Trace-Based Load Characterization for Generating Software Performance Models

C. E. Hrischuk, C.M. Woodside,
J. A. Rolia, R. Iversen

SCE-97-05
March 1997

{ceh, cmw, jar}@sce.carleton.ca,
rod@ObjecTime.com

1125 Colonel By Drive
Real-Time and Distributed Systems Lab
Department of Systems and Computer Engineering,
Carleton University
Ottawa, K1S 5B6
Canada

http://www.sce.carleton.ca/rads/rads.html

Trace-Based Load Characterization for Generating Software Performance Models

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

*Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
** ObjecTime Ltd., Ottawa, Canada

{ceh, cmw, jar}@sce.carleton.ca, 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. Angio tracing 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 angio trace 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

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 [29], queueing network model [15], or a state-based model like a Petri-Net [11]. 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 [47]. Our previous work ([25]) demonstrated an approach to ease this burden by automatically generating a performance model, provided the software components communicated solely by remote procedure call. This paper describes a more developed technique which can develop models of most distributed or parallel programs. This technique is called Trace-based Load Characterization (TLC).

Performance models of distributed and concurrent systems characterize tasks and their interactions. We refer to a sequentially executing, concurrent software component as a *task*.[1] Interactions between tasks are important because they affect the parallelism and resource contention experienced during execution. For example, a heavily used task can queue arriving requests and can even become a bottleneck [28]. The Layered Queueing Network model [9, 40] has been proposed to study these systems. The LQN model extends queueing network models to include contention effects for software resources such as server tasks, as well as contention for hardware devices. It is appropriate for assessing the performance of many kinds of distributed systems, including client-server applications [1], peer-to-peer applications [32], communications switching software [4], transaction processing systems (e.g. ENCINA [24]), and systems based on midware software technologies [5] such as CORBA [35] or DCE [13]. TLC incorporates enough detail to construct an LQN model for these types of systems.

TLC grew out of a technique (reported in [25]) for building models of systems which only communicated with Remote Procedure Calls (RPC). These are synchronous (request-reply) interactions that block the sender and cause delays. However, for performance reasons many systems use asynchronous (send and continue) messages. Sometimes the asynchronous messages are combined into a request-reply interaction. Similarly, a partially synchronous interaction may be formed when a request is followed by a sequence of asynchronous messages through a series of tasks, ending in a reply to the blocked originator of the request; this will be termed a *forwarded RPC*. The layered queueing model supports these interactions, as well as pure asynchronous requests, and TLC has been extended to detect these patterns. It now includes every message and its subsequent processing, so that all workload elements are incorporated. It can identify request-reply patterns even when they are disguised by asynchronous messages. A join interaction between

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.

tasks, and the synchronization delay, is not identified by this version of TLC and it will be added in future work.

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

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, the angio trace event format records information with each user-defined and communication event that implicitly captures their cause and effect relationships. It was introduced in [25] and has been formally described in [26]. Angio traces can be extracted from many sources at various steps of the development, such as annotated specifications (e.g. scenarios, Use-Cases [42] or Message Sequence Charts [7]), functional prototypes, detailed simulations, or the production system. We have implemented it in several environments: a functional prototyping environment (MLOG [16]), a commercial prototyping environment (ObjecTime [36]), a distributed software system simulator called Parasol [34], coarse-grained UNIX tasks [27], and in the DCE RPC environment [37] using data collected by the POET debugger [49]. We concentrate here on traces produced by a design prototype environment, namely ObjecTime.

There are several benefits to using TLC as opposed to a “source code examination” approach for constructing 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. The automated trace processing results in more accurate performance model construction at a lower cost because a larger volume of details is included in the model development. An area where automation has a decided advantage is for correctly identifying the type of an interaction. For example, TLC identifies a synchronous interaction constructed from asynchronous messages. This is important when the nature of an interaction cannot be explicitly identified in the trace. Finally, TLC can also be used to model the production version of the software, to provide full life-cycle support for modelling.

There are other performance engineering techniques that use traces but which are very different from TLC in intention and content. They use statistical measures and visualization for evaluating and tuning the performance of a final product. Where they address model development at all, it is to determine resource demands with little attention to model structure. Examples include: Paragraph [19], SIMPLE [12], ChaosMON [30], AIMS [51], PABLO [2], and W^3 [23].

There are three main contributions presented by this paper. First, TLC extends the model construction technique of [25] to handle interactions by asynchronous messages. Secondly, TLC formalizes the earlier, ad-hoc model builder by using a rule-based graph grammar approach for manipulating the trace. To demonstrate, a new case study is presented that requires the expanded capabilities of TLC. It 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 angio tracing, 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.* [15] or Smith [47]. For the rest of the document, the term *trace* refers to an angio trace and *event* refers to an angio trace event.

2.0 Coupling TLC with Performance Analysis

The steps in applying TLC in a performance engineering context are shown in Figure 1. The first step (#1) is to select *scenarios* which are important for performance modelling (as described

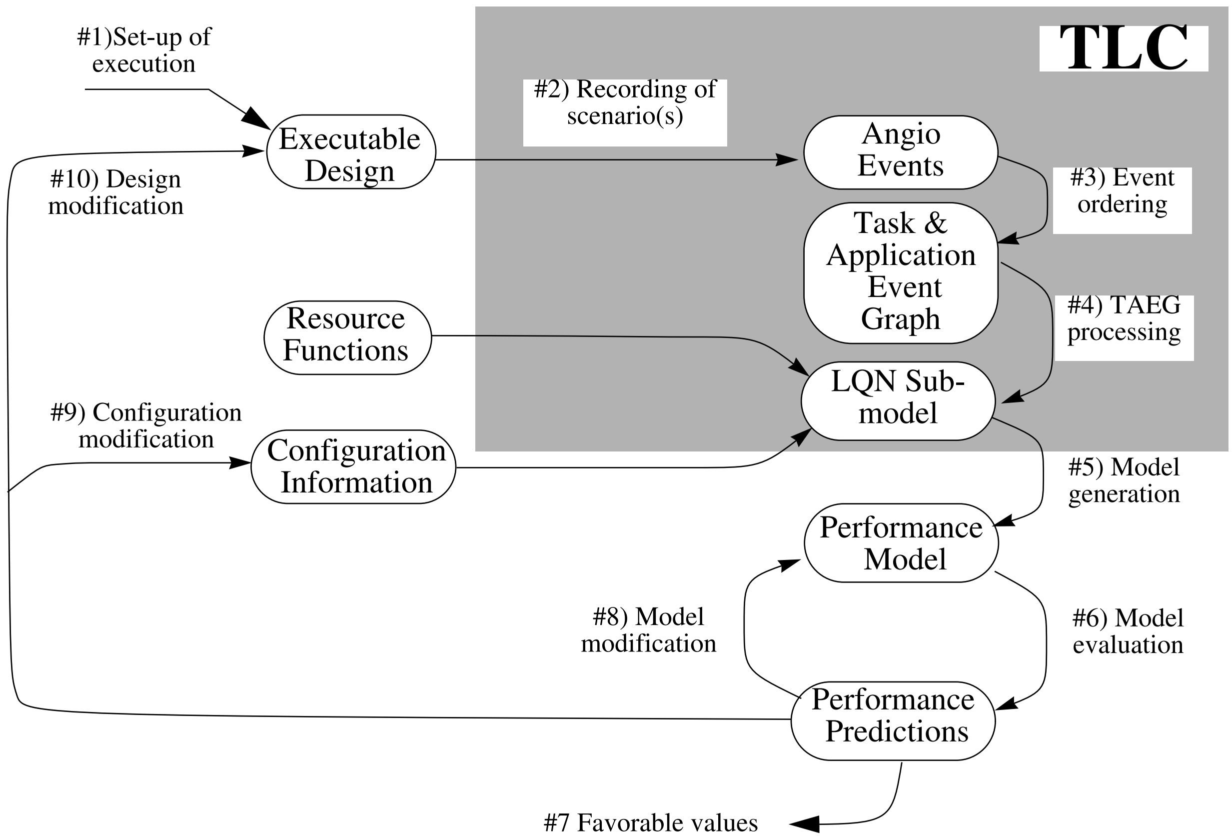


Fig. 1: Integrating TLC and Performance Analysis

in [47]) and to add instrumentation to identify where the execution of each scenario begins and ends. In step #2, angio trace events are recorded during the execution of a scenario.

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). These two steps 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 in the scenarios and they are introduced at this step. 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 [47] and improved (if necessary) as the development progresses. For the purposes of this research it is assumed that a database of resource functions is available.

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 sub-model construction steps (#3 and #4) of TLC in detail.

3.0 Layered Queueing Network Sub-Models

The software components and interactions that contribute to a single scenario are captured in an LQN submodel. The LQN submodel describes the software components as a set of tasks with service entries; interactions are requests between entries. Formally it is a labelled, directed graph whose nodes are *entries* and arcs are *service requests*. An entry characterizes the service a server task provides in response to a service request from another task. A service begins when the responding task accepts the initiating task's service request and the service is assumed to end with the acceptance of the next service request. This section describes the entries, service requests, and the interaction types.

Each service request arc has a source entry for the task that initiated the request, a target entry for the responding task, and an attribute that identifies the type of interaction. An arc is labeled with the average number of requests to the responding task's entry each time the initiating entry accepts a request with the default number of requests set to one.

The arc information is represented in the labeling of the entries: the initiating task's entry has a *service request label* to identify the responding task's entry, and the responding task's entry has a label that identifies the arc's interaction type. Entries labels also include a task identifier, a unique identifier for the entry, and a characterization of the activities that occurred during the service. Recognizing the beginning of a service, the end of a service, and its interaction type is a key problem in constructing the LQM sub-model. These issues are addressed in section 5.0.

The labeling of an entry node (E_n) is described as $E_n = \{\text{interaction type}, \text{task identifier}, \text{list of service requests}, \text{list of activities}\}$. The list of service requests by entry E_n is $V_n = \{n_1, n_2, \ldots\}$ where n_i represents a service request from entry E_n to the entry E_{n_i}. The list of activities is $A_n = \{e_1, \ldots\}$ which identifies the set of activities executed by entry E_n and their corresponding resource functions. 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. This interaction occurs with standard types of blocking remote procedure calls [48] and rendezvous. The initiating task experiences a queueing delay at the responding task.
- *Asynchronous* (type ***A***): an initiating task does not wait for a reply from the responding task. No reply message is needed. This interaction type was not considered in [25].
- *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 behavior occurs when a task acts as an administrator manager by dispatching service requests to a pool of worker tasks [50] 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. We define its type to be ***Cu***. It is a surrogate for the source of the initial stimulus of a scenario.

It may be that two or more entries 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.

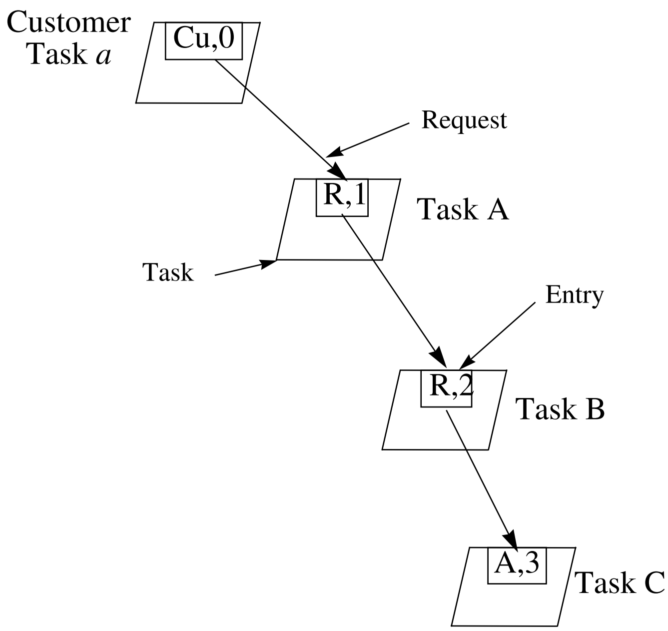
The LQN sub-model splits RPC-based interactions into two phases of execution for the responding task, just like in an LQN model. The first phase consists of the execution between the responding task's request acceptance and subsequent reply (or forwarding) of the request. Any work that occurs after the 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 [45, 39]. 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.

Each trace is analyzed to produce one submodel with a fixed structure. For a set of different applications, a set of LQN submodels is created, and these are merged into one LQN model. This final step is described in section 6.0.

The trace processing used to construct the LQN is described in the next section.



$$E_0 = \{Cu, a, \{1\}, \{\ \}\}$$

$$E_1 = \{R, A, \{2\}, \{2, 3, 4\}\}$$

$$E_2 = \{R, B, \{3\}, \{1, 3\}\}$$

$$E_3 = \{A, C, \{\ \}, \{2\}\}$$

Fig. 2: LQN Model Example

4.0 Event Notation and Event Ordering

The angio trace events record the flow and causality of a scenario, which is then represented as a TAEG (*Task and Application Event Graph*). The TAEG is a node-labelled, directed, binary, acyclic graph [8] whose structure is based on the causal relationships between events. It is constructed by collecting and ordering the events (steps #1, #2, and #3 of Figure 1). Each trace event produces a labelled node in the TAEG. The arcs in the TAEG, which represent cause and effect relationships, are deduced from the node labelling.

A node is labelled with six parameters: two parameters tie it to a task's execution and four parameters describe the execution of the distributed application. The task information is a *task identifier* and a *task node counter* that orders the nodes recorded by a task. The distributed application 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 an application, 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 TAEG 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:

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

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 TAEG 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

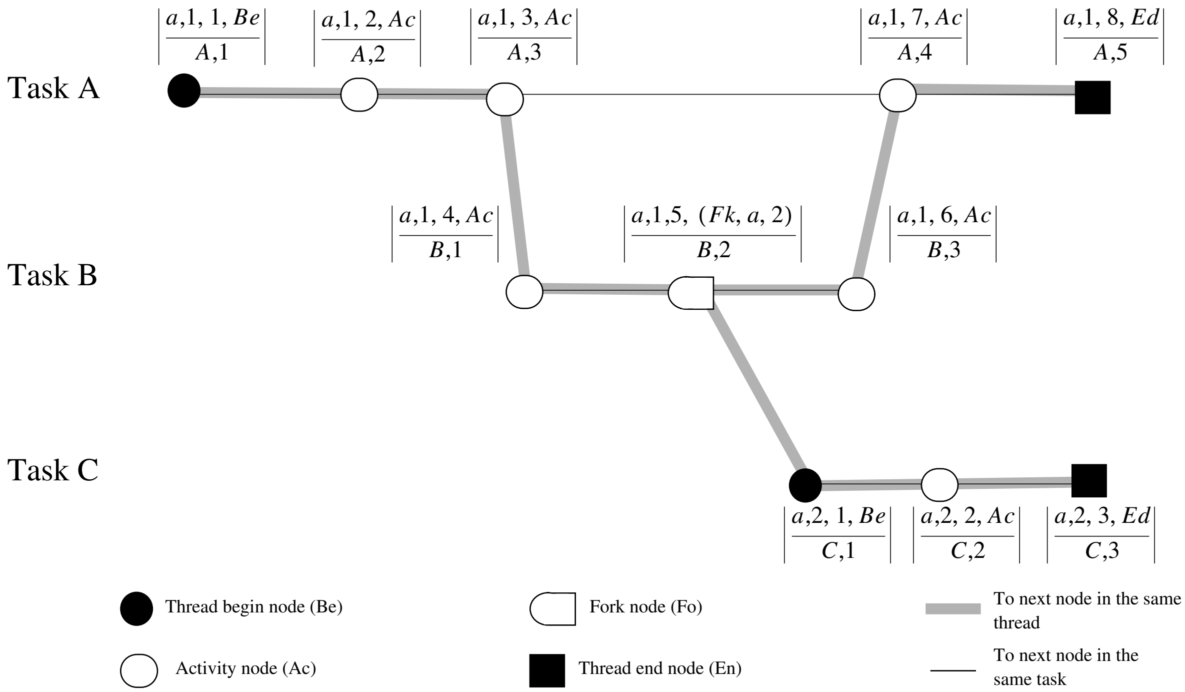


Fig. 3: Example TAEG with Identifier a

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 TAEG 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 TAEG. 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 model development is the algorithm for reducing a TAEG to an LQN submodel, by transforming one graph to another. It uses a rule-based graph analysis approach [18, 31, 22, 41]. The rules have an antecedent part and a consequent part. The antecedent component of a rule is a graph fragment that matches an interaction pattern in the TAEG 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 [14], in a similar way to other authors [38, 10, 21, 43, 44]. The control algorithm is discussed following the presentation of the interaction templates.

5.1 Interaction Templates and their Modeling Operations

An interaction template identifies an occurrence of a certain interaction type. A match also identifies the initiating task(s), the responding task(s), and the possible occurrence of a second phase of execution (to be described later in the section). Once 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. Some implementation optimizations are also presented.

An *interaction template* identifies a pattern of communication that can exist in the target design, it is 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 template matching process proceeds with the control algorithm supplying a TAEG 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 TAEG 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, for example.

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 ($activity$) to an entry ($entry$). The activity is only identified by its task node counter, 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.

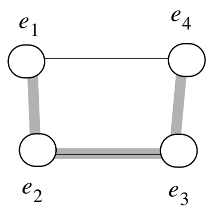
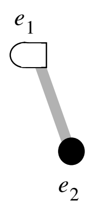
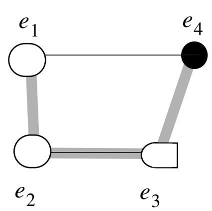
Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(i) Basic RPC $\{e_1, \ldots, e_4\}$	#1	 $e_1 = \left\|\frac{j_1,k_1, m_1, Ac}{i_1,v_1}\right\|$ $e_2 = \left\|\frac{j_1,k_1, m_1+1, Ac}{i_2,v_2}\right\|$ $e_3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

Table 1: Interaction Templates and Modeling Operations

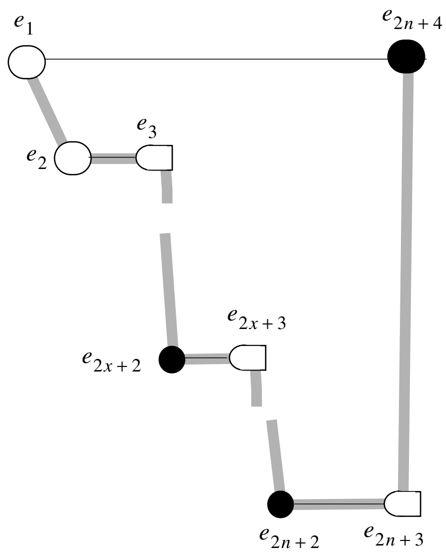
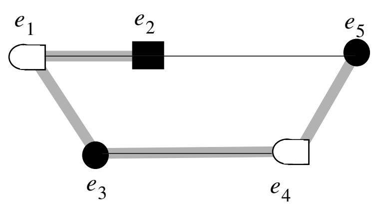
Interaction Type and Node Set	**Order**	**Template Description**	**Modeling Operations (let E represent the entry of task i_1 associated with node e_1)**
(iv) Forwarding interaction with n levels of forwarding $\{e_1, \ldots, e_{2n+4}\}$	#5	 $e_1 = \left\|\frac{j_1, k_1, m_1, Ac}{i_1, v_1}\right\|$ $e_2 = \left\|\frac{j_1, k_1, m_1+1, Ac}{i_2, v_2}\right\|$ $e_3 = \left\|\frac{j_1, k_1, m_2, (Fk, j_1, k_2)}{i_2, v_3}\right\|$ for x := 1 to n do begin $e_{2x+2} = \left\|\frac{j_1, k_{x+1}, 1, Be}{i_{x+2}, v_{2x+2}}\right\|$ $e_{2x+3} = \left\|\frac{j_1, k_{x+1}, m_{x+2}, (Fk, j_1, k_{x+2})}{i_{x+2}, v_{2x+3}}\right\|$ end $e_{2n+4} = \left\|\frac{j_1, k_{n+2}, 1, Be}{i_1, v_1+1}\right\|$	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $addActivity(E, e_1)$ $addActivity(E_n, e_2)$ $phase2(k_1, i_2, v_3)$ for x:=1 to n do begin $newEntry(E_{n+x}, F, i_{x+1})$ $addRequest(E_{n+x-1}, E_{n+x})$ $addTrigger\left(\left\|\frac{j_1, k_{x+1}, 2, l_x}{i_{x+2}, v_{2x+2}+1}\right\|, E_{n+x}\right)$ $phase2\,(k_{x+1}, i_{x+2}, v_{2x+3})$ end
(v) Single RPC using asynchronous primitives $\{e_1, \ldots, e_5\}$	#3	 $e_1 = \left\|\frac{j_1, k_1, m_1, (Fk, j_1, k_2)}{i_1, v_1}\right\|$ $e_2 = \left\|\frac{j_1, k_1, m_1+1, Ed}{i_1, v_1+1}\right\|$ $e_3 = \left\|\frac{j_1, k_2, 1, Be}{i_2, v_2}\right\|$ $e_4 = \left\|\frac{j_1, k_2, m_2, (Fk, j_1, k_4)}{i_2, v_3}\right\|$ $e_5 = \left\|\frac{j_1, k_3, 1, Be}{i_1, v_1+2}\right\|$	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $phase2(k_2, i_2, v_3)$ $addTrigger\left(\left\|\frac{j_1, k_2, 2, l_1}{i_2, v_2+1}\right\|, E_n\right)$ $addTrigger\left(\left\|\frac{j_1, k_3, 2, l_2}{i_1, v_1+3}\right\|, E\right)$

Table 1: Interaction Templates and Modeling Operations

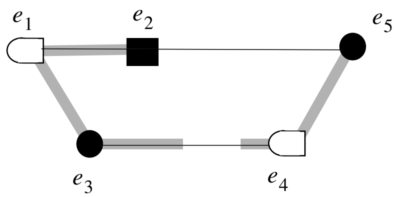
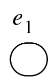
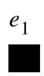
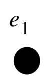
Interaction Type and Node Set	Order	Template Description	Modeling Operations (let E represent the entry of task i_1 associated with node e_1)
(vi) Nested RPC using asynchronous primitives $\{e_1, \ldots, e_5\}$	#4	 $e_1 = \left\|\frac{j_1, k_1, m_1, (Fk, j_1, k_2)}{i_1, v_1}\right\|$ $e_2 = \left\|\frac{j_1, k_1, m_1 + 1, Ed}{i_1, v_1 + 1}\right\|$ $e_3 = \left\|\frac{j_1, k_2, 1, Be}{i_2, v_2}\right\|$ $e_4 = \left\|\frac{j_1, k_3, m_2, (Fk, j_1, k_4)}{i_2, v_3}\right\|$ $e_5 = \left\|\frac{j_1, k_4, 1, Be}{i_1, v_1 + 2}\right\|$	$newEntry(E_n, R, i_2)$ $addRequest(E, E_n)$ $phase2(k_3, i_2, v_3)$ $addTrigger(\left\|\frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1}\right\|, E_n)$ $addTrigger(\left\|\frac{j_1, k_3, 1, l_2}{i_2, v_4}\right\|, E_n)$ $addTrigger(\left\|\frac{j_1, k_4, 2, l_3}{i_1, v_1 + 3}\right\|, E)$
(vii) Activity node $\{e_1\}$	#7	 $e_1 = \left\|\frac{j_1, k_1, m_1, Ac}{i_1, v_1}\right\|$	$addActivity(E, e_1)$
(viii) End node $\{e_1\}$	#8	 $e_1 = \left\|\frac{j_1, k_1, m_1, Ed}{i_1, v_1}\right\|$	-
(ix) Begin node $\{e_1\}$	#8	 $e_1 = \left\|\frac{j_1, k_1, 1, Be}{i_1, v_1}\right\|$	-

Table 1: Interaction Templates and Modeling Operations

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 $e_4 = \left|\frac{a,1,7,Ac}{A,4}\right|$.

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 $addTrigger(\left|\frac{j_1, k_2, 2, l_1}{i_2, v_2 + 1}\right|, E_n)$. In Figure 3, an asynchronous interaction is initiated by node $e_1 = \left|\frac{a,1,5,\ (Fk, a, 2)}{B,2}\right|$.

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., $phase2(k_1, i_2, v_3)$).

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 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 and the nesting depth is referred to by the symbolic value *n*.

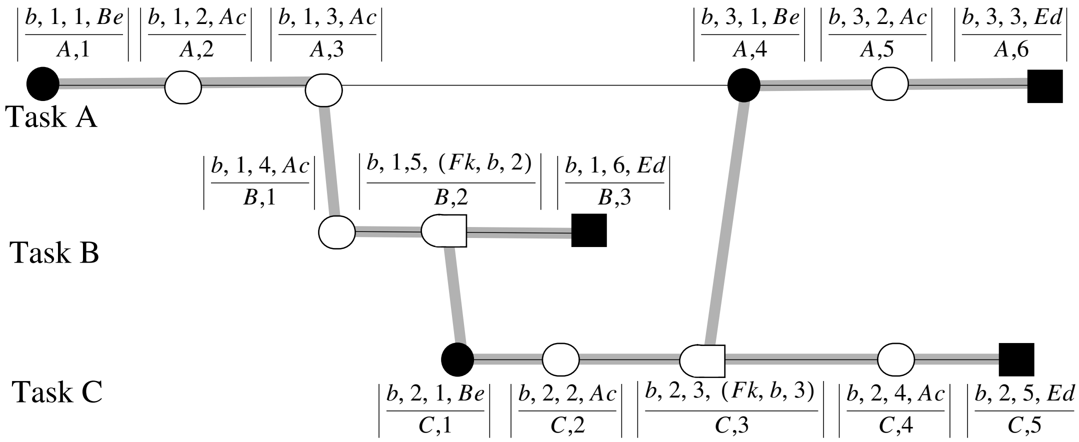


Fig. 4: Simple Forwarding TAEG

Nested forwarding occurs when several tasks process the blocked initiating task's request in a pipeline fashion with an eventual reply.

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 $\{i_3, \ldots, i_{n+2}\}$ are labelled as *forwarding*. The nodes that occur after the nodes $\{e_3, \ldots, e_{2x+3}, \ldots, e_{2n+3}\}$ for $1 \le x \le n$ 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.

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.

The last three templates of Table 1 complete the node processing of the TAEG. 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 TAEG and it is included for completeness.

Each interaction template identifies the start of a service period and the type of task interaction, while the end of a service period is determined in two fashions. First, an end node identifies the end of the service period (e.g. forward, asynchronous, and RPC with phase two execution interactions). Secondly the sending of a reply to a blocked initiating task is the end of service (e.g. basic RPC).

5.2 Template Matching Control Algorithm and Examples

A control algorithm manages the matching of interaction templates with part of the TAEG. It has an initialization component and an analysis component. The control algorithm is outlined in Figure 5. It orders the selection of interaction templates by attempting to match larger templates first. An ordering between the templates was found, so that later ones may be contained within earlier ones. This gives the rank in the "Order" column of Table 1. For example, this prevents a lone asynchronous interaction from matching a component of a forwarding interaction and precluding the detection of the forwarding interaction.

The control algorithm is initialized by putting the first node of the TAEG on the trigger node list ($\left|\frac{a, 1, 1, Be}{A, 1}\right|$), creating an artificial customer task and entry (E_{Cu}), a task and entry for the first event (E_n), and a synchronous request from the customer task to the first task. From the TAEG 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

1 Initialize by inserting the first node of the TAEG 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

9 WHILE (the end template (template (viii) of Table 1) was not matched)

10 WHILE (trigger node list not empty)

Fig. 5: The Control Algorithm

part of a previous template, in turn making each node a trigger node and beginning the template matching. 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. When the control algorithm has terminated all of the nodes in the TAEG have been marked as processed.

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

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

(2) With glue node $\{e_1 = \left|\frac{a, 1, 2, Ac}{A, 2}\right|\}$ the activity template (template (vii)) is matched. The modeling operation is $addActivity(1, e_1)$.

(3) With glue node $\{e_1 = \left|\frac{a,1,3,Ac}{A,3}\right|\}$ the basic RPC interaction template (template (i) is matched and $\{e_2 = \left|\frac{a,1,4,Ac}{B,1}\right|, e_3 = \left|\frac{a,1,6,Ac}{B,3}\right|, e_4 = \left|\frac{a,1,7,Ac}{A,4}\right|\}$. 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 $\{e_1 = \left|\frac{a,1,5,\ (Fk, a, 2)}{B,2}\right|\}$ the asynchronous interaction template (template (ii)) is matched and $\{e_2 = \left|\frac{a,2,1,Be}{C,1}\right|\}$. 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 $\{e_1 = \left|\frac{a,1,8,Ed}{A,5}\right|\}$ 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 $\{e_1 = \left|\frac{a,2,2,Ac}{C,2}\right|\}$ the activity node template is matched (template vii) and the modeling operation is $addActivity(3, e_1)$.

(7) With glue node $\{e_1 = \left|\frac{a,2,3,Ed}{C,3}\right|\}$ 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 TAEG of Figure 4.

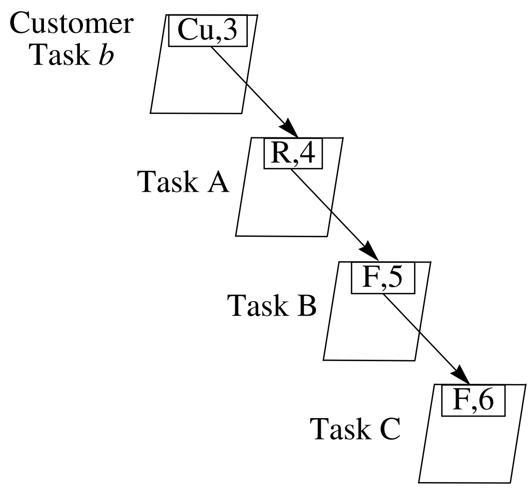


Fig. 6: Simple Forwarding LQN Sub-model Example

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 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 [3, 6] developed with the design protoyping environment called ObjecTime [46, 36]. 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.

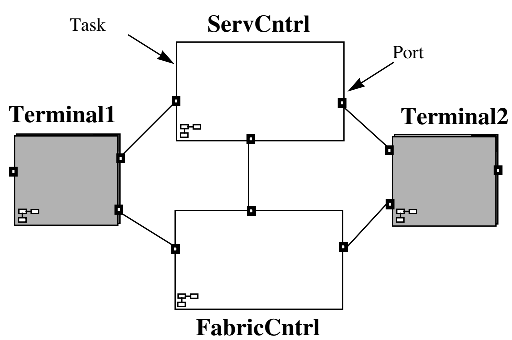


Fig. 7: Call Processing Prototype Structure

The highest level layer of the telephony prototype is shown in Figure 7 (there are several lower layers). The prototype structure consists of tasks[2] and ports.[3] A task is a concurrent object

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

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 [17]. 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 TAEG 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 [7]. 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.

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!

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.

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

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

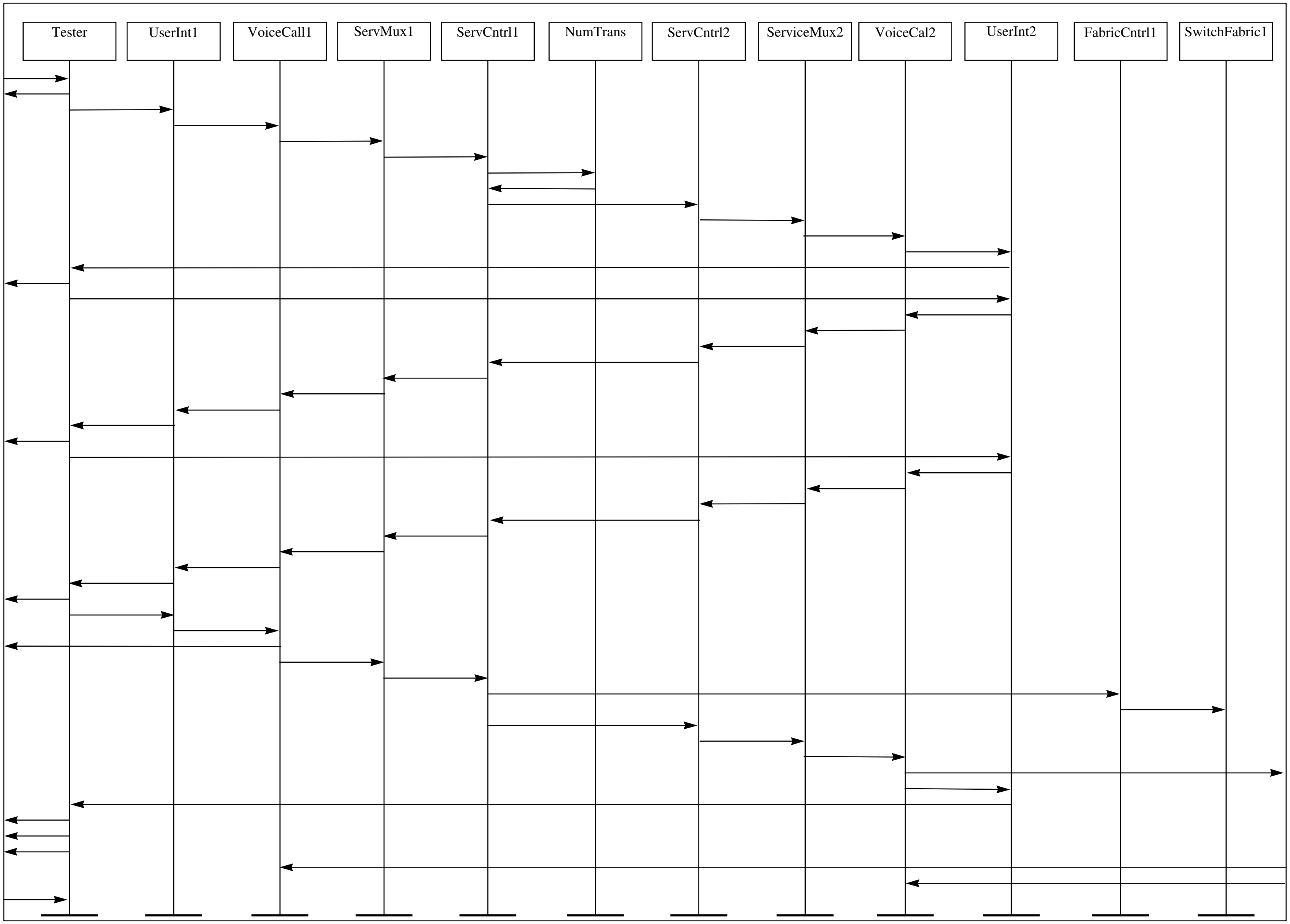


Fig. 8: Angio Trace Example using MSC

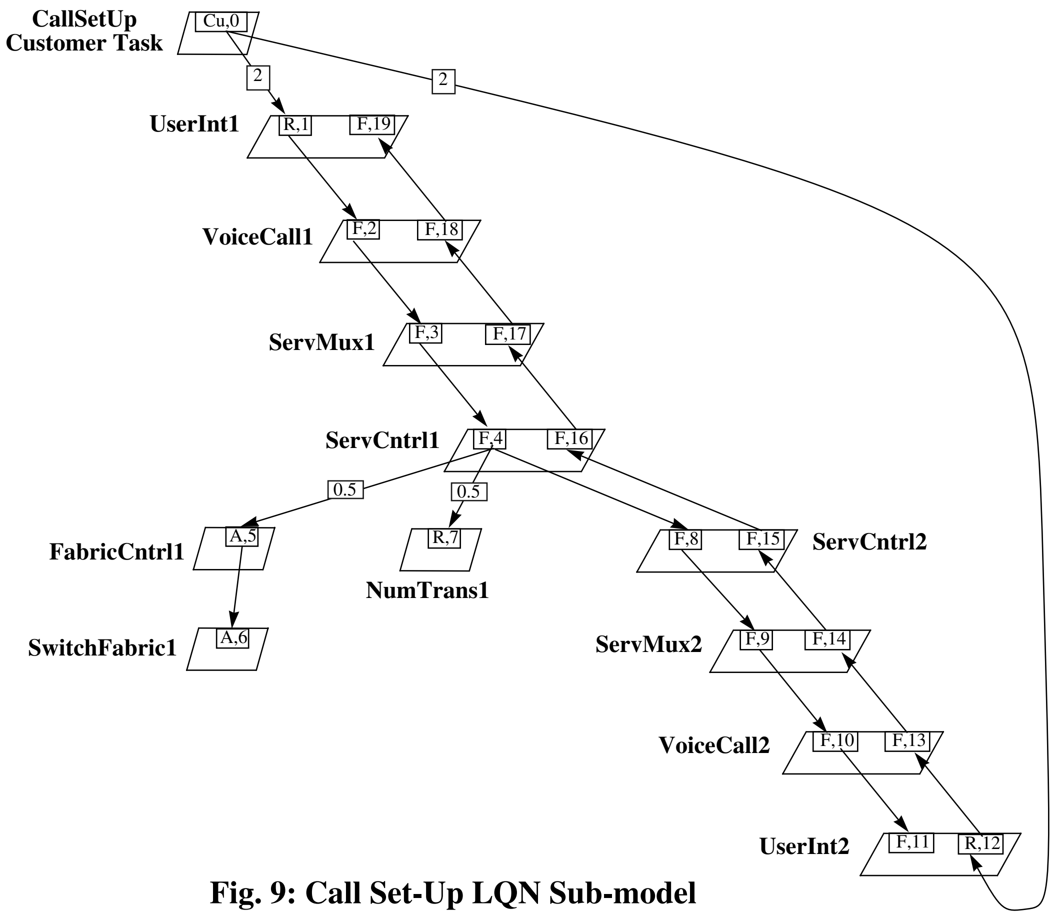


Fig. 9: Call Set-Up LQN Sub-model

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

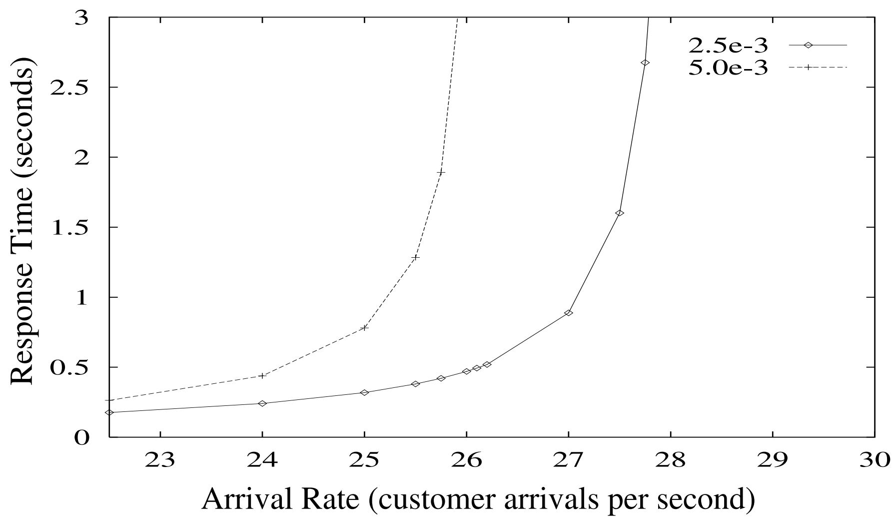


Fig. 10: Effect of NumTrans1 Service Time on the Call Set Up Response Time

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 telephone network 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

Including asynchronous task interactions and using a graph grammar approach has greatly extended the scope of the model building technique from that described in [25]. 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 critically important for analyzing industrial software since many designers introduce asynchronous interactions to improve performance. 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 predominant use of forwarding interactions. Forwarding appears to be an interesting property of concurrent software architectures which has not been prominently mentioned before. It will be interesting to see how often this interaction appears in industrial software.

The LQN sub-model of TLC captures the total workload of a scenario, by identifying when a service begins and ends, as well as the type of the involved interaction. When the LQN performance model is extended to include synchronization between tasks (apart from RPC communication) we believe that TLC can be used to detect these synchronization interactions, as well as other types of interactions that may be important. The rule-based graph grammar analysis approach is inherently modular so that extensions can be easily made. New interaction patterns which appear and are to be modeled can be easily incorporated by adding new templates to the control algorithm. 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 [33, 20]. 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 engineering to be tightly integrated with the software process. With TLC, the performance analyst can focus on the principles of software performance engineering 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, Shikharesh Majumdar, and Fred Kaudel for their helpful comments during the project.

References

[1] R. M. Adler. “Distributed coordination models for client/server computing.” *IEEE Computer*, pages 14–22, April 1995.

[2] 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.

[3] R. Balzer. “Draft report on requirements for a common prototyping system.” *ACM SIGPLAN Notices*, 24(3):93–165, Mar. 1989.

[4] 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.

[5] P. A. Bernstein. “Middleware: A model for distributed system services.” *Communications of the ACM*, 39(2):86–98, February 1996.

[6] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Zullighoven. *Prototyping: An Approach to Evolutionary System Development*. Springer-Verlag, New York, New York, 1992.

[7] CCITT. “Recommendation Z.120: Message Sequence Chart (MSC).” Technical report, Geneva.

[8] J. Clark and D. A. Holton. *A First Look at Graph Theory*. World Scientific, Totteridge, London, 1991.

[9] C.M. Woodside, J.E. Neilson, D.C. 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.

[10] 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.

[11] 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.

[12] 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.

[13] “Distributed Computing Environment: an overview.” Technical report, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, Jan. 1992.

[14] T. Dodd. *Prolog: A Logical Approach*. Oxford University Press, New York, New York, 1990.

[15] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1984.

[16] G.M. Karam. “The MLog user’s guide.” SCE-89-15, Dept. of Syst. and Comp. Engineering, Carleton University, Apr. 1992. 78 pages.

[17] D. Harel. “Statecharts: A visual formalism for complex systems.” *Science of Computer Programming*, 8, 1987.

[18] 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.

[19] M. Heath and J. Etheridge. “Visualizing the performance of parallel programs.” *IEEE Software*, 8(5):29–39, September 1991.

[20] S. Hekmatpour and S. Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley Publishing Co., New York, New York, 1988.

[21] L. Hess and B. Mayoh. “Graphics and their grammars.” In H. Erhig, 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.

[22] L. Hess and B. Mayoh. “The four musicians: Analogies and experts systems - a graphic approach.” In H. Erhig, 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.

[23] 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.

[24] P. J. Houston. “Building the next generation of client/server systems with the Encina monitor.” White paper, Transarc Corporation, 707 Grant St., Pittsburgh PA 15219, 1993.

[25] 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.

[26] C. E. Hrischuk and C. Woodside. “Proper time: Causal and temporal relations of a distributed system.” Technical Report SCE-96-04, Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, http://www.sce.carleton.ca/rads/rads.html, Mar. 1996. Submitted to IEEE Transactions on Parallel and Distributed Systems.

[27] 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.

[28] J.E. Neilson, C.M. Woodside, D.C. Petriu, and S. Majumdar. “Software bottlenecking in client-server systems and rendezvous networks.” *IEEE Transactions on Software Engineering*, 21(9):776–782, September 1995.

[29] D. W. Kelton and A. M. Law. *Simulation Modelling and Analysis*. McGraw-Hill, New York, 2nd edition, 1991.

[30] 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.

[31] 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.

[32] T. G. Lewis. “Where is client/server software headed?” *IEEE Computer*, pages 49–55, April 1995.

[33] Luqi and W. Royce. “Status report: Computer aided prototyping.” *IEEE Software*, pages 77–81, Nov. 1991.

[34] J. Neilson. *Parasol Users’ Manual*. School of Computer Science, Carleton University, Ottawa, Canada, K1S-5B6, 3.0 edition, July 1995.

[35] Object Management Group. “Object Services Architecture.” Technical report, Object Management Group, Inc., Framingham, MA, USA, Aug. 1992.

[36] ObjecTime Limited, Kanata, Ontario, Canada. *ObjecTime User’s Manual*, 1994.

[37] Open Software Foundation. *Introduction to OSF DCE*. Prentice-Hall, 1992.

[38] 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.

[39] D. C. Petriu. “Approximate mean value analysis of client–server systems with multi-class requests.” In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*, pages 77–86, Nashville, TN, U.S.A., May 1994. A.C.M. SIGMETRICS.

[40] J. Rolia and K. Sevcik. “The method of layers.” *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.

[41] 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.

[42] 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.

[43] A. Schürr. “Logic based structure rewriting systems.” In H. J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 341–357, 1993.

[44] A. Schürr. “Logic based programmed structure rewriting systems.” *Fundamenta Informaticae*, XXVII, 1996. To appear.

[45] P. J. Schweitzer and I. F. Akyldiz. “Approximate mean value analysis for non-product form networks of queues with job class changes.” In G. Balbo and G. Serazzi, editors, *Computer Performance Evalution*, pages 61–72. Elsevier Science Publishers, 1992.

[46] B. Selic, G. Gullekson, and P. Ward. *Real-time Object-oriented Modeling*. John Wiley and Sons, 1994.

[47] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Publishing Co., New York, NY, 1990.

[48] B. H. Tay and A. L. Ananda. “A survey of remote procedure calls.” *Operating Systems Review*, 24(3):68–79, July 1990.

[49] D. J. Taylor. “A prototype debugger for Hermes.” In *CASCON 92*, pages 29–42, 1992.

[50] W.M. Gentleman. “Message passing between sequential processes: the reply primitive and the administrator concept.” *Software – Practice and Experience*, 11:435–466, 1981.

[51] 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.