

Trace-Based Load Characterization for Generating Software Performance Models

C. E. Hrischuk^{*}, C. M. Woodside^{**}, J. A. Rolia^{**}, R. Iversen^{***}

^{*}Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada.
curtis@ee.ualberta.ca

^{**}Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
{cmw, jar}@sce.carleton.ca.

^{***}ObjecTime Ltd., Ottawa, Canada.
rod@ObjecTime.com.

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. An ANGIOTRACE™ was developed to gather the necessary information from an executable design and develop a model in an automated fashion. It applies to distributed and concurrent software with synchronous (send-reply or RPC) communications, developing a layered queueing network model. The Trace-based Load Characterization (TLC) technique presented here extends the ANGIOTRACE™ approach to handle software with both synchronous and asynchronous interactions. TLC also detects interactions which are effectively synchronous or partly-synchronous (forwarding) but are built up from asynchronous messages. These patterns occur in telephony software and in other systems. The TLC technique can be applied throughout the software life-cycle, even after deployment.

Keywords: software performance, analytic performance models, layered queueing, software traces, performance engineering, workload characterization, performance prototyping

Copyright 1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in the works must be obtained from the IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

1.0 Introduction

Software performance models of an early design can reduce the risk of performance-related failures. Performance models provide performance predictions under varying environmental conditions or design alternatives and these predictions can be used to detect problems. To construct a model, a software description (e.g. design documents or source code) must be analyzed and translated into the model format which might be a simulation model [23], queueing network model [26], or a state-based model like a Petri-Net [45]. The effort of model development makes it unattractive, so performance is usually addressed only in the final product. Smith has termed this the “fix-it-later” approach and documented the seriousness of the problems it creates [46]. Our previous work ([20]) demonstrated an approach to ease this burden by automatically generating descriptions of a distributed operation’s workload and corresponding performance model(s), provided the software components communicated solely by remote procedure call. This paper describes a more formal technique which can develop models of most distributed or parallel programs. This technique is called Trace-based Load Characterization (TLC).

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 components. 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. To avoid confusion it is emphasized that a distributed operation is not atomic but it is extensive in time with concurrent execution between the tasks. Tasks communicate solely by messages. Communication is reliable, point-to-point, dynamically established at execution, having finite but unpredictable delay, and FIFO order of message delivery is not assumed. The communication protocol may be blocking (i.e., Remote Procedure Call) or non-blocking (i.e., asynchronous). Concurrent and parallel systems are a sub-set of this class of system. 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.

A system that executes distributed operations differs somewhat from the classical parallel or concurrent system model, such as described in [43]. 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 [35], CORBA[33], Java [27, 12], and mobile agents [37] are technologies which are being used to build these types of distributed operations.

Software performance models of distributed operations describe tasks and their interactions because they affect queueing delays, parallelism, and resource contention. For example, a heavily

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.

used task can queue arriving requests and can even become a bottleneck [32]. 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 performance of these systems [48, 38]. The LQN model extends queueing network models to include contention effects for software resources and for hardware devices. TLC incorporates enough detail to construct an LQN model.

A prerequisite for TLC is some executable form of the design. It can be a very abstract executable CASE tool model or a code prototype. (With a code prototype one can obtain performance measurements, but a model is still necessary for extrapolating to scaled up deployments of the software.)

Model construction in TLC has three steps. It begins by recording a special trace of execution that we call an *ANGIOTRACE*², introduced in [20]. The trace is analyzed to produce an *LQN sub-model* that characterizes the involved tasks, their individual activities, and their interactions with each other. Finally, a performance model is developed by merging several LQN sub-models and adding configuration information.

An *ANGIOTRACE* characterizes a distributed operation independent of other simultaneous distributed operations. The name *ANGIOTRACE* 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 *ANGIOTRACE* assigns a different dye to each distributed operation so that they can be distinguished. The *ANGIOTRACE* event format records information with each user-defined and communication event that captures their cause and effect relationships.

An *ANGIOTRACE* can be extracted from many sources at various steps of the development, such as annotated specifications (e.g. Use-Cases [40] or Message Sequence Charts [4]), functional prototypes, executable designs, detailed simulations, or the production system. It has been implemented in several environments: a functional prototyping environment (MLOG [22]), a commercial prototyping environment (ObjecTime [34]), a distributed software system simulator called Parasol [31], coarse-grained UNIX tasks [21], and in the DCE RPC environment [35] using data collected by the POET debugger [47]. We concentrate here on traces produced by a design prototype environment, namely ObjecTime.

There are several benefits to using tracing as opposed to a “source code examination” approach for constructing workload and performance models. Traces incorporate the dynamic details of a design that are difficult to determine from source code or documentation, 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.

There are other performance analysis techniques that use traces but which are very different from TLC. Where TLC is concerned with building predictive models of distributed operations

2. *Angiotrace* is a trademark of Angiograms for Software Analysis Inc. who can be contacted at angio@istar.ca. *ANGIOTRACE* appears in capital letters to signify it is a trademark.

throughout the software life cycle, the other techniques use measurements to evaluate and tune the performance of a final product. Examples include: Paragraph [15], SIMPLE [7], ChaosMON [24], AIMS [49], PABLO [1], and W³ [19]. Many of them are primarily for parallel programs.

The Modeling Kernel [30, 41, 50] and the Parameter based Performance Prediction Tool (P³T) [9] use traces to determine resource demands but they pay little attention to software architecture or software resource constraints. They use the source code, compiler information, hardware platform description, and trace files to produce simulation models for tuning communication or caching performance. The trace files are analyzed to find iteration counts, branch frequencies, message sizes, and basic-block execution times.

Our previous trace-based model builder [20] set off in a new direction, detecting the software architecture (tasks and interactions) and incorporating it, along with the software resource constraints (tasks with message queueing) in a model. The novelty was in the characterization of the concurrency in the distributed operation. However it was limited to systems which solely use a synchronous communication protocol, and it assumed the availability of a global system monitor to record events in the order of occurrence.

Now TLC relaxes both of the restrictions of [20], greatly enlarging the range of application interaction semantics that can be detected and modelled. It analyzes asynchronous interactions and also three kinds of blocking interaction which may be constructed from asynchronous interactions; they are flavors of RPC. It is based on a causal logical clock so that it no longer depends on a global monitor to order the events [43, 10, 29]. TLC now includes every message and its subsequent processing so that all workload elements are incorporated.

A demonstration of the expanded semantics is given below in an example that extracts performance model parameters from a telephony software prototype without *a priori* knowledge of the software components or interactions.

The next section is an overview of performance analysis with TLC. Section 3.0 describes the LQN sub-model. Section 4.0 provides information about an ANGIOTRACE, as well as an alternate trace representation which is used during the trace processing step. Section 5.0 describes how an LQN sub-model is automatically constructed, with Section 6.0 describing how to complete an LQN model using the constructed sub-models. A case study is given in Section 7.0. During our discussion we assume familiarity with basic performance modeling concepts, as described by Lazowska, *et. al.* [26] or Smith [46]. For the rest of the document, the term *trace* refers to an ANGIOTRACE and *event* refers to an ANGIOTRACE event.

2.0 Coupling TLC with Performance Analysis

The steps in applying TLC in a performance analysis context are shown in Figure 1. The first step (#1) is to select *scenarios* which are important for performance modelling. As described in [46], the analyst selects only those *scenarios* judged to be most significant, such as heavily used requests or requests of special importance. A scenario is defined by a *name* and *options*. The scenario name differentiates major types of scenarios. Scenario options reflect variations in the

request data or data state. Each scenario name and set of options are associated with a *distributed operation's trace name* to identify the scenario's resulting workload description, the *LQN sub-model*. Instrumentation is added to identify where the execution of each distributed operation begins, its name, as well as where it ends.

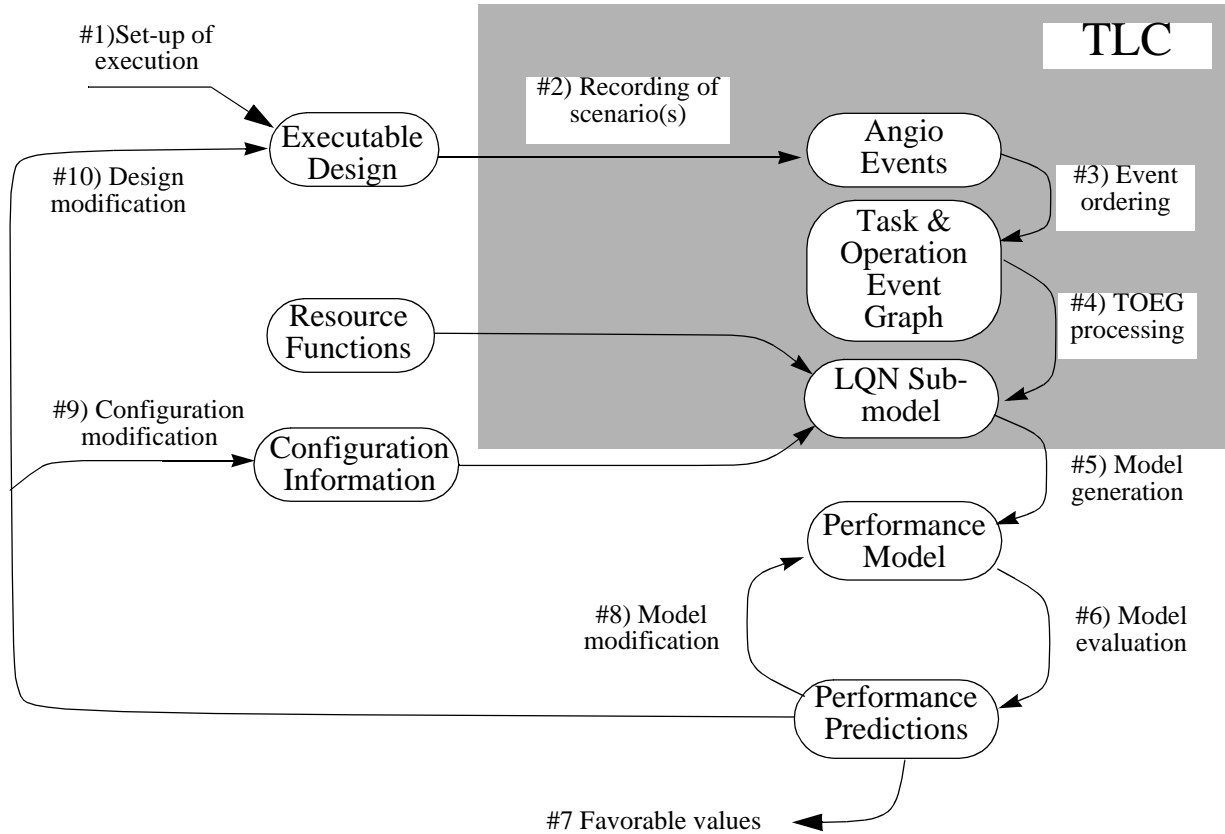


Figure 1: Integrating TLC and Performance Analysis

In step #2, the `ANGIOTRACE` events are recorded. For analysis purposes the events of a trace are reordered into an intermediate format (#3) that is then processed into an LQN sub-model (#4). Steps #3 and #4 are the focus of this paper and are considered in detail in the next sections.

The user completes the LQN construction in step #5 by combining several LQN sub-models with the target system's configuration information, including: workload intensity (e.g. invocations per second by scenario), resource definitions (e.g. scheduling policy, service rate), task information (e.g. internal concurrency level), and resource assignments (e.g. task to CPU assignment). The configuration information can be varied to conduct a “what-if” analysis.

The performance model also requires resource demands for activities and they are introduced using resource functions. An event can record an *activity identifier* which is assigned representative resource demands by a *resource function*. Resource functions may be developed from previous similar projects, rules of thumb, prototypes, or proof-of-concept exercises. Initial estimates for a

resource function can be made by expert judgement [46] and improved as the development progresses.

The remaining steps of Figure 1 illustrate how the constructed performance model can be exploited. Having evaluated the performance model (#6), the performance analysis is complete if the performance predictions meet the performance objectives (#7). Otherwise improvements to the design need to be made. One approach is to manipulate the performance model to compare alternatives without having to alter the existing design (#8). Another option is to change the configuration (#9), such as reallocating tasks to processors, adjusting a task's priority, or providing faster hardware. If necessary, adjustments to the actual design are made (#10) and traces are regenerated. Once an alternative has been selected and the design updated the model construction process is revisited.

We will now explain the LQN sub-model construction steps (#3 and #4) of TLC in detail.

3.0 Layered Queueing Network Sub-Models

TLC builds a layered queueing network (LQN) model, which is a super-set of queueing network models that is adapted for modelling the interactions of software tasks. An LQN includes an entity for each task, as well as for each processor and other devices. Service requests to a task are messages which are queued and served in order and the queueing delays are to be determined. Load originates with other tasks which loop forever and initiate requests. If tasks have no resource limits then an LQN gives the same predictions as a queueing network, but also provides information about the workload and delays broken down by the tasks and their entries. An LQN *entry* is the part of a task that gives a certain service of the task (analogous to a method which executes a service within an object). It translates to a queueing network model *class*.

In TLC each trace determines a *submodel*, describing that part of the system exercised by the trace. A complete LQN model is a union of the submodels determined from different traces.

To create a submodel, the analysis develops a labelled directed graph, like a software call graph, whose nodes are entries and arcs are *service request interactions*. The entry labels describe the services (identifiers for the entry and task, type of interaction, list of activities) and the arc labels define the number of requests made to other entries during a service. A service is defined to begin with acceptance of a request message from a requesting entry and to end before the next request message is accepted. Recognizing the beginning of a service, the end of a service, and its interaction type is a key problem in constructing the LQN sub-model. These issues are addressed in section 5.0.

The labeling of an entry node (E_n) is described as $E_n = \{\text{interactiontype, taskidentifier, listofservicerequests, listofactivities}\}$. The list of service requests from entry E_n to entry E_{ni} is $V_n = \{E_{n1}, E_{n2}, \dots, E_{ni}\}$. The list of activities to identify the resource functions of entry E_n is $A_n = \{a_1, \dots\}$.

The interaction types, with the corresponding interaction type label, are:

- *RPC style or rendezvous* (type **R**): the initiating task waits for a reply to its service request and it resumes execution once it receives a reply message from the task that provided the service.
- *Asynchronous* (type **A**): an initiating task does not wait for a reply from the responding task. This interaction type was not considered in [20].
- *Forwarding* (type **F**): a *forwarding interaction* 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 scenario occurs when a task acts as an administrator manager by dispatching service requests to a pool of worker tasks [11] or as a form of rate control for a task pipeline.

Besides entries that describe service execution, an LQN sub-model has an artificial *customer task* with a single entry that is a surrogate for the source of the initial stimulus of a scenario. We define it as the interaction type **Cu**.

It may be that two or more entries in the same task in an LQN sub-model have the same (or nearly the same) demands and interactions. Similar entries can be merged to simplify the model by averaging their total resource demands and introducing average request counts for the outgoing arcs.

The LQN sub-model splits RPC-based interactions into two phases of execution for the responding task. The first phase consists of the execution between the responding task's request acceptance and subsequent reply (or forwarding) of the request. Any work that occurs after the reply (or forwarding) but before the acceptance of the next request is labelled as a *second phase* of execution. The second phase accounts for the resource contention between the initiating task and the continuing responding task.

LQN models can be solved analytically using approximate Mean-Value Analysis based techniques [38, 48]. These techniques do not exploit sequence information about the demands or service requests of an entry so the LQN sub-model only represents the mean values of demands.

Using our notation we describe an example LQN sub-model w that consists of four entries as: $W = \{E_0, E_1, E_2, E_3\}$ where: $E_0 = \{Cu, a, \{1\}, \{\}\}$, $E_1 = \{R, A, \{2\}, \{2, 3, 4\}\}$, $E_2 = \{R, B, \{3\}, \{1, 3\}\}$, and $E_3 = \{A, C, \{\}, \{2\}\}$. Figure 2 illustrates the sub-model. Parallelograms represent tasks, rectangles are entries, and arcs between entries represent messages or requests to a responding task. Each entry is labelled with its type (**R**, **A**, **F**, or **Cu**) and an entry identifier.

4.0 Event Notation and Ordering

The ANGIOTRACE records the causal flow of a distributed operation's execution by recording a timestamp with each event (step #2 of Figure 1) and then ordering the events by the timestamps

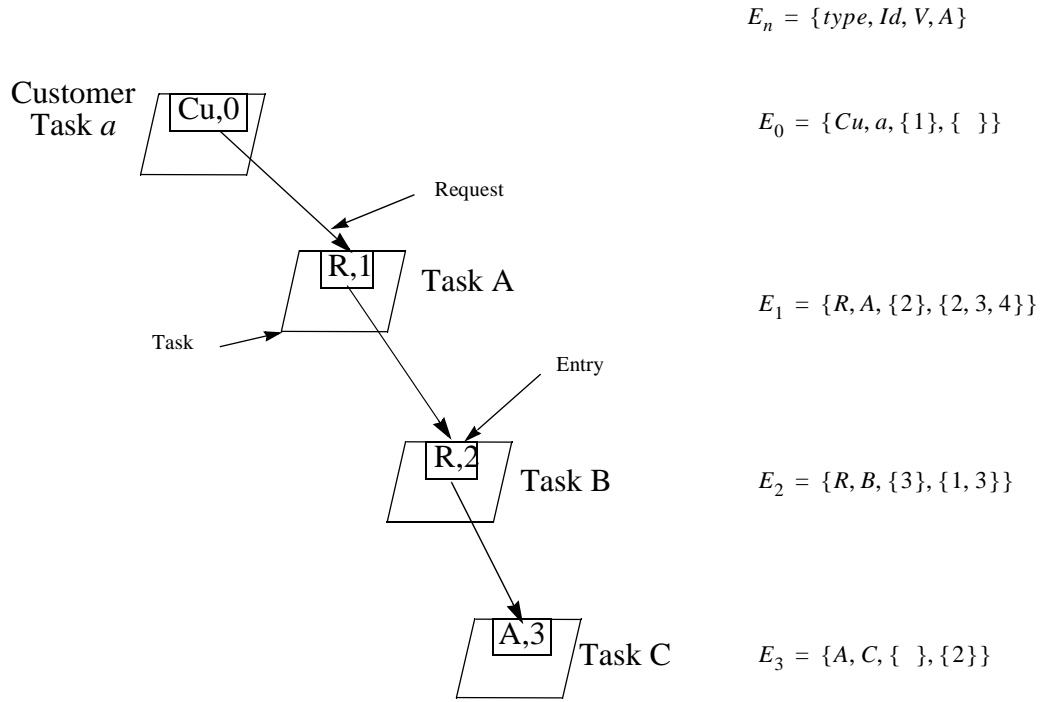


Figure 2: LQN Model Example

(step #3). The event ordering forms a graph which we call a TOEG (*Task and Operation Event Graph*). The TOEG is a node-labelled, directed, binary, acyclic graph [5] whose structure is based on the causal relationships between events in the same distributed operation. Each trace event produces a labelled node in the TOEG. The arcs in the TOEG, which represent cause and effect relationships, are deduced from the event timestamp information.

An event timestamp has six parameters: two parameters tie it to a task's execution and four parameters describe the execution of the distributed operation. The task information is a *task identifier* and a *task node counter* that orders the nodes recorded by a task. The distributed operation information is: a scenario name, a scenario thread identifier, a thread node counter, and a type label for the node. A *scenario name* associates a node with its scenario. A *scenario thread* is a causal sequence of nodes (a linear sub-graph) with the same *scenario thread identifier*. A scenario thread characterizes a single flow of execution in a distributed operation, even across task boundaries. Scenario thread is shortened to thread in the sequel. The *thread node counter* orders the nodes within a thread. The counters begin at one.

The type label of a node takes one of four values to identify the begin, activity, end, and fork nodes. Each thread starts with a *begin node* (Be) and finishes with an *end node* (Ed). An *end node* is used to distinguish between successful termination and a deadlocked or incomplete thread.

Activity nodes (Ac) identify a resource function for an action taken by the program (with optional resource demand information).

Threads in a TOEG are connected together by *fork nodes* (Fk). A fork node represents the introduction of concurrency within a scenario, typically by sending a message to start a parallel thread. A fork node is the cause of two subsequent nodes, where one is placed in the same thread and the other is taken as the beginning of a new thread. To identify the new thread the fork label also records the scenario name and the new thread's identifier.

A node label e has the form:

$$e = \left| \frac{\text{distributed operation information}}{\text{task information}} \right| = \left| \frac{j, k, m, l}{i, v} \right|, \text{ where:} \quad \text{Eq 4.1}$$

j is the scenario name;

k is the scenario thread identifier;

m is the scenario thread's node counter;

$l = \{Be, Ac, (Fk, j, k'), Ed\}$ is the type label. The fork label also includes the scenario name (j) and thread identifier of the forked thread (k');

i is the task identifier;

v is the task node counter.

A thread is identified with its scenario name and thread identifier, namely $|j, k|$.

In this paper our illustration of a TOEG will follow several conventions. Time proceeds from left to right. The consecutive nodes of a task follow a thin line and are at the same vertical level. The consecutive nodes of a thread follow a thick, shaded line. A thread that starts from a fork node is the child thread of that node. The node type information is illustrated by the different node shadings as well as being part of the node labeling.

The TOEG in Figure 3 begins from the external input node labelled $\left| \frac{a, 1, 1, Be}{A, 1} \right|$. There are two threads within this TOEG. The thread $|a, 1|$ crosses task boundaries and ends at node $\left| \frac{a, 1, 8, Ed}{A, 5} \right|$. The thread $|a, 2|$ is forked by the fork node $\left| \frac{a, 1, 5, (Fk, a, 2)}{B, 2} \right|$ which includes the child thread's scenario name and thread identifier.

5.0 Deducing Task Services and Interactions

The core of the automated workload characterization and model development is the algorithm for reducing a TOEG to an LQN submodel, by transforming one graph to another. It uses a rule-

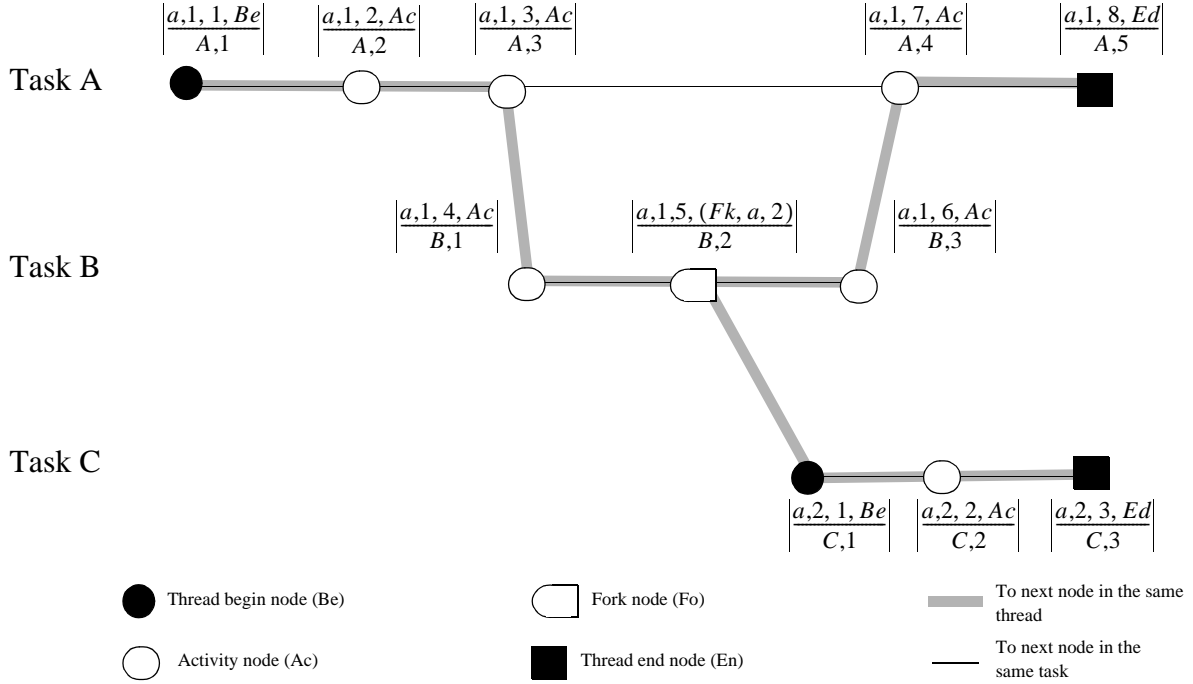


Figure 3: Example TOEG with Identifier a

based graph analysis approach [14, 25, 18, 39]. The rules have an antecedent part and a consequent part. The antecedent component of a rule is a graph fragment that completely matches an interaction pattern in the TOEG which we call an *interaction template*. The consequent component is a sequence of *modeling operations* to develop parts of the LQN sub-model. The order of matching of templates is governed by a *control algorithm*.

This section describes the interaction templates and the control algorithm, which were implemented in first-order predicate logic using Prolog [8], in a similar way to other authors [36, 6, 17, 42]. The control algorithm is discussed following the presentation of the interaction templates.

5.1 Interaction Templates and their Modeling Operations

An *interaction template* identifies a pattern of communication that can exist in the target design. It includes the start of a service period, the end of a service period, and the type of task interaction. A template match identifies the initiating task(s), the responding task(s), and the possible occurrence of a second phase of execution. When a match occurs modelling operations are executed to create the LQN sub-model. Each of these aspects are described in this section. This is followed by template specifications for the interaction types defined in section 3.0.

An interaction template is described as a group of labelled nodes that are connected with arcs deduced from the node labeling. The node labelling follows the notation of Eq 4.1 with the generalization that a field label may be a symbolic variable which is assigned a value during a template match. Variables in the same template with the same subscripts have the same value.

Each interaction template has a root node that is called its *glue node*. The glue node is the start of an interaction. The template matching process proceeds with the control algorithm supplying a TOEG node (called a *trigger node*) that may be the start of an interaction. The variables in the template get their values from the glue node's match with the trigger node. A template match succeeds when all interaction template nodes have corresponding nodes in the TOEG with the same labelling, otherwise it fails.

A template may contain a repeating sequence of nodes. This is supported in our notation by allowing variables to have subscripts which are themselves expressions. The forwarding interaction template uses this notation.

When a template match occurs, specific modeling operations are executed to create a corresponding fragment of the performance model. There are five modeling operations:

- *newEntry(entry, type, taskId)* constructs a new entry (*entry*) for a responding task with the appropriate task (*taskId*) and interaction type (*type*) labels.
- *addRequest(fromEntry, toEntry)* adds a service request label to the initiating task's entry (*fromEntry*) indicating that a request was made to the responding task's entry (*toEntry*).
- *addActivity(entry, activity)* adds the activity information from event (*activity*) to an entry (*entry*). We identify the event with the activity information for further processing by resource functions. Only the task node counter needs to be recorded since the task identity can be deduced from the entry information.
- *phase2(threadId, taskId, taskIndex)* labels nodes with the thread identifier *threadId*, task identifier *taskId*, and task counter values greater than *taskIndex* as being part of phase two execution.
- *addTrigger(node, entry)* appends *node* to the *trigger node list* maintained by the control algorithm. The entry identifier (*entry*) is stored, so that when *node* is later processed the appropriate activities and requests can be assigned to *entry*.

The interaction templates of Table 1 are described next. By convention all nodes labelled e_1 are glue nodes. The graph conventions of Figure 3 are followed in the diagrams. The modeling operations refer to the trigger node's entry as E .

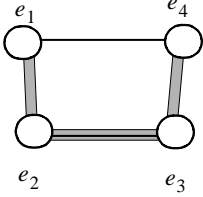
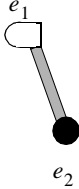
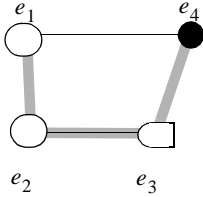
Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(i) Basic RPC $\{e_1, \dots, e_4\}$	#1	 $e_1 = \left \frac{j_1, k_1, m_1, Ac}{i_1, v_1} \right $ $e_2 = \left \frac{j_1, k_1, m_1 + 1, Ac}{i_2, v_2} \right $ $e_3 = \left \frac{j_1, k_1, m_2, Ac}{i_2, v_3} \right $ $e_4 = \left \frac{j_1, k_1, m_2 + 1, Ac}{i_1, v_1 + 1} \right $	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $addActivity(E, e_1)$ $addActivity(E, e_4)$ $addActivity(E_n, e_2)$ $addActivity(E_n, e_3)$
(ii) Asynchronous $\{e_1, e_2\}$	#6	 $e_1 = \left \frac{j_1, k_1, m_1, (Fk j_1, k_2)}{i_1, v_1} \right $ $e_2 = \left \frac{j_1, k_2, 1, Be}{i_2, v_2} \right $	$newEntry(E_n, A, i_2)$ $addRequest(E, E_n)$ $addTrigger\left(\left \frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1} \right , E_n\right)$
(iii) RPC with phase two $\{e_1, \dots, e_4\}$	#2	 $e_1 = \left \frac{j_1, k_1, m_1, Ac}{i_1, v_1} \right $ $e_2 = \left \frac{j_1, k_1, m_1 + 1, Ac}{i_2, v_2} \right $ $e_3 = \left \frac{j_1, k_1, m_2, (Fk j_1, k_2)}{i_2, v_3} \right $ $e_4 = \left \frac{j_1, k_2, 1, Be}{i_1, v_1 + 1} \right $	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $addActivity(E, e_1)$ $addActivity(E_n, e_2)$ $phase2(k_1, i_2, v_3)$ $addTrigger\left(\left \frac{j_1, k_2, 2, l_1}{i_1, v_1 + 2} \right , E\right)$

Table 1: Interaction Templates and Modeling Operations

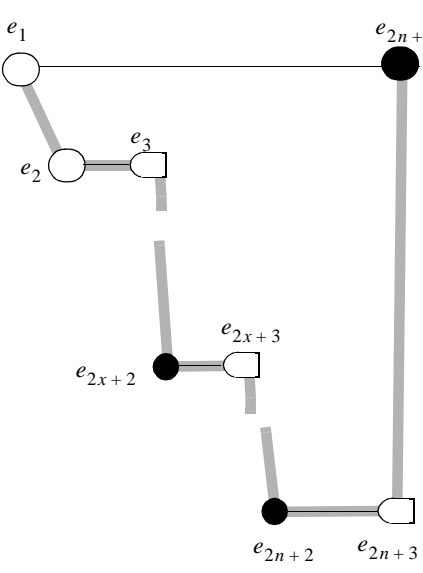
Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(iv) Forwarding interaction with n levels of forwarding $\{e_1, \dots, e_{2n+4}\}$	#5	 $e_1 = \left \frac{j_1, k_1, m_1, Ac}{i_1, v_1} \right \quad e_2 = \left \frac{j_1, k_1, m_1 + 1, Ac}{i_2, v_2} \right $ $e_3 = \left \frac{j_1, k_1, m_2, (Fk, j_1, k_2)}{i_2, v_3} \right $ <p>for $x := 1$ to n do begin</p> $e_{2x+2} = \left \frac{j_1, k_{x+1}, 1, Be}{i_{x+2}, v_{2x+2}} \right $ $e_{2x+3} = \left \frac{j_1, k_{x+1}, m_{x+2}, (Fk, j_1, k_{x+2})}{i_{x+2}, v_{2x+3}} \right $ <p>end</p> $e_{2n+4} = \left \frac{j_1, k_{n+2}, 1, Be}{i_1, v_1 + 1} \right $	<p>$newEntry(E_n, R, i_2)$</p> <p>$addRequest(E, E_n)$</p> <p>$addActivity(E, e_1)$</p> <p>$addActivity(E_n, e_2)$</p> <p>$phase2(k_1, i_2, v_3)$</p> <p>for $x:=1$ to n</p> <p>do begin</p> <p>$newEntry(E_{n+x}, F, i_{x+1})$</p> <p>$addRequest(E_{n+x-1}, E_{n+x})$</p> <p>$addTrigger\left(\left \frac{j_1, k_{x+1}, 2, l_x}{i_{x+2}, v_{2x+2} + 1} \right , E_{n+x}\right)$</p> <p>$phase2(k_{x+1}, i_{x+2}, v_{2x+3})$</p> <p>end</p>

Table 1: Interaction Templates and Modeling Operations

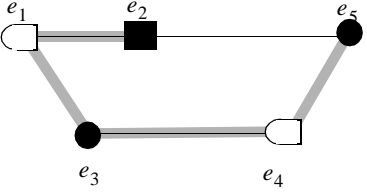
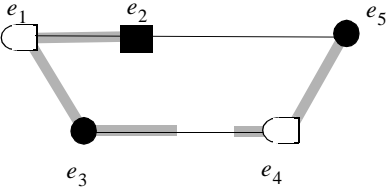
Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(v) Single RPC using asynchronous primitives $\{e_1, \dots, e_5\}$	#3	 $e_1 = \left\lfloor \frac{j_1, k_1, m_1, (Fk, j_1, k_2)}{i_1, v_1} \right\rfloor \quad e_2 = \left\lfloor \frac{j_1, k_1, m_1 + 1, Ed}{i_1, v_1 + 1} \right\rfloor$ $e_3 = \left\lfloor \frac{j_1, k_2, 1, Be}{i_2, v_2} \right\rfloor \quad e_4 = \left\lfloor \frac{j_1, k_2, m_2, (Fk, j_1, k_4)}{i_2, v_3} \right\rfloor$ $e_5 = \left\lfloor \frac{j_1, k_3, 1, Be}{i_1, v_1 + 2} \right\rfloor$	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $phase2(k_2, i_2, v_3)$ $addTrigger\left(\left\lfloor \frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1} \right\rfloor, E_n\right)$ $addTrigger\left(\left\lfloor \frac{j_1, k_3, 2, l_2}{i_1, v_1 + 3} \right\rfloor, E\right)$
(vi) Nested RPC using asynchronous primitives $\{e_1, \dots, e_5\}$	#4	 $e_1 = \left\lfloor \frac{j_1, k_1, m_1, (Fk, j_1, k_2)}{i_1, v_1} \right\rfloor \quad e_2 = \left\lfloor \frac{j_1, k_1, m_1 + 1, Ed}{i_1, v_1 + 1} \right\rfloor$ $e_3 = \left\lfloor \frac{j_1, k_2, 1, Be}{i_2, v_2} \right\rfloor$ $e_4 = \left\lfloor \frac{j_1, k_3, m_2, (Fk, j_1, k_4)}{i_2, v_3} \right\rfloor$ $e_5 = \left\lfloor \frac{j_1, k_4, 1, Be}{i_1, v_1 + 2} \right\rfloor$	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $phase2(k_3, i_2, v_3)$ $addTrigger\left(\left\lfloor \frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1} \right\rfloor, E_n\right)$ $addTrigger\left(\left\lfloor \frac{j_1, k_3, 1, l_2}{i_2, v_4} \right\rfloor, E_n\right)$ $addTrigger\left(\left\lfloor \frac{j_1, k_4, 2, l_3}{i_1, v_1 + 3} \right\rfloor, E\right)$

Table 1: Interaction Templates and Modeling Operations




Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(vii) Activity node $\{e_1\}$	#7	<div>e_1 </div> $e_1 = \left \frac{j_1, k_1, m_1, Ac}{i_1, v_1} \right $	$addActivity(E, e_1)$
(viii) End node $\{e_1\}$	#8	<div>e_1 </div> $e_1 = \left \frac{j_1, k_1, m_1, Ed}{i_1, v_1} \right $	-
(ix) Begin node $\{e_1\}$	#8	<div>e_1 </div> $e_1 = \left \frac{j_1, k_1, 1, Be}{i_1, v_1} \right $	-

Table 1: Interaction Templates and Modeling Operations

Basic RPC Interaction

In the *basic RPC* interaction template (template (i) in Table 1), the initiating task (i_1) begins the RPC and blocks at node e_1 , resuming execution when it receives the reply message at node e_4 . The responding task (i_2) receives the request at node e_2 and replies to the initiating task at node e_3 . The modeling operator $newEntry(E_n, R, i_2)$ constructs a new entry for the responding task with the RPC interaction type. A service request is assigned to the initiating task's entry by $addRequest(E, E_n)$. The activity nodes are assigned to the appropriate entries using $addActivity()$. An example match of the basic RPC occurs in Figure 3, with the nodes $e_1 = \left| \frac{a, 1, 3, Ac}{A, 3} \right|$, $e_2 = \left| \frac{a, 1, 4, Ac}{B, 1} \right|$, $e_3 = \left| \frac{a, 1, 6, Ac}{B, 3} \right|$, and $e_4 = \left| \frac{a, 1, 7, Ac}{A, 4} \right|$.

Asynchronous Interaction

In the *asynchronous* interaction template (template (ii) in Table 1) the node e_1 forks the child thread k_2 that begins with the node e_2 . A new entry is constructed for the receiving task, with an asynchronous interaction type (e.g., $newEntry(E_n, A, i_2)$). To ensure the nodes of the forked thread are processed, a trigger node is added to a *trigger node list* by the modeling operator $addTrigger\left(\left| \frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1} \right|, E_n\right)$. In Figure 3, an asynchronous interaction is initiated by node $e_1 = \left| \frac{a, 1, 5, (Fk, a, 2)}{B, 2} \right|$.

RPC with Phase Two Interaction

The *RPC with phase two* template (template (iii)) differs from a basic RPC by the responding task (i_2) continuing execution after replying to the initiating task. For this reason, the reply is sent back as a separate asynchronous interaction. The responding task's new entry is labelled as RPC and the nodes of the responding task that occur after e_3 are labeled as phase two nodes (e.g., $phase2(k_1, i_2, v_3)$).

Forwarding RPC Interaction

The *forwarding RPC* interaction was introduced in section 3.0. Its simplest form is given in Figure 4, where: the initiating task (Task A) sends the RPC request and blocks, the first responding

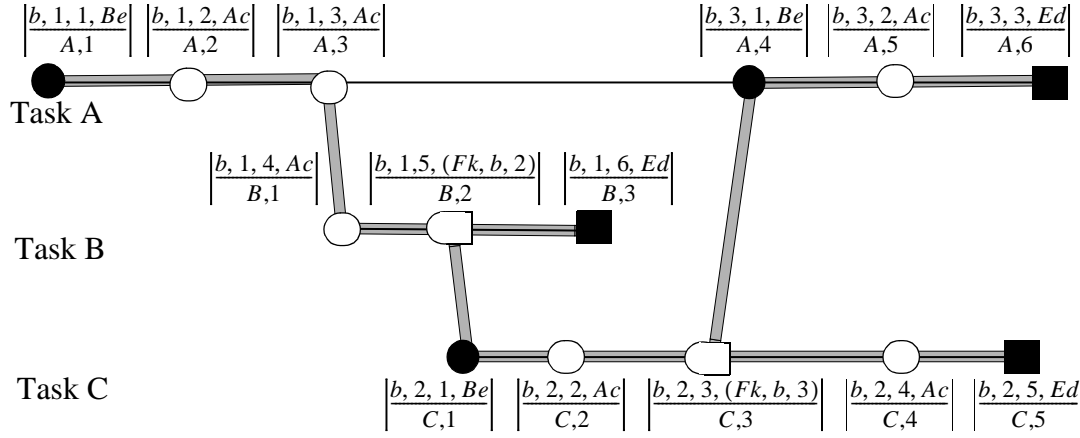


Figure 4: Simple Forwarding TOEG

task (Task B) processes the request, and passes it to another responding task (Task C) which replies to the initiating task. The example of Figure 4 has a forwarding depth of one. Nested forwarding interactions can occur.

In the forwarding template (template (iv)) the nesting depth value (n) and the corresponding nodes are determined during the matching. Our notation illustrates this by this by assigning to nodes symbolic subscripts that are incremented with each nesting level that is matched. The new entries of the responding tasks $\{i_3, \dots, i_{n+2}\}$ are labelled as *forwarding*. The nodes that occur after the nodes $\{e_3, \dots, e_{2x+3}, \dots, e_{2n+3}\}$ for $1 \leq x \leq n$ are labeled as phase two nodes. For example, in Figure 4 the nodes $\left| \frac{b, 1, 6, Ed}{B, 3} \right|$, $\left| \frac{b, 2, 4, Ac}{C, 4} \right|$ and $\left| \frac{b, 2, 5, Ed}{C, 5} \right|$ are labeled as phase two execution. The nested forwarding interactions are a feature of the case study.

RPC using Asynchronous Messages

There are two templates for identifying an RPC interaction constructed from asynchronous messages. The two templates differentiate the cases of non-nested (template (v) of Table 1) and nested RPC interactions (template (vi)) that are constructed exclusively from asynchronous messages. In both cases, after having sent the service request the initiating task immediately blocks to wait for a reply (e_1 and e_2). The distinguishing characteristic between these two templates is whether the request acceptance node (e_3) and reply to the initiating task node (e_4) have the same thread identifier; if so, then it is a non-nested RPC.

Remaining Templates

The last three templates of Table 1 complete the node processing of the TOEG. The activity nodes which are not part of any interaction are assigned to the appropriate entry (template (vii)). When an end node is detected there is no other processing needed (template (viii)). Template (ix) is used during initialization to match the node that begins a TOEG and it is included for completeness.

5.2 Template Matching Control Algorithm

A control algorithm manages the matching of interaction templates with part of the TOEG. The control algorithm is outlined in Figure 5. It has an initialization component and an analysis component.

The control algorithm is initialized by putting the first node of the TOEG on the trigger node list ($\left| \frac{a,1,1,Be}{A,1} \right|$), creating an artificial customer task and entry (E_{Cu}), creating a task and entry for the first event (E_n), and adding a synchronous request from the customer task to the first task. From the TOEG in Figure 3, the modeling operations for initialization are: $newEntry(0, Cu, a)$, $newEntry(1, R, A)$, $addRequest(0, 1)$.

The analysis component of the control algorithm consists of two do-while loops. The outermost loop (line 2 to line 10) ensures that all of the nodes in the trigger node list are processed. The inner loop (line 4 to line 9) sequentially processes all the events in a thread that has not been part of a previous template, in turn making each node a trigger node and beginning the template matching. The control algorithm ensures that the earliest available node that can initiate an interaction (the trigger node in a given thread) is the earliest node assigned to start an interaction (the glue node in a template). Once a thread has been fully analyzed (line 9) the first node on the trigger node list is made the new trigger node and the analysis continues with the new trigger node's thread.

Once the initialization component of the control algorithm is complete for Figure 3, the order of template matching is:

- 1) With glue node $\left\{ e_1 = \left| \frac{a,1,1,Be}{A,1} \right| \right\}$ the begin node template (template (ix)) is matched. There are no modeling operations to perform.
- 2) With glue node $\left\{ e_1 = \left| \frac{a,1,2,Ac}{A,2} \right| \right\}$ the activity template (template (vii)) is matched. The modeling operation is $addActivity(1, e_1)$.

- 1 Initialize by inserting the first node of the TOEG into the trigger node list, creating the customer task, its customer entry, an entry for the trigger node's task, and adding a synchronous request.
- 2 DO
- 3 take the first node off the trigger node list and make it the trigger node
- 4 DO
- 5 match a template using the template descriptions of Table 1, in the order specified.
- 6 execute modeling operations of the current template as described in Table 1, including appending of any new trigger nodes to the trigger node list (by the modeling operation *addTrigger*)
- 7 mark the matched nodes of the matched template as “processed”, and they are disqualified from any future template comparisons.
- 8 the trigger node's closest successor node that has not been labelled as “processed” is made the new trigger node

Figure 5: The Control Algorithm

- 3) With glue node $\left\{ e_1 = \left| \frac{a,1,3,Ac}{A,3} \right| \right\}$ the basic RPC interaction template (template (i)) is matched and $\left\{ e_2 = \left| \frac{a,1,4,Ac}{B,1} \right|, e_3 = \left| \frac{a,1,6,Ac}{B,3} \right|, e_4 = \left| \frac{a,1,7,Ac}{A,4} \right| \right\}$. The modeling operations are: *newEntry*(2, R, B), *addRequest*(1, 2), *addActivity*(1, e_1), *addActivity*(1, e_4), *addActivity*(2, e_2), and *addActivity*(2, e_3).
- 4) With glue node $\left\{ e_1 = \left| \frac{a,1,5,(Fk,a,2)}{B,2} \right| \right\}$ the asynchronous interaction template (template (ii)) is matched and $\left\{ e_2 = \left| \frac{a,2,1,Be}{C,1} \right| \right\}$. The modeling operations are: *newEntry*(3, A, C), *addRequest*(2, 3), and *addTrigger*($\left| \frac{a,2,2,Ac}{C,2} \right|$, 3). The node that is added to the trigger node list is $\left| \frac{a,2,2,Ac}{C,2} \right|$ because the begin node $\left| \frac{a,2,1,Be}{C,1} \right|$ has been marked as processed (it matched the asynchronous interaction).
- 5) With glue node $\left\{ e_1 = \left| \frac{a,1,8,Ed}{A,5} \right| \right\}$ the end node template (template (viii)) is matched and there are no associated modeling operations. The control algorithm recognizes that this thread has been fully processed, so the head node of the trigger node list becomes the new trigger node.
- 6) With glue node $\left\{ e_1 = \left| \frac{a,2,2,Ac}{C,2} \right| \right\}$ the activity node template is matched (template (vii)) and the modeling operation is *addActivity*(3, e_1).

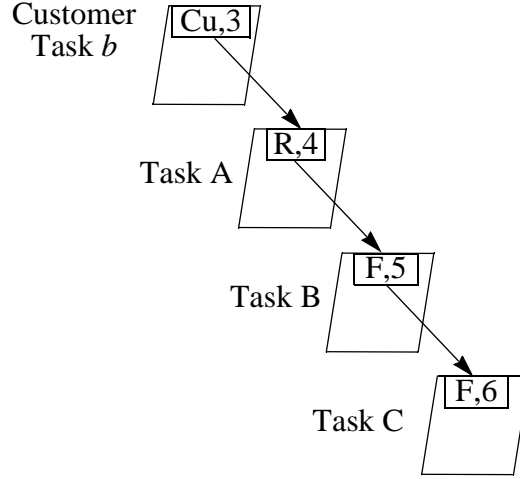


Figure 6: Simple Forwarding LQN Sub-model Example

- 7) With glue node $\left\{ e_1 = \left| \frac{a,2,3,Ed}{C,3} \right| \right\}$ the end node template is matched (template viii) completing analysis of this thread. Since the trigger node list is empty there are no remaining threads to process and the analysis is complete. The LQN sub-model of Figure 2 is the result of this analysis.

The example LQN sub-model of Figure 6 incorporates forwarding. It is developed by applying the control algorithm to the simple forwarding TOEG of Figure 4.

5.3 Correctness and Coverage of TLC

To ensure that every event is properly matched to a template, we rely on the fact that all send and receive events are captured and that the application proceeds in such a way as to generate events that match the templates. These attributes follow from the assumptions that the distributed operation has sequential tasks with no explicit synchronization, and communication is by reliable, point-to-point synchronous and asynchronous protocols (which follow the templates).

The same assumptions ensure that the model structure is correct once the templates are matched, since the model is based directly on the semantics of the interaction templates. For example, three templates (iv), (v), and (vi) contain the asynchronous message template (ii) and they are tested first on the assumption that if they occur, the blocking behavior they imply is present. Templates are defined for the RPC, asynchronous, and forwarding interaction types because they are integral to the layered model definition.

All events are matched because the control algorithm examines each scenario thread, assigning the trigger node in a depth-first fashion, and matching each trigger node to a template. If the choice of template is unique it matches the interaction of the application at that point. If it is not unique the template selection order does not matter because the larger templates are just concatenation of smaller ones so the coverage of events is identical and a valid sub-model is produced. In this work,

the control algorithm gives priority to larger templates on the assumption that a larger template match provides a more accurate or more descriptive model. The ordering is shown by rank in the “Order” column of Table 1

The adequate coverage of behavior by scenarios is an important question that cannot be resolved here. The problem is similar to test coverage. 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. A well defined set of scenarios will reveal enough of the important behavior to give a useful performance model. Criteria for scenario coverage are an important (but difficult) future goal.

Tracing errors and lost events pose problems for TLC which are not addressed here. If a lost event nonetheless leads to a complete template match the loss will be undetectable and some system feature will be left out of the model. In some cases we expect that a single lost event (a single send or receive for instance) can be detected and also corrected. Multiple lost events may well be unrecoverable.

6.0 Developing an LQN Model from LQN Sub-Model(s)

After the analyst has selected the scenarios to be investigated and has developed the appropriate LQN sub-models, an LQN model is completed by: including the workload parameters, providing the target configuration, and assigning resource demands to the activities. The first two items can be varied by the analyst to construct differing performance models.

The workload parameters of the LQN model include, 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. If there is more than one LQN sub-model, then any entries with the same task identifier are grouped together. The routing of work between scenarios is modeled by combining the customer task’s of the appropriate sub-models, forming a larger sub-model.

The target configuration information that completes the model includes the number of processors, processor speeds, scheduling disciplines, task priorities, and the assignment of tasks to processing elements. If it is decided that a task should be multi-threaded or cloned the number of replications is specified.

Lastly, the activity nodes are converted into resource demands using resource functions. Consecutive resource demands within the same entry and phase can be grouped together before evaluating the model.

Once an LQN model is produced it may be simplified. For example, entries of a task with the same interaction type can be merged by averaging their resource demands and the number of requests, weighted by the expected frequency of invocation of the entries to be merged. The averaging maintains the entry-entry request ratios and the aggregate resource demands of the LQN sub-model. This is done in the case study.

7.0 Call Processing Prototype Case Study

The purpose of the case study is to illustrate the benefits of TLC using a modest executable design. The subject of the case study is a call-processing prototype [2, 3] developed with the design prototyping environment called ObjecTime [44, 34]. The ObjecTime run-time system was instrumented to automatically record significant prototyping language statements. A third party had previously constructed the design prototype without our involvement so it provides a blind test for the technique. After describing the prototype, the resulting LQN sub-model is presented. The developed sub-model was verified to be correct by a code walk-through.

The telephony prototype has three different scenarios of execution which would be recorded as individual traces. These scenarios are “set-up”, “use”, and “hang-up”. Normally, each scenario would have an LQN sub-model developed and all three sub-models would later be combined. Due to space constraints, only the LQN sub-model for the “set-up” scenario is presented since it is representative of the technique.

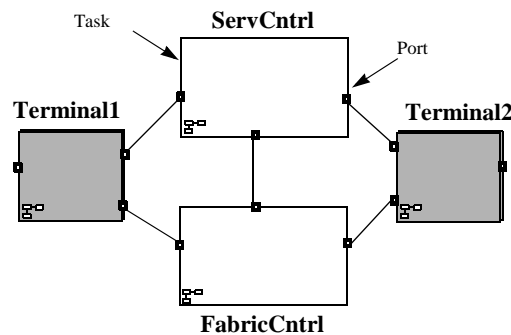


Figure 7: Call Processing Prototype Structure

The highest level layer of the telephony prototype is shown in Figure 7 (there are several lower layers). The prototype structure consists of tasks and ports.³ A task is a concurrent object that requests services from other tasks by sending messages to ports. The behavior of a task is specified using an extended state machine syntax that is similar to StateCharts [13]. A task can inherit the behavior and structure of another task.

To generate a trace of the “call set-up” scenario a “set-up” message is sent from “Terminal1” to the “ServCntrl” task. Using this trace a TOEG is developed that is too large to present. However, the Message Sequence Chart of Figure 8 is useful for illustrating the inter-task communication of the scenario [4]. It shows tasks as vertical bars with time proceeding downwards and each inter-task message is a directed arc from sender to receiver. The task name is identified at the top of the vertical bar (the numbers in parenthesis). Only the communication events (i.e. send and receive) are shown although other events were recorded.

3. ObjecTime uses the term “actor”. We will use the more generic term “task”. A port is a service entry in the LQN sub-model.

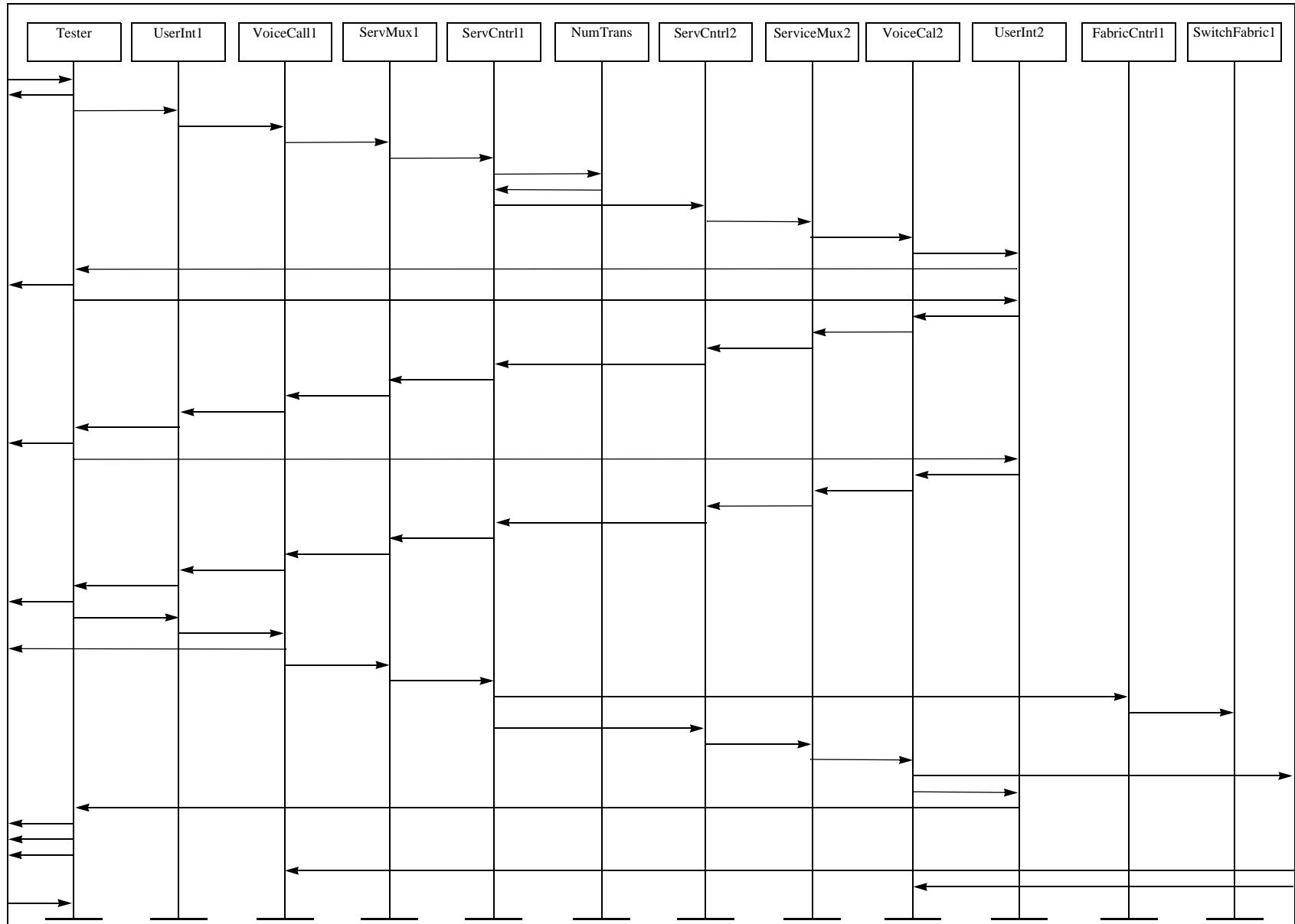


Figure 8: ANGIOTRACE Example using MSC

Manual modelling of this system would have been difficult because message reception at a task is anonymous. The identification of tasks would have involved analyzing the task inheritance hierarchy and task attributes. There were 11 tasks, some of which are created and destroyed with each request. A manual investigation would virtually require stepping through the design which is analogous to the steps of TLC!

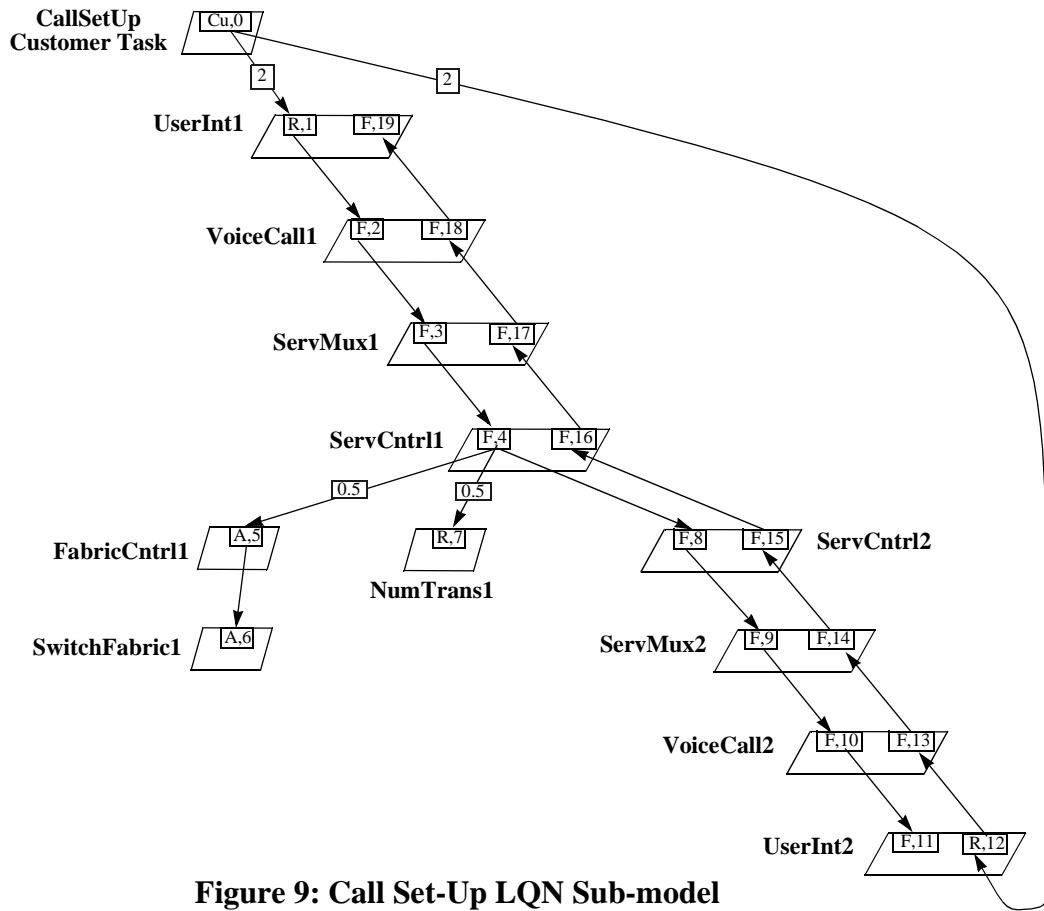


Figure 9: Call Set-Up LQN Sub-model

Using TLC the task identifiers and interactions of the scenario were automatically recovered from the prototype. The “call set-up” trace processing produced the LQN sub-model of Figure 9. The sub-model begins with the “Call Set-Up Customer Task.” The execution then proceeds as two separate task pipelines that flow in opposite directions. The two task pipelines are formed by the chain of the forwarding entries (*F*). The customer task initiates two requests to the entries (*R*, 1) and (*R*, 12), indicating that both entries are called twice per “call set-up” scenario. Each request to these entries initiates subsequent processing.

Processor	Task Name
Processor 1	UserInt1, VoiceCall1, ServMux1, ServCntrl1
Processor 2	UserInt2, VoiceCall2, ServMux2, ServCntrl2
Processor 3	FabricCntrl1, SwitchFabric1, NumTrans1

Table 2: Processor Allocation for Example LQN Sub-model

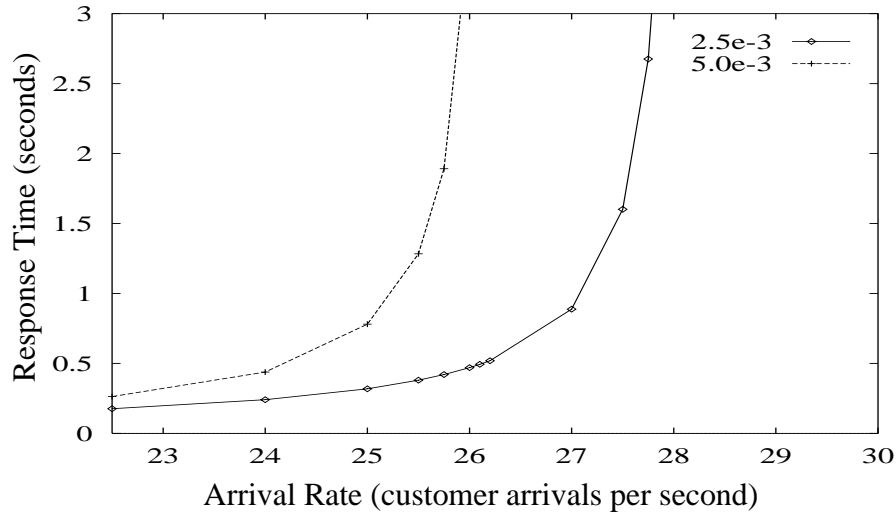


Figure 10: Effect of NumTrans1 Service Time on the Call Set Up Response Time

An RPC interaction constructed from asynchronous messages was identified, with the initiating entry (F, 4) of ServCntrl1 and the responding entry (R, 7) of task NumTrans1. This RPC service request occurs once per scenario but the initiating entry (F, 1) is called twice per scenario. Then the average number of service request for this RPC interaction is 0.5. This averaging operation maintains the aggregate resource demand and service requests for the LQN sub-model when the entries were merged.

The LQN sub-model can provide performance predictions once the configuration information and resource demand estimates are provided. For example, the developed LQN sub-model of Figure 9 can be used to estimate the delay a customer experiences when trying to set up a call for a varying traffic load. An objective of this example is to understand the response time impact of the CPU demand of the task “NumTrans1.” The target configuration has three processors with the task assignment as per Table 2. Each task is allocated a CPU cost budget of one milli-second to satisfy an accepted service request, except for the NumTrans1 (number translation task) which involves a database search. The CPU cost of NumTrans1 are estimated to be between 2.5 and 5.0 milli-seconds. With this configuration the response time predictions are shown in Figure 10 so the expected characteristics of the system (for this configuration) will fall between the two curves. This is very important capacity information since there are timing requirements for call set up time.

The value of TLC in this example is more evident when potential uses of the LQN sub-model are considered. First, the LQN sub-model can be used to help understand the task architecture and the role of each task within the scenario. This is even more valuable when other sub-models, such as the “call use” and “hang-up” scenarios, are added to the picture. Secondly, one can use a knowledge of the architecture to expand the performance model to describe a larger system with many users and many sites, with a variety of processors and networks, and including variations of the task architecture. This can be done by modifying the performance model to explore

configuration issues and performance limits. Lastly, one could expand the study to include more calling features (such as call waiting or voice mail), recording the traces and combining the resulting LQN sub-models.

8.0 Conclusions and Further Work

Trace-based Load Characterization (TLC) recovers the software structure and execution behavior of an executable design, which is useful for developing a performance model. It captures the total workload of a scenario by including all its activities. In our additional work we have found that the technique scales-up well. For example, an LQN-submodel was generated from a trace of more than 10,000 events with 73 potentially concurrent software components.

Including asynchronous task interactions and using a template matching approach has greatly extended the scope of the model building technique from that described in [20]. The trace processing provides accurate models by detecting elusive RPC interactions that are constructed from asynchronous messages, which may be overlooked during a manual examination. This is important for analyzing industrial software since designers may introduce asynchronous interactions to improve performance, unknowingly implementing blocking interactions anyway. It is essential to have this feature in order to provide a quantitative assessment of performance.

A surprising aspect of the case study was the frequent use of forwarding interactions, which we had hypothesized before beginning. Forwarding appears to be an interesting property of concurrent software architectures which has not been prominently mentioned before.

In principle the template-matching technique appears to be extensible to include synchronization between tasks and other interaction types, such as multicasts. The development of resource functions which map the activities onto their resource demands is ongoing research.

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. The advantages of TLC are most fully realized if it is part of an evolutionary approach to software development, moving from an early executable design or partial implementation to a complete product [28, 16]. As an executable design is refined into an implementation, TLC transparently incorporates more detail into the model. This allows performance expectations to be tracked through the development cycle, and performance analysis to be tightly integrated with the software process. With TLC, the performance analyst can focus on the principles of software performance analysis rather than model building.

9.0 Acknowledgments

This research was supported by the Strategic Grant Program of the Natural Sciences and Engineering Research Council (NSERC) of Canada and by it, Bell Canada, BNR, Gandalf and DY-4, Inc., through an Industrial Research Chair. We are grateful to Bran Selic of ObjecTime for his

support. We are thankful to Dorina Petriu, and Shikharesh Majumdar for their helpful comments during the project, and to the anonymous reviewers for the improvements they suggested.

References

- [1] 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.
- [2] R. Balzer. “Draft report on requirements for a common prototyping system.” *ACM SIGPLAN Notices*, 24(3):93–165, Mar. 1989.
- [3] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Zullighoven. *Prototyping: An Approach to Evolutionary System Development*. Springer-Verlag, New York, New York, 1992.
- [4] CCITT. “Recommendation Z.120: Message Sequence Chart (MSC).” Technical report, Geneva.
- [5] J. Clark and D. A. Holton. *A First Look at Graph Theory*. World Scientific, Totteridge, London, 1991.
- [6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe. “Graph grammars and logic programming.” 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 221–237, 1991.
- [7] 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.
- [8] T. Dodd. *Prolog: A Logical Approach*. Oxford University Press, New York, New York, 1990.
- [9] 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.
- [10] C. Fidge. “Logical time in distributed computing systems.” *IEEE Computer*, pages 28–33, August 1991.
- [11] W. M. Gentleman. “Message passing between sequential processes: the reply primitive and the administrator concept.” *Software – Practice and Experience*, 11:435–466, 1981.
- [12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.
- [13] D. Harel. “Statecharts: A visual formalism for complex systems.” *Science of Computer Programming*, 8, 1987.
- [14] F. Hayes-Roth and D. Waterman. “Principles of pattern-directed inference systems.” In D. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 577–601. Academic Press, 1978.
- [15] M. Heath and J. Etheridge. “Visualizing the performance of parallel programs.” *IEEE Software*, 8(5):29–39, September 1991.
- [16] S. Hekmatpour and S. Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley Publishing Co., New York, New York, 1988.
- [17] L. Hess and B. Mayoh. “Graphics and their grammars.” In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, number 291 in *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 1987.
- [18] L. Hess and B. Mayoh. “The four musicians: Analogies and experts systems - a graphic approach.” In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, number 532 in *Lecture Notes in Computer Science*, pages 430–445. Springer-Verlag, 1990.
- [19] J. Hollingsworth and B. Miller. “Dynamic control of performance monitoring on large scale parallel systems.” *Proceedings of International Conference on Supercomputing*, pages 19–23, July 1993.
- [20] 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>.
- [21] 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.
 - [22] G. M. Karam. “The MLog user’s guide.” SCE-89-15, Dept. of Syst. and Comp. Engineering, Carleton University, Apr. 1992. 78 pages.
 - [23] D. W. Kelton and A. M. Law. *Simulation Modelling and Analysis*. McGraw-Hill, New York, 2nd edition, 1991.
 - [24] 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.
 - [25] 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.
 - [26] E. D. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1984.
 - [27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
 - [28] Luqi and W. Royce. “Status report: Computer aided prototyping.” *IEEE Software*, pages 77–81, Nov. 1991.
 - [29] 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.
 - [30] 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.
 - [31] J. Neilson. *Parasol Users’ Manual*. School of Computer Science, Carleton University, Ottawa, Canada, K1S-5B6, 3.0 edition, July 1995.
 - [32] 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.
 - [33] 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.
 - [34] ObjecTime Limited, Kanata, Ontario, Canada. *ObjecTime User’s Manual*, 1997.
 - [35] Open Software Foundation. *Introduction to OSF DCE*. Prentice-Hall, 1992.
 - [36] F. Parisi-Presicce, H. Ehrig, and U. Montanari. “Graph rewriting with unification and composition.” 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 496–514, 1987.
 - [37] S. Prasad. “Models for mobile computing agents.” *ACM Computing Surveys*, 28:53, Dec. 1996.
 - [38] J. Rolia and K. Sevcik. “The method of layers.” *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.
 - [39] 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.
 - [40] 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.
 - [41] 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.

- [42] A. Schürr. “Logic based structure rewriting systems.” In H. J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 341–357, 1993.
- [43] R. Schwarz and F. Mattern. “Detecting causal relationships in distributed computations: in search of the Holy Grail.” *Distributed Computing*, 7(3):149–174, 1994.
- [44] B. Selic, G. Gullekson, and P. Ward. *Real-time Object-oriented Modeling*. John Wiley and Sons, 1994.
- [45] 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.
- [46] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Co., New York, NY, 1990.
- [47] D. J. Taylor. “A prototype debugger for Hermes.” In *CASCON 92*, pages 29–42, 1992.
- [48] 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.
- [49] 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.
- [50] 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.