TOEG graph schema of Figure 48 is quite simple. It has two different node classes. The TaskEvent node class is part of the definition of the event types that are associated with a task while the Event node class is used to define the external event type. Each node and edge type of the TOEG is included in the schema.

The synthesized graph schema is straightforward and is shown in Figure 49. The reader should note that it has been developed so that the activity node type of the TOEG is the same as the activity node type of the sub-model.

The constructed graph rewrite rules follow three conventions that arise from the LQN sub-model schema. The first convention is that there are no intervening nodes between an entry node and an and-join node (Figure 47). Secondly, activity information is kept separate from the nodes which describe the flow of execution (e.g., and-fork, and-join). Lastly, and-fork nodes are mapped to a sub-model's request node, an asynchronous interaction edge, and a corresponding new entry node.

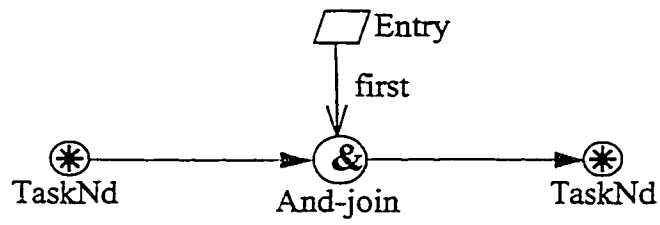


Figure 47: And-Join Sub-graph

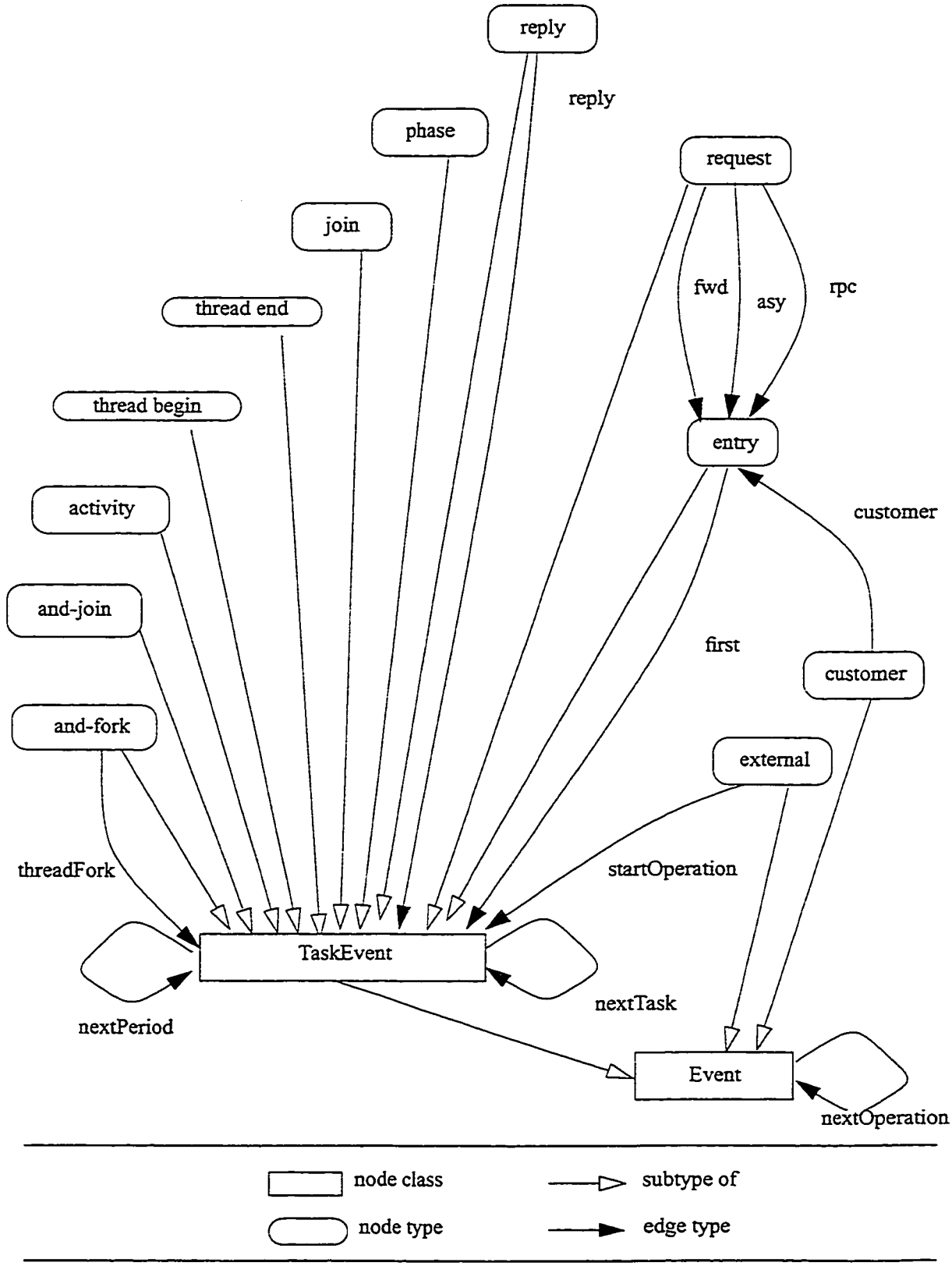


Figure 49: Combined Sub-model and TOEG Graph Schema

6.5.3 Construction of a Graph Rewrite Rule

Graph rewrite rules transform portions of a TOEG into a sub-model by replacing a set of TOEG nodes with nodes from the sub-model graph schema. The construction of a simple graph rewrite rule and a complex graph rewrite rule are considered next. The reader may want to refer back to section 3.3 where graph rewrite rules were introduced.

Each graph rewrite rule follow several conventions. First, a common input parameter to all graph rewrite rules is the node in the host graph that is potentially the beginning of a graph rewrite rule match (called a *trigger node*). Secondly, all graph rewrite rules return the new trigger node to be considered next, as well as any service periods to be processed at a later time. Lastly, when a graph traversal is needed, the graph rewrite rules will seek the shortest causal path using a depth-first search on the task event graph, crossing over to different tasks only when necessary.

6.5.3.1 Simple Graph Rewrite Rule Construction

The simplest graph rewrite rule is a production. A production has a left-hand side sub-graph (LHS) and a right-hand side sub-graph (RHS).

The LHS is used to match a portion of the TOEG using the TOEG graph schema. A LHS match is attempted relative to the trigger node supplied by the control algorithm. For this reason, each production has a *glue node* that is bound to the *trigger node* when a match is attempted. Once the glue node is bound to the trigger node, the LHS matching process continues by attempting to bind the LHS edges and remaining nodes to other nodes in the TOEG since all nodes are structurally related to the glue node. Once the LHS nodes and edges are bound in the TOEG, any supplied operational constraints are tested for validity. A LHS match succeeds if all of the nodes and edges occur in the TOEG and all of the operation constraints are satisfied. If the LHS is matched then the TOEG is rewritten according to the RHS sub-graph.

The LHS of a production includes all of the nodes and edges that are involved in an interaction and are modified by the RHS. It also can include nodes which are not modified but are used to anchor the replacement sub-graph of the RHS.

In a production, the RHS removes or replaces the TOEG node types and edges identified by the LHS. Only the sub-model node types, edge types, and node classes are allowed in the RHS.

Several features of PROGRES allowed the number of productions to be reduced by factoring out the common characteristics of productions. For example, general productions were specialized by passing the type of a node in the LHS as a parameter value. Another useful feature is the ability to specify optional nodes or edges in both the LHS and RHS. The number of productions could have been reduced still further except that PROGRES does not allow the type of an edge to be passed in as a parameter to a production.

6.5.3.2 Complex Graph Rewrite Rule

The complex graph rewrite rule is a transaction. A transaction is a graph rewriting algorithm that either completely succeeds or fails. A transaction is programmed like a procedure except that it can include productions or other transactions.

The forwarding interaction detection was programmed as a transaction because it was broken into two other sub-transactions: the first one detected the start and end of a forward, while the second sub-transaction found the causal execution path to connect the start and end of the forwarding interaction. Each sub-transaction was itself decomposed into productions and a control algorithm. The number of productions was reduced by decomposing the LHS of the sub-graph into common elements which were themselves partitioned into individual productions. This approach reduces the execution time during the matching phase because backtracking occurs for the smaller sub-transactions and productions, compared to a larger transaction. Only when the matching sub-transactions succeeded in entirety was the host sub-graph modified. The approach proved to be very

useful for detecting forwarding because recursion was used to find the causal execution path between the forwarding tasks.

6.5.4 Control Algorithm

A control algorithm manages the order in which graph rewrite rules are selected and applied to a TOEG. It has an initialization component and an analysis component. The control algorithm used in this transformation is outlined in Figure 50. The general operation of the control algorithm is to consider each task's service period in entirety before analyzing another task's service period. This is done to ensure that events are processed in a manner consistent with operation causality, so that events *are not* considered out of order which could cause interpretation difficulties.

The initialization component of the control algorithm begins by taking all of the external nodes and inserting them into the trigger node list. The head of the trigger node list is made the trigger node and the analysis process begins.

The analysis component of the control algorithm consists of three do-while loops and an if-then statement. The outermost loop (line 2 to line 19) ensures that all of the nodes in the trigger node list are processed. The inner loop (line 9 to line 15) processes all the nodes in a service period of a task, sequentially making each node a trigger node, and skipping over those nodes that were part of a previous match. Once a service period has been fully analyzed the head node on the trigger node list is made the new trigger node and the analysis continues. When the control algorithm has terminated only LQN sub-model TOEG node and edge types remain.

The analysis component must reschedule the analysis of an operation thread if an and-join node is encountered (lines 8, 15, 16). The and-join nodes are interface points for two operation threads and the order in which these joining operation threads are analyzed is significant. Figure 51 illustrates this point. If the control algorithm begins its analysis with the operation thread of nodes 7 to 13, it will misinterpret the interaction at and-fork node 4

```
1   Initialize by inserting all the external nodes into the trigger node list.
2   DO
3       DO
4           take the head node off the trigger node list and make it the trigger node
5           IF (the trigger node is an and-join node)
            THEN
6               place the trigger node at the end of the trigger node list
7           ENDIF
8       UNTIL (trigger node is not an and-join node)
9       DO
10          IF (this is not a request, phase, reply, or join node type)
            THEN
11              match the left-hand side of a graph rewrite rule described in Table 8.
12              execute the right-hand side of the matched graph rewrite rule,
                including appending any new trigger nodes to the trigger node list
13          ENDIF
14          the trigger node's closest successor node that is in the TOEG
            graph schema is made the new trigger node
15      WHILE ((the end operation thread graph rewrite rule was not matched) and (the
        trigger node is not an and-join node))
16      IF (trigger node is an and-join node)
        THEN
17              place the trigger node on the end of the trigger node list
18      ENDIF
19  WHILE (trigger node list not empty)
20  convert all external nodes to sub-model customer nodes
```

Figure 50: The Control Algorithm
(Node types are in the schema of Figure 49)

and the and-join node 12. It will be interpreted as an asynchronous interaction rather than a reply to an RPC constructed from asynchronous messages. To avoid this problem the analysis is suspended when an and-join node is detected, the current trigger node is added to the trigger node list, and another trigger node is selected from the trigger node list for analysis. The graph rewrite rules have been developed so that when the appropriate graph rewrite rule matches the and-join, the RHS of the graph rewrite rule replaces the and-join node. The control algorithm performs this rescheduling with the DO-WHILE loop of lines 3 to 8 and the IF-THEN statement of lines 16 to 18.

The complexity of the analysis is of order $nmpr$ and the worst case is n^2mr where:

n is number of nodes in the graph,

m is number of graph rewrite rules,

p is average number of nodes examined by a graph rewrite rule,

r is the number of interactions in the graph.

The worst case complexity arises when each node is examined by each graph rewrite rule and the last graph rewrite rule matches, which is the RPC interrupted from a causally unconnected event (Figure 44). In this situation, all interactions are interrupted RPCs so that all nodes are examined on each asynchronous interaction ($p = n$).

6.5.5 TOEG to Sub-model Example

Figure 52 is an example TOEG that will be used to illustrate the manner in which the control algorithm operates. The TOEG is converted into the LQN sub-model of Figure 53.

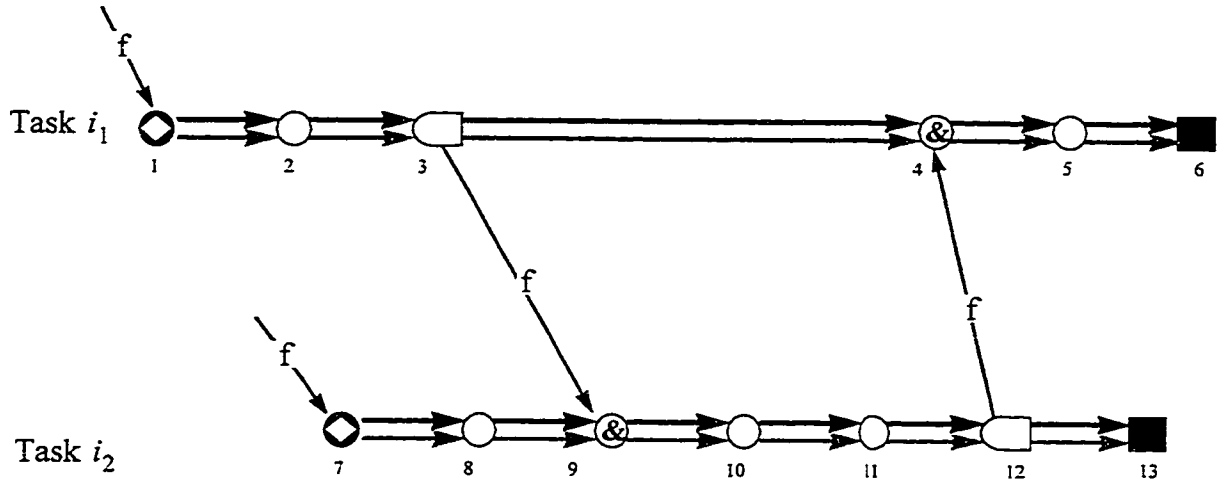


Figure 51: Example of Operation Thread Analysis Scheduling

To avoid confusion, the nodes in the TOEG are numbered while the nodes in the LQN sub-model are identified by letter. The steps in the transformation are:

1) The initialization of the control algorithm will insert node 2 into the trigger node list because it is the target of an external node. It is then made the trigger node.

2) Node 2 matches the begin node graph rewrite rule which has no effect except to advance the trigger node to the next node in the same operation thread (i.e., node 3).

3) Node 3 matches a plain activity production so it is left unchanged, becoming node *c*. Node 4 becomes the trigger node.

4) Nodes {4, 5, 7, 12} match the plain RPC interaction production. Since they are all activity nodes they are retained and become nodes $\{d, i, f, m\}$ in the LQN sub-model. To characterize the RPC interaction the task request node *e* and entry node *h* (i.e., labelled E2) are added, along with the RPC edge. The reply node *n* is added along with its reply edge to indicate that node *f* is the next activity of Task A after

receiving the reply. Node 8 is added to the trigger node list so that the task period it is associated with will be processed at a later time. Node 6 is made the trigger node because it is the next node in the task's period that has not been involved in a production match.

5) Node 6 matches the end thread production which stops processing of this task period. Node 8 is removed from the trigger node list and made the new trigger node.

6) Node 8 matches a plain activity production so it is left unchanged, becoming node *j*. Node 9 becomes the trigger node.

7) Nodes {9, 10, 13, 16} match the production for an RPC interaction using asynchronous messages. These nodes are all removed and replaced with the nodes {*k*, *o*, *r*} to identify the RPC interaction. Node 14 is added to the trigger node list because it is the start of an operation thread that will be processed later. Node 11 is made the trigger node.

8) Node 11 matches the plain activity production so it becomes activity node *l*. All of the remaining nodes in the operation thread have been part or a previous production match the next trigger node is taken from the trigger node list. Node 14 is removed from the trigger node list and made the new trigger node.

9) Node 14 matches the plain activity production and node 15 is made the trigger node.

10) Node 15 matches the plain activity production. Node 17 is made the trigger node because it is the nearest node in the task period that has not been involved in a production match.

11) Node 17 matches the thread end production. There is no successor trigger node and the trigger node list is empty so the main loop of the control algorithm exits.

12) The external node, its start edge, and the target thread begin node {1, 2} are replaced with a Customer node, a customer edge, and a target entry {a, b}. The control algorithm halts.

Although it is not discussed here, a sanity check is made on the resulting LQN sub-model. The sanity check passes if only LQN sub-model node or edge types remain.

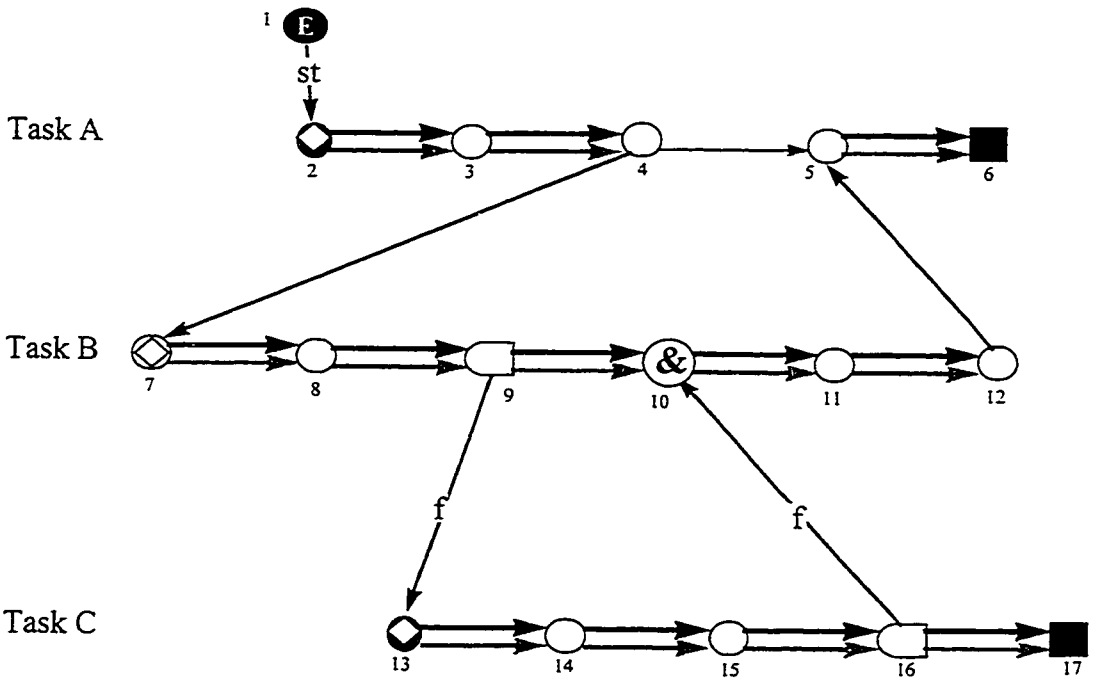


Figure 52: Example TOEG with Identifier *a*

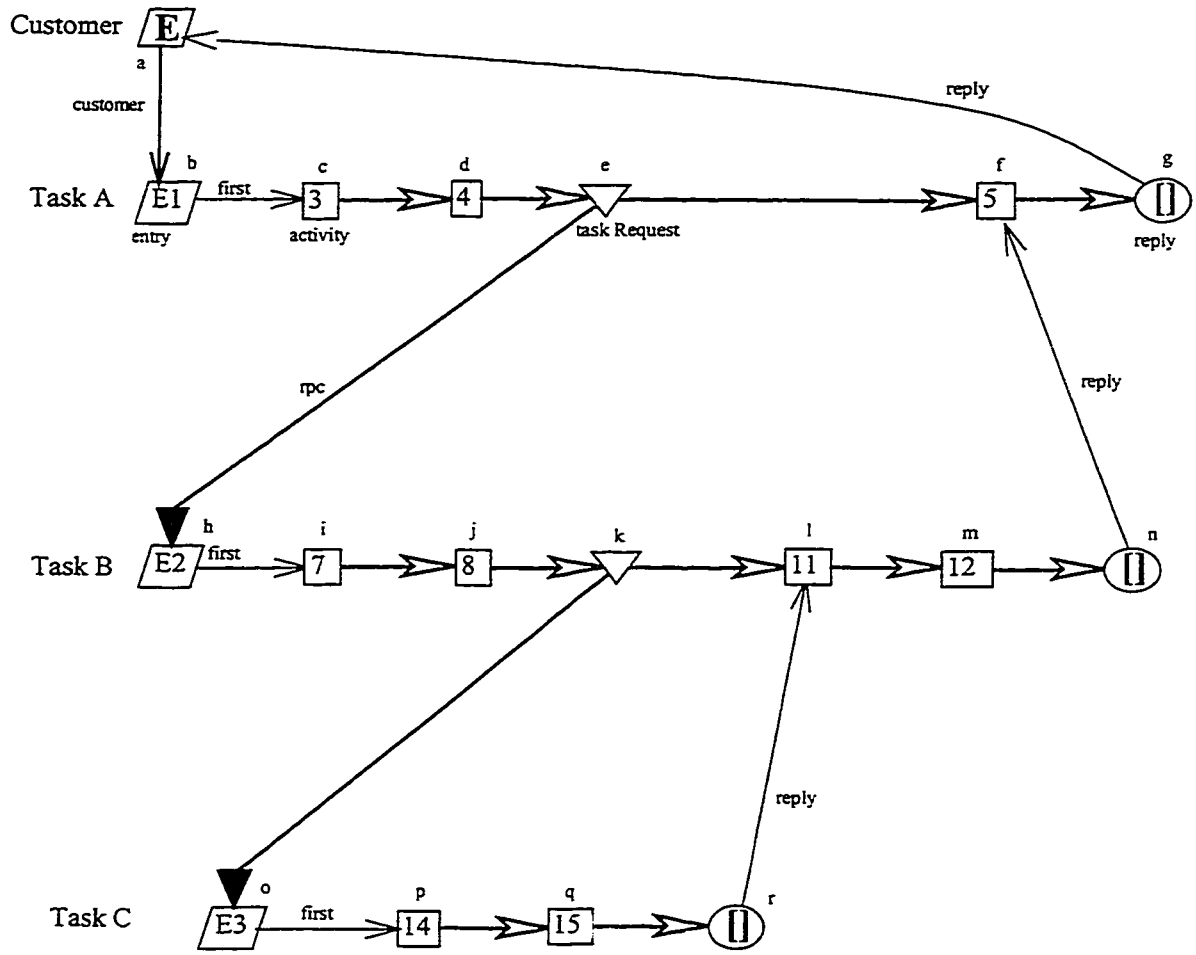


Figure 53: Example LQN Sub-model with Identifier *a*

6.6 The Consistency and Completeness of Proper Time

This section identifies the types of events (i.e., node types) and their relationships (i.e., edge types) that are needed in proper time. This defines the requirements for the angio trace specification because an angio trace must be able to be transformed into a TOEG. It is also shown that all valid distributed operations are characterized by proper time. This information is used by the angio trace specification to ensure that it covers all possible execution behaviors.

This analysis begins by identifying the types of causal relationships that can exist between two events and an edge type is defined for each type of causal relationship (section 6.6.1). Then the node types are identified, as well all of the valid relationships between two nodes (section 6.6.2) which constrains the valid node connections to those listed in Table 6. Finally, it is shown that all distributed operations are characterized by Table 10 which identifies the set of valid predecessor and successor nodes for a given item in Table 6 (section 6.6.3). Table 10, in conjunction with Table 6, define the requirements of the angio trace by identifying, for each event type, what predecessor and successor relationships must be captured in the information stored with each event in the trace.

6.6.1 Defining the Edge Types using Causal Relationships

An edge type defines the type of causal relationship that exists between two nodes. To identify the necessary edge types the various types of causal relationships between two events must be elaborated. There are three different types of causal (precedence) relationships that are identified here: realized causality, potential causality, and operation causality.

6.6.1.1 Realized Causality

Realized causality produces an event ordering that is consistent with both the purpose of the software (as intended by the analyst) and a particular execution of that software.

Realized causality is summarized as "*an event e_1 is a cause of event e_2 if event e_2 cannot occur unless event e_1 has already occurred.*" However, characterizing realized causality is impossible in practice because it necessitates full knowledge of the behavior of each task, the initial value of variables, the execution environment, and the external world. Approximations to realized causality are defined for this research.

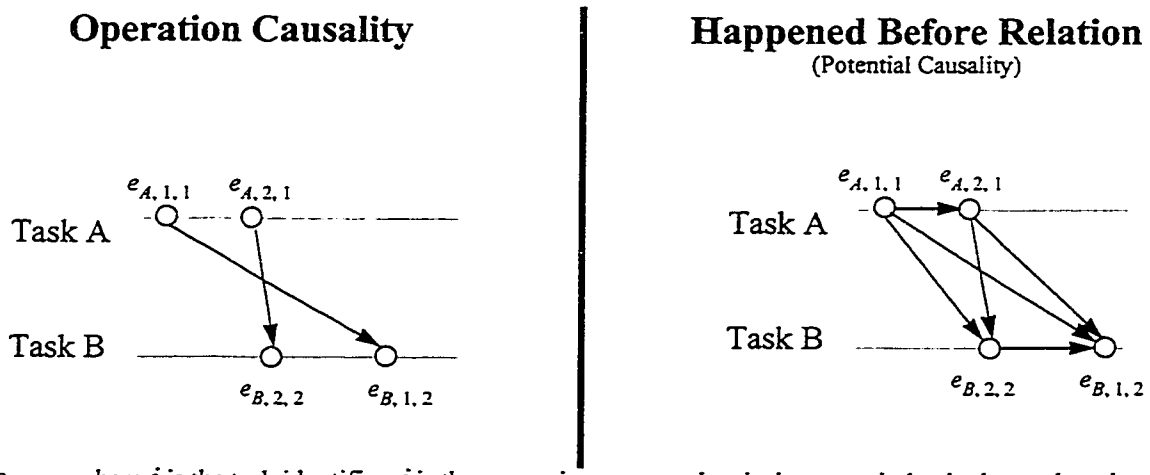
6.6.1.2 Potential Causality

Potential causality produces a temporal ordering of events. It orders events just as they occur in the system, one after the other, using the happened before ordering relation. Potential causality is characterized as the future not being able to influence the past [97]: "*an event e_1 is the cause of event e_2 if event e_1 occurs earlier than event e_2.*" Previous work which used this definition are [97, 161, 4, 41, 162, 143]. It is a weak characterization of realized causality because it results in all previous events being potential causes for later events.

Potential causality provides a limited amount of information to analyze the trace of a distributed system's execution. It assumes all communication is non-blocking, making it difficult to characterize blocking and synchronization interactions. A more restrictive limitation is that it introduces ordering relationships between the events of different operations, treating independent operations as if they were part of a single, system-wide operation. A simple example will illustrate this second point.

In the example of Figure 54 there are two distributed operations. Each operation consists of Task A sending a message to Task B. As shown, two independent external events cause each operation to execute, recording the events of the first operation as $\{e_{A,1,1}, e_{B,1,2}\}$ and the events of the second operation as $\{e_{A,2,1}, e_{B,2,2}\}$. However, as the operation causal ordering shows in Figure 54, there is a delay such that the second message sent by Task A (event $e_{A,2,1}$) overtakes the first message it has sent (event

$e_{A,1,1}$). The happened before ordering of the events is shown in Figure 54, including the transitive ordering components. The happened before ordering includes the additional event orderings of $\{ (e_{A,1,1} \rightarrow e_{A,2,1}), (e_{A,1,1} \rightarrow e_{B,2,2}), (e_{B,2,2} \rightarrow e_{B,1,2}), (e_{A,2,1} \rightarrow e_{B,1,2}) \}$ which are not cause-and-effect orderings because the operations are independent.



$e_{i,j,m}$ where i is the task identifier, j is the scenario name, and m is the event index in that task and scenario

Figure 54: Contrasting Operation Causality and the Happened Before Relation

Potential causality is characterized in the TOEG using the edge types of the task event graph. A temporal, system level view of the execution can be constructed with the next period edges connecting different TOEGs and the next task edges ordering the events in each task period. This is allowed because the target of a task's next period edge can be a service period in the same operation (same TOEG) or in a different operation (different TOEG), as long as it is the same task. Then the whole system execution can be ordered provided there are interacting tasks which are common to the different operations.

6.6.1.3 Operation Causality

Operation causality is a subset of realized causality, including only those causal relationships that are certain to be valid. Operation causality is a conservative form of

realized causality, limiting an events influence to the operation it was recorded for. Operation causality is described as: "*an event e_1 is a cause of event e_2 if there is a sequence of events from event e_1 to event e_2 in the same distributed operation.*" Provided the assumptions of section 6.1 are met then operation causality is equivalent to realized causality, which is sufficient for developing an LQN.

Operation causality is characterized by the edge types of the operation event graph. A start edge identifies a node that is immediately caused by an external stimulus. Nodes in the same thread of execution (i.e., operation thread) are ordered by the next operation edge. Forked operation threads are identified and ordered by the operation thread's fork edge. There is no 'operation thread join edge' because this behavior is characterized by the and-join node and adding this type of edge would unnecessarily expand the number of edge types.

The difference between potential and realized causality is well known [176] but many papers gloss over or ignore the difference (such as [124, 40, 22, 57, 93, 132, 165]). This difference was highlighted in Figure 54 and is captured by the logical fallacy "*Post hoc ergo propter hoc* [154]", or, "If e_1 is earlier than e_2, then e_1 is the cause of e_2." Simply because one event follows another does not mean that one is dependent on the other occurring.[4] The different edge types clearly identify this difference, where the task event graph's edge types characterize potential causality and the operation event graph's edge types characterize operation causality.

6.6.2 Characterizing Operation Causality between Any Two Nodes

This sub-section proves that the node connections of Table 6 are the only valid ways to connect nodes and characterize operation causality.

4. Sproul tells us the philosopher Hume asserted that what we perceive in such sequences is not necessarily a causal relation but a relation of *contiguity*.

This proof is by enumeration and it has three steps. First a general representation of a TOEG node and edge is identified. Then all of the ways in which a TOEG node can be connected to its preceding and succeeding node are enumerated. Lastly, those node connections whose causal interpretation is not consistent with operation causality are eliminated.

The general notation for a TOEG node is the six-port device of Figure 31. For this proof the type of an edge is not important. An edge's type identifies the type of operation causality between two nodes but the proof only needs to identify that a causal relation (an edge) exists.

Based on this six-port building block model, consider all the combinations of ways in which these ports may have arcs attached to them or not. For each port, assign a binary value 1 if it has an arc attached or a 0 if not. A binary number, called the *node connection value* in Table 9, can be constructed where the bit positions are, from most significant bit to least significant bit:

InOpPd, InTask, InOpExt, OutOpPd, OutTask, OutOpExt

This means there are 64 possible values. However, many of these connections are invalid because they violate the causal interpretation of the operation and task event graphs, as well as the TOEG.

The constraints on the causal relationships fall into three categories:

- *Structural constraints*: each node and edge type are unique, having specific interpretations that allow, or prevent, the connecting with other node types.

- *Consistency constraints*: operation and potential causality must be consistent.

- *Interpretation constraints*: the TOEG must be unambiguous in its characterization of causality.

Each constraint type is described below and its effect on the possible node connections considered.

The structural constraints ensure each node has unique properties. The *thread end node* is the only node type to finish an operation thread. The *thread start node* is the only node type that is allowed to begin an operation thread. The *external node* has no cause. The *and-join node* has two causes from different operation threads. The *and-fork node* is allowed to be the cause of events in two operation threads.

The *consistency constraints* ensure both the task and operation event graphs have a consistent causal interpretation. These constraints are derived directly from the assumptions:

- For an operation to proceed it needs a task to execute on its behalf.
- A task only executes on behalf of an operation.
- An operation always successfully completes (i.e., it does not deadlock).

There are two results that follow from these constraints. First, a node cannot be the source of an output next task edge without also being the source of an output next operation edge. Secondly, a node cannot be the target of an input next task edge without also being the source of an input next operation edge. Note that a node can be the source of a next operation edge without being the source of a next task edge because the operation can be continued by another task (i.e., RPC reply).

There are two *interpretation constraints* which are really assumptions. First, a task can either accept a service request or send a service request, but both cannot occur simultaneously for the same node. Otherwise the event ordering is ambiguous because it is not known if the request acceptance preceded the sending of the request, or if the sending of the request preceded the request acceptance. The second interpretation constraint is that

tasks complete their current service request before starting another service request to avoid the interleaving of service periods.

The valid node connections can be found by enumerating the possibilities and identifying those that are consistent with the constraints described above. This is summarized in Table 9 and expanded in Table 1 of Appendix A.

Node Connection Value	Explanation	Invalidated Node Connection Values
N/A	Nodes with no edges are not allowed because an operation thread must have at least two nodes: a begin node and an end node.	0
1	Only an external node is allowed to have an effect without a cause. This is item {A} in Table 6.	2, 3, 4, 5, 6, 7
N/A	Cannot receive a message and send a message in the same node (not (InOpExt and OutOpExt)).	9, 11, 13, 15, 23, 25, 27, 29, 31, 41, 43, 45, 47, 57, 59, 61, 63
N/A	The operation is stopped if there is not an outgoing task edge (OutOpPd) without an outgoing operation edge (OutOpPd or OutOpExt).	10, 26, 50, 58
N/A	A next operation edge output in the same task period (OutOpPd) must have a corresponding next task output edge (OutTask), otherwise the task is deadlocked.	12, 20, 28, 36, 44, 52, 60
14	The receiving task is blocked (no InTask edge) and it becomes unblocked by accepting a message (InOpExt). These are items {E, I, M} in Table 6.	
N/A	A node must have a next operation thread edge as an input (either InOpPd or InOpExt) to proceed to the next node, otherwise the task executes without an operation.	17, 18, 19, 20, 21, 22
30	The task is blocked (i.e., not InOpPd), becoming unblocked by accepting a message on InOpExt, continuing execution of the operation by sourcing edges on OutOpPd and OutTask. This is items {G, K, O} in Table 6.	
N/A	A node must have a task input edge if it has an operation input edge in the same task period.	33, 34, 35, 37, 38, 39, 42, 46
48	The thread end node is the only node type that is allowed to terminate the task and operation event graphs. This is item {B} in Table 6.	8, 16, 24, 32, 40, 56
49	Sending of the reply to an initiating task in an RPC interaction and the replying task finishes its service period. This is item {F} in Table 6.	
51	A blocking request interaction initiation. This is recorded by item {D} in Table 6.	
N/A	N/A. A node cannot have an output operation edge in the same task (OutOpPd) without a corresponding task output edge (OutTask) because an operation cannot progress in the same task without the task progressing.	53

Table 9: Identification of Valid and Invalid Node Connections

Node Connection Value	Explanation	Invalidated Node Connection Values
54	The distributed operation continues in the same task. This is item {C} in Table 6.	
55	An operation thread is forked. This is item {H} in Table 6.	
62	A message reception is accepted and the accepting task was already processing a service request (InTask). This is characterized by items {L, J, N} in Table 6.	

Table 9: Identification of Valid and Invalid Node Connections

A single node connection value may give rise to several of the node connections in Table 6 which are distinguished by the node and edge types. Those sets of nodes which are differentiated by the type information are: {J, L, N}, {K, G}, and {E, M, I}.

6.6.3 Characterizing All Valid TOEG's by Determining Node Reachability

This sub-section proves that proper time characterizes all valid distributed operations because it identifies all valid causal connections between any two events (i.e., TOEG nodes).

This proof of completeness is by enumeration. The reachability table (Table 10) identifies how nodes are connected based on the port definition and edge type information. It is constructed by identifying the valid predecessor and successor node connection types for each node connection of Table 6. This is done by inspection; for each node connection in Table 6, the set of possible predecessor and successor node connection types are identified by matching the outgoing and incoming edge types. However, this list must be pruned by removing the node connections which do not fit the roles of the tasks or where the causal interpretation is invalid.

Node Connection	Node Type	Previous Node Connection in the Same Task Event Graph (InOpPd and InTask)	Previous Node Connection in a Different Task Event Graph (InOpExt)	Successor Node Connection in the Same Task Event Graph (OutOpPd and OutTask)	Successor Node Connection in a Different Task Event Graph (OutOpExt)
A	External	-	-		M, N, O
B	Thread end	C, E, G, H, I, J, K, L, M, N, O	-	E, I, M	-
C	Activity	C, E, G, H, I, J, K, L, M, N, O	-	B, C, D, F, H, J, L, N	-
D	Activity	C, E, G, H, I, J, K, L, M, N, O	-	G, K, O	E, J
E	Activity	B, F	D	B, C, D, F, H, J, L, N	
F	Activity	C, E, G, H, I, J, K, L, M, N, O	-	E, I, M	G
G	Activity	D	F	B, C, D, F, H, J, L, N	-
H	Fork	C, E, G, H, I, J, K, L, M, N, O	-	B, C, D, F, H, J, L, N	I, K, L
I	Thread begin	B, F	H	B, C, D, F, H, J, L, N	-
J	And-join	C, E, G, H, I, J, K, L, M, N, O	D	B, C, D, F, H, J, L, N	-
K	And-join	D	H	B, C, D, F, H, J, L, N	-
L	And-join	C, E, G, H, I, J, K, L, M, N, O	H	B, C, D, F, H, J, L, N	-
M	Thread begin	B, F	A	B, C, D, F, H, J, L, N	-
N	And-join	C, E, G, H, I, J, K, L, M, N, O	A	B, C, D, F, H, J, L, N	-
O	And-join	D	A	B, C, D, F, H, J, L, N	-

Table 10: Reachable Node Connections for a Given Node Connection Type

There are two node connections which are invalid because their interpretation is not consistent with the role of the task and its causal interpretation. They are:

- Node connection D cannot source a next operation edge from port OutOpExt to the target port InOpExt of a target node connection G. The justification is that an RPC initiation (D) should not be a reply message that unblocks another initiating task already in an RPC interaction (G).

- Node connection F cannot source a next operation edge from port OutOpExt to target port InOpExt of a target node connection E or J. The justification is that an RPC reply (F) cannot be considered to be the initiation of another RPC interaction (E or J).

The usefulness of enumeration was made evident when, to the author's surprise, these invalid node connections were found. These invalid node connections are not shown in Table 10.

An unexpected node connection that was uncovered while checking for valid connections was an interrupted RPC due to an external event, item (O) in Table 6.

6.7 Correctness of the TOEG to Sub-model Transformation

Correctness of the TOEG to sub-model transformation is concerned with completeness, consistency, and validation.

Completeness of the transformation means that a graph rewrite rule is provided for all nodes and their possible connections. Completeness was addressed at four levels. First, a default graph rewrite rule was provided for all node types where appropriate (e.g., activity node). Secondly, all of the possible types of interactions identified by the reachability table (Table 10) were enumerated (taking into account the task's role) and a graph rewrite rule was provided for each unique case. Third, all of the possible interactions that are important for constructing an LQN were enumerated in section 6.5.1 and a unique graph rewrite rule is provided for each one. Lastly, it is ensured that the control algorithm matches all of the

events in a TOEG because it examines each event in each service period, sequentially making each node in the service period a trigger node, and matching each trigger node to a graph rewrite rule.

Consistency of the transformation means that the application of the graph rewrite rules is deterministic. Consistency was addressed by ordering the graph rewrite rules to avoid conflicts and choosing the "best" graph rewrite rule when several alternatives were available. The ordering was arrived at by partitioning the graph rewrite rules based on the node type of the glue node, and then ordering the graph rewrite rules within each partition. The control algorithm gave priority to larger graph rewrite rule because a larger graph rewrite rule match provides a more descriptive model. An example of the ordering is that the graph rewrite rule to detect an RPC constructed from two asynchronous messages will match before a graph rewrite rule to detect a plain asynchronous interaction because the RPC is more descriptive behavior.

The implementation was validated by providing 85 individual test cases so that each graph rewrite rule was tested at least once. A sanity check is incorporated in the control algorithm so that, if an interaction cannot be matched then the control algorithm halts. For example, if the invalid node connections described in section 6.6.3 occurred, the control algorithm halted because they could not be matched against a graph rewriting rule.

While developing the proof of correctness and consistency the author was surprised to find that several types of interactions were found that had been missed. This included the interrupted RPC, and the forwarding of a request using a blocking communication protocol. This proved the usefulness of the formal approach.

Chapter 7.0 Angio Tracing and the Transformation to a TOEG

Tracing a sequential program is used to understand how the program executes and it is easily done. Tracing a concurrent application on a single processor is more difficult because events are recorded in an interleaved fashion in the address space of each task, so it is difficult to determine the event ordering. Tracing a concurrent application on a tightly coupled multi-processor is even more difficult because events can be recorded simultaneously by the different processors. However, a loosely coupled distributed system is the most difficult type of system to trace. In this environment there is no common clock reference so it is even harder to order the events. The ordering of events is further complicated because a task will record events from different, unrelated distributed operations. The angio trace was designed to trace a distributed system with multiple, simultaneously executing distributed operation.

Proper time defines the requirements for the angio trace specification because each angio trace is used to construct a TOEG. So, an angio trace must have enough information to recover the TOEG's node and edge types of Table 6, as well as the way in which the nodes can be connected together (Table 10). This requires the angio trace specification to characterize operation causality, potential causality, the communication protocol elements, and task synchronization.

A simple definition of an angio trace is that it is a set of events and ordering relations that order the events. This is emphasized here because the figures in this chapter which depict an angio trace will often employ the TOEG icon notation to show the trace as a graph (see section 6.2). This makes it easier for the reader to understand how the trace events are used to construct a TOEG.

This chapter defines an angio trace specification and the transformation from an angio trace to a TOEG (step #3 of Figure 1). The instrumentation specification is independent of the host-language, operating system interface, processor architecture, or network connections.

7.1 Angio Trace Requirements and Assumptions

Angio tracing uses a new type of logical clock to characterize potential and operation causality, independent of environmental factors such as scheduling. It does not require a global system clock or clock synchronization mechanisms so it can be used to monitor a distributed system (section 2.5). Angio tracing allows events to be interleaved so that multiple angio traces can be recorded simultaneously. It is appropriate for the type of environment described in section 1.3. The angio trace specification also satisfies the requirements of proper time (section 6.1).

The assumptions of angio tracing are:

- The trace captures all of the appropriate information for constructing a performance model. This is a *closed world assumption.*

- Time stamp information can be sent with a message.

- Events are recorded and gathered by a data collection system that does not miss any events (e.g., an instrumentation system [164] or monitor [172, 71]).

- Message communication is reliable but messages may be delivered in an arbitrary order.

- Monitoring does not change the behavior of the distributed operation.

- If a task maintains state information about a distributed operation's execution, this information does not affect the behavior of other distributed operations.

- The cause of a message arrival at a task is the event that sent it.

The manner in which angio tracing characterizes operation causality relies on the last two assumptions.

7.2 Definition of an Angio Trace

The novelty of the angio trace is that it constructs two different types of graphs from a set of events whereas the classical partial-order approach can only construct a single type of graph. Because of this difference an angio trace uses a different strategy for ordering the events.

Each angio trace event records a *task time stamp* to construct the task event graph and an *operation time stamp* to construct the operation event graph. Instead of a single event ordering relation, an angio trace uses several ordering relations to order the events. Each ordering relations is associated with a TOEG edge type. An *event predicate* identifies the type of an event and it serves as guard condition for selecting an *ordering relation* to order two events. An angio trace is:

Definition 7.1 An angio trace is a set of recorded events N that have angio trace time stamps assigned in accordance with an angio trace specification.

Definition 7.2 An angio trace specification is $G_{\text{Trace}} = (A_n, M_n, P, O, M_e)$ where:

A_n is the alphabet of event time stamps,

$M_n : N \rightarrow A_n$ are the rules for assigning time stamp values to events,

P is a set of event predicates,

O is a set of partial-ordering relations,

$M_e : P \rightarrow (O_{e1}, \ldots, O_{en})$ is the mapping of a predicate to one, or more, valid ordering relations.

Each of these items is discussed next.

7.2.1 Angio Trace time stamp Values

The fields that make up the task and operation time stamps are quite different. They were developed using the same considerations as proper time, namely the structural, consistency, and interpretation constraints of section 6.6.2.

In addition to these constraints, there are three properties that the time stamp values and ordering relations must satisfy to be used as a logical clock, and these are proven by inspection. First, each time stamp must have a unique value or the event ordering relation(s) must provide a default scheme for ordering events with identical time stamps. The adopted approach was to ensure that each time stamp value is unique. Secondly, the time stamp values should be monotonically increasing, although there can be gaps in the time stamp

values. The adopted approach was that operation time stamps are sequentially indexed so that missing events can be detected (i.e., any gaps are attributed to missing events). Task time stamp value are sequentially indexed. An additional property needed for angio tracing is that the two time stamp values of successive events in the same task must be synchronized: two events A and B cannot have time stamp values where the task time stamps indicate that event A occurred *before* event B and the operation time stamp indicates that event A occurred *after* event B, and *vice-versa*.

A *task time stamp* consists of:

- A unique *task identifier* for each task event graph.
- A *task period index* is a counter that increments each time a period starts. It orders the service periods of a task.
- A *task event index* is a value that orders the events within a service period.

Task time stamp monotonicity is a result of the period and event index values always increasing. The task identifier provides uniqueness of the time stamp values.

An *operation time stamp* consists of:

- A unique *operation name* that associates an event with a scenario.
- A unique *operation thread identifier* that is assigned when an operation thread begins.
- A *thread event index* is a counter which increments when an event is recorded. It orders the events of an operation thread.
- *Event type* information that is used by the event predicates.

The operation time stamp monotonicity is provided by the thread event index values always increasing. Uniqueness of an operation time stamp is provided by the operation name and operation thread identifier: operation thread identifiers are unique within the scope of an operation name and the operation name must be globally unique.

The event type information closely follows the node types of the TOEG, so only the differences are discussed. The event types are:

- *External event*: it is a marker for the initiation of an operation and it is also the start of an operation thread.

- *Begin event*: identifies the start of an operation thread.

- *Activity event:* records an identifier for a message received, a message sent, or an action taken by a task.

- *Fork event*: connects the forked child operation thread with its parent operation thread. To identify the forked child operation thread the fork event records the new operation thread's identifier.

- *Half-join event*: signals the end of the current operation thread but not the service period of the task. To identify the child operation thread the half-join event records the new operation thread's identifier. It is identified with the icon  .

- *End event*: the operation thread and the task's service period both end.

To identify the start of an operation thread, the first event of an operation thread has either a begin or external event type. It should be emphasized that to identify that an operation thread has completed, the ending event is either a half-join or thread end event type. Additional information is recorded with the event type which is used to order the operation threads.

The event notation that is used combines the task time stamp and the operation time stamp as follows:

Definition 7.3 An event e has the time stamp values

$$e = \left[\frac{\text{Operation Timestamp}}{\text{Task Timestamp}}\right] = \left[\frac{j,k,\ m,l}{i,c,v}\right], \text{ where:}$$

j is the *operation name* for each operation event graph,

k is the *operation thread identifier*,

m is the *thread event index,*

$l = \{Ex, Be, Ac, Fk, HJo, En\}$ is the event type information including information specific to each event type,

i is the *task identifier* for each task event graph,

c is a task service *period index*,

v is a *task event index.*

An operation thread is identified with the operation name and the operation thread identifier, such as $|j,k|$. If an object-oriented system is being monitored then the task identifier should include the class name and instance number of the executing object.

Some fields require a particular initialization value. They are defined as: v_o is the initial value for the task event index, c_o is the initial value for the task period index, and m_o is the initial value for the thread event index. When a time stamp field is cleared it is assigned the *empty value* which is designated as $\varnothing$. The empty value $\varnothing$ means that no valid time stamp value is assigned to that field.

7.2.2 Comparing the TOEG And-join node with the Angio Trace Half-join Event

A half-join event differs from the TOEG and-join node. The TOEG *and-join* node is preceded by two events in different operation threads. The purpose of the half-join event is to order the two operation threads involved in a synchronization. In an angio trace, two *half-join* events precede a new operation thread that results from a synchronization. One half-join event ends the current operation thread of a task. The second half-join event is recorded with the operation time stamp of the received message, so, the ordering of the send and receive events can always be established. If the second operation thread is the result of a fork event then the thread begin event will also be recorded. Both half-join events are ordered using the task time stamp. Two half-join events usually form a synchronization, hence the name *half-join event.*

The manner in which the half-join events characterizes the five types of synchronization (Figure 37 to Figure 40, and Figure 43) are shown in Table 11. This table shows the equivalence of a TOEG sub-graph and the angio trace event orderings. In each case, the TOEG and-join node is characterized by several events that are enclosed in the box and the half-join node is shown as . Each angio trace characterization is unique so that each case is easily identified.

The event predicates are discussed next.

Interpretation	TOEG Model Characterization	Anglo Trace Model Characterization
Responder task synchronization from an RPC initiation (Figure 37).	Task A Task B	Task A Task B
Responder task synchronization from an asynchronous interaction (Figure 38.	Task A f Task B	Task A f Task B
Initiating task receiving a reply to its RPC and the responding ask continues execution (Figure 39).	Task A f Task B	Task A f Task B
Synchronization from an external event (Figure 40).	E st Task A	Task A
RPC interruption (Figure 43).	E st Task A	Task A

Table 11: Angio Trace Characterization of Proper Time Synchronization

7.2.3 Angio Trace Event Predicates for Selecting an Ordering Relation

The event predicates serve several purposes. They define what information is recorded with the event type of an event. Some event predicates are used to determine the type of an event. They can also be used to determine information for identifying a succeeding or preceding event, which is used when defining the ordering relations. A convention for using the event predicates is that a time stamp field with a "-" may take on any acceptable value. The event predicates that are defined are:

fork(e, (j,k)) True if event e is a *fork* event that forked the operation thread $|j,k|$, otherwise it is false. This is deduced from: (1) the parent event e is a fork event type, (2) event e recorded the child operation thread's identifier, and (3) the child begin event recorded the operation time stamp of its parent fork event. To test if event e is a fork event the operation thread field can take on any value (e.g., *fork*(e, -)).

hJoin(E, (j,k)) If operation thread $|j,k|$ is caused by one or more half-join events, then the half-join events are assigned to the set E and the predicate is true, otherwise the predicate is false. This is deduced from: (1) the half-join event(s) being identified by their event type, (2) the half-join event(s) record the resulting operation thread's identifier, and (3) the begin event of the resulting operation thread records the operation time stamp of its parent half-join event(s).

isHJoin(e) True if event e is a half-join event, otherwise it is false.

external(e) True if event e is an external event, otherwise it is false.

begin(e) True if event e is a begin event, otherwise it is false.

end(e) True if event e is an end event, otherwise it is false.

activity(e, V) True if event e is an activity event that also recorded the application level information V, otherwise it is false. To test if event e is an activity event the application information field can take on any value (e.g., *activity*(e, -).

last(i, c, e) True if event e is the last event recorded in period c of task i, otherwise it is false. This is found by traversing the task event graph of task i in period c until the period index changes or there are no further events recorded for the task.

exist(e) True if event e is an event that occurs in the trace, otherwise it is false.

These predicates are used to define the event ordering relations which is the next topic.

7.2.4 Angio Trace Event Ordering Relations

An angio trace has six event ordering relations to identify a given event's succeeding or preceding event in the task event graph or the operation event graph. The ordering relations are $O = \{>^T, <^T, >^{At}, >^{Ao}, <^{At}, <^{Ao}\}$. Each relation is antisymmetric and transitive. In general, if an ordering relation cannot be satisfied then the null event ($\varnothing$) is assigned to the unassigned event. The ordering relationships are interpreted as follows:

$>^T(e_1, e_2)$ orders events in the task event graph. Given the event e_1, it assigns the succeeding event in the task event graph to e_2. If there is no succeeding event in the task event graph then the null event is assigned to

e_2.

$>^{At}(e_1, e_2)$ orders succeeding events in the same operation thread. Given the event e_1, it assigns the succeeding event in the same operation thread to e_2. Otherwise the null event is assigned to e_2.

$>^{Ao}(e_1, e_2)$ orders succeeding operation event graph events that are not in the same operation thread (e.g., a fork event and its child begin event). Given the event e_1, it assigns the succeeding operation event graph event that is not in the same operation thread to e_2. Otherwise the null event is assigned to e_2.

The inverse ordering relationships are $<^T(e_1, e_2)$, $<^{At}(e_1, e_2)$, and $<^{Ao}(e_1, e_2)$, but in these cases the event e_2 is provided and the preceding event e_1 is sought. By convention, the direction of the "<" or ">" indicates which of the two events is being ordered: "<" is used to look for the preceding event and ">" is used to look for a succeeding event. For example, $<^T(e_1, e_2)$ indicates that event e_2 is provided and the preceding event in the same task period is begin sought.

The interpretation of the different ordering relations is directly related to the representation of a TOEG node as a six-port building block (Figure 31). The ordering relations are associated with the TOEG node's ports as follows:

- $>^T(e_1, e_2)$ and $<^T(e_1, e_2)$ mean that port OutTask of source node e_1 is connected to port InTask of target node e_2.

- $>^{At}(e_1, e_2)$ and $<^{At}(e_1, e_2)$ mean that port OutOpPd of source node e_1 is connected to port InOpPd of target node e_2.

- $>^{Ao}(e_1, e_2)$ and $<^{Ao}(e_1, e_2)$ mean that port OutOpExt of source node e_1 is connected to port InOpExt of target node e_2.

These ordering relations are defined in Table 12 and Table 13 using the event notation of Definition 7.3 and the event predicates described in section 7.2.3. The predicate evaluation follows a short circuit approach for the sake of simplicity: when a predicate evaluates to true then no other predicates are evaluated. There are two conventions that are used in Table 12 and Table 13 that should be explained. First, the relation $\rightarrow$ is logical inference (i.e., "$x \rightarrow y$" is interpreted to mean "if (x) then y") and it is not the happened before relation. Secondly, as described in the previous sub-section, a time stamp field with a "-" may take on any acceptable value.

A single angio trace instrumentation statement (Figure 1) may record several events. In this situation the ordering of the sequence of events is significant, so several rules constrain the ordering to be consistent with the TOEG. First, if an instrumentation statement records one or more and-fork events(s) and an activity event, the activity event is the earliest event. Conversely, if an instrumentation statement records an activity event along with a begin or and-join event(s), the activity event is the last event recorded. For instance an N-way fork is described by a sequence of 2-way and-fork events, preceded by an activity event.

Event Type of Given Event $\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert$	Successor Event e_2 in the Task Event Graph $>^{T}(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, e_2)$	Predecessor Event e_1 in the Task Event Graph $<^{T}(e_1, \left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$
Activity event $activity(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, -)$	$exist(\left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert) \rightarrow e_2 = \left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert \vee$ $(exist(\left\lvert \frac{-,-,-,-}{i,c+1,v_o} \right\rvert) \wedge last(i,c,\left\lvert \frac{j,k,m}{i,c,v} \right\rvert)) \rightarrow \left\lvert \frac{-,-,-,-}{i,c+1,v_o} \right\rvert \vee$ $e_2 = \varnothing$	$(v \neq v_o) \rightarrow e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert \vee$ $(c \neq c_o \wedge v = v_o) \rightarrow last(i,c-1,e_1) \vee$ $e_1 = \varnothing$
External event $external(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, -)$	$e_2 = \left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert$	$(c \neq c_o) \rightarrow last(i,c-1,e_1) \vee$ $(isHJoin(\left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert)) \rightarrow (e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert) \vee$ $e_1 = \varnothing$
Fork event $fork(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, -)$	$e_2 = \left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert$	$e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert$
End event $end(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$	$\neg exist(\left\lvert \frac{-,-,-,-}{i,c+1,v_o} \right\rvert) \rightarrow e_2 = \varnothing \vee$ $e_2 = \left\lvert \frac{-,-,-,-}{i,c+1,v_o} \right\rvert$	$e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert$

Table 12: Task time stamp Ordering Relations For a Given Event

Event Type of Given Event $\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert$	Successor Event e_2 in the Task Event Graph $>^{T}(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, e_2)$	Predecessor Event e_1 in the Task Event Graph $<^{T}(e_1, \left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$
Begin event $begin(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$	$e_2 = \left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert$	$(c \neq c_0 \wedge v = v_0) \rightarrow last(i,c-1,e_1) \vee$ $(isHJoin(\left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert)) \rightarrow (e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert) \vee$ $(activity(\left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert, -)) \rightarrow (e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert)$ $\vee\ e_1 = \varnothing$
Half-join event $isHJoin(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$	$e_2 = \left\lvert \frac{-,-,-,-}{i,c,v+1} \right\rvert$	$e_1 = \left\lvert \frac{-,-,-,-}{i,c,v-1} \right\rvert$

Table 12: Task time stamp Ordering Relations For a Given Event

Event Type of Given Event $\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert$	Successor Event e_2 in the Same Operation Thread $>^{At}(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, e_2)$	Successor Event e_2 in a Different Operation Thread $>^{Ao}(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert, e_2)$	Predecessor Event e_2 in the Same Operation Thread $<^{At}(e_1, \left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$	Predecessor Event e_2 in a Different Operation Thread $<^{Ao}(e_1, \left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$
Half-join event $isHJoin(\left\lvert \frac{j,k,m,-}{i,c,v} \right\rvert)$	$e_2 = \varnothing$	$e_2 = \varnothing$	$e_1 = \left\lvert \frac{j,k,m-1,-}{-,-,-} \right\rvert$	$e_1 = \varnothing$

Table 13: Operation Ordering Relations For a Given Event

7.2.5 An Example Angio Trace

An example angio trace is shown in Figure 55 by overlaying the trace events on the TOEG that would be constructed from the events. The trace involves a client task (Task A), a middle server task (Task B), and a pure server task (Task C). It begins at the external input event $\left|\frac{a,1,\ 1,\ Ex}{A,1,\ 1}\right|$ which is the first event of the first period of Task A. Task A then initiates an RPC ($\left|\frac{a,1,\ 3,\ Ac}{A,\ 1,3}\right|$) which completes with its next event ($\left|\frac{a,4,\ 4,\ Ac}{A,\ 1,4}\right|$). Task B receives the request, does some processing, sends a message to Task C using an asynchronous protocol ($\left|\frac{a,1,6,\ (Fo,\ (a,2))}{B,\ 1,3}\right|$), and waits for a response. Task C receives the request and sends a response to Task B ($\left|\frac{a,2,4,\ (Fo,\ (a,3))}{C,\ 1,4}\right|$). After Task B receives Task C's response, it does some processing and then replies to Task A ($\left|\frac{a,4,\ 3,\ Ac}{B,\ 1,9}\right|$). This is a synchronization and it is composed of the events $\left|\frac{a,1,\ 7,\ (HJn,\ (a,4))}{B,\ 1,4}\right|$, $\left|\frac{a,3,\ 1,\ (Be,\ (a,2,4))}{B,\ 1,5}\right|$, $\left|\frac{a,3,\ 2,\ (HJn,\ (a,4))}{B,\ 1,6}\right|$, and $\left|\frac{a,4,\ 1,\ (Be,\ (a,\ 1,\ 7),\ (a,\ 3,\ 2))}{B,\ 1,7}\right|$.

The TOEG that would be generated from the angio trace is shown in Figure 52. One change that occurs in this transformation is that the angio trace external event in Figure 55 is expanded into a TOEG external node and a begin node. Another change is that Task B's events that identify its synchronization with Task C have been replaced with a TOEG and-join node just like Table 11 indicates.

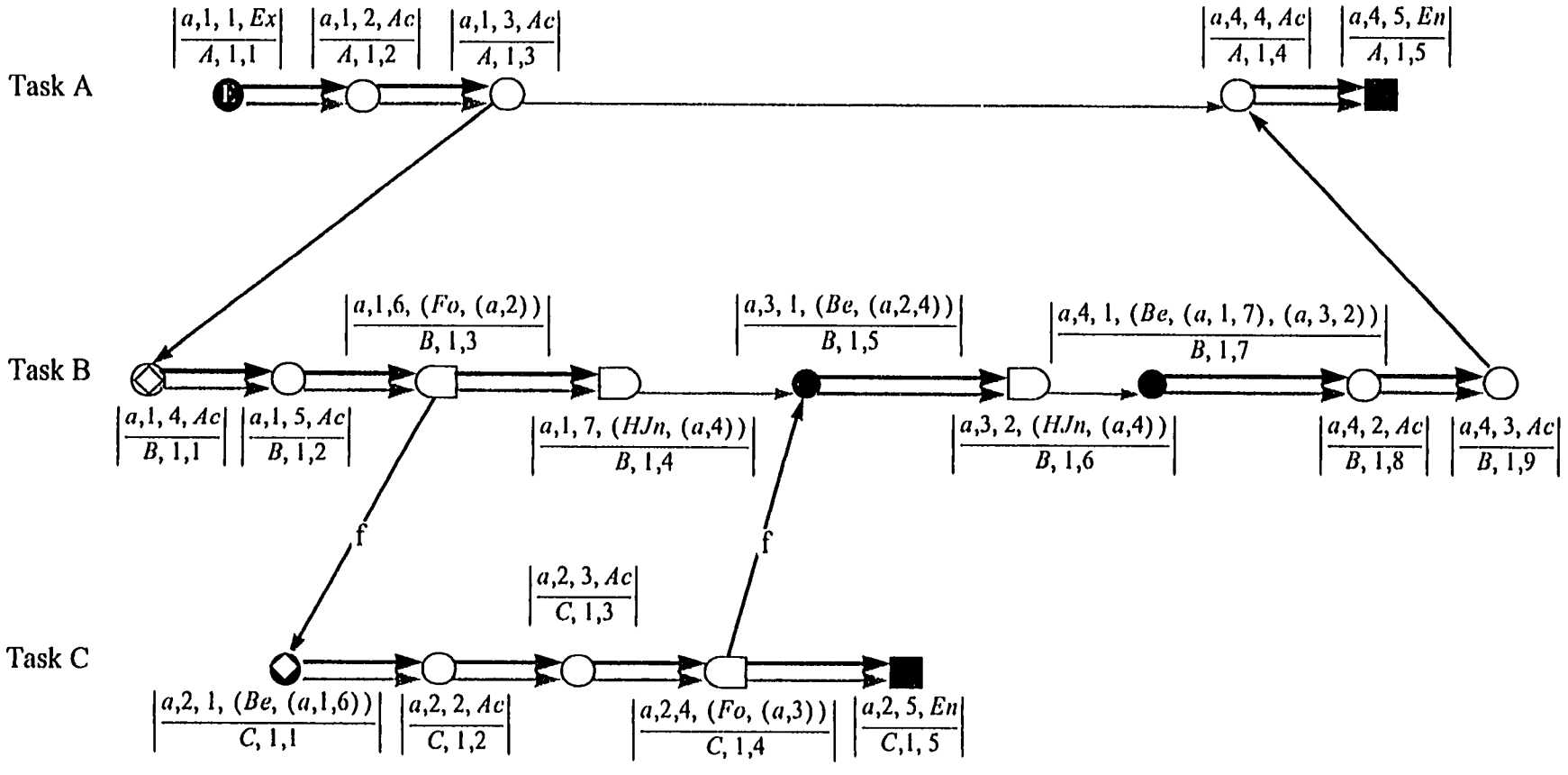


Figure 55: Example Angio Trace with Identifier *a*

7.3 Correctness of the Angio Trace Specification

For the angio trace specification to be correct it must be able to order the adjacent pairs of events that can occur in a trace and the ordering must be consistent.

Proper time determines the set of possible event orderings that the angio trace specification must accommodate. An angio trace constructs a TOEG and there is a close correspondence between the TOEG's node and edge types and the angio trace's event types and ordering relations. So, the correctness of the angio trace specification is directly related to the TOEG.

To determine the set of all valid event orderings first requires determining the set of all valid TOEG node connections between any two node types. This is already defined by Table 6. This is then expanded upon by replacing each and-join node in Table 6 (connections J, K, L N, and O) with the appropriate half-join characterization of Table 11. The last modification to this table is to remove the TOEG external node and replace it with the angio trace external event. The set of all valid event orderings can be determined by enumerating the possible successor and predecessor events for a given event type, similar to the construction of the reachable node connections listed in Table 10. Then the ordering relations of Table 12 and Table 13 are checked to make sure that every valid event ordering is included.

The specification is also checked to ensure that the transitivity of each ordering relation and its inverse relation are consistent. For example, if $>^T(e_1, e_2) \wedge >^T(e_2, e_3)$ is true then $<^T(e_1, e_2) \wedge <^T(e_2, e_3)$ must also be true.

To avoid a conflict when ordering two events it must be ensured, that for a given event type and ordering relation, there is only one possible successor time stamp value.

When this examination was done the author was (yet again) surprised to find that the formal approach corrected errors in his initial angio trace specification, again proving the usefulness of a formal approach.

7.4 Accommodating Task Period Interleaving

It is common in practice for a a task to interleave the processing of several service requests, maintaining state information for each outstanding request. This is typically implemented as a responding task polling several message queues and servicing the first message it finds. Angio tracing can accommodate this service period interleaving without violating proper time's assumption that a task's service period should not be interleaved. This is done using the two instrumentation operators *suspend* and *resume*. They are defined as:

$suspend(ts_i, ts_t)$ Copies task i's time stamp value ts_i into the temporary time stamp storage location ts_t. Then task i's time stamp value is updated by clearing its operation time stamp, incrementing its task period index, and resetting its task period index value to v_o.

$resume(ts_t, ts_i)$ Copies the contents of the temporary time stamp storage location ts_t into task i's time stamp value ts_i.

In each case an activity event is recorded with its application information set to either "suspend" or "resume", so that post-processing of the trace can determine if service periods are being interleaved.

An example of service period interleaving is shown in Figure 56, where Task B performs a suspend at activity $\left|\frac{a,1,6,Ac}{B,1,3}\right|$ and resumes the service at activity $\left|\frac{a,1,7,Ac}{B,1,4}\right|$

(the application information is not shown). RPC interactions are used for the example because they are the simplest case but an asynchronous interaction could occur just as well. However, care must be taken when instrumenting the application code.

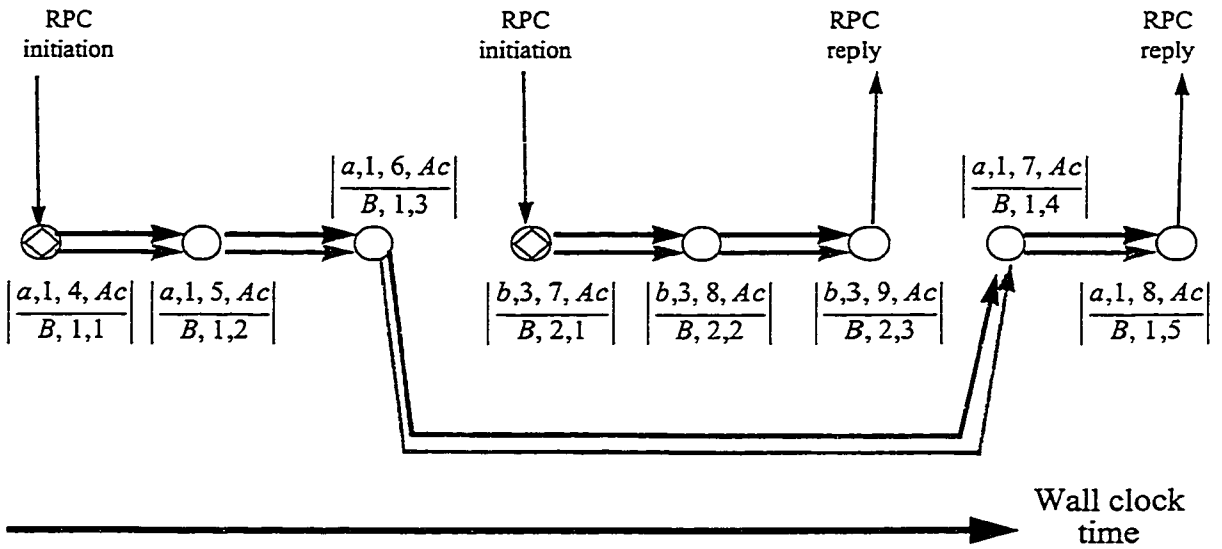


Figure 56: Example of Service Period Interleaving by Task *B*

7.5 Transformation to a TOEG

The transformation from an angio trace into a TOEG largely consists of converting events to nodes, adding the edges, and then replacing the half-join and external event types. The control algorithm to do this is shown in Figure 57. Its first step is to convert each event into a node so that the rest of the transformation can use graph rewrite rules.

The control algorithm is very similar in structure to the proper time control algorithm of Figure 57, with the largest differences being the initialization component. Here the initialization component constructs the host graph whereas the proper time control algorithm assumes the host graph exists. The analysis component then applies graph

rewrite rules to remove angio trace node and edge types so that only the proper time node and edge types remain.

```
FOR (each event)
    convert the event into a node of the appropriate type
    store the time stamp values as attributes of the node
ENDFOR

FOR (each node)
    add the task and operation event graph's edges according to Table 14
ENDFOR

apply the productions of Table 16 to simplify the resulting TOEG graph using a control algorithm
similar to Figure 50
```

Figure 57: The Control Algorithm For Angio Trace to TOEG Transformation

The graph schema of Figure 58 is used in this transformation. It is the TOEG graph schema with the addition of the half-join event type. The Event class has attributes to store the operation time stamp and the TaskEvent class has attributes to store a task time stamp.

The initialization and analysis components are discussed next.

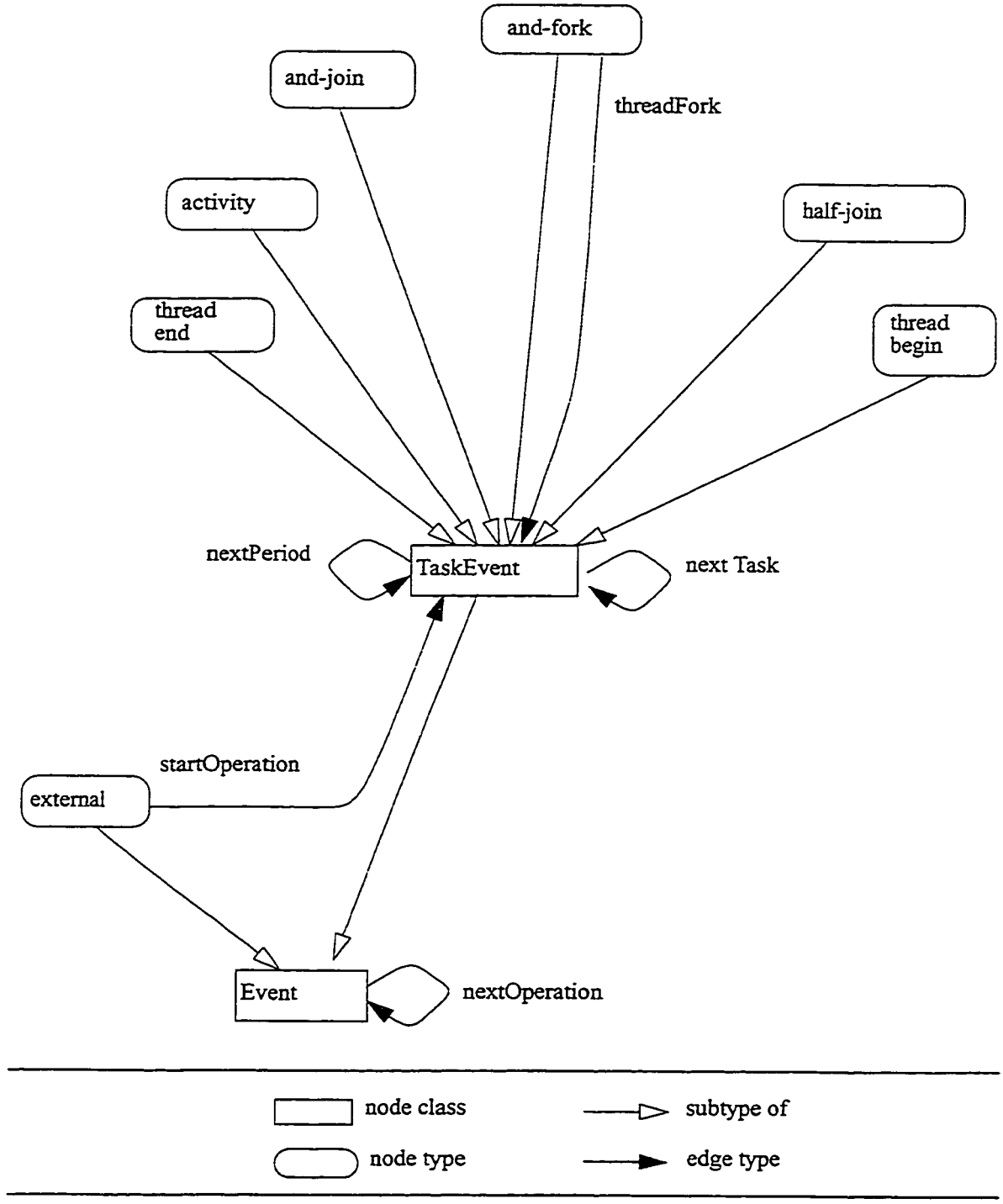


Figure 58: Combined TOEG and Angio Trace Graph Schema

7.5.1 Initialization Component of the Control Algorithm

The initialization component of the control algorithm consists of steps 1 to 7 in Figure 57. Steps 2 to 5 convert each event into an appropriate TOEG node type using the event type information.

Once all of the events are converted into nodes the edges are added to form the host graph (steps 5 through to 7 of Figure 57). There are four operators that can add a TOEG edge type between two adjacent nodes:

- $nextTask(e_1, e_2)$ adds a next task edge from the source node e_1 to the target node e_2.
- $nextPeriod(e_1, e_2)$ adds a next period edge from the source node e_1 to the target node e_2.
- $nextOpTh(e_1, e_2)$ adds a next operation edge from the source node e_1 to the target node e_2.
- $andFork(e_1, e_2)$ adds an and-fork edge from the source node e_1 to the target node e_2.

An operator is not needed for the start operation edge because they are added later by the graph rewrite rules.

The information needed to add the edges for a node is summarized in Table 14. The table is devised so that the source node e_1 is given and the successor target node e_2 is to be found. The first step to add an edge is to use an event predicate to identify the type of e_1 to determine which row of Table 14 applies. Each ordering relation column determines the successor event e_2 for a different edge type. If an ordering relation assigns a valid event

(node) to e_2, then the identified operator is executed to add the edge from e_1 to e_2. The activity event and end event have an additional test to determine if the edge type to be added is a next task edge or next task period edge.

Event type of e_1 where $e_1 = \left\lvert \frac{j_1, k_1, m_1}{i_1, c_1, v_1} \right\rvert$ and $e_2 = \left\lvert \frac{j_2, k_2, m_2}{i_2, c_2, v_2} \right\rvert$	Successor Event e_2 in the Task Event Graph $>^{T}(e_1, e_2)$	Successor Event e_2 in the Same Operation Thread $>^{At}(e_1, e_2)$	Successor Event e_2 that is not in the Same Operation Thread $>^{Ao}(e_1, e_2)$
Activity event $activity(e_1, -)$	$(c_1 = c_2) \rightarrow nextTask(e_1, e_2) \vee$ $(c_1 \neq c_2) \rightarrow nextPeriod(e_1, e_2)$	$nextOpTh(e_1, e_2)$	N/A
External event $external(e_1, -)$	$nextTask(e_1, e_2)$	$nextOpTh(e_1, e_2)$	N/A
Begin event $begin(e_1)$	$nextTask(e_1, e_2)$	$nextOpTh(e_1, e_2)$	N/A
Fork event $fork(e_1, -)$	$nextTask(e_1, e_2)$	$nextOpTh(e_1, e_2)$	$andFork(e_1, e_2)$
Half-join event $isHJoin(e_1)$	$nextTask(e_1, e_2)$	N/A	N/A
End event $end(e_1)$	$(c_1 = c_2) \rightarrow nextTask(e_1, e_2) \vee$ $(c_1 \neq c_2) \rightarrow nextPeriod(e_1, e_2)$	N/A	N/A

Table 14: Edge Type Assignment from Nodes e_1 to e_2

The complexity of searching for a successor node is reduced to a couple of tests for each successor event if hashing is employed. Each time stamp value is unique so it can be used as a key in a hash table.

7.5.2 Analysis Component of the Control Algorithm

There are two stages to the analysis component of the control algorithm. The first stage applies graph rewrite rules to replace the half-join and external node type with the appropriate TOEG sub-graphs. These graph modifications are described in Table 16. where the RHS nodes in the box ☐ are replaced. The adjacent nodes in the LHS and RHS are numbered to ensure the new sub-graph is properly embedded in the host graph. The external nodes have an additional attribute that prevents them from being the subject of a graph rewrite rule twice.

Description	Sub-graph to Replace (RHS)	Replacement Sub-graph to Embed (LHS)
Responder task synchronization from an RPC interaction.		
Responder task synchronization from an asynchronous interaction.		
Initiating task synchronization from an RPC interaction.		
Responder task synchronization from an external event.		
RPC interruption		
An external event is converted to the TOEG external node, start operation edge, and begin operation thread node.		
Same as above.		

Table 15: Graph Rewriting Rules To Make a TOEG

The second stage of the analysis simplifies the generated TOEG, for example replacing an interrupted RPC with an equivalent set of interactions. The simplifications are described by Table 16.

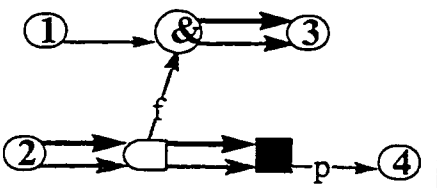
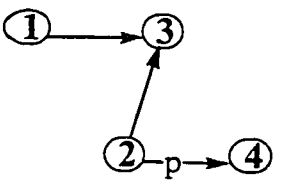
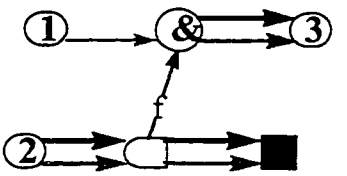
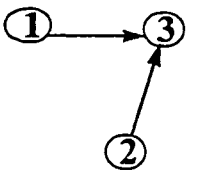
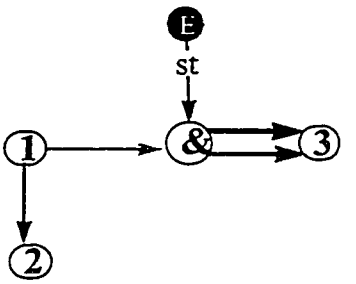
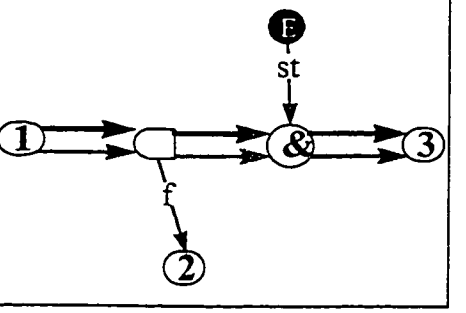
Description	Sub-graph to Replace (RHS)	Replacement Sub-graph to Embed (LHS)
An RPC interaction that ends with an asynchronous message and an immediate thread end node is changed to an activity reply. Node 1 is an activity node.		
Same as above.		
RPC interruption from an external event.		

Table 16: Graph Rewriting Rules To Simplify a TOEG

Chapter 8.0 Angio Tracing Instrumentation

This chapter provides a description of an angio trace implementation in a general fashion, independent of any particular environment (step #2 of Figure 1). It is a distributed algorithm, where the global state is reconstructed from the local state of the tasks and the information sent along with each message. The instrumentation is presented in a tabular format to show how it is associated with a TOEG. Each row in the instrumentation table is associated with a row in the node connection table of the TOEG (Table 6). Each row identifies the angio trace events record, their time stamp values, the instrumentation operations that record the events, the pre-conditions for executing the operations, and the resulting task state. The TOEG icon notation is also used to depict the recorded events as a sub-graph.

Other implementation considerations are briefly presented at the end of the chapter.

8.1 The General Operation of the Task Instrumentation

The instrumentation is defined with respect to a single task because in a distributed system global information is not available. Each task sets aside memory to store a task and operation execution time stamp in a vector (j,k,m,l,i,c,v). When a task is created it is assigned a task identifier, the operation time stamp fields (j,k,m,l) are set to the empty value, its task period c is set to the initial value c_o, and its task period index is set to the initial value v_o.

The instrumentation embedded in a task operates like an extended state machine. The reachable node connections of proper time (Table 10) describes that state machine. Each

row in this table is an *instrumentation state* that the task may be in and the column "Successor Node Connection in the Same Task Event Graph" describes the possible next instrumentation state of the task for a given state, which are the candidate events which may be recorded next. This column describes the local behavior of a task in response to recording an event, starting an angio trace, or receiving a message. The task interactions that are possible from a given instrumentation state are identified by the column "Successor Node Connection in a Different Task Event Graph" which is used to determine the information to send with each message.

There are several conventions that are used in the implementation of the task's instrumentation state machine. First, recording an event increments the task event and thread indexes in accordance with the angio trace specification. Second, when a task initiates an RPC, its operation time stamp is cleared when the message is sent because the next message accepted will have the replacement operation time stamp. This is also a precautionary measure that can be used to check for errors, such as recording events without an operation time stamp. Third, when a message is received with an operation time stamp whose thread event index is one, then the receiving task records a begin event. This distinguishes between node connection strategies {E, I} and {J, K, L}. Another point is that the instrumentation for the and-fork, operation thread begin, and half-join events may record several trace events for one software event (as described in section 7.2).

The last convention to make is that the task event index determines where the task period boundaries are and when a message reception is, or is not, a synchronization. The criterion that is used is when a message is received and the task event index is not set to the initial value v_o, then it is considered to be a synchronization. This requires that when a task's period ends, the operation time field values are set to the empty value, the task period value increments, and the task period index is set to its initial value.

8.2 Application Level Instrumentation Routines

An application is instrumented using high-level routines which hide the details of the task's instrumentation state machine. The instrumentation routines that are suggested are as follows:

- Initialize the angio trace event buffers.
- Enable or disable the recording of angio trace events.
- Initialization of a task time stamp and operation time stamp data structures.
- The recording of an external event, indicating if it begins a new task period.
- Recording an and-fork event which also stores the time stamp information of the forked operation thread in a data structure.
- When a message is received, the time stamp information is retrieved and the appropriate events are recorded.
- Recording a thread end event.
- Suspending an operation thread.
- Resuming an operation thread.

Several of these functions update the task's local state information.

When a message is received the receiving task must determine which node connection it should make a transition to, with the eligible ones being {J, L, G, K, E, I} of Table 6. To differentiate these cases there are three pieces of information that are used and they describe the truth table of Table 17. The operation time stamp fields of task i_1 are checked to see if

they have been assigned the empty value which would indicate that the task is waiting to receive a message. Then the task event index is checked to determine if task i_1 is beginning a new service period or in the middle of a service period. The thread event index of the operation time stamp in the received message is checked to see if the preceding event is a fork event.

Node Connection from Table 6 for Task i_1	Task i_1's Operation time stamp fields are set to the Empty Value	Task i_1 has Not Started a Task Period	Task i_1's Received Message's operation time stamp Thread Index is One
J	No	No	No
L	No	No	Yes
Not Applicable	No	Yes	No
Not Applicable	No	Yes	Yes
G	Yes	No	No
K	Yes	No	Yes
E	Yes	Yes	No
I	Yes	Yes	Yes

Table 17: Task i_1's Truth Table for the Events it Should Record upon a Message Reception

8.3 Monitor Interface

An angio trace event is recorded by instrumentation embedded within an operation that interfaces to the program monitor. A minimal set of operations must be supported by the program monitor and they are described here.

Operation time stamp information is added to a message before it is sent to implement the ordering relations $\{>^{Ao},<^{Ao}\}$ of section 7.2.4. A message carries the operation time stamp S_1 of the sending task's event that is the cause of the receive event. The operation

time stamp S_2 will replace the current operation time stamp value of the receiving task. The sending task is responsible for generating S_2.

The monitor provides four operators. The first two are used to manipulate the time stamp values of a message. They are

- $send(e, S_1, S_2)$ appends the operation time stamps S_1 and S_2 to the message that is associated with the send event e.
- $rcv(e, S_1, S_2)$ retrieves the operation time stamps S_1 and S_2 that were sent with the message received by event e.
- $record(e)$ atomically records and stores the event e.
- $unique(x)$ assigns a globally unique value to variable x.

The $unique(\ )$ operator can generate values by concatenating unique identifiers together. For example, a globally unique value could an be constructed by concatenating a node's IP address, the process ID of the task executing the operator, and the CPU's microsecond counter value.[1] Further optimizations are possible by only providing a globally unique value for the operation name and reducing the operation thread identifier's scope to be unique within the operation name.

8.4 Angio Trace Implementation

The instrumentation presented in this sub-section implements the angio trace specification in a straight forward fashion. The properties of this implementation are: it

1. A 64 bit micro-second counter has a roll-over time of 213503982 days.

minimizes the post-processing effort, if a single event is lost then analysis is still possible, and all of the ordering relations of section 7.2.4 are supported.

The instrumentation is listed in Table 18 and it is defined by the last three columns. The instrumentation defines the time stamp information that each event must record and *not* how the instrumentation is to be coded or executed. Each row in the table can be cross-referenced to Table 6, Table 12, and Table 13. Each row in the table should be interpreted as follows: "if the precondition values of task i_1 are met, then execute the instrumentation operations to record the identified events." The instrumentation for the suspension and resumption of an operation thread is presented as (P) and (Q). The conventions that are used to describe the instrumentation follow.

The documentation columns of the table are the columns "*Event Connection Interpretation*" and "*Instrumentation Comments*". The "*Event Connection Interpretation*" column describes the purpose of the specification. The "*Instrumentation Comments*" column details the finer points of each event connection specification, identifying the purpose for recording each event.

The "*Recorded Event Observations*" is the most important column because it illustrates the recorded events and their ordering, using the conventions of proper time (section 6.3) the time stamp fields of Definition 7.3 and the icon for the half-join event. The illustrations show the recorded events with the angio trace time stamp information overlaid against the proper time edges and nodes (where applicable). For the sake of simplicity there are two exceptions. First, the icon of ⊛ is added to represent any type of event provided it has the specified time stamp field values. Secondly, the domain data recorded with the activity event information is not shown because it is operation dependent.

Each illustration identifies the recorded events, as well as their preceding and succeeding event. Events in boxes are the events recorded by the instrumentation

and, if there is more than one, they are recorded together atomically. The time stamp field values of the recorded events are the actual values that are recorded.

In all the illustrations, event e_1 precedes the event(s) to be recorded by the instrumentation. The events which may succeed the recorded events are also shown. In some cases, boundary conditions exist. The illustrations will show additional events to describe the boundary condition, but these events will not be recorded in all cases. For example the source or target node of a task's next period edge may not exist.

The table refers to recording events for the task i_1 and its instrumentation state vector may be used to determine which events to record. The "*Precondition State of Task* i_1" column lists the predicates and conditions which must all be true for the instrumentation operations to be executed. This state information is the task's instrumented state just prior to recording the events; it is not the state of the task when event e_1 was recorded.

The executed instrumentation operations are described in a column of the same name. The instrumentation operations also identify task i_1's instrumentation state vector values after executing the operation.

The time stamps follow the Definition 7.3 with some additions. A field value may be a symbolic, subscripted variable. Variables in the same event connection specification with the same subscripts have the same value. A time stamp field value with a "-" can take on any value. A field with the place holder value "-" can take on the empty value. All time stamp values are natural numbers, beginning at one. The empty value is assigned the value zero, $\varnothing = 0$. The time stamp field initialization values are: the task event index ($v_o = 1$), the task period index ($c_o = 1$), and the thread event index ($m_o = 1$).

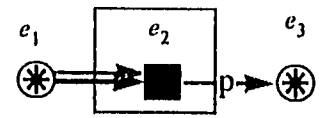
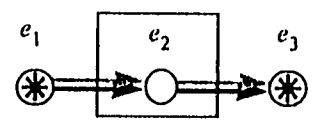
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(A) External system request.	See (M), (N), or (O)			
(B) End of the task period and operation thread.	Event e_3 will not occur if the task i_1 is not involved in any further interactions.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, -}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_1, m_1 + 1, En}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{-, -, -, -}{i_1, c_1 + 1, 1} \right\rvert$	$record(e_2)$; $j \leftarrow \varnothing$; $k \leftarrow \varnothing$; $m \leftarrow \varnothing$; $c \leftarrow c_1 + 1$; $v \leftarrow 1$;
(C) An activity event.		$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, -}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_1, m_1 + 1, Ac}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_1, m_1 + 2, -}{i_1, c_1, v_1 + 2} \right\rvert$	$record(e_2)$; $m \leftarrow m_1 + 2$; $v \leftarrow v_1 + 2$;

Table 18: Angio Trace Instrumentation Example

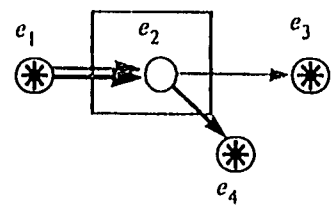
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(D) Initiation of an RPC interaction.	Event e_2 is the activity event for the sending of the request.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, -}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_1, m_1 + 1, Ac}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{-, -, -, -}{i_1, c_1, v_1 + 2} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_1, m_1 + 2, -}{i_2, -, -} \right\rvert$	$send(e_2, S_1, S_2)$ where $S_1 \leftarrow (j_1, k_1, m_1 + 1)$, $S_2 \leftarrow (j_1, k_1, m_1 + 2)$; $record(e_2)$; $j \leftarrow \emptyset$; $k \leftarrow \emptyset$; $m \leftarrow \emptyset$; $v \leftarrow v_1 + 2$;

Table 18: Angio Trace Instrumentation Example

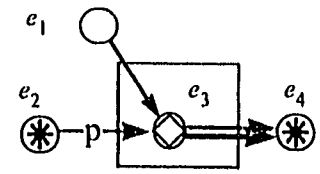
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(E) Acceptance of a service request sent as an RPC interaction.	Event e_3 is the activity event for the acceptance of the request.	$rcv(e_3, S_1, S_2)$ where $S_1 = (j_1, k_1, m_1)$, $S_2 = (j_1, k_1, m_1 + 1)$; $j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1 + 1$; $v = 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, Ac}{i_2, -, -} \right\rvert$, $e_2 = \left\lvert \frac{-, -, -, -}{i_1, c_1, -} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_1, m_1 + 1, Ac}{i_1, c_1 + 1, 1} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_1, m_1 + 2, -}{i_1, c_1 + 1, 2} \right\rvert$	$record(e_3)$; $j \leftarrow j_1$; $k \leftarrow k_1$; $m \leftarrow m_1 + 1$; $v \leftarrow 2$;

Table 18: Angio Trace Instrumentation Example

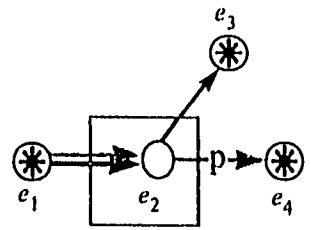
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(F) Sending the reply to a service request sent as an RPC interaction. The responding task's service period ends.	Event e_3 records the activity event for the reply acceptance. Event e_4 will not occur if the task i_2 is not involved in any further interactions.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, -}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_1, m_1 + 1, Ac}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_1, m_1 + 2, -}{i_2, -, -} \right\rvert$, $e_4 = \left\lvert \frac{-, -, -, -}{i_1, c_1 + 1, 1} \right\rvert$	$send(e_2, S_1, S_2)$ where $S_1 \leftarrow (j_1, k_1, m_1 + 1)$, $S_2 \leftarrow (j_1, k_1, m_1 + 2)$; $record(e_2)$; $j \leftarrow \varnothing$; $k \leftarrow \varnothing$; $m \leftarrow \varnothing$; $c \leftarrow c_1 + 1$; $v \leftarrow 1$;

Table 18: Angio Trace Instrumentation Example

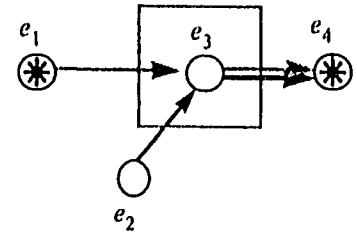
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(G) A blocked initiating task in an RPC interaction receives the reply to its service request. The replying task ended its service period after it sent the reply.	Event e_3 records the activity event for the reply acceptance.	$rcv(e_3, S_1, S_2)$ where $S_1 = (j_1, k_2, m_2)$, $S_2 = (j_1, k_2, m_2 + 1)$; $j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{-,-,-,-}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_2, m_2, -}{i_2, -, -} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_2, m_2 + 1, Ac}{i_1, c_1, v_1 + 1} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_1, m_2 + 2, -}{i_1, c_1 v_1 + 2} \right\rvert$	$record(e_3)$; $j \leftarrow j_1$; $k \leftarrow k_2$; $m \leftarrow m_2 + 2$; $v \leftarrow v_1 + 2$;

Table 18: Angio Trace Instrumentation Example

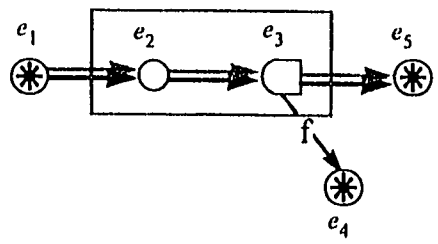
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(H) There are three interpretations: (1) An initiating task initiates an asynchronous interaction. (2) A responding task sends the reply to a service request sent as an RPC interaction and the responding task does not end it service period but continues executing after sending the reply. (3) A forwarding task forwards the service request to another responding task.	There are two events recorded. Event e_2 is the activity event that identifies the statement executed and event e_3 characterizes the forked operation thread.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, -}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_1, m_1 + 1, Ac}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_1, m_1 + 2, (Fo, (j_1, k_2))}{i_1, c_1, v_1 + 2} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_2, 1, (Be, (j_1, k_1, m_1 + 2))}{i_2, -, -} \right\rvert$, $e_5 = \left\lvert \frac{j_1, k_1, m_1 + 3, -}{i_1, c_1, v_1 + 3} \right\rvert$ where $fork(e_3, (j_1, k_2))$	$unique(k_2)$; $send(e_3, S_1, S_2)$ where $S_1 \leftarrow (j_1, k_1, m_1 + 2)$, $S_2 \leftarrow (j_1, k_2, 1)$; $record(e_2)$; $record(e_3)$; $m \leftarrow m_1 + 3$; $v \leftarrow v_1 + 3$;

Table 18: Angio Trace Instrumentation Example

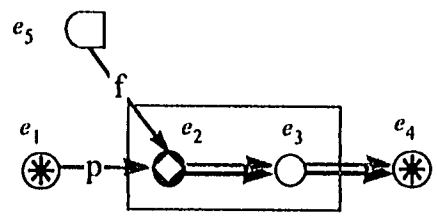
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(I) A blocked task that is **not** processing a previously accepted service request now accepts a service request that was sent asynchronously	There are two events recorded. Event e_2 is the activity event that is associated with the actual acceptance of the request and event e_3 is the beginning of the forked operation thread.	$rcv(e_3, S_1, S_2)$ where $S_1 = (j_1, k_1, m_1)$, $S_2 = (j_1, k_2, 1)$; $j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1 + 1$; $v = 1$;	 $e_1 = \left\lvert \frac{-,-,-,-}{i_1, c_1, -} \right\rvert$, $e_2 = \left\lvert \frac{j_1, k_2, 1, (Be, (j_1, k_1, m_1))}{i_1, c_1 + 1, 1} \right\rvert$, $e_3 = \left\lvert \frac{j_1, k_2, 2, Ac}{i_1, c_1 + 1, 2} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_2, 3, -}{i_1, c_1 + 1, 3} \right\rvert$, $e_5 = \left\lvert \frac{j_1, k_1, m_1, (Fo, (j_1, k_2))}{i_2, -, -} \right\rvert$ where $fork(e_5, (j_1, k_2))$	$record(e_2)$; $record(e_3)$; $j \leftarrow j_1$; $k \leftarrow k_2$; $m \leftarrow 3$; $v \leftarrow 3$;

Table 18: Angio Trace Instrumentation Example

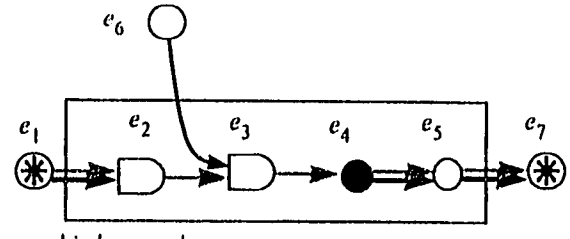
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(J) A blocked task that is processing a previously accepted service request completes a synchronization by accepting a service request. The service request was sent as an RPC interaction.	There are four events that are recorded: e_2, e_3, e_4, and e_5. The half-join event e_2 indicates that the operation thread $\lvert j_1,k_1 \rvert$ is ending but the task's service period is not over. The half-join event e_3 indicates that the operation thread $\lvert j_2,k_2 \rvert$ is ending but the service period is not over. Each of these half-join events indicates that the new operation thread $\lvert j_1,k_3 \rvert$ is the result of the two operation threads joining. Event e_4 is the start of the resulting operation thread $\lvert j_1,k_3 \rvert$. Event e_5 is the activity event that records the statement used to accept the request.	$rcv(e_2, S_1, S_2)$ where $S_1 = (j_2,k_2,m_2)$, $S_2 = (j_2,k_2,m_2+1)$; $j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,-}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,m_1+1,(HJo,(j_1,k_3))}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,k_2,m_2+1,(HJo,(j_1,k_3))}{i_1,c_1,v_1+2} \right\rvert$, $e_4 = \left\lvert \frac{j_1,k_3,1,(Be,(j_2,k_2,m_2+1),(j_1,k_1,m_1+1))}{i_1,c_1,v_1+3} \right\rvert$, $e_5 = \left\lvert \frac{j_1,k_3,2,Ac}{i_1,c_1,v_1+4} \right\rvert$, $e_6 = \left\lvert \frac{j_2,k_2,m_2,Ac}{i_2,-,-} \right\rvert$, $e_7 = \left\lvert \frac{j_1,k_3,3,-}{i_1,c_1,v_1+5} \right\rvert$ where $hJoin(\{e_2,e_3\},(j_1,k_3))$	$unique(k_3)$; $record(e_2)$; $record(e_3)$; $record(e_4)$; $record(e_5)$; $k \leftarrow k_3$; $m \leftarrow 3$; $v \leftarrow v_1 + 5$;

Table 18: Angio Trace Instrumentation Example

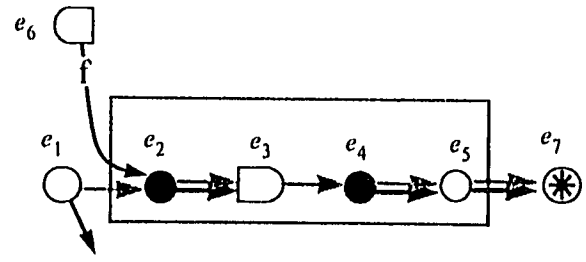
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(K) A blocked initiating task in an RPC interaction becomes unblocked by receiving the reply to its service request and the replying task did not end its service period after it sent the reply. The blocked initiating task may also become unblocked due to an RPC interruption.	There are four events that are recorded: e_2, e_3, e_4, and e_5. The begin event e_2 is needed to start the forked operation thread $\lvert j_2,k_3 \rvert$. The half-join event e_3 indicates that the operation thread $\lvert j_2,k_3 \rvert$ is ending but the service period is not over. It indicates that the resulting operation thread $\lvert j_1,k_4 \rvert$ is the result of the synchronization. Event e_4 is the start of the new operation thread $\lvert j_1,k_4 \rvert$. Event e_5 is the activity event that records the statement used to accept the request. Only one half-join event is needed because there is only one operation thread that is ending, but this is still a synchronization because a message is accepted within an active service period.	$rcv(e_2, S_1, S_2)$ where $S_1 = (j_2,k_2,m_2)$, $S_2 = (j_2,k_3,1)$; $j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,Ac}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_2,k_3,1,(Be,(j_2,k_2,m_2))}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,k_3,2,(HJo,(j_1,k_4))}{i_1,c_1,v_1+2} \right\rvert$, $e_4 = \left\lvert \frac{j_1,k_4,1,(Be,(j_2,k_3,2))}{i_1,c_1,v_1+3} \right\rvert$, $e_5 = \left\lvert \frac{j_1,k_4,2,Ac}{i_1,c_1,v_1+4} \right\rvert$, $e_6 = \left\lvert \frac{j_2,k_2,m_2,(Fo,(j_2,k_3))}{i_2,-,-} \right\rvert$, $e_7 = \left\lvert \frac{j_1,k_4,3,-}{i_1,c_1,v_1+5} \right\rvert$ where $fork(e_3,(j_1,k_4))$ and $hJoin(\{e_3\},(j_1,k_4))$	$unique(k_4)$; $record(e_2)$; $record(e_3)$; $record(e_4)$; $record(e_5)$; $j \leftarrow j_1$; $k \leftarrow k_4$; $m \leftarrow 3$; $v \leftarrow v_1 + 5$;

Table 18: Angio Trace Instrumentation Example

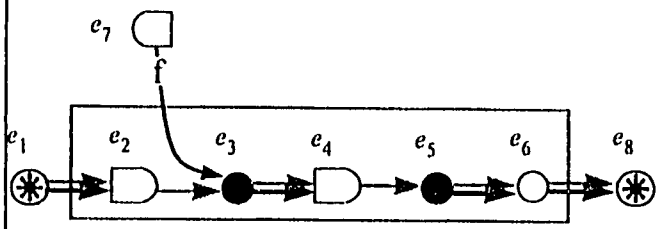
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(L) A blocked task that is processing a previously accepted service request completes a synchronization by accepting a service request. The service request was sent as an asynchronous interaction	There are five events that are recorded: e_2, e_3, e_4, e_5, and e_6. The half-join event e_2 indicates that the operation thread $\lvert j_1,k_1 \rvert$ is ending but the service period is not over. The begin event e_3 is needed to start the forked operation thread $\lvert j_2,k_3 \rvert$. The half-join event e_4 indicates that the operation thread $\lvert j_2,k_3 \rvert$ is ending but the service period is not over. Each of these half-join events indicates that the new operation thread $\lvert j_1,k_4 \rvert$ is the result of the two operation threads joining. Event e_5 is the start of the operation thread $\lvert j_1,k_4 \rvert$. Event e_6 is the activity event that records the statement used to accept the request.	$rcv(e_2, S_1, S_2)$ where $S_1 = (j_2,k_2,m_2)$, $S_2 = (j_2,k_3,1)$; $j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,-}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,m_1+1,(HJo,(j_1,k_4))}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,k_3,1,(Be,(j_2,k_2,m_2))}{i_1,c_1,v_1+2} \right\rvert$, $e_4 = \left\lvert \frac{j_2,k_3,2,(HJo,(j_1,k_4))}{i_1,c_1,v_1+3} \right\rvert$, $e_5 = \left\lvert \frac{j_1,k_4,1,(Be,(j_2,k_3,2),(j_1,k_1,m_1+1))}{i_1,c_1,v_1+4} \right\rvert$, $e_6 = \left\lvert \frac{j_1,k_4,2,Ac}{i_1,c_1,v_1+5} \right\rvert$, $e_7 = \left\lvert \frac{j_2,k_2,m_2,(Fo,(j_2,k_3))}{i_2,-,-} \right\rvert$, $e_8 = \left\lvert \frac{j_1,k_4,3,-}{i_1,c_1,v_1+6} \right\rvert$ where $fork(e_7, (j_2,k_3))$ and $hJoin(\{e_2, e_4\}, (j_1,k_4))$	$unique(k_4)$; $record(e_2)$; $record(e_3)$; $record(e_4)$; $record(e_5)$; $record(e_6)$; $k \leftarrow k_4$; $m \leftarrow 3$; $v \leftarrow v_1 + 6$;

Table 18: Angio Trace Instrumentation Example

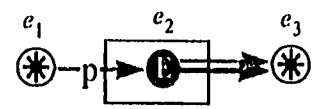
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(M) A blocked task that is **not** processing a previously accepted service request now begins an operation thread because of an external request.	Event e_1 will not occur if it is the task's first period, in which case c_1 is one.	$j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1 + 1$; $v = 1$;	 $e_1 = \left\lvert \frac{-,-,-,-}{i_1,c_1,-} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,1,Ex}{i_1,c_1+1,1} \right\rvert$, $e_3 = \left\lvert \frac{j_1,k_1,2,-}{i_1,c_1+1,2} \right\rvert$	$unique(j_1)$; $unique(k_1)$; $record(e_2)$; $j \leftarrow j_1$; $k \leftarrow k_1$; $m \leftarrow 2$; $v \leftarrow 2$;

Table 18: Angio Trace Instrumentation Example

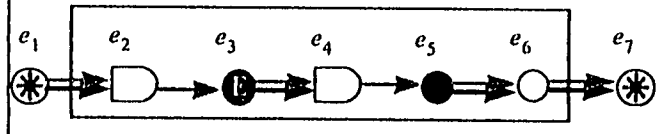
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(N) A blocked task that is processing a previously accepted service request completes a synchronization by accepting an external request (e.g., a received message without time stamp information is an external event).	There are five events that are recorded: e_2, e_3, e_4, e_5, and e_6. The half-join event e_2 indicates that the operation thread $\lvert j_1,k_1 \rvert$ is ending but the service period is not over. The external event e_3 starts the new operation thread $\lvert j_2,k_3 \rvert$. The half-join event e_4 indicates that the operation thread $\lvert j_2,k_3 \rvert$ is ending but the service period is not over. Each of these half-join events indicates that the new operation thread $\lvert j_1,k_4 \rvert$ is the result of the two operation threads joining. Event e_5 is the start of the operation thread $\lvert j_1,k_4 \rvert$. Event e_6 is the activity event that records the statement used to accept the request.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,-}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,m_1+1,(HJo,(j_1,k_4))}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,k_3,1,Ex}{i_1,c_1,v_1+2} \right\rvert$, $e_4 = \left\lvert \frac{j_2,k_3,2,(HJo,(j_1,k_4))}{i_1,c_1,v_1+3} \right\rvert$, $e_5 = \left\lvert \frac{j_1,k_4,1,(Be,(j_2,k_3,2),(j_1,k_1,m_1+1))}{i_1,c_1,v_1+4} \right\rvert$, $e_6 = \left\lvert \frac{j_1,k_4,2,Ac}{i_1,c_1,v_1+5} \right\rvert$, $e_7 = \left\lvert \frac{j_1,k_4,3,-}{i_1,c_1,v_1+6} \right\rvert$ where $jhJoin(\{e_2,e_4\},(j_1,k_4))$	$unique(j_2)$; $unique(k_3)$; $unique(k_4)$; $record(e_2)$; $record(e_3)$; $record(e_4)$; $record(e_5)$; $record(e_6)$; $k \leftarrow k_4$; $m \leftarrow 3$; $v \leftarrow v_1 + 6$;

Table 18: Angio Trace Instrumentation Example

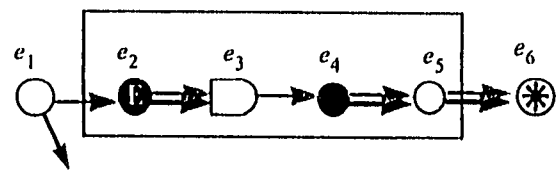
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(O) A blocked initiating task in an RPC interaction becomes unblocked due to an RPC interruption caused by an external event.	There are four events that are recorded: e_2, e_3, e_4, and e_5. The external event e_2 interrupts the RPC initiated by event e_1 and it starts the operation thread $\lvert j_2,k_2 \rvert$. The half-join event e_3 indicates that the operation thread $\lvert j_2,k_2 \rvert$ is ending but the service period is not over. It indicates that the resulting operation thread $\lvert j_1,k_3 \rvert$ is the result of the synchronization. Event e_4 is the start of the new operation thread $\lvert j_1,k_3 \rvert$. Event e_5 is the activity event that records the statement used to accept the request. Only one half-join event is needed because there is only one operation thread that is ending.	$j = \varnothing$; $k = \varnothing$; $m = \varnothing$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1, k_1, m_1, Ac}{i_1, c_1, v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_2, k_2, 1, Ex}{i_1, c_1, v_1 + 1} \right\rvert$, $e_3 = \left\lvert \frac{j_2, k_2, 2, (HJo, (j_1, k_3))}{i_1, c_1, v_1 + 2} \right\rvert$, $e_4 = \left\lvert \frac{j_1, k_3, 1, (Bc, (j_2, k_2, 2))}{i_1, c_1, v_1 + 3} \right\rvert$, $e_5 = \left\lvert \frac{j_1, k_3, 2, Ac}{i_1, c_1, v_1 + 4} \right\rvert$, $e_6 = \left\lvert \frac{j_1, k_3, 3, -}{i_1, c_1, v_1 + 5} \right\rvert$ where $external(e_2)$ and $hJoin(\{e_3\}, (j_1, k_4))$	$unique(j_2)$; $unique(k_2)$ $unique(k_3)$; $record(e_2)$; $record(e_3)$; $record(e_4)$; $record(e_5)$; $j \leftarrow j_1$; $k \leftarrow k_3$; $m \leftarrow 3$; $v \leftarrow v_1 + 5$;

Table 18: Angio Trace Instrumentation Example

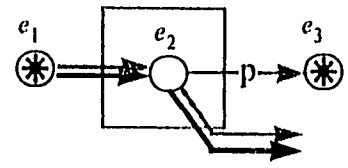
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(P) Suspending a service period to allow task period interleaving.	The operation thread is $\lvert j_1,k_1 \rvert$ suspended at event e_2 and a new task period is begun at e_3.	$j = j_1$; $k = k_1$; $m = m_1 + 1$; $c = c_1$; $v = v_1 + 1$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,-}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,m_1+1,Ac}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,-,-,-}{i_1,c_1+1,1} \right\rvert$	$record(e_2)$; $j \leftarrow \varnothing$; $k \leftarrow \varnothing$; $m \leftarrow \varnothing$; $c \leftarrow c_1 + 1$; $v \leftarrow 1$;

Table 18: Angio Trace Instrumentation Example

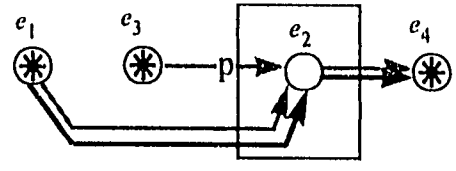
Event Connection Interpretation	Instrumentation Comments	Precondition State of Task i_1	Recorded Event Observations	Instrumentation Operations
(Q) Resuming a service period to allow task period interleaving.	The operation thread $\lvert j_1,k_1 \rvert$ that was suspended at event e_1 resumes at event e_2. There may be any number of intervening event recorded between e_1 and e_2.	$j = j_2$; $k = k_2$; $m = m_2$; $c = c_2$; $v = v_2$;	 $e_1 = \left\lvert \frac{j_1,k_1,m_1,Ac}{i_1,c_1,v_1} \right\rvert$, $e_2 = \left\lvert \frac{j_1,k_1,m_1+1,Ac}{i_1,c_1,v_1+1} \right\rvert$, $e_3 = \left\lvert \frac{j_2,k_2,m_2,-}{i_1,c_2,v_2} \right\rvert$, $e_4 = \left\lvert \frac{j_1,k_1,m_1+2,-}{i_1,c_1,v_1+2} \right\rvert$	$record(e_2)$; $j = j_1$; $k = k_1$; $m = m_1 + 2$; $c = c_1$; $v = v_1 + 2$;

Table 18: Angio Trace Instrumentation Example

8.5 Correctness of the Instrumentation

Aside from programming errors, the instrumentation is correct if it is complete and consistent. Completeness is ensured because the instrumentation maps back directly to the possible node connections of proper time, with the addition of the transparent suspend and resume instrumentation operations.

Consistency of the instrumentation is examined on three levels. First, the instrumentation must be compatible with the angio trace specification. Secondly, conflict must be avoided so each pre-condition must be unique. Lastly, the instrumentation operations and pre-conditions must be synchronized for each allowed event ordering - the instrumentation operations of a previous event are the pre-condition to the next successor event. The converse must also be true, that the pre-condition of a given event must be compatible with the result of the instrumentation operations of its previous event. This is considered by enumerating and evaluating all of the possibilities.

During the course of considering the correctness of the instrumentation it was found that an extra variable had been defined which was not needed. A Boolean variable for determining if a task had started a service period had been defined but it has been removed.

8.6 Implementation Considerations

Although the instrumentation defines the event types and their time stamp values some implementation considerations need to be addressed. To minimize the effort to add the instrumentation it is intended that the instrumentation be embedded in the system-level operations. Then the instrumentation effort is limited to identifying the external events, task periods, and application level information. There are also standardization concerns that need to be discussed for use in a heterogenous environment.

To amortize the instrumentation effort, it is expected that the angio tracing instrumentation would be permanently embedded in the message passing system functions of the:

- Distributed system programming language.
- System libraries.
- Interface Definition Language compilers.
- Operating system kernel calls.

The system functions to be instrumented need to be selected. The selection criterion suggested here are concerned with the development of LQN performance models. For this purpose all system functions that involve task interactions or that have noticeable service times are instrumented. Obviously, the accessibility of the source code, program loader type, project time constraints, and ingenuity of the analyst will impact the instrumentation effort.

The analyst will need to add instrumentation to identify where the execution of each distributed operation begins and ends. This instrumentation can be added by hand but there are some ways to alleviate this effort. First, the instrumentation for the end of an operation can be reduced if it is assumed that an operation implicitly ends when another operation begins. Secondly, some external interactions can occur transparently by automatically generating an operation name. For example, a signal handler or software interrupt routine is a good candidate. Lastly, the generation of angio traces may be transparently incorporated into the design testing effort with little additional cost, as well as providing additional information for debugging.

Application level instrumentation is also needed to identify the beginning of a task period, the end of a task period, and any activities to be considered. The additional

application level information recorded by the activity events may be automated or hand instrumented. This is similar to code coverage which uses automation extensively. The instrumentation to identify task periods requires careful consideration because it may be a heavier burden than identifying where an operation begins because there may be a larger number of tasks.

There are three independent approaches that can be taken to reduce the task period instrumentation. First, the start of a task period can be deduced automatically for some system operations such as an RPC message acceptance or task creation. This is also true for the ending of a task period. The operation level information stored with the activity events may be an additional source of information.

A second approach is to identify where synchronization between operation threads occurs. A service period serves as a boundary between different scenarios and it identifies synchronization between operation threads. However, separation between scenarios can be deduced from the scenario name values in the operation time stamp. So, if the synchronization points are instrumented then the service periods can be deduced. For example, synchronization can be automatically identified by nested accept statements in ADA, nested, interleaved RPC interactions, or synchronization barriers in parallel programs [30].

The third approach is to introduce constraints on the operation and use heuristics to deduce the start and end of a task period. If an operation is constrained to being initiated by a single external event then the history of the operation can be used to infer the start of a task period, the end of a task period, and where a synchronization occurs. If a single test-driver is used to initiate an operation then this is a feasible approach.

The selection of which approach to adopt should be assessed for each environment. However, it should be pointed out that the task period information is important design

documentation, which is generally not captured. It is recommended here that it should become part of the documentation.

The node connection specifications can be extended to include several task interactions that do not send messages, provided the run-time system can be instrumented. Table 19 lists several task interactions and how the node connection specifications can be applied.

Task Interactions	Initiating task	Responding Task
An initiating task creates a child task.	An asynchronous message send for the child task to "create itself".	A message reception that begins a new task period.
An initiating task removes another task.	Asynchronous message send for the other task to "remove itself"	The message reception introduces a synchronization and the task exits as described in this table.
An initiating task waits for another task to exit	An RPC interaction from the initiating task to the responding task.	The message reception introduces a synchronization and a reply is sent to the initiating task when the responding task exits.
An initiating task blocks another task from executing	An asynchronous message send from the initiating task to the responding task to "stop executing."	The message reception introduces a synchronization. The responding task then waits to receive another message which will be a synchronization with a "resume execution" message.
An initiating task unblocks another task that was previously blocked	An asynchronous message send from the initiating task to the responding task to "resume execution."	The message reception introduces a synchronization and the task continues execution.
An initiating task exits.	The operation thread and task period ends so an end event is recorded.	N/A

Table 19: Task Interactions and their Message Based Representation

This concludes the discussion about the automated development process and the case study is discussed next.

Chapter 9.0 Case Study

In this case study an LQN is generated from the execution of a detailed simulation of an On-Line Transaction Processing system (OLTP). An instrumented OLTP simulator was used as a surrogate for the real target system. It captured the important hardware and software details of the OLTP system, being faithful to its functionality, operation, and structure, especially the task concurrency and interactions. The angio trace events that are recorded by the simulator are identical to those which would be generated from a target system. The exercise of building the LQN performance model demonstrates the feasibility and accuracy of the process, as it recovers from a trace the architecture and performance parameters of a scenario, and it copes with the complexity and scale of the system, which (apart from its timing parameter values) represent a real product. The simulator of the OLTP was developed previously and it is used in place of the real environment. It is very similar to the OLTP system presented in section 2.1.

The case study follows the model building process of TLC (Figure 1). The steps involved are:

1) Instrument the application tasks.

2) Add the angio trace instrumentation to the operating system, which in this case is the simulation engine.

3) Execute the instrumented system to record angio trace events that describe several scenarios.

4) Construct a Task and Operation Event Graph from the angio trace events of each scenario.

5) Transform each TOEG into an LQN sub-model.

6) Combine the LQN sub-models into an LQN model for evaluation purposes.

Each of these steps is considered here. The tool Performance Analysis by Model Building (PAMB) automates the latter steps. PAMB was discussed in section 5.3.

9.1 A Description of the Tandem OLTP System

The OLTP used in this case study served as an "intelligent network" database for a telephone system (e.g., for 800-number translation) with short transactions and stringent performance requirements. A description of the main components of the OLTP system follow. For a more complete description of the OLTP system architecture provided by Tandem Computers, the reader is referred to [20, 12, 13, 95, 175, 19, 11, 61, 18, 159].

9.1.1 Hardware

The OLTP hardware architecture is composed of processing nodes with 2 to 16 CPUs per node. Communication between CPUs on the same node use two buses that are collectively referred to as the *Dynabus*. Each CPU has an *I/O channel*, which is a bus dedicated to interfacing to the outside world via I/O controllers. There are two main types of I/O controllers: communication controllers and disk controllers. Communication controllers interface to user terminals, LANs, etc. Disk controllers can interface up to eight disk drives.

9.1.2 Database

The database is composed of SQL tables which are made up of several components. A database definition defines the data fields of an SQL table. The SQL table is itself made up of data records. Each SQL table has a primary index to quickly access a data record. Optional secondary indices can be defined for an SQL table.

Each SQL table is a file that can is composed of file segments. A file segment can reside on any disk. Primary and secondary indexes are files and they too are made up of file segments.

9.1.3 Communication

All inter-task communication is by point to point message passing. All messages are sent by value. Each task has a message queue, where messages are queued in either a FIFO manner or according to the sender's priority.

In general, the remote procedure call is the communication protocol used for inter-task communication because it provides a positive acknowledgment of completion. However, asynchronous communication can be used.

9.1.4 Software Architecture

The software architecture of this OLTP system consists of many small tasks, with each one providing a single service. So, operating system functions are implemented using intertask messages instead of procedure calls. The database transactions follow this paradigm, being managed by user-written tasks. For fault tolerance purposes, a task is context free in the sense that it retains no memory from the servicing of one request to the next.

A pool of tasks can be dedicated to providing the same service. Requests are then sent to the *task pool* rather than to an individual task and any free task in the task pool can accept the request. This allows requests of the same type to be executed concurrently. For example, there can be more than one instance of a particular application task, allowing the same type of transaction to be executed by several tasks in parallel.

The service provided by each system task is organized around the transaction. The tasks are similar to those described in section 2.1, and they are:

- Communication task: It is responsible for terminal input and output.

- Dispatcher task: Manages screen operations, field validation, and the routing of application requests.

- Name server: Provides address information for inter-task messages by determining which queue a message should be sent to.

- Application task: It manages a transaction, as well as translating the high level SQL request into lower-level SQL database operations.

- Disk task: Translates low-level SQL statements into disk accesses.

- Memory management: Provides the virtual memory management functions.

- Audit task: Performs the logging (journaling) of transactions so that transactions can be backed out in the event of failure.

The Dispatcher and Disk tasks are discussed a little further.

Three types of services are provided by the Dispatcher and modeled in the case study. They are:

- Screen request: A terminal operator's screen format is defined by files on a disk. The Dispatcher manages the terminal interface by retrieving screen format files and then interpreting them for display on the terminal.

- Validation request: The data entered by the operator is validated to ensure that correct data was entered (e.g. checking that a correct date has been entered in a field). It can also perform a computation with entered data (e.g. adding two fields).

- Application request: A transaction is requested and is sent to the appropriate application task.

The Dispatcher does not block when sending a transaction request to an application task.

The disk task controls all access to a disc volume, including cache management, lock management, journaling, field projection, predicate evaluation, and set-oriented access. A disk task or disk task pool will receive low-level SQL operations and convert them into commands to the appropriate disk controller.

9.1.5 An Example Transaction

The manner in which the tasks interact to satisfy a user's transaction is:

1) User: Inputs the data to the terminal and indicates that the transaction is ready to be performed.

2) Communication task: Receives the terminal's transmission and converts it to an inter-task message. This message is then sent to the Dispatcher.

3) Dispatcher task: Receives the transaction request and parses the data to make sure it was entered properly. Assuming the data was entered properly, the Dispatcher must determine the application task's message queue, so it requests this information from the Name Server.

4) Name server: It sends a message back to the Dispatcher with an appropriate message queue identifier.

5) Dispatcher task: The Dispatcher sends the transaction request to the application task.

6) Application task: The application task receives the request and begins to satisfy it by converting it into SQL operations. These SQL operations are decomposed into low-level SQL statements and then sent as messages to one, or more, disk tasks.

7) Disk task: The disk task receives a low-level SQL operation and performs all of the disk I/O to satisfy the SQL operation. The results of the SQL operation are sent back to the requesting application task. The disk task will also send messages to the audit task so that the transaction can be backed out if necessary.

8) Application task: The application task receives the results of the low-level SQL operations. Once all of the SQL operations have been performed, and all of the data received and operated on, the application task sends a message back to the Dispatcher. The application task will also periodically send messages to the audit task so that the transaction can be backed out.

9) Dispatcher task: It receives the data back from the application task and then sends it to the communication task. A blocking request is made to the Name Server to retrieve the message queue identifier of the appropriate communication task.

10) Communication task: It receives the reply and formats it for block transmission over the I/O channel to the communication controller and the user's terminal.

11) User: Receives the reply to the transaction request.

Having described the key elements of the OLTP system, the simulation toolkit that is used to construct the OLTP simulator is described.

9.2 Simulation of the Hardware and Operating System

The OLTP simulator was constructed using PARASOL, a discrete-event simulator package intended to simulate a modern distributed and/or parallel computer system. PARASOL's simulated execution environment is built up of multi-processor nodes interconnected with communication devices of various types and capacities (Figure 59).

The concurrent software, which may represent both application and operating system software, is simulated explicitly with user-defined *tasks* written in C or C++. A *simulation driver* looks after running the simulation. Parasol can also capture performance statistics in a fashion equivalent to making measurements. Operationally speaking, a PARASOL-based simulator runs as a single task in a POSIX compliant host environment. PARASOL is written entirely in ANSI C and is therefore highly portable and easily extended. This description of PARASOL is taken from [110, 109].

The PARASOL software interface may be thought of as a sort of primitive distributed operating system, containing a number of functions to support: task management, including dynamic task creation and destruction as well as task migration; inter-task communication and various shared memory mechanisms; and task synchronization, through spin locks and semaphores. Many of its function calls resemble those of thread packages and distributed operating systems. The busy waiting spinlock and counting semaphore synchronization mechanisms were not used and are not discussed further.

PARASOL is well suited to simulating the OLTP system. It allows true concurrency to be simulated in a multi-processor system, with scheduled concurrent tasks. The message passing and task execution is described in detail.

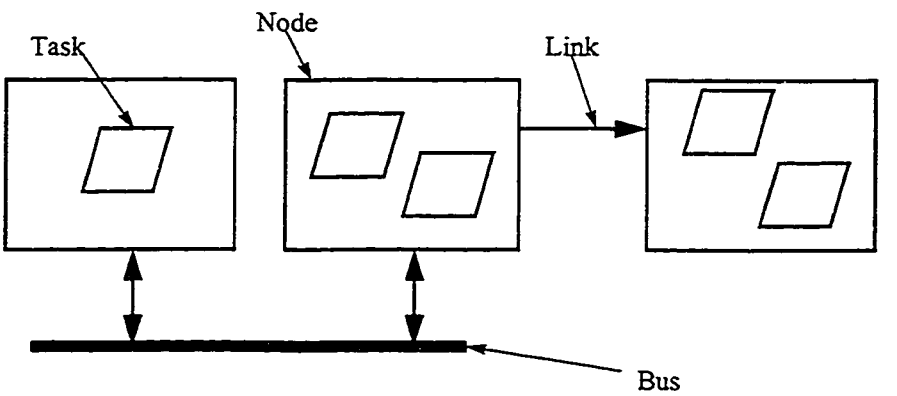


Figure 59: Components in a Parasol Simulation

9.2.1 Execution Environment

The simulated execution environment on which PARASOL tasks execute is constructed from nodes, one-way communication links, and multi-way communication buses. Each of the "hardware" components has a finite capacity to characterize queueing for devices.

A PARASOL node may have one or several processors. Each node has a single ready-to-run queue and is managed by either a built-in scheduler or by a user-defined scheduler.

Network connections are made with user-configurable shared buses or dedicated point-to-point one-way links.

9.2.2 Tasks

PARASOL tasks are sequential software entities that serve a number of purposes. The most obvious application is for a PARASOL task to simulate a real system task. They are also frequently used to perform peripheral duties (e.g., as workload generators and statistics gatherers). A task may also act as a manager of other tasks.

9.2.3 Communication Mechanisms

PARASOL uses a port-based style of inter-task communication derived from Mach [1], where ports are the receptors of messages. A port is really a mailbox from which the owning task can retrieve a message. So, when a message is sent, it is directed to a port rather than another task. Tasks are created with one standard port but they may own more if appropriate. Extra ports may be passed from task to task along with any queued messages or grouped into port sets to enable a task to listen on several ports at the same time. As well, a global type of port, known as the shared port, is available which permits any task to retrieve messages from it.

A message send operation will never block the sending task. Aside from the point-to-point message sending operation, PARASOL provides operations to send a message

- Transaction types are defined by user written application tasks that perform SQL operations.

- The workload is simulated as users making transaction or terminal requests.

Figure 60 shows the example simulator configuration that was traced in this case study. Unless otherwise shown in the figure, a PARASOL task has its own CPU to execute on to correctly characterize concurrency.

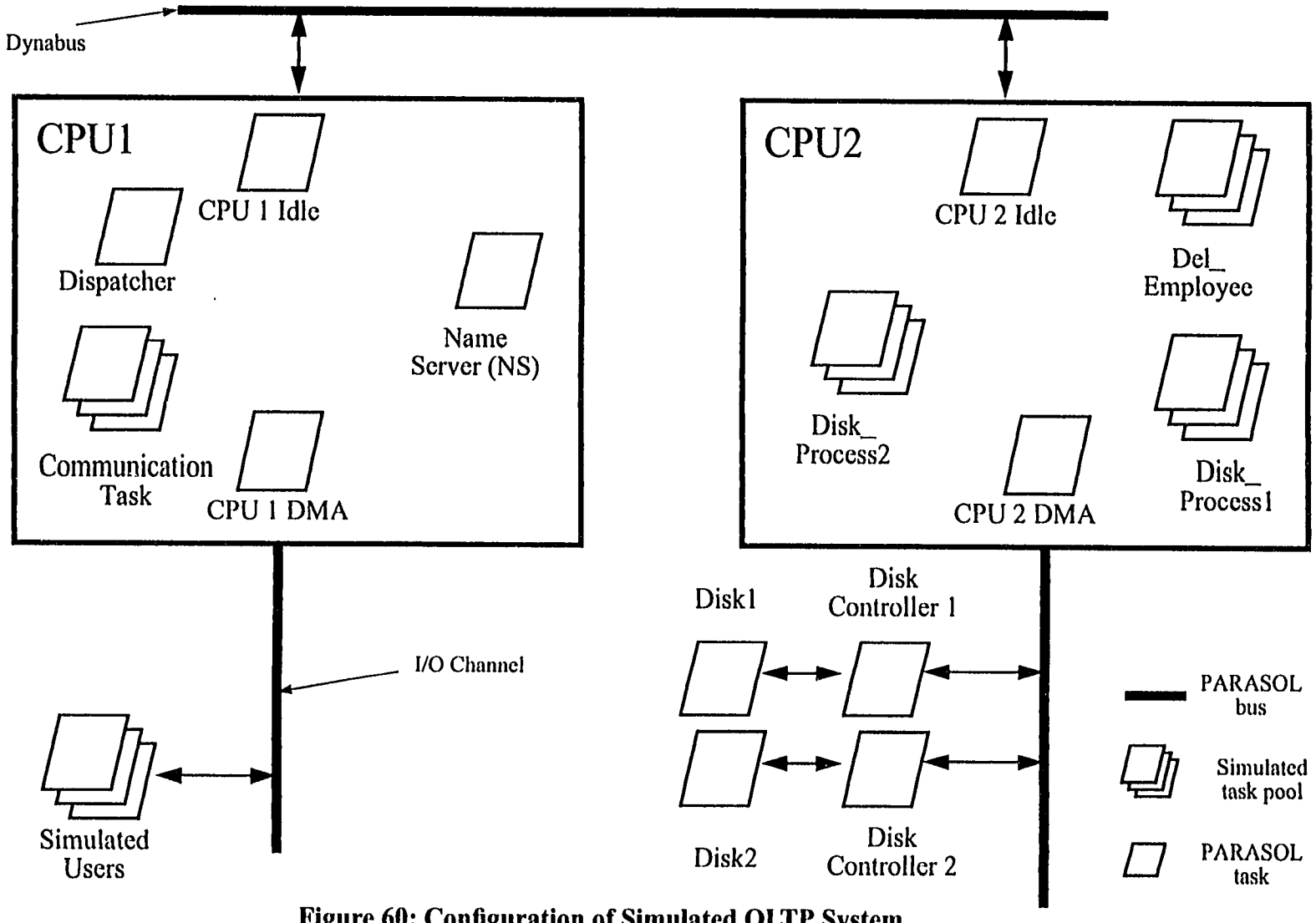


Figure 60: Configuration of Simulated OLTP System

The example database has several SQL tables with file segments that are split across the two disks. The details are not give here because they are not essential to the demonstration of the model building.

9.3.3 Users

A PARASOL task simulates the actions a user at a terminal performs and it is referred to as a *simulated user*. The simulated user can perform many actions. In the example simulation each simulated user can make any of the three types of requests discussed in section 9.1.4. Several transaction requests were developed in the example although only the "delete user" transaction will have an angio trace recorded and a sub-model constructed.

Each simulated user has a dedicated communication task associated with it. The creation of the communication task is performed transparently when the simulated user is created.

9.3.4 Software Architecture Simulation

The software architecture of the OLTP was faithfully reproduced in the simulator with the exceptions of the memory management and audit task (they were not included). Some of the simulated OLTP tasks use at least two Parasol ports for intertask communication: an event port and a data port. The event port is used as input for a task's state machine while the data port provides parameters for database operations. Simulated OLTP tasks which communicate over the I/O Channel have an additional port dedicated to I/O channel communication.

To simplify the construction of the OLTP simulator a high-level program interface was developed to construct a task pool. The task pool was simulated using a PARASOL task to manage the task pool and a set of PARASOL tasks for the worker tasks. This arrangement allowed statistics to be gathered for a set of worker tasks.

The operation of the simulated task pool follows this pattern:

- The worker signals the manager when it is free, blocking until the manager sends it a request.

- A service request from an initiating task is received by the task pool manager. The manager will randomly pick a free worker to satisfy a request and then send the request to the worker.

- The worker awakens when it receives a request and processes the request.

- When the worker is finished processing the request it will reply to the initiating task.

- The worker will then signal the manager that it is ready for more work, and again block until it receives a request.

Other messages are also exchanged between a worker task and its manager for management purposes (e.g., initialization of the task pool).

9.3.5 Simulator Limitations

The limitations of the simulator fall into three categories. It does not simulate the fault-tolerant nature of the OLTP computer system. It does not simulate advanced resource management policies. Finally, exception conditions are not simulated. The limitations are:

- Complex read and write buffering schemes are not simulated.

- Disk controllers use a FIFO algorithm to service requests.

- DMA operations are performed in bulk and interrupt the processor.

- Verified disk writes are not supported.

- Backup task pairing is not simulated.

- Virtual memory management is not simulated.
- Message transfers are not abortable by either the requester or responding task.
- The disk does not "pre-fetch" data.
- The database operations are not logged.

Even with these limitations the OLTP simulator properly describes the concurrency, blocking, synchronization, and task architecture.

9.4 Operating System Instrumentation

Angio tracing instrumentation was added to PARASOL which is analogous to instrumenting an operating system. An earlier attempt to instrument PARASOL with the first generation of angio tracing helped because it indicated where to add the third generation angio trace instrumentation.

The first instrumentation change made was to add the task and operation time stamp data structures to the PARASOL task data structure. Then the PARASOL message data structure was modified to include two operation time stamp values, one for the sender and one for the receiver. These were the only changes needed to the data structures.

The PARASOL functions that were used to send and receive messages were instrumented according to the descriptions of Table 18. If a message reception timed-out this was handled as an activity and not as an external event because it was considered to be application level information. The forking of a task was also instrumented to look like a message send and reception.

9.5 Application Level Instrumentation

The existing OLTP simulation of Figure 60 was instrumented without first examining the documentation. Instrumentation was added to identify the beginning of a task cycle for the following tasks: DMA, Idle, disk controller, disk worker, responding task worker, name server, dispatcher, and the communication task. Instrumentation was also added to enable the recording of the angio trace and to identify the external event of each simulated user's request type.

This initial instrumentation effort did record an angio trace but the automated post-analysis of the trace showed that the traces were invalid. Examination of the traces showed that the Dispatcher task and the various task pool managers were interleaving their service periods, as described in section 7.4. This was verified by examination of the code. A second round of instrumentation solved this problem by recording the suspending and resumption of operation threads, to capture the service period interleaving.

The traces also revealed that different communication protocols were used in the simulator. It had been forgotten that many of the OLTP simulated tasks use two messages to simulate one OLTP message: one for the event port and one for the data port. In particular, the communication protocol used by the disk worker task, disk controller task, and disk task appeared to be so unusual that it was manually verified to make sure the trace captured the actual execution.

The simulator had other realistic details. It violated the assumption that a task only executed on behalf of an operation because several PARASOL tasks did house-keeping chores that were not associated with an operation. Tasks also performed activities that did not have a duration (i.e., took place in zero simulation time).

9.6 Trace Analysis

The trace analysis was straightforward once the traces passed the sanity test and the instrumentation was validated.

However, the trace analysis did have to be adapted for PARASOL. An interesting property of PARASOL is that a task can sleep for the duration of a simulation which is equivalent to the task being killed since it does not execute. This is used because killing a task will also kill its descendents, which is often not wanted. This posed a difficulty for the trace analysis because, when this situation occurred, the task's operation thread and period did not end so the trace was thought to be incomplete. This was overcome by post-processing the trace to detect the "sleep forever" activity and replacing it with a thread-end event.

There were three types of simulated user requests that were traced and analyzed using the PAMB tool: a validation request, a screen request, and a "delete an employee" transaction request. The sub-models that were generated for each of these request types are shown below.

9.6.1 Visualization of the Generated Sub-models with Dot

An option of PAMB automatically generates figures that show a sub-model's task interactions and these figures are used in the following sub-sections. The program that generates the figures is called 'dot' [44].

The generated figures have a slightly different notation than the LQN icons of Table 5. The automatically generated figures show the task as being a collection of boxes, with the left-most (or right-most) box having the task name and the remaining named boxes are entries.

The directed edges that connect two entries identify the interaction type as well as providing ordering information with two digits: the first digit identifies the initiating task's

phase that the interaction occurs within and the second digit is an index for the interaction in that phase. For example, an edge label of "2.2 rpc" indicates that this is an RPC interaction that occurs in the second phase of the initiating task, and it is the second interaction of the task in that second phase. This information was useful for connecting the model to the sequence of evens in the trace, and verifying that all the events were captured in the model.

The last component of the figures is the Customer node whose start edge identifies the first entry that initiates the distributed operation, which is the task "User 0" in Figure 61. The resource demands are not shown in the figures.

9.6.2 Validation Request Sub-model

A validation request sub-model was developed from an angio trace and it is shown in Figure 61. The first entry that initiates the distributed operation is the task "User 0" which is identified by the Customer node's start edge.

The validation request sub-model has more elements than the comparable sub-model of Figure 11. The sub-model of Figure 61 includes the additional tasks that are used to model the DMA activity on the Dynabus ("CPU 1 dynabus idle") and on the I/O channel ("CPU1 DMA"). Many of the interactions are properly interpreted as RPC interactions even though they are constructed from asynchronous messages (e.g. between entry E008 of "User 0" and entry E006 of "User 0 Line Hand"). On the other hand, the asynchronous interactions between the "Dispatcher" and "CPU 1 dynabus idle" task are correctly interpreted as not being RPC interactions because the Dispatcher records "suspend" and "resume" activities due to the service period interleaving.

An example of the two-stage protocol for simulating an OLTP message occurs when the Entry E001 of "CPU 1 dynabus idle" sends two messages to the Dispatcher, one for the event port and one for the data port. An and-join activity provides the synchronization of entries E009 and E003 of the Dispatcher. An item that will recur is that all inter-task

communication is routed through the Idle tasks because they route the messages across the Dynabus if needed.

9.6.3 Screen Request Sub-model

The screen request sub-model of Figure 62 is a little more complicated than the validation request because it involves a disk task pool.

The disk task pool manager is "Disk_Process2" and the worker task is "Disk_Process2 Drone1". The operation of the task pool begins with the screen request being received at entry E017 of "Disk_Process2" and this request is sent to entry E014 of worker task "Disk_Process2 Drone1". The worker task consumes CPU time for transferring the screen information from CPU1 to CPU2 using the RPC interaction (edge "1.1 rpc") with "CPU1 dynabus idle". Then it sends a message to the Dispatcher via the same Idle task asynchronously (edge "1.2 asy"). The worker task then informs its manager task that it is free to do more work (edge "1.3 asy"). The disk process does not perform any disk accesses because it is assumed that the screen files remain in cache.

9.6.4 Delete Employee Sub-model

The delete employee sub-model of Figure 63 involves many more tasks than the other sub-model, including PARASOL tasks which are used to represent hardware components. A disk controller is represented by the "Disk_Controller2" task and an attached disk is task "Disk2". There are two task pools that are shown: the "del employee" application task pool and the disk task pool "Disk_Process2". The protocol between the "Disk_Process2 Drone1" worker task, the "Disk_Controller2" task, and the "Disk2" task is not straightforward but it was verified.

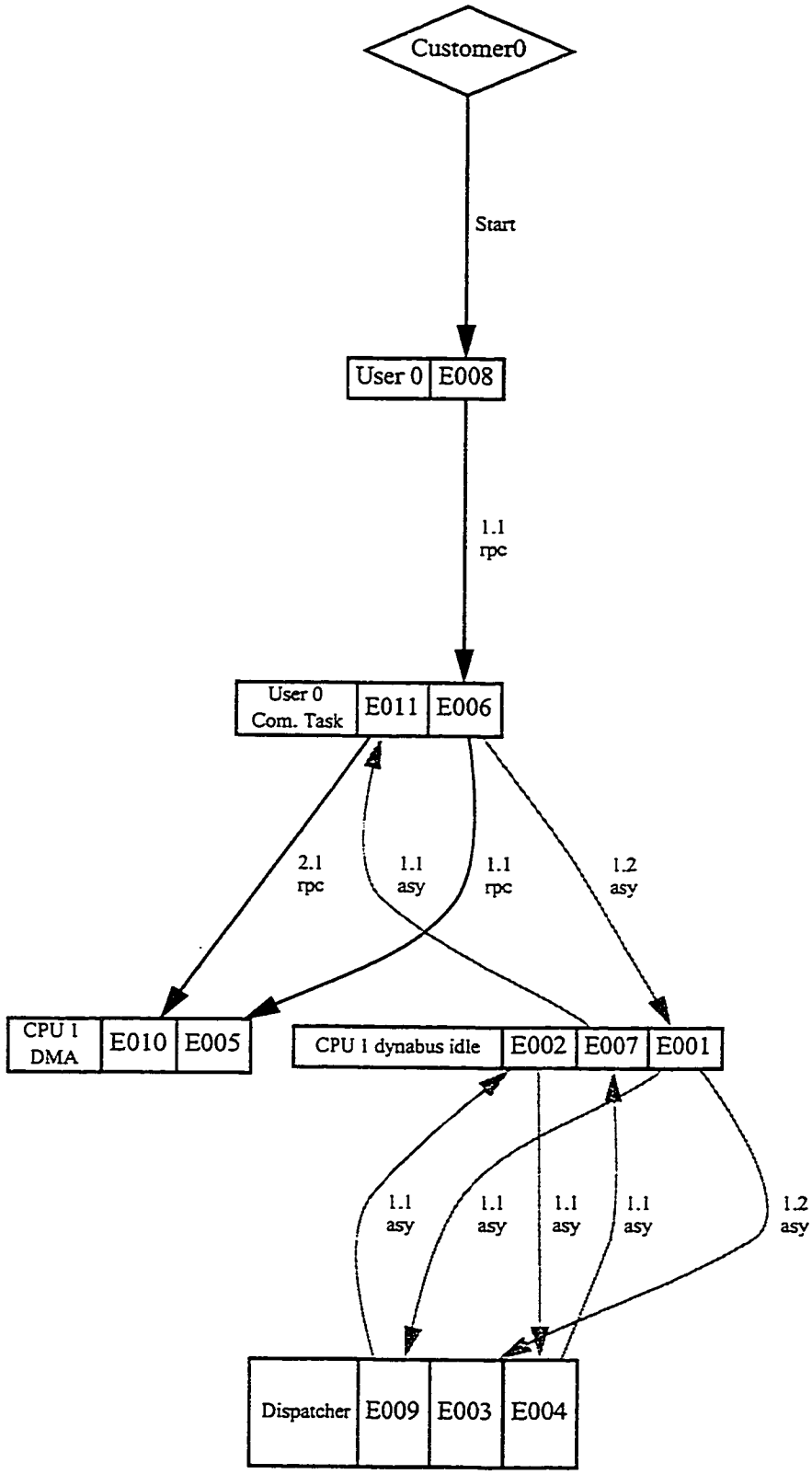


Figure 61: Validation Request Sub-model

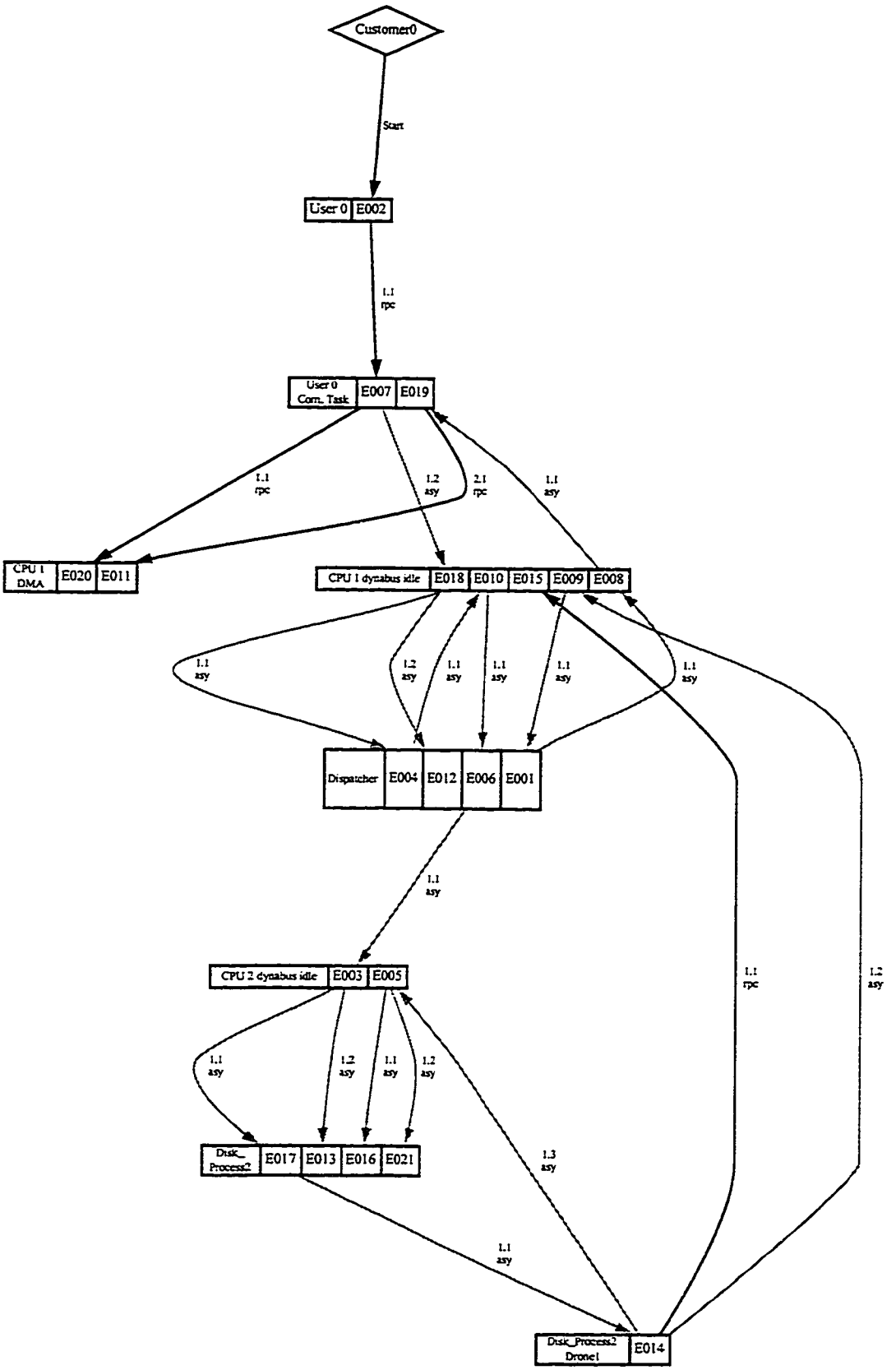


Figure 62: Screen Request Sub-model

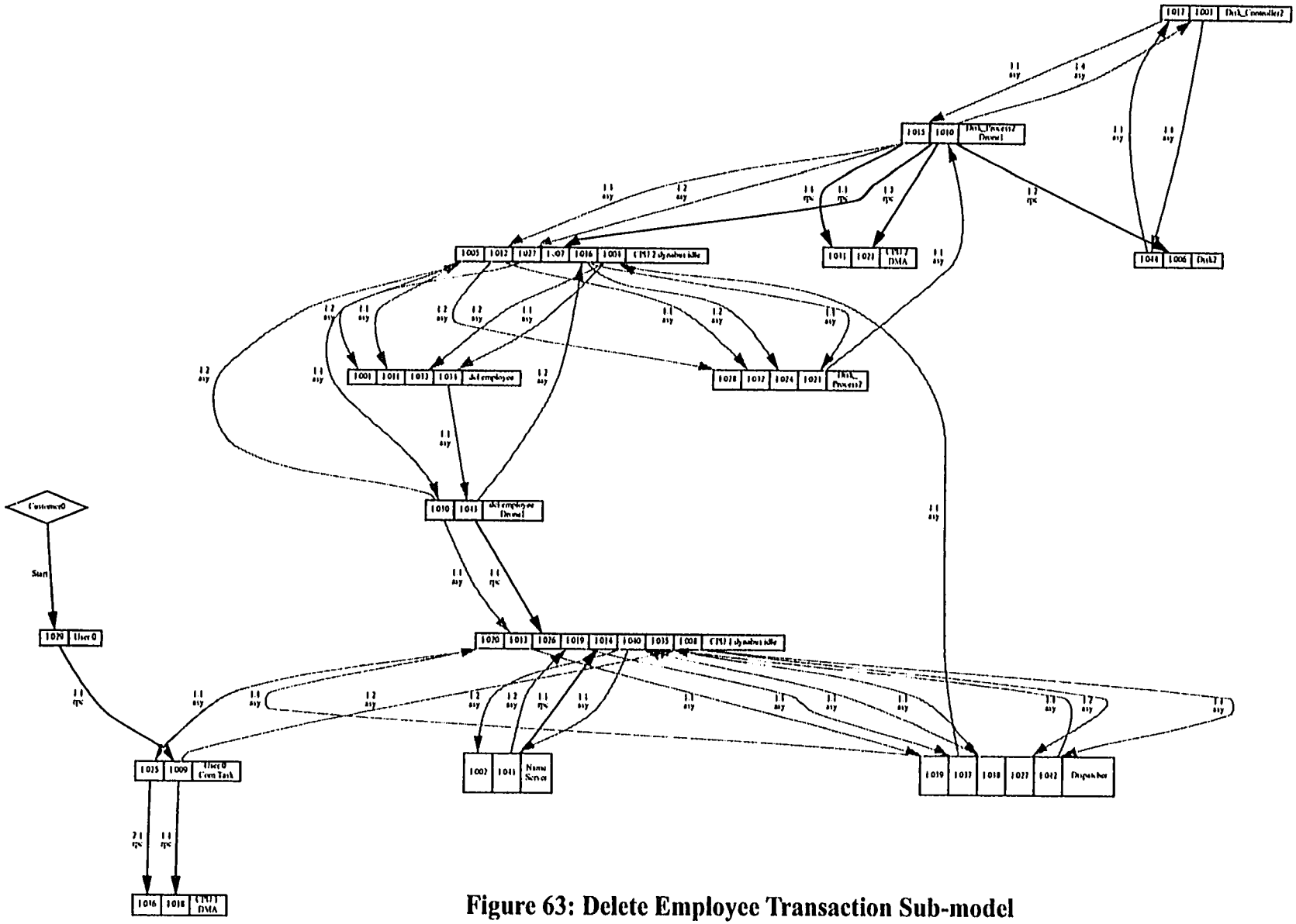


Figure 63: Delete Employee Transaction Sub-model

9.7 Building an LQN from LQN Sub-models

Building an LQN using the Performance Analysis by Model Building tool was straightforward, with one exception that limited the range of experimentation. As described in section 5.8, the combination of a responding multi-server task needing to synchronize with an asynchronous message before replying is currently not supported by the LQN toolset. This restricted the Dispatcher task to being a single server. The generated LQN was solved by a simulation tool.

The model building technique produced a high quality model because it did not leave out important details. The angio traces captured more detail than was expected, including identifying information that would have been overlooked had it not been captured in the trace. The trace processing faithfully characterized the task architecture, task interactions, and synchronization of the angio trace scenarios, when generating the LQN sub-models.

In this example, activity identifiers were not used since the service times were part of the simulator. The simulator's service time values were derived from Tandem Computer data. If the values were not available then the important activities could be found by inspection and are listed in Table 2. What is not described in this table are the additional parameter values that can be added to the resource functions make the service times more realistic. For example, all message based activities should be qualified by the size of the messages.

The one area that is complicated for developing a resource function is the mapping of SQL operations to physical disk accesses. This is difficult because the logical representation of the SQL tables insulates the analyst from the location and physical representation of the data, making it difficult to relate the SQL operations to the appropriate disk [50]. The physical file characteristics (e.g. file partitioning, file placement) also influence how SQL statements are translated and therefore affect resource selection and demands. Often, an SQL optimizer program uses this information to minimize disk

accesses [158, 79]. It should be possible to use the output of the SQL optimizer to provide the resource function values directly for a given database configuration and query.

9.8 Discussion

During the model building process many details were uncovered that had been forgotten, illustrating that detailed knowledge of the system is not essential for building a correct model. The approach also seems to scale, dealing with small and large traces

The trace analysis correctly identified an RPC constructed using asynchronous messages. The generated sub-models could have been further simplified by using the graph rewrite rules to simplify the two-stage communication protocol between the Idle task and a task pool. This would have made it easier to understand the resulting model. It also suggests that the abstraction process can be extended by defining graph rewrite rules to simplify angio traces from a particular environment.

Manually constructing an LQN is a subjective matter so comparing a manual approach to building a model with TLC is difficult. It seems to the author that the generated LQN's structure is at least as good as could have been constructed manually, or even better in that it captured structural details that were forgotten.

Chapter 10.0 Conclusions and Future Research

10.1 Discussion

TLC captures the total workload and the task architecture of a scenario so that an LQN performance model can be generated in an automated fashion. TLC is appropriate for a message passing distributed system where tasks interact by point-to-point communication.

This research has extended and formalized the various models used in TLC's automation process beyond that described in [65]. TLC now characterizes asynchronous messages and the synchronization between tasks. To accomplish this, two formal model types were developed: the angio trace and proper time. *Proper time* characterizes the execution of a distributed operation by cause-and-effect event relationships, as well as identifying blocking, non-blocking, and synchronization interactions between tasks. A proper time model is a Task and Operation Event Graph (TOEG) which is a new graph grammar. An *angio trace* is an instrumentation specification for generating a TOEG and it is a new, causal logical clock. The LQN *sub-model* is a third model type that is used to assemble several distributed operation scenarios into an LQN.

The Performance Analysis by Model Building tool (PAMB) provides a common user interface to the several tools that implement the automation process. PAMB guides the analyst through the steps to develop a model.

Some instrumentation and monitoring limitations can be accommodated by TLC. Instrumentation trace errors are identified by the tools and manual intervention is required to resolve an instrumentation problem. The angio trace specification of Chapter 8.0 can accommodate for a single lost event but multiple lost events may well be unrecoverable.

importance. The automation in TLC helps by allowing a very large number of scenarios to be used, allowing TLC to piggy-back on the testing effort.

The next sub-section suggests a high-level approach for applying TLC.

10.2 Applying TLC for Performance Analysis or Tuning Purposes

The steps in applying TLC in a performance analysis or tuning context are shown in Figure 64.

The first step (#1) is to select scenarios which are important for performance modelling and assigning operation names to them. In step #2, the angio trace events are recorded for each scenario and LQN sub-models are developed. From the configuration and environment information a performance experiment is defined and, in step #3, a performance model is generated. The model is evaluated (step #4) to produce performance predictions. The performance analysis is complete if the performance predictions meet the performance objectives (step #5).

If improvements need to be made, the automation of TLC allows a software developer to experiment with an LQN model and test possible improvements. One approach is to manipulate the model to compare alternatives without having to alter the existing design (step #6). Another option is to change the environment (step #7), such as reallocating tasks to processors, adjusting a task's priority, or providing faster hardware. If necessary, adjustments to the actual design are made (step #8) and the analysis is revisited.

Finding a solution to a difficult performance problem is still the domain of the performance analyst. Although TLC does not eliminate the need for the performance analyst, it provides a model with which the software developer(s) and performance analyst can communicate effectively to resolve the problem.

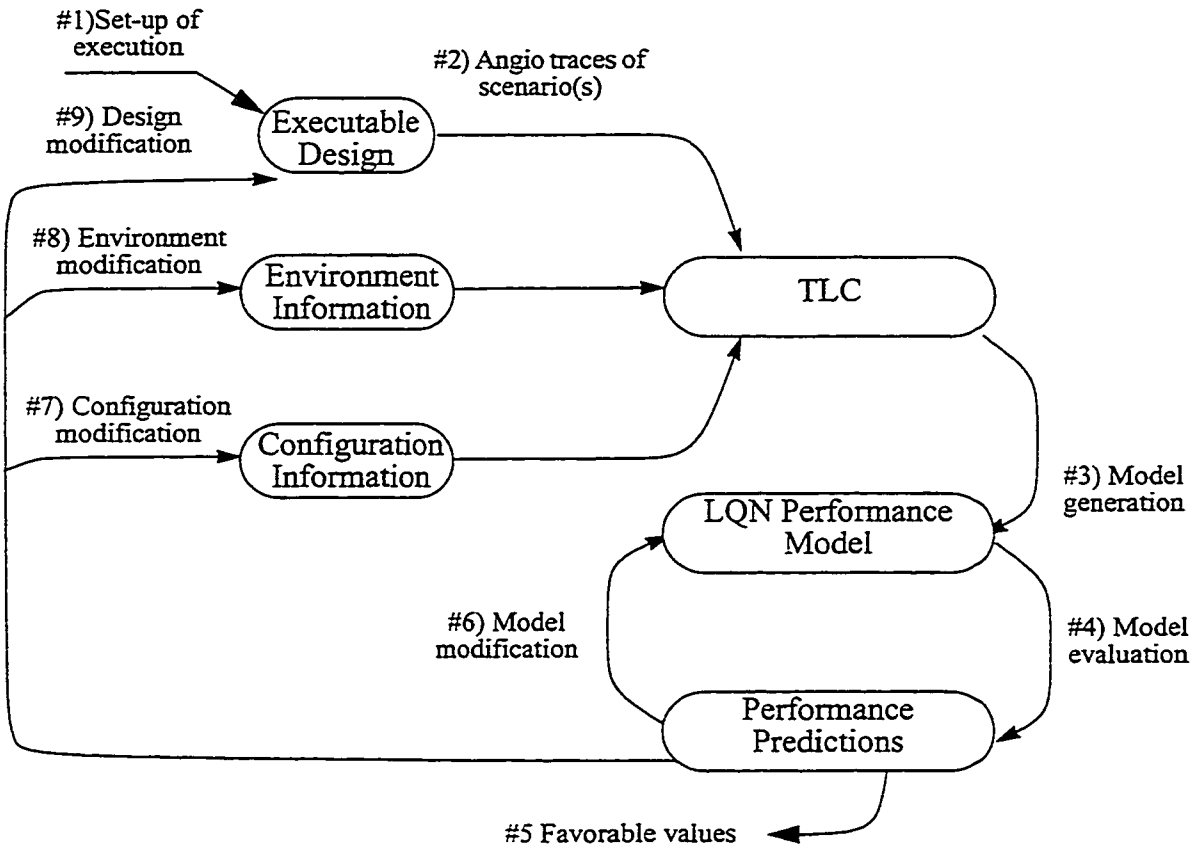


Figure 64: Performance Analysis and Tuning with TLC

10.3 Future Directions

A short-term goal is to relax the assumptions associated with angio tracing, proper time, or the sub-model. This includes generating models with internal concurrency by extending the model building tools. Another extension is to incorporate other communication protocols (e.g., multi-cast communication) into the characterization.

A longer term research goal is to generate models for different types of task interactions, such as state-based synchronization (e.g., semaphore). The development of resource functions is also ongoing research.

There are other research opportunities that are associated with each model type of TLC.

Proper time can be extended to characterize looping or alternative execution path selection so that it could describe a family of scenarios. It could also incorporate other dimensions of software, such as the inheritance hierarchy of software components. These extensions would make proper time more appropriate for validation purposes, where a scenario's specification is compared with its execution. It should be possible to translate a specification of a scenario's behavior (e.g., such as annotated specifications (e.g. Use-Cases [130], Message Sequence Charts [27], or Use-case Map [24]) into a proper time model, which could then be validated against a proper time model developed from the execution.

The angio trace appears to have properties that are different than other tracing techniques which result from its formal foundation. One property is that it can serve as a logical clock, which are could have many known applications, such as: distributed algorithms [104], global state recording [28], design recovery [84], trace replay [112, 89], the visualization of system execution [84], and debugging [98].

10.4 Perspective

TLC has several benefits and qualities that distinguish it from other transformational abstract model development techniques (section 2.3). First it uses formal models to bridge the semantic gaps that exist between the software description, software execution model, performance model, and the implementation. Secondly, the use of formal models allows the model transformation to be automated. Third, as a design evolves, increasing detail is easily incorporated into the LQN model to allow more complex performance concerns to be examined (just as MIDAS does for the executable model development technique), providing full life-cycle support for modelling. Fourth, there is traceability from the software components to the performance predictions and (theoretically) from the

performance predictions back to the software components using by-products of the automation process. Fifth, it provides a single input format (the angio trace) that allows various factors to be accommodated, such as software architecture [45], software design approach (e.g., object-oriented programming [130]), and implementation technologies (e.g., client-server [148], multi-threaded tasks [69]). Sixth, it can be used in a dynamic, message-passing distributed system (a Multiple Program Multiple Data paradigm) which the other development strategies had difficulty accommodating. Lastly, different types of models can be developed (as suggested by Ammar [7]), allowing advances in model solution techniques to be transparently included. Another type of performance model can be generated by providing an appropriate transformation for step #5 of Figure 1.

The automation provided by TLC has the main benefit of making the model construction less prone to analyst error and reducing the construction cost. This is most beneficial for large systems which have the greatest potential for performance failures. With TLC, the performance analyst can focus on the principles of software performance analysis rather than model building.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. *USENIX*, pages 93–112, July 1986.

[2] R. M. Adler. Distributed coordination models for client/server computing. *IEEE Computer*, pages 14–22, April 1995.

[3] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, Jhy-Chun, and D. A. Reed. An integrated compilation and performance analysis environment for data parallel programs. Technical Report 1902, University of Illinois, 1995.

[4] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, Texas, 1991.

[5] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. AWK – A pattern scanning and processing language. *Software – Practice and Experience*, 9:267–279, 1979.

[6] M. W. Alford. A requirements engineering methodology for real time processing requirements. *IEEE Trans. on Soft. Eng.*, SE3(1):60–69, Jan. 1977.

[7] R. A. Ammar and C. P. Rosiene. Visualizing a hierarchy of performance models for software systems. *Software - Practice and Experience*, 23(3):293–315, Mar. 1993.

[8] S. Andler. Predicate path expressions. *Proceedings 6-th ACM Symposium on Principles of Programming Languages*, pages 226–236, Jan. 1979.

[9] R. L. Bagrodia and C. Shen. MIDAS: Integrated design and simulation of distributed systems. *IEEE Trans. on Software Engineering*, pages 1042–1058, Oct. 1991.

[10] J. W. Baker, D. Chester, and R. T. Yeh. Software design by stepwise evaluation and refinement. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspectives*, pages 236–273. Van Nostrand Reinhold, New York, NY, 1990.

[11] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in Tandem Computer Systems. In

A. Avizienis, H. Kopetx, and J. C. Laprie, editors, *Dependable Computing and Fault-Tolerant Systems*, pages 55–77. Springer-Verlag, New York, New York, 1987.

[12] J. F. Bartlett. A "nonstop" operating system. In *Proc. 11th Hawaii International Conference on System Sciences*, volume 3, 1978.

[13] J. F. Bartlett. A nonstop kernel. *Proceedings of the Eighth Symposium on Operating System Principles (SIGOPS)*, pages 22–29, 1981.

[14] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM SIGOPS*, 24(1):11–22, Jan. 1989.

[15] R. L. Bennett, J. A. Lindner, R. W. Michelsen, and D. J. Rypka. SDL in 5ESS switching system development. In *IEE Sixth International Conference on Software Engineering for Telecommunication Switching Systems*, pages 184–189, London, April 1986. Institution of Electrical Engineers.

[16] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[17] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[18] A. Borr. Robustness to crash in a distributed database: A non-shared memory multi-processor approach. Technical Report Tandem Technical Report 84.2, Tandem Computers Inc., Sept. 1984.

[19] A. Borr and F. Putzolu. High performance SQL through low-level system integration. *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 342–349, 1988.

[20] A. J. Borr. Transaction monitoring in encompass: Reliable distributed transaction processing. *Proceedings of Seventh International Conference on Very Large Data Bases*, pages 155–164, Sept. 1981.

[21] B. Bruege and P. Hibbard. Generalized path expressions: A high-level debugging mechanism. *The Journal of Systems and Software*, 3:265–276, 1983.

[22] D. Bryan. An algebraic specification of the partial orders generated by concurrent Ada computations. In *Proceedings of Tri-Ada*, pages 225–241, New York, N.Y., 1989.

A.C.M. Press.

[23] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Int. Journal of Computer Simulation*, volume 4, pages 155–182, April 1994.

[24] R. Buhr and R. Casselman. *Use Case Maps for Object Oriented Systems*. Prentice Hall, 1996.

[25] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, August 1993.

[26] R. Campbell and A. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer-Verlag, New York, 1974.

[27] CCITT. Recommendation Z.120: Message Sequence Chart (MSC). Technical report, Geneva.

[28] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[29] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[30] L. Clarke and I. Glendinning. The MPI message passing interface standard. In K. Decker and R. Tehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218. Birkhauser Verlag, 1994.

[31] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[32] P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sotz. ZM4/ Simple: A general approach to performance measurement and evaluation of distributed systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 286–309. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[33] Distributed Computing Environment: an overview. Technical report, Open Software

Foundation, 11 Cambridge Center, Cambridge, MA 02142, Jan. 1992.

[34] C. Diehl and C. Jard. Interval approximations of message causality in distributed executions. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 363–374. Springer-Verlag, February 1992.

[35] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69, 1979.

[36] H. Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 3–14, 1987.

[37] H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 24–37, 1991.

[38] J. Engelfriet and G. Rozenberg. Graph grammars based on node rewriting: An introduction to NLC graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 12–23, 1991.

[39] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Institute for Software Technology and Parallel System, Wien University, Austria, September 1993.

[40] C. Fidge. Partial orders for parallel debugging. In *Proceedings of ACM SIGPLAN/ SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, 1988.

[41] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, August 1991.

[42] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of 10th International Conference on Distributed Systems*, pages 134–141, 1990.

[43] G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton University, Ottawa, Canada, K1S-5B6, 1998. In preparation.

[44] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, Mar. 1993.

[45] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowlede Engineering*, volume I. World Scientific Publishing Company, 1993.

[46] G. Geist, M. Heath, B. Peyton, and P. Worley. A user's guide to PICL. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, March 1991.

[47] W. M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software – Practice and Experience*, 11:435–466, 1981.

[48] M. Glinz. An integrated formal model of scenarios based on statecharts. In W. Schäfer and P. Botella, editors, *European Software Engineering Conference ESEC '95*, pages 254–271, September 1995.

[49] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[50] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[51] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology, Atlanta, GA, 1994.

[52] M. C. Hao, A. H. Karp, A. Waheed, and M. Jazayeri. VIZIR: An integrated enironment for distributed program visualization. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 288–292, 1995.

[53] J. C. Harden, D. S. Reese, M. B. Evans, S. Kadambi, G. J. Henley, C. E. Hudnall, and

[63] C. Hrischuk. The automatic generation of software performance models using prototypes. Master's thesis, Dept. of Systems and Computer Engineering, Carleton University, 1995.

[64] C. Hrischuk. Implementing angio trace analysis using the graph rewriting tool PROGRES. Technical Report SCE-96-09, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6, 1996.

[65] C. Hrischuk, J. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 399–409, 1995. http://www.sce.carleton.ca/ftp/pub/rads/syncWthread.ps.Z.

[66] C. Hrischuk, C. M. Woodside, J. Rolia, and R. Iversen. Trace-based load characterization for generating software performance models. Technical Report SCE-97-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6, 1997. To appear in IEEE Trans. on Software Engineering.

[67] A. Hubbard, C. M. Woodside, and C. Schramm. DECALS: distributed experiment control and logging system. In *Proc. of CASCON'95, Meeting of Minds*, Toronto, Canada, November 1995.

[68] O. C. Ibe, H. Choi, and K. S. Trivedi. Performance evaluation of client-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):1217–1229, Nov. 1993.

[69] I.E.E.E. *Threads Extension for Portable Operating Systems*, POSIX P1003.4a edition, 1990.

[70] M. Jackel. Ada-concurrency specified by graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 262–279, 1987.

[71] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, New York, 1991.

[72] D. Janssens and G. Rozenberg. Basic notions of actor grammars. In H. Ehrig, M. Nagl,

G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 280–298, 1987.

[73] D. Jefferson and H. Sowizral. Fast concurrent simulation with the time warp mechanism. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 63–69, July 1985.

[74] H. F. Jordan. Performance: The need for an in-depth view. In *Performance Instrumentation and Visualization*, Frontier Series, chapter 13, pages 195–212. Addison-Wesley Publishing, New York, 1990.

[75] K. Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *CACM*, 25(2):126–134, Feb. 1982.

[76] M. J. Kaelbling and D. Ogle. Minimizing monitoring costs: Choosing between tracing and sampling. *23rd International Hawaii Conference on System Sciences*, Volume I:314–320, January 1990.

[77] G. M. Karam. The MLog user's guide. SCE-89-15, Dept. of Syst. and Comp. Engineering, Carleton University, Apr. 1992. 78 pages.

[78] D. W. Kelton and A. M. Law. *Simulation Modelling and Analysis*. McGraw-Hill, New York, 2nd edition, 1991.

[79] S. Khoshafian, A. Chan, A. Wong, and H. K. T. Wong. *Client/Server SQL Applications*. Morgan Kaufmann Publishers, Inc., San MAteo, CA, 1993.

[80] N. Kiesel, A. Schuerr, and B. Westfechtel. GRAS, a graph-oriented database system for (software) engineering applications. In H.-Y. Lee, T. F. Reid, and S. Jarzabek, editors, *Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering CASE 93*, pages 272–286, 1993.

[81] C. Kilpatrick and K. Schwan. ChaosMON - application-specific monitoring and display of performance information for parallel and distributed systems. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[82] B. Kirkerud. *Object-Oriented Programming with Simula*. Addison-Wesley, Reading, MA, USA, 1989.

[83] M. Korff. Application of graph grammars to rule-based systems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 505–519, 1991.

[84] T. Kunz and J. P. Black. Using automated process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6):515–527, June 1995.

[85] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[86] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience*, 20(12):1241–1258, Dec. 1990.

[87] J. R. Larus. Efficient program tracing. *I.E.E.E. Computer*, 26(5):52–61, May 1993.

[88] E. D. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1984.

[89] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[90] T. G. Lewis. Where is client/server software headed? *IEEE Computer*, pages 49–55, April 1995.

[91] D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1994.

[92] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[93] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, April 1995.

[94] Luqi and M. Ketabchi. A computer-aided prototyping system. *IEEE Software*, pages 66–72, Mar. 1988.

[95] J. Lyon. Tandem's remote data facility. *The Thirty-Fifth IEEE Computer Society*

International Conference (COMPCON), pages 562–567, 1990.

[96] S. Majumdar, C. M. Woodside, J. E. Neilson, and D. C. Petriu. Performance bounds for concurrent software with rendezvous. *Performance Evaluation*, 13(4):207–236, 1991.

[97] F. Mattern. Time and global states of distributed systems. In *Proceedings International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1988. Bonas, France, North-Holland.

[98] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[99] P. Mehra, M. Gower, and M. A. Bass. Automated modeling of message-passing programs. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems MASCOTS '94*, pages 187–192. IEEE Computer Press, 1994.

[100] S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maintaining potential causality. In *Proceedings 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 231–239, Montreal, Canada, 1991.

[101] E. Melin, A. Thiolliere, C. Hrischuk, and C. Woodside. Transaction processing simulator user's guide. Technical Report SCE-92-23, Department of Systems and Computer Engineering, Carleton University, June 1992.

[102] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, November 1995.

[103] B. Mohr. Standardization of event traces considered harmful or is an implementation of object-independent event trace monitoring and analysis systems possible? In *Proceedings CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 103–124, 1993.

[104] C. Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20(4):189–194, May 1985.

[105] M. Nagl. A tutorial and bibliographical survey on graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 70–126, 1979.

[106] M. Nagl. Set theoretic approaches to graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 41–54, 1987.

[107] M. Nagl. *Building Tightly-Integrated (Software) Development Environments: The IPSEN Approach*. Number 1170 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[108] M. Nagl and A. Schürr. A specification environment for graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 599–609, 1991.

[109] J. E. Neilson. *PARASOL Users' Manual*. School of Computer Science, Carleton University, Ottawa, Ontario, Canada, version 3.1 edition, 1996.

[110] J. E. Neilson and P. Morin. *A PARASOL Tutorial*. School of Computer Science, Carleton University, Ottawa, Ontario, Canada, version 3.1 edition, 1996.

[111] J. E. Neilson, C. M. Woodside, D. Petriu, and S. Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(9):776–782, September 1995.

[112] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings Supercomputing '92*, pages 502–511, Minneapolis, 1992.

[113] Object Management Group. Object Services Architecture. Technical report, Object Management Group, Inc., Framingham, MA, USA, Aug. 1992.

[114] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA) Revision 1.2*. Object Management Group, Framingham, Mass., revision 1.2 edition, Dec. 1993. OMG TC Document 93.12.43.

[115] ObjecTime Limited, Kanata, Ontario, Canada. *ObjecTime User's Manual*, 1997.

[116] D. M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[117] Open Software Foundation. *Introduction to OSF DCE*. Prentice-Hall, 1992.

[118] J. K. Ousterhout. *Tcl and Tk Toolkit.*. Addison-Wesley, 1994.

[119] H. G. Patil. *Efficient Program Monitoring Techniques*. PhD thesis, University of Wisconsin - Madison, 1996.

[120] S. Paynter. Structuring the semantic definitions of graphical design notations. *Software Engineering Journal*, 10(3):105–115, May 1995.

[121] D. C. Petriu, S. Majumdar, J. Lin, and C. E. Hrischuk. Analytic performance estimation of client-server systems with multi-threaded clients. In *MASCOTS 94*, pages 96–100, 1994.

[122] J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.

[123] S. Prasad. Models for mobile computing agents. *ACM Computing Surveys*, 28:53, Dec. 1996.

[124] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, February 1996.

[125] J. Rolia. *Predicting the Performance of Software Systems*. PhD thesis, University of Toronto, Jan. 1992.

[126] J. Rolia and K. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.

[127] S. J. Rosenschein. The production system: Architecture and abstraction. In D. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 525–538. Academic Press, 1978.

[128] G. Rozenberg. An introduction to the NLC way of rewriting graphs. In H. Ehrig,

M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 55–66, 1987.

[129] G. Rozenberg, B. Courcelle, H. Ehrig, G. Engels, D. Janssens, H. Kreowski, and U. Montanari. *Handbook on Graph Grammars: Foundations*, volume Vol. 1. World Scientific, 1997.

[130] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632 USA, 1991.

[131] S. R. Sarukkai, P. Mehra, and R. J. Block. Automated scalability analysis of message-passing parallel programs. *IEEE Parallel and Distributed Technology*, 3(4):21–32, Winter 1995.

[132] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proceedings 3rd International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 219–232. Springer-Verlag, Berlin, 1989.

[133] H. J. Schneider. Chomsky-Systeme f"ur partielle Ordnungen. Arbeitsbericht 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen, 1970.

[134] B. A. Schroeder. On-line monitoring: A tutorial. *Computer*, 28(6):72–78, June 1995.

[135] A. Schürr. *A Guided Tour through the PROGRES Environnement*. RWTH Aachen, D-52056 Aachen, Germany.

[136] A. Schürr. Introduction to PROGRES, an attribute graph grammar based specification language. In M. Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 151–165, 1990.

[137] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen Anwendungsbeispiele und Werkzeugunterstützung*. Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[138] A. Schürr. PROGRES: A VHL-language based on graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 641–

659, 1991.

[139] A. Schürr. PROGRES-editor: A text-oriented hybrid editor for PROgrammed Graph REwriting SyStems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, page 67, 1991.

[140] A. Schürr. Logic based structure rewriting systems. In H. J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 341–357, 1993.

[141] A. Schürr, A. J. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, 1995.

[142] B. F. Schutz. *A First Course in General Relativity*. Cambridge University Press, 1990.

[143] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[144] P. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*, Amsterdam, 1979.

[145] Scientific and Engineering Software, Inc., 4301 Westbank Drive, Austin, Texas, U.S.A. *SES/workbench User's Manual - Release 2.1*, Feb. 1992.

[146] A. C. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, pages 242–254, May 1978.

[147] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[148] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7):77–98, July 1992.

[149] C. U. Smith. Robust models for the performance evaluation of software/hardware designs. *Int. Workshop on Timed Petri Nets, Torino, Italy*, pages 172–180, July 1985.

[150] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Co., New York, NY, 1990.

[151] C. U. Smith and L. G. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on Software Engineering*, 19(7):720–741, July 1993.

[152] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, 1988.

[153] *Unified Modeling Language v1.1 Semantics*, version 1.1. Rational Software Inc., Sept. 1997. http://www.rational.com/uml.

[154] R. Sproul. *Not a Chance*. Baker Books, Grand Rapids, Michigan, 1994.

[155] S.S. Lavenberg. A perspective on queueing models of computer performance. *Performance Evaluation*, 10:53–76, 1989.

[156] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2 edition, 1986.

[157] J. A. Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, 1991.

[158] Tandem Computers Inc. Nonstop SQL optimizer: Basic concepts. In *Tandem Systems Review*, pages 14–21. Tandem Computers Inc., July 1988.

[159] Tandem Computers Inc. *NonStop SQL Physical Database Design*, part. no. 53361 edition, 1990.

[160] D. J. Taylor. A prototype debugger for Hermes. In *CASCON 92*, pages 29–42, 1992.

[161] C. Valot. Characterizing the accuracy of distributed timestamps. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 43–52, May 1993.

[162] R. van Renesse. Causal controversy at Le Mont St.-Michel. *ACM Operating System Review*, 27(2):44–53, 1993.

[163] V. Vetland. *Measurement-Based Composite Computational Work Modelling of*

Software. PhD thesis, University of Trondheim, Aug. 1993.

[164] A. Waheed, V. Melfi, and D. T. Rover. A model for instrumentation system management in concurrent systems. *Proceedings of the Twenty Eigth Hawaii International Conference on System Sciences*, January 1995.

[165] G. Winskel. An introduction to event structures. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 364–397. Springer-Verlag, Berlin, 1989.

[166] C. M. Woodside, J. E. Neilson, D. Petriu, and S. Majumdar. The rendezvous network model for performance synchronous multi-tasking distributed software. Sce-89-8, Dept. of Systems & Computer Engineering and School of Computer Science, Carleton University, Mar. 1989.

[167] C. M. Woodside, J. E. Neilson, D. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(1):20–34, Jan. 1995.

[168] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous multi-tasking distributed software. *IEEE Transactions on Software Engineering*, 1994. Accepted for publication.

[169] C. M. Woodside and C. Schramm. Complex performance measurements with NICE (Notation for Interval Combinations and Events). *Software - Practice and Experience*, 24(12):1121–144, December 1994.

[170] M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov. A wideband approach to software performance prediction and improvement. In *Workshop on Research in OO Real-Time Modeling*, January 1998.

[171] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, September 1992.

[172] D. Wybranietz and D. Haban. Monitoring and measuring distributed systems. In *Performance Instrumentation and Visualization*, Frontier Series, chapter 2, pages 27–45. Addison-Wesley Publishing, New York, 1990.

[173] J. Yan. Performance tuning with an automated instrumentation and monitoring system for multicomputers AIMS. *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, January 1994.

[174] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software Practice and Experience*, 25(4):429–461, April 1995.

[175] H. Zeller. Parallel query execution in nonstop SQL. In *Proc. 35th IEEE Computer Society Intl. Conf. COMPCON*, pages 484–487, 1990.

[176] P. Zwart. *About Time*. North-Holland Publishing Company, Oxford, 1976.

Appendix A: Evaluating Node Connections

Explanation	InOpPd	InTask	InOpExt	OutOpPd	OutTask	OutOpExt
(0) N/A. Nodes with no edges are not allowed because an operation thread must have at least two nodes: a begin node and an end node.	N	N	N	N	N	N
(1) Only an external node is allowed to have an effect without a cause. This is item (A) in Table 6.	N	N	N	N	N	Y
(2) N/A. See (1).	N	N	N	N	Y	N
(3) N/A. See (1).	N	N	N	N	Y	Y
(4) N/A. See (1).	N	N	N	Y	N	N
(5) N/A. See (1).	N	N	N	Y	N	Y
(6) N/A. See (1).	N	N	N	Y	Y	N
(7) N/A. See (1).	N	N	N	Y	Y	Y
(8) N/A. See (48).	N	N	Y	N	N	N
(9) N/A. A node cannot receive a message and send a message.	N	N	Y	N	N	Y
(10) N/A.The operation stops if there is an outgoing task edge but there is no outgoing operation edge.	N	N	Y	N	Y	N
(11) N/A. See (9).	N	N	Y	N	Y	Y

Table A1: Enumeration and Evaluation of TOEG Node Connections

Explanation	InOpPd	InTask	InOpExt	OutOpPd	OutTask	OutOpExt
(12) N/A. A next operation edge output in the same task period must have a corresponding next task output edge, otherwise the task is deadlocked.	N	N	Y	Y	N	N
(13) N/A. See (9).	N	N	Y	Y	N	Y
(14) The task is blocked (i.e., no InTask edge) and it becomes unblocked by accepting a message on InOpExt. The cases are items {E, I, M} in Table 6.	N	N	Y	Y	Y	N
(15) N/A. See (9).	N	N	Y	Y	Y	Y
(16) N/A. See (48).	N	Y	N	N	N	N
(17) N/A. A node must have a next operation edge as an input, either InOpPd or InOpExt, to proceed to the next node, otherwise the task executes without an operation.	N	Y	N	N	N	Y
(18) N/A. See (17).	N	Y	N	N	Y	N
(19) N/A. See (17).	N	Y	N	N	Y	Y
(20) N/A. See (12).	N	Y	N	Y	N	N
(21) N/A. See (17).	N	Y	N	Y	N	Y
(22) N/A. See (17).	N	Y	N	Y	Y	N
(23) N/A. See (9).	N	Y	N	Y	Y	Y
(24) N/A. See (48).	N	Y	Y	N	N	N
(25) N/A. See (9).	N	Y	Y	N	N	Y
(26) N/A. See (10).	N	Y	Y	N	Y	N

Table A1: Enumeration and Evaluation of TOEG Node Connections

Explanation	InOpPd	InTask	InOpExt	OutOpPd	OutTask	OutOpExt
(27) N/A. See (9).	N	Y	Y	N	Y	Y
(28) N/A. See (12).	N	Y	Y	Y	N	N
(29) N/A. See (9).	N	Y	Y	Y	N	Y
(30) The task is blocked (i.e., not InOpPd), becoming unblocked by accepting a message on InOpExt, continuing execution of the operation by sourcing edges on OutOpPd and OutTask. This are items {G, K, O} in Table 6.	N	Y	Y	Y	Y	N
(31) N/A. See (9).	N	Y	Y	Y	Y	Y
(32) N/A. See (48).	Y	N	N	N	N	N
(33) N/A. A node must have a task input edge if it has an operation input edge in the same task period.	Y	N	N	N	N	Y
(34) N/A. See (33).	Y	N	N	N	Y	N
(35) N/A. See (33).	Y	N	N	N	Y	Y
(36) N/A. See (12).	Y	N	N	Y	N	N
(37) N/A.See (33).	Y	N	N	Y	N	Y
(38) N/A. See (33).	Y	N	N	Y	Y	N
(39) N/A. See (33).	Y	N	N	Y	Y	Y
(40) N/A. See (48).	Y	N	Y	N	N	N
(41) N/A. See (9).	Y	N	Y	N	N	Y
(42) N/A. See (33).	Y	N	Y	N	Y	N
(43) N/A. See (9).	Y	N	Y	N	Y	Y
(44) N/A. See (12).	Y	N	Y	Y	N	N
(45) N/A. See (9).	Y	N	Y	Y	N	Y

Table A1: Enumeration and Evaluation of TOEG Node Connections

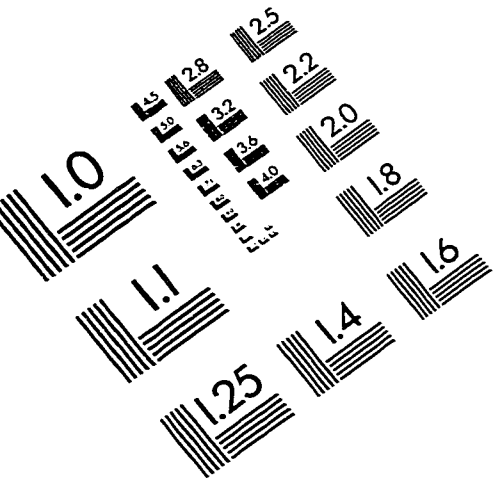
Explanation	InOpPd	InTask	InOpExt	OutOpPd	OutTask	OutOpExt
(46) N/A. See (33).	Y	N	Y	Y	Y	N
(47) N/A. See (9).	Y	N	Y	Y	Y	Y
(48) The thread end node is the only node type that is allowed to terminate the task and operation event graphs. This is item (B) in Table 6.	Y	Y	N	N	N	N
(49) Sending of the reply to an initiating task in an RPC interaction and the replying task finishes its service period. This is item (F) in Table 6.	Y	Y	N	N	N	Y
(50) N/A. See (10).	Y	Y	N	N	Y	N
(51) Initiation of an RPC interaction. This is item (D) in Table 6.	Y	Y	N	N	Y	Y
(52) N/A. See (12).	Y	Y	N	Y	N	N
(53) N/A. A node cannot have an output operation edge in the same task (OutOpPd) without a corresponding task output edge (OutTask) because an operation cannot progress in the same task without the task progressing.	Y	Y	N	Y	N	Y
(54) The distributed operation continues in the same task. This is item (C) in Table 6.	Y	Y	N	Y	Y	N
(55) An operation thread is forked. This is item (H) in Table 6.	Y	Y	N	Y	Y	Y
(56) N/A. See (48).	Y	Y	Y	N	N	N
(57) N/A. See (9).	Y	Y	Y	N	N	Y

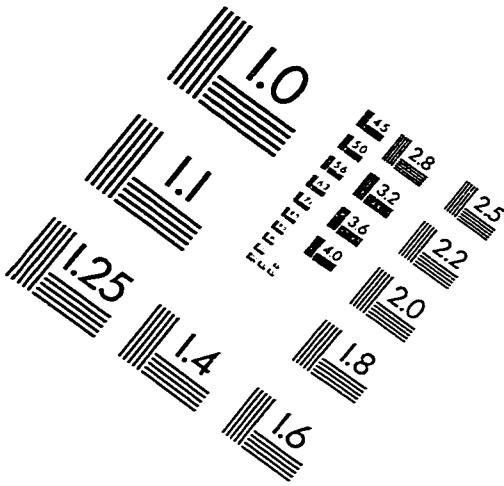
Table A1: Enumeration and Evaluation of TOEG Node Connections

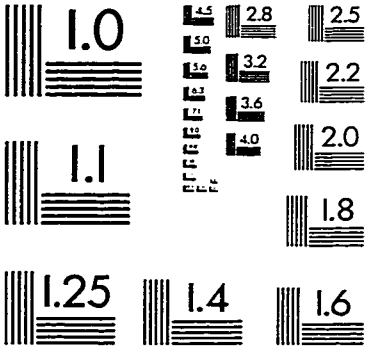
Explanation	InOpPd	InTask	InOpExt	OutOpPd	OutTask	OutOpExt
(58) N/A. See (10).	Y	Y	Y	N	Y	N
(59) N/A. See (9).	Y	Y	Y	N	Y	Y
(60) N/A. See (12).	Y	Y	Y	Y	N	N
(61) N/A. See (9).	Y	Y	Y	Y	N	Y
(62) A message reception is accepted and the accepting task was already processing a service request (InTask). This is characterized by items {L, J, N} in Table 6.	Y	Y	Y	Y	Y	N
(63) N/A. See (9).	Y	Y	Y	Y	Y	Y

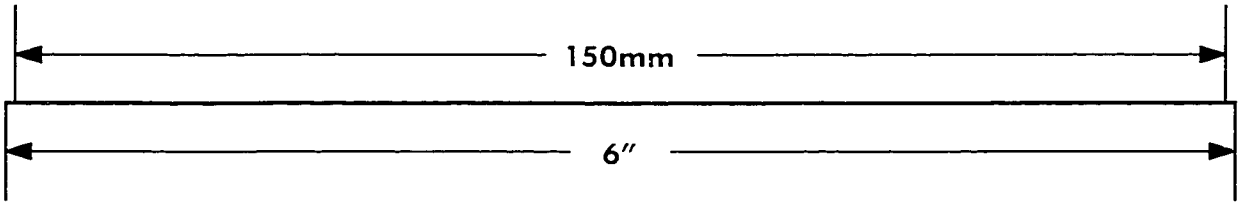
Table A1: Enumeration and Evaluation of TOEG Node Connections

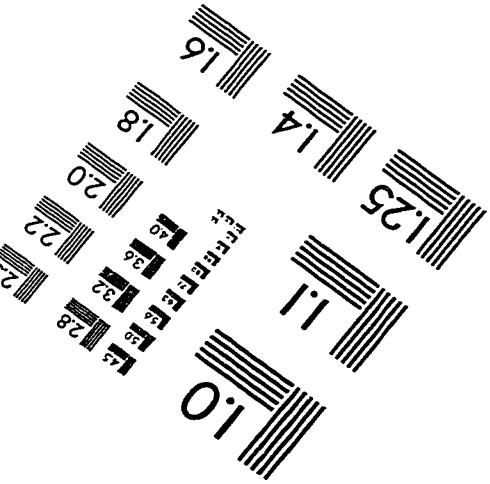
IMAGE EVALUATION
TEST TARGET (QA-3)











APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989