Complex Performance Measurements with NICE
(Notation for Interval Combinations and Events)

Murray Woodside, Cheryl Schramm
Telecommunications Research Institute of Ontario,
Dept. of Systems and Computer Engineering
Carleton University, Ottawa
1125 Colonel By Drive, Ottawa, Ontario, Canada, K1S 5B6
cmw@sce.carleton.ca
copyright December 1994

Abstract: This paper presents a notation for extracting complex performance measurements of parallel software from an event log. NICE defines a "complex interval" as a sequence of events in the event log that match a rather general template using an Interval Monitor Process (IMP). The IMP process model combines an extended FSM with parameterized event descriptors to provide a generalized description of a sequence of subintervals each one of which may be measured for duration An interval matching algorithm provides a strategy for distributing the events in the log to multiple concurrent IMPS, thereby handling interval occurrences that overlap and/or share the same event. Practical issues of implementing and using NICE are addressed in a discussion of our performance monitoring tool called Finale.

Keywords : Event monitoring, distributed systems, performance measurements

INTRODUCTION

While investigating the performance of experimental applications on multiple processors, we have found it useful to generalize the usual concept of a time interval as the time from a "start" event to an "end" event. For example, there may be additional events before an interval to define a context for its occurrence; alternatively, additional events may be required during the interval to define its path between beginning and end. In general, we find that a time-interval is embedded in a certain sequence of events, which may be quite complex. The NICE *Interval Definition* uses Extended Finite State Machines (EFSMs) to represent an event sequence as a sequence of transitions between states. As such, the interval definition can be used as the process model for an *Interval Monitor Process* (IMP). The IMP accepts events from the event log that match the interval definition, traversing the FSM of the interval definition until a complete instance of the interval has been identified and its time value computed. This approach allows a powerful open-ended concept of an interval, which includes the following :

- The notion of a generalized interval that matches complex patterns of events, including multiple event sequences, alternative sequences, and overlapping sequences,
- A flexible definition of interval duration in which subintervals may be identified and individually included in or omitted from the calculation of a *time value* for the interval,
- Parameterized descriptions allowing one to broadly define types of behaviours.

NICE is a description language for encoding the semantics of concurrent run-time behaviours in terms of event sequences. NICE comprises the FSM syntax for interval definitions described above, and an algorithm for interval matching that manages multiple interval definitions and/or multiple, concurrent instances of an interval definition. It separates the performance descriptions from the program, supporting an investigative approach to performance monitoring by allowing (1) performance problems to be isolated by repeatedly refining the interval definitions and re-applying them to the same event log, and (2) re-use of the same interval definitions on different traces of the same system. NICE is a general purpose language whose open event format is independent of any implementation language or operating environment. We have written a postprocessor tool called Finale which is a partial implementation of NICE. Finale was originally written for a bus-based multiprocessor system, but has also been adapted to handle logs from a transputer system and from UNIX processes. The Finale event format is used to provide concrete examples throughout the paper.

The paper proceeds as follows. An informal introduction to the major concepts in NICE is given in the first three sections : the notion of a complex interval, the model for an interval definition, and the interval matching algorithm. The fourth section describes the NICE syntax in detail. Two examples follow in the fifth section. The sixth section relates our experiences with Finale. A discussion of related works and our conclusions are given in the final two sections.

A COMPLEX INTERVAL

An interval in NICE differs in two ways from the ordinary notion of the time period from event e_1 to event e_2. First, an interval is a sequence of events (e_1, e_2 ... e_n) that conforms to certain pattern-matching rules (described below). Second, it has a sequence of subinterval durations for subintervals (e_j, e_{j+1}) and an overall *time value* T which is the sum of certain selected subintervals - a selective accumulation.

Two examples illustrate the approach and its flexibility with regard to performance measurements, based on scenarios drawn from a client/server program pictured in Figure I. Three CLIENTs (C1,C2,C3) send requests to two top-level SERVERs (S1,S2). A top-level SERVER acts on the request, replying when complete. On occasion, in order to fulfill a CLIENT's request, a top-level SERVER may send a nested request to the lower SERVER (S3) before replying. All interprocess communication is synchronous with a send-receive-reply event sequence in which the sender is blocked until the receiver replies.

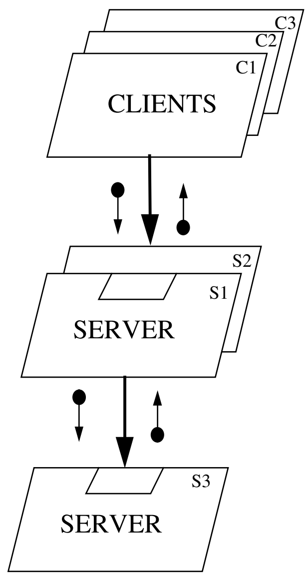

Figure I Client Server Program

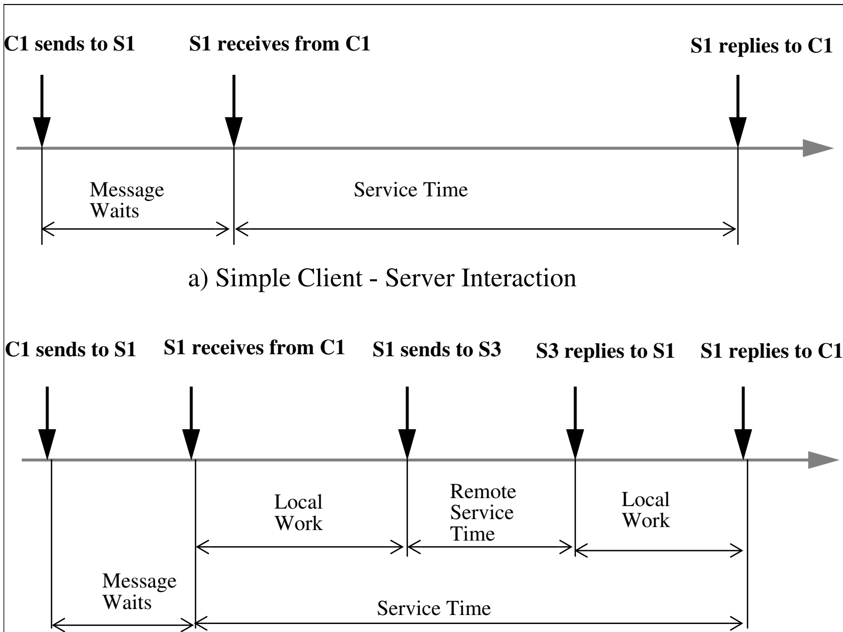

Figure II Examples of NICE Intervals

Example 1 : A Simple Client - Server Interaction (Figure II(a))

The basic client-server interaction occurs when a client sends a request to a top-level server and the server performs the request itself. The interaction is illustrated by the following event sequence :

i = (C1 sends to S1, S1 receives from C1, S1 replies to C1).

As shown in Figure II(a), there are two subintervals in a simple client-server interaction. The "Message Waits" subinterval is associated with the time to signal the request to S1 (possibly over a network) and to wait until S1 is ready to accept a request. The "Service Time" subinterval represents the time taken by S1 to perform any work associated with the request. *Selective time accumulation* gives access to each individual subinterval. Depending on what performance measurements are required, the interval's time value can be defined as either the first, the second, or both subintervals.

Example 2 : A Nested Client-Server Interaction (Figure II(b))

In this scenario, the top-level server must relay a client's request to the lower-level server. The interval is illustrated by the following event sequence :

i = (C1 sends to S1, S1 receives from C1, S1 sends to S3, S3 replies to S1, S1 replies to C1)

As shown in Figure II(b), the "Service Time" subinterval identified in Example 1 now comprises three subintervals that can be used to determine the nature of the work done. By summing the two subintervals labelled "Local Work", one can distinguish between the work done by the server locally and the time spent in rendezvous with the lower-level server (Remote Service Time). The "Remote Service Time" is itself an interval with the same structure as the local request, and could be broken down into the "Message Waits" and "Service Time" subintervals.

Both examples emphasize that a NICE interval has two aspects : an event sequence and a time value defined by selective time accumulation.

A NICE INTERVAL DEFINITION

An *interval definition* is a template that specifies an event sequence and a time value. In addition, we have found it valuable to generalize an interval definition further to support parameterization, complex events patterns and concurrent occurrences.

Parameterization : The intervals depicted in Figure II involve specific processes : C1 and S1. Instead of an interval definition that specifies C1 and S1, it is more useful to capture interactions between any client-server pair using parameters. Parameters provide the facility to specify a set of events that differ only at their components levels. They generalize the notion of an interval definition from identifying a particular event sequence to specifying a type of event sequence.

Complex Event Patterns : In Examples 1 and 2, the events must all occur in the given order. In more complex patterns, a behaviour may also involve alternatives (**or**) or indeterminacy (**and**). By indeterminacy, we mean a set of events that must all occur, but in any order. This may arise, for example, in networked applications where variable communications delays may alter the event sequence for non-connected events. Alternative event sequences are required when there are more than one possible set of events that belong to the same type of event sequence.

Concurrency : Given a complex and parameterized interval definition, there may be overlapping occurrences involving different participants. In Examples 1 and 2, there will be multiple concurrent occurrences whenever multiple clients send a request at the same time to either the same or different servers.

A variety of notations would be suitable. One which immediately suggests itself is the regular expression [1]. However, we could not find a suitable representation for selective time accumulation within regular expressions, so we abandoned them for a state-machine notation. The state machine has equivalent power for defining sequences and is (we feel) easier to annotate for time accumulation. The essential point is not the state-machine notation itself, but the concept of an event-driven process (called an IMP below) that can react to an input stream of events, and recognize an interval as a series of events embedded in that input stream.

The IMP Model

As just described, a very general notion of an "interval" can be obtained by associating an interval definition with a process that can detect an interval instance within the event stream. The process is controlled by an extended Finite State Machine (EFSM) which embodies the interval definition. We call the process an IMP, or *Interval Monitor Process*. An IMP instance is created when a starting event for an interval definition is detected. Then, in a given state, it can accept events that are defined to trigger transitions from that state; it ignores other events. On transitions, it does time accumulation, if indicated. When it reaches a terminal state, the data for the interval instance is complete and the IMP instance terminates. We think of the IMP as parsing the event stream during the process of interval matching, except that, unlike a parser, it ignores events that it cannot match.

The EFSM for the interval definition is defined by states and transitions in a conventional way [2]. A sample interval definition is shown in Figure III . It captures the "Remote Service Time", which is the amount of time a client's request is serviced by the low-level Server, a subinterval within a nested client-server interaction (Figure II(b)). The interval definition is designed to eliminate cases with no nesting (via ABORT) and to end after one nested interaction; the time value comprises the duration of the subinterval of State 3 only. The state and transition are now examined in turn. We postpone some syntax details until the next section.

The IMP is in a *state* during a subinterval whose start and end points are defined by the incoming and outgoing event transitions, respectively. An interval definition is made up of an initial state, one or more subinterval states, and an optional[1] terminal state, where the initial and terminal states represent the time before and after the interval. States are numbered from 0 to MAXINT, with State 0 reserved for the initial state. NICE provides three other reserved states:

1. **COMPLETE** : The terminal state.
2. **ERROR :** An IMP instance is sent to this state when an event sequence that should never occur is identified. This state is useful for debugging interval definitions.
3. **ABORT** : Used in so-called *tentative* interval definitions that include both allowable event sequences and disqualifying event subsequences. An IMP instance tentatively tracks a valid event sequence; if a disqualifying event is encountered, the IMP instance is aborted. For instance, in Figure III, all client-server interactions are tracked but those which are not nested (ie, which don't include "same SERVER sends to S3") are aborted. The **ABORT** state is distinct from the **ERROR** state because disqualified intervals are not considered erroneous. Instead, **ABORT** provides a clean exit.

A transition between states identifies either the specific event or a set of events that link the two states. The transition from the initial State 0 is called the triggering event. As shown in Figure III, a state may have multiple outgoing transitions, representing alternative event sequences. A transition may loop back to the same state to either capture repetitive event sequences or to ignore events. However, a loop may never include a transition back to initial State

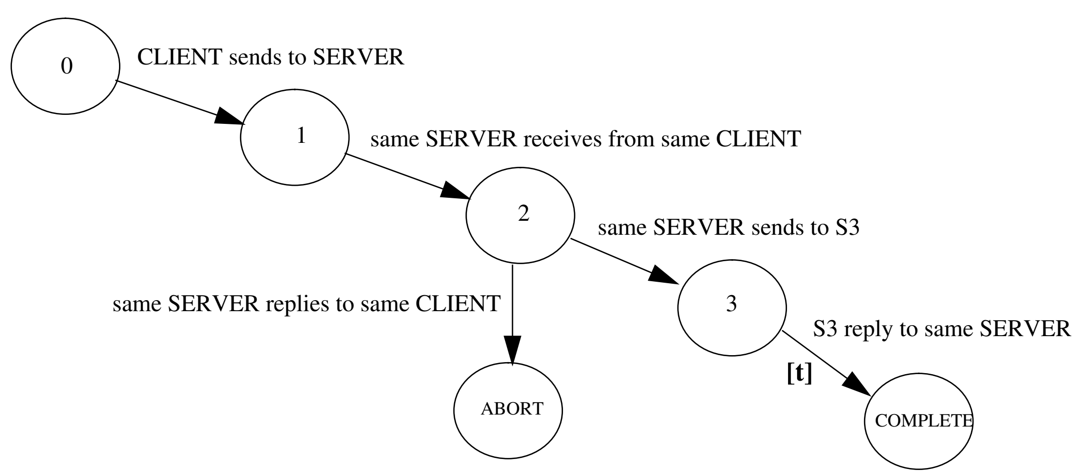

Figure III : Interval Definition for "Remote Service Time"

1. The optional terminal state is seen later in connection with open intervals.

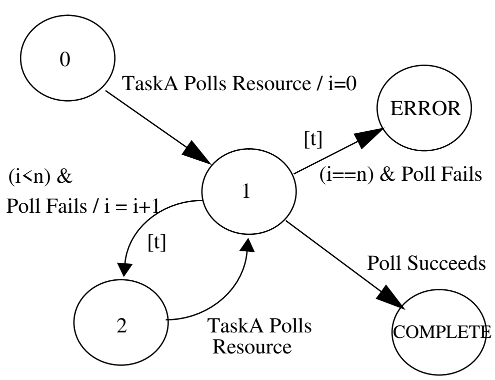

Figure IV Polling Time

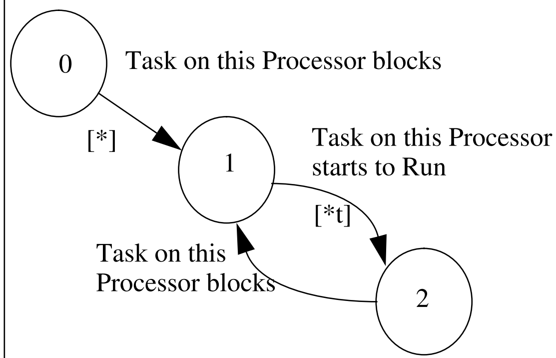

Figure V : Processor Idle

0 because State 0 is semantically "before" the interval.

The transition definition can be optionally extended with a condition predicate and with an action, both of which operate on variables associated with the FSM. The condition is ANDed (&) with the transition event while the action is performed when the transition is triggered (/). A trivial example is a loop counter by which an IMP instance is terminated if the loop in the FSM is traversed more than *n* times (Figure IV) .

Associated with each state is a *subinterval duration variable* (t_{state}) containing the duration of the subinterval represented by the state. Special actions, called duration qualifiers, govern the accumulation of each t_{state}. The qualifier **[t]** is applied to an outgoing transition to indicate that the subinterval terminated by the transition is to be added to t_{state}. t_{state} is accumulated each time the state is exited on a qualified transition. Because State 0 represents the time before an interval, transitions from State 0 cannot be qualified by **[t]**; instead, the triggering transition implicitly initializes all subinterval duration variables to zero. When the IMP terminates, each t_{state} is retained and the interval's time value is computed as the sum of all t_{state}. In Figure III, because **[t]** is used only on the transition from State 3, the interval's time value will include only the remote service time : from the SERVER's send to S3 until S3's reply.

The **[t]** qualifier is applied to the transition, not the state, because the accumulation of a subinterval's time value can be transition-dependent. Return to the interval definition in Figure IV. This interval definition describes the time spent by a task polling a resource (possibly over a network), **excluding** any time spent by the task in doing other work between failed polls. An IMP instance will track the number of failed polls (i), adding the duration of each to the interval's time value (transition from State 1 to State 2) until successful. If the number of failed polls reaches some limit (n), the IMP instance terminates and an ERROR is flagged. Due to the absence of a [**t**] qualifier, the time taken for the final successful poll is not included (State 1 to COMPLETE), nor is the work done by TaskA between polls included (State 2 to State 1). To include either of these as a wider measure of polling time, the [**t**] need only be added to these transitions.

A final question about time accumulation is how to treat intervals that are incomplete when the event log itself ends. We make the interval definition force a transition either to COMPLETE or to ABORT. The interval definition is *open* if it goes to COMPLETE (so that partially finished intervals are included) or it is *closed* if it aborts (so that they are ignored). In this work, open interval definitions are labelled by attaching a qualifier **[*]** to the triggering transition from State 0. They are useful for gathering data about the entire experiment or to monitor system states. For example, the interval definition in Figure V measures the time for which a processor is idle. State 1 represents idle subintervals while State 2 represents busy subintervals. As with many open interval definitions, there is no COMPLETE state, so an IMP instance will continue until the end of the event log. At this point, the duration of the final subinterval is added to the t_{state} for the current state, if that state has an outgoing transition qualified with **[*t]** (as does State 1 in Figure V) . Closed intervals are typically used to monitor activities where the average behaviour of all the completed interval instances is of interest. Incomplete instances are ignored because their prematurely-terminated time values will skew any mean calculations. Because there is no **[*]** on the triggering transition in Figure III, it is a closed interval.

THE INTERVAL MATCHING ALGORITHM

Interval matching is the process of identifying event sequences in an event log that match the interval definitions (called interval instances), and computing their time values. The *interval matching algorithm* provides a control strategy to manage multiple concurrent IMP instances. The algorithm initiates and aborts the IMP instances, and distributes the incoming events from the event log to the IMP instances. The algorithm must consider :

- How to decide that an event begins a new interval instance, requiring a new IMP instance.
- How to decide to which IMP instance(s) to give the next event, when there are several IMPs instantiated.

A very general policy is needed to resolve all the possible situations. For example, the same event might trigger a transition in more than one IMP instance, which might be either multiple instances of the same interval definition or instances of different interval definitions. This might be intentional. Consider an interval definition from an event E to the next event E. Once started, there will always be one active IMP instance of this interval definition; each event E will trigger the completion transition of the active IMP instance and then trigger the initial transition of a new IMP instance.

With this in mind we have adopted a priority rule for binding events and intervals with priority to the interval instance that started first. The interval matching algorithm is :

```
For each event (e_i)  in the event log in turn  {
        For each interval definition (IDEF_j)  {
            If (there exist any active IMP instances for IDEF_j ) then
                    Offer e_i to each one, beginning with the oldest instance,
                    stopping when it is accepted by one IMP instance.

            If (no active IMP instance exists) or
               (none accepted e_i) or
               (an active IMP accepte e_i and was terminated by it)   then
                    If (e_i matches any of the triggering transitions of IDEF_j)  then
                                Create a new IMP instance for IDEF_j
            /** Timeout clause **/
        }
}
```

Each event is applied separately to every interval definition. This approach allows a single event to be bound to more than one type of interval. Each event may be accepted by only one IMP instance per interval definition. When multiple possibilities exist, the event is applied to the oldest instance. An event is applied to an active IMP before the creation of a new IMP instance is considered. If an event terminates one IMP instance by sending it to either COMPLETE, ABORT or ERROR, it is allowed to start a new IMP instance of the same interval definition. This exception is used for cyclic interval definitions in which the starting and terminating events are the same.

Correctness Checks

In developing interval definitions with the generality of NICE, errors may be made. The user may need help in checking for errors, or in finding their source. We wish to ensure that : (1) every event sequence in an event log that matches an interval definition is found, and (2) every instance identified is indeed a correct interpretation of an interval definition. Two sources of errors have been identified in our experience:

1. The event log itself may be erroneous. In a network application, depending on the clock synchronization method in place, events may occur out of order in the event log. Alternatively, events may be lost when the event rate exceeds the monitoring capabilities, leaving gaps in the event sequences and giving the appearance of out-of-order events (for example, given that B always follows A, if an event A is lost, its partner event B will appear to precede the next occurrence of A).

2. The interval definitions may be incomplete, either because a possible event sequence was not accounted for or because the specification of the transition event descriptor was not restrictive enough.

The two error sources manifest themselves in similar ways. Consider the interval definition in Figure VI and the three accompanying event logs. The same interval instances are extracted whether the error source was a badly formed interval definition (boxed transition is missing) or a lost event (boxed B's are missing events).

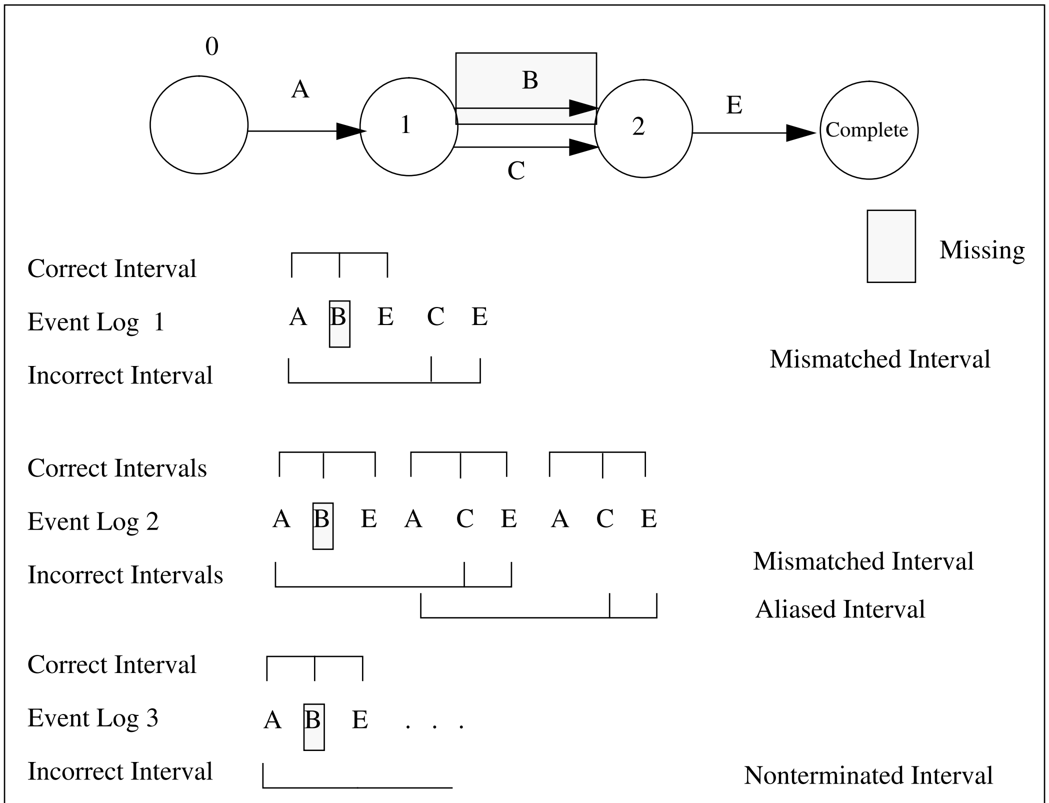

Figure VI Error Sources and Their Impact on Interval Bindings

A mismatched interval occurs when an event is applied to a wrong instance of the interval, as in the first event log where the interval extracted is (ACE) rather than (ABE).

An aliased interval occurs when the interval overlaps a mismatched interval of the same interval definition. With two IMP instances of the same interval definition waiting for the same event, the interval matching algorithm will apply the event to the oldest IMP instance. In the second event log, a phase shift occurs with the second event C because the first event C is paired with the first event A, rather than with the event A immediately preceding. This error is a serious because it is propagated as long as two active IMP instances of the same interval definition overlap; in our example, as long an event A is followed by an event C.

A nonterminated interval occurs instead of a mismatch when the event sequence does not occur again and the IMP instance is left in an orphan state because events that should match do not.

NICE provides several correctness checks for the interval matching algorithm. They are intended to verify the events and intervals captured, not the behaviour of the underlying programs described by the intervals.

1. The reserved **ERROR** state may be used to explicitly define an error condition as part of the interval definition. It is useful for tightly-defined behaviours, when the completeness of the event log is suspect and for debugging interval definitions.

2. Timeouts on the subinterval durations provide a warning mechanism for nonterminated interval instances whereby active IMP instances that are not triggered by an event within a selected time limit are flagged and (optionally) abandoned. Timeouts are implemented as an additional clause in the interval matching algorithm, which is applied after each event to all active IMP instances of all interval definitions to test the time elapsed since their last transition.

3. Correctness Predicates define conditions or properties that are monitored for consistency during the interval matching process. Correctness predicates may be defined globally as part of the interval matching algorithm. Alternatively, correctness predicates particular to a given interval may be defined locally as part of the interval definition. Two examples of global predicates follow. Each is explained generally and must be tailored to an application

   Example 3 : Connected Events

   Connected events [3] are those which always occur in a particular order; for instance, in the send-receive-reply messaging paradigm, each receive is always paired with a reply that follows it. Given a list of connected events, a correctness predicate would monitor the event log, flagging any inconsistency as a possible missing event.

   Example 4 : Exclusivity of Intervals

   The exclusive property of an interval definition limits the concurrency of its instances and is intended to capture aliased intervals. It requires that multiple instances of the same interval definition be exclusive : that is, that the same event from the event log should not be acceptable by more than one instance of the same interval definition. Looking at the example for aliased intervals in Figure VI, the exclusivity predicate would issue a warning that the first event C could apply to two active IMP instances.

The correctness checks provide a measure of confidence in the interval statistics calculated. They identify potential errors in the interval matching process and then allow the process to continue. If an error is detected, it is still up to the user to decide whether an error indeed occurred, and if so, determine the source as a missing event in the event log, a misunderstanding of the system's behaviour, or as a badly formed interval definition.

THE SYNTAX OF NICE

The preceding sections have described the principles and underlying model of NICE. In this section, the syntax of NICE is given, with concrete examples drawn from our own implementation called Finale. There are two constructs in NICE : event descriptors and interval definitions. Both constructs are templates to be matched against an event log : event descriptors match single events, interval definitions match event sequences. To describe either construct first requires a representation for the event record in the event log. It is here that we begin.

Parameters and Values

NICE expects as input an event log made up of a variable-length list of event records. As a general-purpose language, NICE provides a flexible and extensible representation for an event record that an implementation must customize to its particular event format. The representation is a parameter list.

Each event record in an event log is a set of fields, where each field contains a value of a specific *value type*. Some systems will have a fixed event format in which the number of fields is always the same. More often, systems will have a variable event format in which the number of fields and the value type of each field depends on the event record being logged. For flexibility, NICE attaches a parameter with an associated value type to each field *that may occur in any event record* in the log. After reading in from the event log and decoding, and before matching to intervals, an event record is represented as a vector with a value for each parameter.

To read an event log, NICE requires : (1) a definition of the parameters for the given implementation including the value type associated with each parameter, (2) and a definition of the value types, including a set of valid values for each value type, how they are encoded in the event log (eg. bitstring), how they are referenced by NICE (eg. a character string), and a mapping between the two. While NICE imposes no requirements on the parameters defined by an implementation, use of NICE for performance measurements implies minimally two parameters (an event code and a timestamp) and typically includes a third parameter to identify the software component logging the event. Under these assumptions, NICE provides an extensible and user-configurable definition of an event record as a vector of three standard parameters plus an open-ended list of implementation-dependent parameters which we call attributes:

$$\text{Event} = (\text{T, E, Ti, Attr}_1 \ldots \text{Attr}_n)$$

where

T : identifier for the component (eg. task, processor, procedure) logging the event

E : an event code

Ti : the timestamp value. NICE assumes globally consistent times; in a network, some preprocessing may be necessary to estimate the offsets of different clocks

Attr_i : a data value containing additional information about the event. Possibilities include correspondent identifiers, message types, message lengths, task entries or ports.

NICE does not impose an order on the parameters.

As a concrete example of an event vector, consider the definition of a Finale event.

$$\text{Finale Event} = (\text{T, E, C, P, CP, Ti})$$

where :

T : the identifier of the task logging the event (task_type)

E : an event code (event_code_type)

C : the identifier of the correspondent task, with a significant value only for events involving messages. In a "send message", it is the destination of the message; in a "receive message", it is the source. (task_type)

P : an identifier for the processor of task T (processor_type)

CP : an identifier for the processor of task C (processor_type)

Ti : the timestamp value in terms of clock ticks (timestamp_type)

Finale has four value types. The event_code_type maps a set of valid event codes encoded as 8-bit unsigned integers into strings such as "Send", "Receive" and "Reply". Likewise, the task_type maps a set of known task identifiers to task names, such as "S1" or "C1". The processor_type and timestamp_type are both simply integers.

As well as value types, we can define a class as a set of values of a single value type, which allows us to group values together in specifying types of events. For instance, two task classes could be defined for the client-server program in Figure I : task class SERVER=(S1,S2,S3) and task class CLIENT=(C1,C2).

The Event Descriptor Syntax

An *event descriptor* is a template that can be matched against an event record from the log. It is the primitive construct in NICE and is used both for the monitoring of single event occurrences and as a component in the interval definitions. We introduce the syntax for an event descriptor by example, and then formalize the notation.

Following is a Finale event descriptor that will match any event record in which "S1 sends to C1".

(T=S1, E=Send, C=C1)

The Finale event descriptor is a collection of expressions in the Finale parameters defined above, where each expression describes the value to be matched for a parameter. The above example uses *data expressions* in which the exact value of a field is specified by the expression "Parameter = Value" (eg. T=S1). It is also a partial description of an event record because the Finale parameters P, CP and Ti are not given in the event descriptor; any value for these fields will be accepted as a match.

To generalize the event descriptor, we add a *class expression* where a class of values for a parameter is given by the expression "Parameter < Class". To capture all event records in which any SERVER sends to any CLIENT, we use the following event descriptor :

(T<SERVER, E=Send, C<CLIENT)

Formally, an event descriptor is a collection of expressions in the parameters defined for the event record where (1) there must be at most one expression for any one parameter and (2) unspecified parameters are considered to be wildcards. The possible expressions are listed below, with alternative definitions placed on successive lines.

expression :	data_expression	(Identifies a single value)
	class_expression	(Identifies a set of values)
data_expression :	parameter = value	(Data Equality)
	parameter != value	(Data Inequality)
class_expression :	parameter < class	(Class membership)
	parameter !< class	(Class non-membership)

The parameters, values and classes are implementation-dependent.

Example 5: Some sample Finale Event Descriptors

Finale Event Descriptor	Semantics
(P=3, E=Running)	Any task on Processor 3 becomes RUNNING
(T=S3)	Any event logged by the S3 task
(T<CLIENT, E<BLOCKED)	Any CLIENT task becomes blocked
(E=Send, CP=3)	Any task sends to any task on Processor 3
(E=Send, C<SERVER)	Any task sends to any SERVER task
(T<SERVER, E=Send, C=S3)	Any SERVER task SENDs to S3

Event matching is the process of matching an event from the event log with an event descriptor. An event record matches an event descriptor if all the fields in the event record match the corresponding parameter expression in the event descriptor. The rules for matching an individual field are :

1. If the expression in the event descriptor is a data expression, the value for this field in the event record must match the value given in the expression.
2. If the expression in the event descriptor is a class expression, the value for this field in the event record must be a member of the given class.
3. If there is no expression in the event descriptor for a field in the event record, (ie. wildcard), any value for this field in the event record will match.

The Interval Definition Syntax

Any standard notation for extended finite state machines can be used to define the EFSM that constitutes an interval definition. Below, we summarize how the EFSM does this by describing the tabular notation used in Finale.

An interval definition has four parts : an interval name, variable declarations, the FSM itself and correctness predicates. Variable declarations are needed for user-defined variables used in conditions or actions; subinterval variables and the context variables (introduced below) are declared implicitly. The FSM is a set of state-transition statements of the following form, in which mandatory parts are bold face, and optional parts are in italics :

State *Qualifier* **=>** *Condition* & **Event** / *Action* **=>** **Next State**

The FSM must conform to the following rules :

1. State is a number from 0 to MAX_INTEGER. State 0 is reserved for the initial state; others are called user-defined states. Associated with each user-defined state *i* is a subinterval duration variable t_i which is implicitly initialized to zero by the triggering transition.
2. Next State is either a user-defined state that is used in the State column or one of three defined terminal states : COMPLETE, ERROR, ABORT.

3. Every FSM must have an initial state and at least two states, excluding ERROR and ABORT.

4. A transition from a state is minimally an event descriptor. There must be at least one transition from every non-terminal state (ie. State 0 and user-defined states). There may be multiple transitions to and from a state. No transition may go to State 0.

5. A transition may be extended with a condition and action(s). A condition is any legal[1] boolean expression of the FSM variables. An action is any legal assignment to a FSM variable. FSM variables include subinterval duration variables, context variables and user-defined variables.

6. The qualifiers available are [*] and [t], or merged together as [*t]. [t] cannot be applied to a transition from State 0.

   An open interval must have a [*] qualifier in at least one transition from State 0, and a [*t] qualifier on at least one transition from at least one other state.

   A closed interval must have at least one transition from at least one state qualified by [t] (otherwise, no time calculations will be done).

7. A closed interval must have a transition to a COMPLETE state; an open interval need not.

Example 6 : Finale Interval Definition for Polling Time (Figure IV)

Interval Name : Polling Time

int i;

	State *Qualifier* =>	*Condition* & **Event** / *Action* =>	**Next State**
Line 1	0	(T=TaskA, E=Polls) / i = 0	1
Line 2	1	(T=TaskA, E=Poll_Succeeds)	COMPLETE
Line 3	1 [t]	(i<5) &(T=TaskA, E=Poll_Fails) / i = i+1	2
Line 4	1 [t]	(i==5) & (T=TaskA, E=Poll_Fails)	ERROR
Line 5	2	(T=TaskA, E=Polls)	1

"Polling Time" is a closed interval, completed when a poll succeeds (Line 2) or flagged as an error if the number of failed polls (i) exceeds five (Line 4). The variable i is initialized to zero by the triggering event (Line 1), and incremented each time a poll fails (Line 3). It is used in boolean expressions to limit the number of failed polls (Line 3 and 4). The interval's time value is accumulated only for failed polls (Line 3 and 4), not for successful polls (Line 2) nor for the time elapsed between successive polls (Line 5).

1. Operators for comparing and assigned FSM variables must be provided by the implementation. In the simplest case, all FSM variables are integers so that "+" means integer addition and "==" means "integer equality". If an implementation allows variables to be declared as character strings, the implementation also has to provide operators for comparison and assignment (eg. the strcmp() and strcpy() utilities in C).

In the example above, the event descriptor in every transition identifies a single task called TaskA. Alternatively, we could define a class called TASKS of which TaskA is one member and then replace Line 1 with the following to capture the polling time for any task in the class TASKS :

0	(T<TASKS, E=Polls) / i = 0	1

This form, however, incurs a problem of "context" in subsequent transitions . For instance, transitions from State 1 and 2 should restrict events to those that involve *the same task as that which triggered the current IMP instance* in order to avoid mismatched intervals when there are several tasks polling simultaneously. For this, we introduce context variables, a particularly powerful feature of NICE. For each parameter defined for the event record, a context variable of the same value type is declared. For example, Finale has a six context variables : T_same, E_same, C_same, P_same, CP_same and Ti_same. When an IMP instance is created, these variables are initialized to the parameter values for the triggering event (or to NIL if the triggering event does not use the parameter). Example 6 is re-written below using task classes and context variables.

Example 7 : Finale Interval Definition for Polling Time (Figure IV) - Class Version

Interval Name : Polling Time

int i;

State [*Qualifier*]	**=>** *Condition* & **Event** / *Action* **=>**	**Next State**
0	(T<TASKS, E=Polls) / i = 0	1
1	(T=T_same, E=Poll_Succeeds)	COMPLETE
1 [t]	(i<5) &(T=T_same E=Poll_Fails) / i = i+1	2
1 [t]	(i==5) & (T=T_same, E=Poll_Fails)	ERROR
2	(T=T_same, E=Polls)	1

Context variables provide the notion of persistence between events within an interval. By default, context variables contain the instantiation of the triggering event descriptor - the parameter values of the triggering event. Like other FSM variables, context variables can be reassigned by a transition's action, which is typically done when a context variable is undefined in the triggering event or when the triggering event is simply a precondition for a later event which is in the actual subinterval of interest. In general, the context variables contain the identifying values for an interval instance. For this reason, they can be thought as an identifier for an IMP instance that can be used when compiling results. For instance, Finale uses the context variables to group IMP instances of the same interval definition according to their triggering event. In this way, statistics can be compiled not only for the interval definition as a whole, but also for groups within this interval definition. In the example above, Finale would compile performance measurements for each individual task that polls, and then combine the tasks' time values to provide an overall measurement for the polling time.

Local correctness predicates were discussed in the section titled "The Interval Matching Algorithm". These are boolean expressions associated with a single interval definition, relating the events making up an interval instance, their context variables, subinterval durations and user-defined variables. Like the condition on a transition, their form depends on the value types defined for the parameters and the user-defined variables.

EXAMPLES

The Client/Server Example

We return to the client/server scenario to show how the interval matching algorithm works, and how multiple interval definitions can be combined to produce a composite of performance measurements. A comparison is made of the service times for simple and nested client-server interactions (Figure II), relating their relative frequency and time values. The service time is the period from when the top-level SERVER begins receiving a request until it has returned the reply to the requesting CLIENT. In addition, the cycle times of the SERVERs are monitored as an indication of the servers' workloads, and also as an illustration of a cyclic interval definition. The three interval definitions as implemented in the Finale tool are presented below.

Interval Name : Simple Service Time

State	*Qualifier* **=>**	**Event**	**=> Next State**
0		(T<SERVER, E=Receive, C<CLIENT)	1
1	[t]	(T=T_same, E=Reply, C=C_same)	COMPLETE
1		(T=T_same, E=Send, C=S3)	ABORT

Interval Name : Nested Service Time

State	*Qualifier* **=>**	**Event**	**=> Next State**
0		(T<SERVER, E=Receive, C<CLIENT)	1
1		(T=T_same, E=Reply, C=C_same)	ABORT
1	[t]	(T=T_same, E=Send, C=S3)	2
2	[t]	(T=T_same, E=Reply, C=C_same)	COMPLETE

Interval Name : Server Cycle Time

State	*Qualifier* **=>**	**Event**	**=> Next State**
0		(T<SERVER, E=Receive)	1
1	[t]	(T=T_same, E=Receive)	COMPLETE

All three interval definitions are closed. The simple and nested service times are *tentative* interval definitions in which an event sequence is aborted if a disqualifying event is encountered. In the case of the "Nested Service Time", the same event drives transitions from different states (from State 1 to ABORT, and from state 2 to COMPLETE), meaning that the order of events is important. The server cycle time is a cyclic interval in which the same event occurrence both completes an active IMP instance and triggers a new one.

The three interval definitions share common events so that a single event occurrence triggers transitions in each one. Whenever a server receives from a client, the IMP instance currently active for the cycle time is completed, and three new IMP instances are created - one each for local service, remote service and cycle time. The local and remote IMP instances will concurrently track the event sequence for the client/server pair until one is disqualified and the other completes.

Using the context variables to identify the interval instances, the service times can be extracted per server, per client, per client/server pair or as a whole. In addition, the interval definitions can be combined to provide secondary information; for instance, using the number of interval instances as counters, the request ratio to the second level server could be calculated as the ratio of remote to local requests.

Parallelism Measurement with Subinterval Durations

In this section, the calculation of average parallelism is used to demonstrate the power of selective time accumulation. The average parallelism of a program is defined as the average number of processors that are busy during the program execution, given by :

$$A = \frac{\sum_i i t_i}{t}$$

where

t_i is the total time that i processors are busy

t is the total time of the program execution, where $t = \sum_i t_i$

The following interval definition monitors the parallelism of three processors, but can be extended as needed. A state is labelled with each value of *i* so that the subinterval duration (t_i) is the total time when *i* processors running. The interval's time value (the sum of all subinterval durations) then gives *t*.

Interval Name : Degree of Parallelism

State	*Qualifier* **=>**	**Event**	**=> Next State**
0[*]		(E=Start_Logging)	1
4[*t]		(E<START_RUNNING)	1
4[t]		(E<BLOCKED)	ERROR
1[*t]		(E<START_RUNNING)	2
1[t]		(E<BLOCKED)	4
2[*t]		(E<START_RUNNING)	3
2[t]		(E<BLOCKED)	2
3[*t]		(E<START_RUNNING)	ERROR
3[t]		(E<BLOCKED)	2

Two events classes are used to signal changes in the number of processors running : START_RUNNING events (all events signalling when a task begins running) and BLOCKED events (all events signalling when a task blocks and relinquishes the processor). To begin with a known value of parallelism, logging is started at a point when the initialization program holds all processors blocked except its own, and the IMP is then triggered by the **Start_Logging** event. From this point, the IMP increments to the upper neighbour state upon a START_RUNNING event, and decrements to the lower neighbour state upon a BLOCKED event. State 4 (or n+1 where n=number of processors) represents the idle case where no processors are running. Two illegal event sequences are defined for the limit cases : (1) in State 3, another processor cannot start running because all processors are already running, and (2) in State 4 (the idle state), no processor can block because all processors are currently idle. The interval is open and will continue for the duration of the program execution. For this reason, there is no transition to COMPLETE from any state. Whenever the end of the event log is reached, the final subinterval will be included in the time value calculations because a transition from each state is qualified with **[*t]**.

EXPERIENCES WITH FINALE

Finale is an implementation of NICE. It was originally written for a multiprocessor system running a real-time message-passing kernel called Unison[4]. However, Finale could be used for any concurrent message-passing system, as long as the system's events were mapped to Finale's event record format. Currently, Finale interval definitions are only basic FSMs, meaning that conditions and actions associated with a transition are not supported. Consequently, context variables always describe the triggering event. There is selective time accumulation but the separate t_{state} values are not reported, only their total. Finally, Finale does not support any inequalities, only "=" and "<". The interval definitions in the Section "EXAMPLES" are currently supported, although, without the separate subinterval durations, the interval definition for the "Degree of Parallelism" is not very useful; a different approach is used with the current version of Finale to get a parallelism profile.

The flow of data in Finale is shown in Figure VII . Most of the data is stored in files because of the amount of information involved, and because the files serve as documentation and promote re-use amongst different performance tests. Figures VIII to IX show the contents of the files for the client/server example of the previous section.

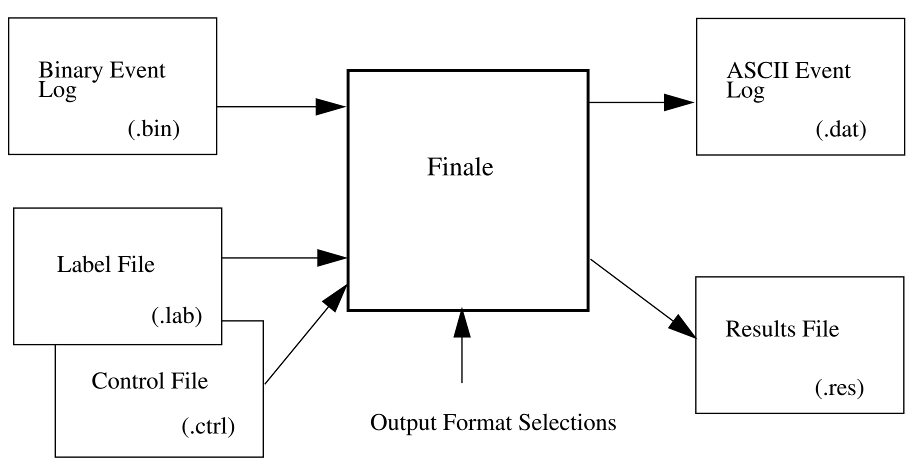

Figure VII The Flow of Data in Finale

The primary input is a binary event log for an application program. The label file contains the value type definitions (Figure VIII). A label is declared for each task and event to be processed. It comprises an ASCII string name for the binary task identifier or event code stored in the binary event log , and the name of the (task or event) class of which this component is a member. The control file contains the interval definitions, event counters (event descriptors that request a count of single event occurrences in the event log), and some optional processing controls (Figure IX). *Event filters* (*keep* or *remove*) reduce computation times by providing a minimal event log containing only events that will be used in the postprocessing. By default, processing proceeds from the first until the last event occurrence in the log. *Limit events* (*start* and *stop)* control the portion of the filtered event log used by the postprocessor. For instance, the *start* event is often used to ignore a startup period.

The outputs of Finale include an optional ASCII version of the event log and a results file (Figure X). A complete listing dumps all the information regarding the event counts and the interval instances extracted from the event log. For each interval instance, the starting and ending timestamps and its time value are given. Performance statistics compile information about the interval instances, including the minimum and maximum time value, the mean time value and its standard deviation, and the relative time value compared to the total time of the event log. Histograms offer distribution plots of the time values of the interval instances. For both statistics and histograms, the interval instances can be grouped together by their context variables to investigate effects at a finer granularity than the interval definition. For example, in Figure XI, the simple (and similarly, the nested) service times are found for each client/server pair. The listing shows two groups for the Simple Service Time - (S1/C1) and (S1/C2)- each with a number of occurrences. Statistics are compiled individually for each group, and then summary statistics for the interval definition as a whole are compiled by combining the groups. The classification capability currently supported by Finale is primitive, only allowing one to group together instances of the same triggering event (ie. same task, event and correspondent). Extensions to the facility are required to provide finer classification by task, event, correspondent or combinations thereof.

A number of implementation issues were encountered in developing Finale. Currently, Finale allows a task or event to belong to only one class. This restriction avoids the ambiguity encountered when overlapping classes are used in event descriptors within the same interval definition. We have found, however, that subdividing a class into (non-intersecting) subclasses would be useful, and will be expanding Finale to allow this.

Of more import is the assignment of labels. Finale uses static assignment of labels. For event codes, this approach is satisfactory because kernel event codes are pre-defined in the TEMPO event monitor and user event codes are logged by an explicit probes in the source. For task identifiers, however, the approach is limiting. Unison task identifiers are made up of a static task class identifier (eg. 127) and a dynamic instance identifier (a pointer to the task's control block). Finale uses the static part because it is defined in the source code, making the association of labels to task identifiers easy. It does, however, limit a task class to having a single instantiation, or else it confuses tasks of the same class. Discernment of multiple instantiations of the same task class requires separate static identifiers for each possible instantiation, as is done in FigureVIII for both the servers and the clients. This solution does not allow dynamic allocation of tasks. Extensions to handle multiple instantiations appear to be straightforward and would not affect the post-processing by Finale. The static identifier would be used as the task class template, and an instance counter would be mapped from each new dynamic identifier, without affecting the runtime. The task name used in this paper would be replaced by name.counter. The counter value of an instance would depend on its order of appearance in the event log, and could vary in different execution runs. Using an event descriptor with "T=CLIENT.1" could not be used to describe a particular task instance with certainty. The assignment of unique names to each instance of a class without involving the runtime remains an open question.

```
identifer | task name            |   class
   --------------------------------------------={
    110   S1                        SERVER
    111   S2                        SERVER
    112   S3                        SERVER
    125   C1                        CLIENT
    126   C2                        CLIENT
    127   C3                        CLIENT
  }=---------------------------------------------

event code | event name          |   class
   --------------------------------------------={
    4     Send                      BLOCKED
    5     Reply                     BLOCKED
    12    Receive                   RUNNING
    20    Start_Logging             LOG
    21    Stop_Logging              LOG
  }=---------------------------------------------
```

Figure VIII Exerpts from the Label File for the Client/Server Example

```
- keep event = (E<BLOCKED)
- keep event = (E=Receive)

- start event = (T<CLIENT, E=Send, C<SERVER)
- stop event = (E=Stop_Logging)

-event count = (T<SERVER, E=Receive)

 Interval Definitions ... (as in the Client/Server Example in the Previous
                           Section)
```

Figure IX Exerpts from the Control File for the Client/Server Example

LISTINGS ********************

Interval = Simple Service Time

Triggering Event = S1(3)&Receive&C1(5)

Number of Occurrences = 7
Minimum Duration Occurrence = 8
Maximum Duration Occurrence = 33

	Start	End	Time Value
0	48872	48882	8
.			
.			
6	49914	49932	16

Triggering Event = S1(3)&Receive&C2(6)

Number of Occurrences = 15
Minimum Duration Occurrence = 10
Maximum Duration Occurrence = 41

	Start	End	Time Value
0	48901	48909	10
.			
.			
14	50401	50437	36

STATISTICS *********************

Interval = Simple Service Time

Triggering Event	#	Min Duration	Max Duration	Total Duration	Relative Duration	Mean Duration	Std Dev
S1(3)&Receive&C1(5)	7	10	33	147	0.0847	21.00	8.04
S1(3)&Receive&C2(6)	15	8	41	376	0.2200	25.06	9.37
...							

SUMMARY STATISTICS (For each interval definition) ********************

Interval	#	Min Duration	Max Duration	Total Duration	Relative Duration	Mean Duration	Std Dev
Simple Service Time	23	8	41	523	0.30	23.77	11.12
Nested Service Time							
Server Cycle Time							

Figure X Exerpts from a Sample Finale Results File for the Client/Server Example

RELATED RESEARCH

The notion of a "compound event" is a closely related concept to our interval definition, seen in the literature. Bemmerl et al, in describing the TOPSYS monitor [5], defined compound events as being made up of a set of atomic events structured by three operators :

- one of a set of events occurs ($e_1 \vee e_2$)
- one happens, then the other ($e_1 \rightarrow e_2$)
- both happen in any order ($e_1 \wedge e_2$)

The compound events are used by an integrated toolset - including a debugger, an execution visualizer, performance analyzer and a load balancer - as a means of gathering runtime data and presenting multiple views of the data at different levels of abstraction. With expressions for both order and nondeterminism, compound events nicely express parallelity events - those events due to the interaction of multiple components and that normally come in pairs when an event in one component triggers an event in a corresponding component. Event sets and multiple, indeterminate (order-nonspecific) and alternative event sequences are provided for in TOPSYS. Their compound events are unable, however, to handle unbounded loops, and it is unclear whether overlapping instances are covered. Furthermore, TOPSYS cannot provide selective time accumulation. A compound event is treated as an atomic occurrence with internal duration. Duration remains a simple delay between an event pair (the beginning and end) and is modelled as an attribute of a compound event to be computed by the monitor upon the compound event's occurrence.

Compound events are also seen in Rapide[6,7]. Rapide is a language for prototyping concurrent systems that is based on partially ordered event sets (*posets*). Concurrent modules monitor the system poset (using well-defined scoping rules), triggering on an event pattern and generating output events that are added to the system poset and that serve as input to other component modules. The event pattern is written in an event pattern language derived from Luckham's earlier work on TSL[3]. It is an expression of atomic events constructed from four operators : or , and, causally followed by, and causally independent. In addition to triggers for the concurrent modules, event patterns are used to specify constraints on the prototype behaviour, describing a pattern of events that should or should not appear in an poset.

Clearly, Rapide and NICE have a common basis in the analysis of complex event patterns. Both use concurrent processes (Concurrent Reactive Processes in Rapide and IMPs in NICE) to match an incoming event stream with an event pattern. The process model allows for multiple concurrent activations of the same event pattern, and overlapping intervals. The rules for matching are both based on the notion that - in Rapide's terminology - the earliest event in a event stream that can participate in a match is used. Rapide uses placeholders to allow general event descriptions including the notion of data types.

The languages differ, however, in their approach to the syntax for compound events, Rapide using regular expressions and NICE using a FSM notation. The reason lies in their different perspectives. Rapide constraints are designed primarily as a tool for testing and debugging a prototype where the question reduces to "Did this happen". For this, an atomic compound event is suitable. For performance monitoring, NICE extends the notion so an interval is itself perceived as a process with its own internal behaviours and its own concurrency. NICE provides :

- an alternative formulation of the ideas in Rapide and TSL, to further satisfy the needs of performance monitoring for clear flexible descriptions of performance quantities,
- an alternative approach to a definition language, featuring program -independent flexibility and emphasizing the sequential structure of intervals .

As a final note, our approach has something in common with protocol testing, where the protocol tester process must accept the event token stream going to the process under test. However, an IMP can skip over any event in the log, so the questions of determining what interval is present are quite different from the questions posed in testing.

DISCUSSION

Through a partial implementation, NICE has shown itself to be a flexible and powerful notation for performance measurements on parallel software. It allows the user to concisely express complex temporal patterns of events and selectively extract subintervals within these patterns. Wildcard event parameters and dynamic binding of variables to each interval instance are key capabilities. NICE also defines a strategy for distributing events to interval instances, and for handling several interval definitions concurrently. The paper has shown how useful these capabilities are with examples drawn from our experience with a partial implementation running on a multiprocessor.

Essentially, NICE defines an interval by a process model (a finite state machine) that extracts complex event sequences from an event stream. The generality of this process model gives it its power. Events that it cannot accept are simply ignored, in contrast with many other uses of FSMs where unacceptable events indicate errors. The necessity of using multiple instances of the same FSM operating at the same point in the event log and competing for events leads to several practical questions. The paper has discussed how to deal with questions of correct interval definition, problems in debugging erroneous definitions, and how to deal with errors in the event stream, such as missing events.

ACKNOWLEDGEMENTS

This work was funded through grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Telecommunications Institute of Ontario (TRIO). Original coding of Finale was done by Jerome Bacelli. A graphical user interface was written by Sanjay Gera, with advice from Ron Casselman. Practical experiences of working with Finale were provided by Shikaresh Majumdar and Srinivas Vunnava.

REFERENCES

1. Zvi Kohavi; *Switching and Finite Automata Theory*; McGraw-Hill Book Computer Science Series, 1978, Chapter 16.

2. D.J. Hatley and I.A. Pirbhai; Strategies for Real-Time System Specification; Dorset House Publishing Co., Inc., New York, NY 10014, 1987, Chapter 6.

3. D.C. Luckham, D.P. Helmbold, S. Meldal, D.L. Bryan, M.A. Haberler; *Task Sequencing Language for Specifying Distributed Ada Systems;* System Development and Ada, Springer-Verlag, pp. 249-305, 1987.

4. Multiprocessor Toolsmiths Inc.; *Unison User's Guide, Software Release 1.1;* Kanata, Ontario, Canada, November 1987.

5. Thomas Bemmerl, Robert Lindhof, Thomas Treml; *The Distributed Monitor System of TOPSYS*; Proceedings CONPAR 1990 - VAPP IV September 1990; Springer-Verlag, pp. 756-765

6. David C. Luckham, James Vera, Doug Bryan, Larry Augustin, Frank Belz; *Partial Orderings of Events Sets and Their Application to Prototyping Concurrent Timed Systems*; Technical Report CSL-TR-92-515, Computer Systems Laboratory, Stanford University, Stanford, California, April 1992

7. Doug Bryan; *Rapide-0.2 Language and Tool-Set Overview*; Technical Report CSL-TN-92-387, Computer Systems Laboratory, Stanford University, Stanford, California, February 1992.