Run-time Fault Detection in Monitor Based Concurrent Programming

Jiannong Cao, Nick K.C. Cheung, Alvin T.S. Chan
Software Development and Management Lab.
Dept. of Computing, Hong Kong Polytechnic University, Hung Hom., Kowloon, Hong Kong

Abstract

The monitor concept provides a structured and flexible high-level programming construct to control concurrent accesses to shared resources. It has been widely used in a concurrent programming environment for implicitly ensuring mutual exclusion and explicitly achieving process synchronization. This paper proposes an extension to the monitor construct for detecting run time errors in monitor operations. Monitors are studied and classified according to their functional characteristics. A taxonomy of concurrency control faults over a monitor is then defined. The concepts of a monitor event sequence and a monitor state sequence provide a uniform approach to history information recording and fault detection. Rules for detecting various types of faults are defined. Based on these rules, fault detection algorithms are developed. A prototypical implementation of the proposed monitor construct with run-time fault detection mechanisms has been developed in Java. We shall briefly report our experience with and the evaluation of the robust monitor prototype.

1. Introduction

The monitor construct [6, 8] has been widely used as a high level process synchronization mechanism in modern operating systems, as well as in concurrent programming languages as a language level construct [1, 5, 7, 10, 14]. It is employed as a means to control the flow of process interactions, which is one of the essential sources of difficulty for both design and validation of concurrent programs. As such, it is very important to ensure correctness and reliability of monitor operations. The original version of the monitor construct, however, is specified as a device for defining shared abstract objects and for scheduling accesses to them, without provision for handling malfunction of monitor procedures and run time errors that occur during a monitor operation.

A number of methods [8, 9, 13] have been developed for proving correctness of a monitor's functional operations. Most of these methods are based on Hoare's axiomatic system for proving data representation (because of the analogy between a monitor and a data representation) and provide proof rules which emphasize reasoning on the invariants relating values of the permanent variables of the monitor. These axioms are further developed and applied by Howard [9]. Howard used history variables and axiomatization of the properties of *Wait* and *Signal* operations to prove some more interesting properties of monitors, not only of functional but also scheduling properties. Verification of specifications of monitor primitives to be implemented is also studied. Saxena and Bredt verified a specification of monitor primitives in terms of input/output assertions, from both the procedure viewpoint and the process viewpoint [13].

Verification is essentially a fault-prevention technique. Correctness does not necessarily imply reliability. Many factors can lead to run time errors even for a program that is proved to be correct. Furthermore, not all the properties of a monitor can be proved from the monitor definition alone. For example, the *external consistency* of a monitor, defined as the observation of a sequential constraint upon the order of procedure invocation that may be initiated by any individual user, must be proved separately for each program that uses the monitor. Run-time mechanisms, therefore, are needed to handle exceptions of monitor operations and usage raised during execution.

The fact that a monitor is an abstract data type makes it relatively easy to add new functional components and control capabilities to the monitor construct to improve the reliability of concurrent programs. In this paper, we propose an extension to the monitor functionality by introducing an underlying fault-detection mechanism, which can be applied to the implementation and run-time execution of monitor primitive operations. We describe the fault detection model, the development of the fault detection algorithms, and a prototypical implementation of the augmented monitor construct using Java.

The rest of the paper is organized as follows. In Section 2, functional characteristics of different kinds of monitors are analyzed and taxonomy of monitor faults is introduced. We present the fault detection model and a classification of the abnormal behaviors of monitor

0-7695-1101-5/01 $10.00 © 2001 IEEE

operations that may result in run-time concurrent control errors. In Section 3, we define the concepts of a monitor event sequence and a monitor state sequence, which will be used for structuring and recording the monitor execution history information. A set of rules governing well-defined monitor sequences is proposed. Based on these rules the fault detection algorithms are developed. Section 4 discusses the implementation issues and describes a prototypical implementation of the proposed monitor construct in Java. Finally, Section 5 concludes the paper.

2. A Taxonomy of monitor concurrency control faults

Based on the concept of an abstract data type, the monitor encapsulates both local data and operations on the data, and provides an interface which is the only means for user processes to request the operations. The data represents the status of the shared resource being controlled by the monitor while the local procedures operate on the local data variables to change the status of the shared resource. Typically, the specification of a protected resource must include the integrity (consistency) constraints and the scheduling (timing order of events) constraints. In a monitor, the scheduling constraints are coded in its implementation and the integrity constraints are coded in its procedures. Essentially, at any one time, one and only one process is allowed to be inside a monitor. In this way, concurrency control is integrated into the monitor specification so that it does not have to be considered in the use of the shared data by the programmer.

Apart from automatically ensuring mutual exclusion of contending processes, monitors could also manage the conditional synchronization of processes sharing the resources. Hoare proposed the concept of condition variables [8], which are local variables of a monitor representing the synchronization status of the shared resources. A condition variable is used to delay processes executing in a monitor and is represented as a queue initialized to be empty.

Several implementations of monitors have been proposed, which depend on primitives at a lower level [2, 8, 7, 11]. In the context of this paper, a monitor implementation is specified at a higher level, and consists of a set of four procedures, namely *Enter*, *Wait*, *Signal*, and *Exit*. The *enter* primitive ensures mutually exclusive access to the requested monitor while the *exit* primitive releases the mutual exclusion of the monitor being accessed by other requesting processes. The *wait* primitive blocks the execution of the calling process and releases the mutual exclusion of the monitor that is on hold by other requesting processes. The *signal* primitive activates one of the processes waiting in either the entry queue or the condition queues of the specified monitor and releases the control of the signaling process to the awakened one. As the signaling processes have finished using the resource and are no longer inside their critical sections, they normally exit the monitor right after issuing the signaling operation [8]. The signal and exit primitives can thus be naturally combined into one, named *signal-exit*, so as to reduce process-switching overheads.

In this section, we present taxonomy of concurrency control faults in the monitor mechanism. The classification serves several purposes. First, it serves as a guide for building the error detection algorithm. Second, validation requires a system specification against which the actual results of operations can be assessed. A fault classification gives a systematic way to check which parts of the specification are violated. Third, it provides information about the frequency of each fault. For example, if a particular kind of fault appear frequently we could use a variety of methods to reduce the incidence of it.

2.1 Classification of monitors

We first present a classification of monitors, which provide us insight into the requirements and structures of different types of monitors, and thus a guide for identifying the faults to be detected.

Processes may interact in two ways: (a) *Directly by communicating messages via a common data area.* This often happens when a number of processes cooperate on some common tasks, each of them is aware of the other's existence and function and depends directly on data or signals produced by the others. (b) *Indirectly by competing for the same resources.* Each process may be functionally independent so it may not know the existence of others. Therefore, concurrent control can be divided into two main subcategories: resource control and communication control. Accordingly, we can classify monitors into three types, according to their functional characteristics.

- *Communication Coordinator*: Communication between processes requires that they be synchronized during data exchange. This type of monitor allows pairs of processes to communicate via data exchanges that are controlled by the monitors. All interprocess communication is performed by calls to entries of shared buffers known to each other and governed by the implicit mutual exclusion of the monitor calls. Each process simply calls the monitor procedures "Send" and "Receive" in their respective programs. Synchronization and resource operations are combined together in the monitor, i.e., the monitor takes care of both scheduling access to and operations on the buffer. There is no requirement on the order of procedure calls: either Send or Receive can be called before or

after each other. However, in order to ensure normal operation of this type of monitors, there are several integrity constraints, which need to be observed by the monitors.

1) A process calling "Send" can be delayed if and only if the buffer is full.
2) A process calling "Receive" can be delayed if and only if the buffer is empty.
3) The number of successful monitor procedure calls of "Receive" cannot exceed the number of successful monitor procedure calls of "Send".
4) The number of successful monitor procedure calls of "Send" cannot exceed the sum of the maximum buffer capacity and the number of successful procedure calls of "Receive".

- *Resource-Access-Right Allocator*: When a number of processes compete for the exclusive use of the same resource they must exclude each other in time and maintain invariants for shared resources. Furthermore, when more than one resource are to be shared and/or if a user needs to access more than one resource, deadlock prevention or avoidance in resource allocation needs to be implemented. Usually there is an allocator for each shared resource to guarantee mutually exclusive accesses to the resource. A monitor of the allocator type can be used for this purpose. A process must declare its desire to use the resource to the allocator by initiating a request. When it is granted the right to access the resource from the monitor it can do predefined operations on the resource. After finishing the use of the resource the process must release the resource through the allocator. Note that the resource access operation is separated from resource allocation. The monitor only handles "request" and "release", it does not mediate the use of the resource. One of the constraints to be observed by this type of monitor is the partial ordering of procedures. This partial ordering is declared in the monitor specification explicitly. For example, a procedure call to "Release" cannot precede a procedure call to "Request" by the same process for each use of the resource. A process must invoke the procedure "Release" after it has completed its use of the resource. Failure to observe this sequence clearly represents possible misuse of the resource.
- *Resource Operation Manager*: Synchronization can be provided by a monitor either explicitly or implicitly. In explicit synchronization, processes have to explicitly request access to the resource, perform the operations and finally release the resource. On the other hand, in implicit synchronization, monitors and resources are combined into shared modules; Processes only need to issue the access operation to the module and the monitor will handle all the operations including the requesting and releasing operations. Monitors of the Resource-operation-manager type are just for providing the processes with implicit synchronization. This approach has the benefit of more modularity and preventing user processes from possible misuses of the resources.

2.2 A taxonomy of faults

We first define what is meant by a "concurrency control fault". A monitor performing concurrency control over a shared resource is considered to be operating correctly if and only if the following conditions hold:

1) It guarantees mutual exclusion and enforces synchronization on accesses to the shared resource.
2) It is free of deadlock and starvation in sharing the resources by concurrent processes.
3) It maintains consistency of the states of the shared resource.
4) It preserves the specified behavior of processes that are using the shared resource, that is, the exact execution sequence of the processes is observed and is not influenced by the monitor.

Any event that causes one or more errors which violate the above concurrency control properties of a monitor is declared to be a concurrency control fault. Faults could be software faults or hardware faults, design faults or system faults. Attempting to catch all the faults requires a complete specification of both monitor functions (procedures) and monitor implementations. Since monitor procedures are application dependent, it is impossible to know what the internal function of each monitor procedure is. Therefore, in our classification, we only consider the effects of monitor procedures, rather than their internal logic.

Based on the taxonomy of monitors, we identify the following events as concurrency control faults. They are classified into three levels: the implementation level, the monitor procedure level, and the user process level.

I. Implementation level faults

Four types of faults are identified at this level - the *Enter* procedure fault, the Wait procedure faults, the Signal-exit procedure faults, and finally the internal process termination fault.

a) *Enter procedure faults:* arising in the following situations
   1. Mutual exclusion is not guaranteed - two or more processes have entered the monitor at the same time.
   2. The requesting process is lost - the process is neither queued up for entering the monitor nor will it be allowed to enter the monitor.
   3. The requesting process has not received a response - the process is queued up indefinitely for entering the monitor or the process is blocked

when there is no process currently running inside the monitor.

4. Entry is not observed - the process that is running inside the monitor has not invoked the Enter primitive.

b) *Wait procedure faults:* arising in the following situations
   1. Synchronization is not guaranteed - the calling process is not blocked to queue up for condition but continues to run inside the monitor.
   2. The calling process is lost - the process is neither queued up for condition nor continues to run inside the monitor.
   3. Entry waiting processes are not resumed - none of the processes waiting on the entry queue are resumed when the calling process is blocked.
   4. Entry waiting process is starved - the process waiting on the entry queue is never resumed but wait indefinitely.
   5. Mutual exclusion is not guaranteed - more than one process waiting on the entry queue is resumed to enter the monitor when the calling process is blocked to queue up for condition.
   6. Monitor is not released - the calling process is blocked to queue up for condition but has failed to release the monitor for other waiting processes to access.

c) *Signal-Exit procedure faults:* arising in the following situations
   1. Waiting processes are not resumed - none of the processes waiting on condition queues or on the entry queue are resumed when the calling process exits the monitor.
   2. Monitor is not released - the calling process exits the monitor but the monitor is not released for other waiting processes to access.
   3. Mutual exclusion is not guaranteed - more then one process is resumed to access the monitor at the same time when the calling process exits the monitor.
   4. *Internal process termination fault.* Process is terminated inside the monitor - the process never exits the monitor after entered it but terminated inside the monitor.

II. Monitor procedure level faults

This type of faults refers to monitor procedure operations, which result in inconsistent states. These faults cause the states of the shared resources to be inconsistent and thus violate the integrity constraints of the communication coordinator type monitors described in Section 2.1. Four kinds of faults are identified which are simply violations of the integrity constraints.

a) A process calling the monitor procedure "Send" is delayed when the buffer is not full, or the buffer is full but the calling process is not delayed.
b) A process calling the monitor procedure "Receive" is delayed when the buffer is not empty, or the buffer is empty but the calling process is not delayed.
c) The number of successful calls of "Send" is less than the number of successful calls of "Receive".
d) The number of successful calls of "Send" is larger than the sum of the maximum buffer capacity and the number of successful calls of "Receive".

III. User process level faults

This class of faults refers to logic design errors or run-time errors in executing the monitor procedures. Three kinds of faults are identified which are simply violations of the partial-ordering constraint.

a) Ordering of monitor procedure calls is incorrect - a process tries to release a resource without first acquiring the resource.
b) Resource is not released - a process never releases a resource after it acquires the resource.
c) Process is deadlocked - the process acquired a resource and attempts to acquire the same resource again without first releasing the resource.

In total, twenty-one concurrency control faults are identified and classified into different types and levels. Only the user process level faults (the last three) should be detected during real time execution, as the execution sequence of the monitor procedures of the resource-access-right allocator type monitors must be kept correct. Others can be checked against during a certain execution time frame since they induce no immediate significant errors or disaster.

3. A monitor construct augmented with run-time fault detection

Extensions are made to the monitor construct in two respects: the *visible* part and the *invisible* part. For the visible part the users need to supply some information. Here, we require the partial ordering of procedure calls within a monitor be specified in the monitor declaration. A convenient way to specify the partial order relation is *path-expression* like notation [3]. The invisible part refers to internal control operations that implement the monitor construct. A database for collecting history information needs to be defined and maintained and fault detection procedures need to be incorporated into the monitor implementation.

3.1. History information

For the purpose of detecting faults we need to maintain history information about monitor scheduling operations, against which the run-time behavior of a monitor can be checked. The history information

includes which processes are invoking the monitor, at which time an operation is executed, and the resource and queue states. They can be classified into two categories, which define the concepts of *scheduling event* and *scheduling state*.

A *scheduling event* is the event of invoking one of the three monitor primitives - `Enter`, `Wait`, and `Signal-Exit`. The set of scheduling events, EVENTset, for a monitor is defined as follows:

EVENTset = {Enter(Pid, Pname, t, flag),
Wait(Pid, Pname, Cond, t, flag),
Signal-Exit(Pid, Pname, Cond, t, flag) }

Each event identifies the time t at which the event occurs, the process Pid that caused the event and the procedure Pname involved. A flag is associated with the events `Enter` and `Wait` to indicate whether the corresponding monitor primitive has been successfully completed (e.g., blocked or continued). With an unsuccessful operation, the flag is set to 0 and later changed to 1 when the invoking process is resumed; the time t is set to the time at resumption. For `Signal-Exit` the flag indicates whether a process waiting on the condition queue has been resumed.

The runtime operation of a monitor can be modeled as a finite sequence of scheduling events, $L = l_1 l_2 \ldots l_n$. A scheduling state of a monitor is a 3-tuple <EQ, CQ[], R#>, where EQ denotes the external waiting queue, CQ[ ] is the array of condition queues, and R# denotes the number of currently available resources. Because each of the above three events will cause a new scheduling state to be generated, so for each scheduling event sequence $L = l_1 l_2 \ldots l_n$ there will be exactly one corresponding scheduling state sequence $S = s_1 s_2 \ldots s_n$ such that s_i is generated by l_i and

1) l_i precedes l_j in L if and only if $i < j$;
2) s_i precedes s_j in S if and only if $i < j$;
3) $l_i <_L l_j$ if and only if $s_i <_S s_j$

We will use the symbol $<_L$ and $<_S$ to denote "precede in L" and "precede in S", respectively. In addition, we define L_{ij} to be a subsequence of L between l_i and l_j.

The dynamic scheduling behavior of a monitor then is implied in its sequence of scheduling events and states. Under correct monitor operations the sequences must be consistent and correspond to the specified value.

3.2. Fault detection rules

Before we present the fault detection algorithm we first show that every class of concurrency control faults in the taxonomy can be detected. To do this, we first define a set of rules in terms of the scheduling event sequence and then prove that every fault in the taxonomy is a violation of at least one of the rules. Based on the proof, a detecting algorithm that checks a given scheduling event sequence against these rules can be developed.

Let *Tio* denote the timeout period for interpreting deadlock or starvation, Q denote a queue (either EQ or any condition queue) and $|Q|$ be the number of processes waiting on Q. Then according to the definition of correct concurrency control, a *valid* scheduling sequence is a scheduling sequence $L = l_1 \ldots l_n$, $S = s_0 \ldots s_n$ satisfying the following rules:

FD-Rule 1 : Mutually exclusive access to the monitor

a) $l_i = Enter(P,Pr,t_r,1) \rightarrow$
$\forall j<r\ (l_j = Enter(P',Pr',t,1) \rightarrow \exists k(s<k<r \wedge$
$(l_k = Signal\text{-}Exit(P',Pr',Cond,t_k,0/1) \vee$
$l_k = \text{Wait}(P',Pr',\text{Cond},t_k,0))))$

This rule requires that a process be allowed to enter the monitor only if no process currently uses the monitor.

b) $l_i = Wait(P,Pr,Cond,t,0/1) \vee$
$l_i = Signal\text{-}Exit(P,Pr,Cond,t_i,0) \rightarrow$
$(|s_{i-1}.EQ| \neq 0 \rightarrow (|s_i.EQ| = |s_{i-1}.EQ| - 1 \wedge$
$\exists j < i\ (l_j = Enter(P',Pr',t_i,1))))$

c) $l_i = Signal\text{-}Exit(P,Pr,Cond,t_i,1) \rightarrow$
$(|s_{i-1}.CQ[Cond]| \neq 0 \rightarrow (|s_i.CQ[Cond]| =$
$|s_{i-1}.CQ[Cond]| - 1 \wedge$
$\exists j < i\ (l_j = Wait(P',Pr',Cond,t_i,1))))$

FD-Rules 1.b) and 1.c) requires that, if the waiting queue Q is not empty the proper `Signal-Exit` or `Wait` will activate exactly one of the processes awaiting on Q.

d) $l_i = Wait(P,Pr,Cond,t_r,0/1) \vee$
$l_i = Signal\text{-}Exit(P,Pr,Cond,t_i,0/1) \rightarrow$
$\exists j<i\ (l_j = Enter(P,Pr,t_r,1)$

This rule requires that every process operating inside a monitor must have called `Enter`.

FD-Rule 2 : Nontermination inside a monitor

$l_i = \text{Enter}(P,Pr,t_r,1) \rightarrow$
$\exists j>r\ (t_j - t_r \leq Tmax \wedge$
$l_j = \text{Signal-Exit}(P,Pr,\text{Cond},t_j,0/1))$

FD-Rule 3 : Fair response

$l_i = Enter(P,Pr,t_i,0) \rightarrow$
$\exists j<i\ (l_j = Enter(P',Pr',t_r,1) \wedge \neg\exists k\ (j<k<i \wedge$
$(l_k = Signal\text{-}Exit(P',Pr',Cond,t_k,0/1) \vee$
$l_k = Wait(P',Pr',Cond,t_k,0))))$

Rule 3 requires that a requesting process can be delayed only when the monitor is in use already.

FD-Rule 4 : Free of starvation and losing process

$l_i = Enter(P,Pr,t_i,0) \vee l_i = Wait(P,Pr,Cond,t_i,0) \rightarrow$
$n - i < Tio \wedge |s_i.Q| = |s_{i-1}.Q| + 1$

FD-Rule 5 : Correct synchronization

a) $l_i = Wait(P,Pr,Cond,t_r,1) \rightarrow$
$i \neq r \wedge l_r = Signal\text{-}Exit(P,Pr,Cond,t_r,1)$

A process waiting on a condition queue can only be resumed by a process calling `Signal` on the condition.

b) $l_i = Enter(P,Pr,t_r,1) \rightarrow$
$l_r = Wait(P',Pr',Cond,t_k,0/1) \vee$
$l_r = Signal\text{-}Exit(P',Pr',Cond,t_r,0)$

A process waiting on the entry queue can only be resumed by a process calling `Wait` or non-Signal Exit.

FD-Rule 6 : Consistency of resource states

a) Let r and s denote the number of successful calls of `Receive` and `Send`, respectively. Let *Rmax* denote the maximum number of resources. For a communication coordinator monitor, the following invariant holds:

$$0 \le r \le s \le r + Rmax$$

b) $l_i = Wait(P,\underline{Send},Cond.full,t_r,0/1) \rightarrow s_i.R\# = 0$

c) $l_i = Wait(P,\underline{Receive},Cond.empty,t_r,0/1) \rightarrow s_i.R\# = Rmax$

FD-Rule 7 : Correct ordering of procedure calls

a) $l_i = Enter(P,\underline{Acquire},t_r,0/1) \rightarrow \exists j>i\ (l_j = Enter(P,\underline{Release},t_j,0/1) \wedge \neg\exists k\ (i<k<j \wedge l_k = Enter(P,\underline{Acquire},t_k,0/1)))$

b) $l_i = Enter(P,\underline{Release},t_i,1) \rightarrow \exists j<i\ (l_j = Enter(P,\underline{Acquire},t_j,1) \wedge \neg\exists k\ (i<k<j \wedge l_k = Enter(P,\underline{Release},t_k,1)))$

It can be shown that, having the history information database, every level of concurrency control faults in the taxonomy can always be detected. This is because each fault at the implementation level will lead to a violation of at least one of the FD-Rules 1 – 5, and each fault at the user process level will lead to a violation of at least one of the FD-Rules 6.a), 6.b) and 6.c).

3.3. Fault detection algorithms

Fault detection is achieved by detecting routing invoked periodically or when trouble is suspected. This approach leaves the main program logic largely unaltered by the detection logic and modifications to the detection routine should have little effect on the main program system. Our fault detection strategy includes two phases: real-time checking of calling orders of monitor procedures, which is applied only to Resource-access-right-allocator type monitors for correct orders of using shared resources, and periodical checking of other errors. Let *Tmax* and *Tmin* denote the maximum and minimum number of times any process can be inside a monitor, respectively. The frequency of periodical invocation of the detection routine is determined by a unit of time T, where $Tmax < T$. Therefore, whenever T is reached the detection routine is automatically invoked.

For the sake of efficiency in both time and memory space, we use a fault-detection the following strategy. First, we find out the correct changes of monitor states and event relations guaranteed by the above rules. Then, indirectly, we detect faults by checking the given event sequence to see whether it maintains such consistent changes. In this way, events can be viewed as functions mapping one consistent monitor state into another consistent state. Only the states at the last checking time and the current checking time are recorded for checking the mapping; the state sequence in between is not needed. Furthermore only a small amount of information needs to be kept (in the last checking state) for later detection; most of the information can be removed after being used.

We collect the information about the monitor events and states between the last checking time and the current checking time:

At the last checking time p :

$$s_p = (EQ_p,\ CQ[Cond]_p,\ R\#_p)$$

.

.

. $L = l_p...l_t$ (recorded event information)

.

At the current checking time t_t :

$$s_t = (\ EQ_t,\ CQ[Cond]_t,\ R\#_t\ )$$

With the above information, faults causing run-time concurrency control errors can then be detected by checking the information against the rules. The checking is based on state transformations – derive s_t from s_p and L according to the state transition rules; If every step in the derivation gives a consistent state and finally s_t can be derived, then we say that no fault has occurred. On the other hand, any inconsistency in monitor states notifies that concurrency control faults have occurred during the execution of monitor operations between the checking time frames. If one step in the derivation gives an error there must be an error in the event sequence, e.g., l_i = *Wait*(P), l_{i+1} = *Signal-Exit*(P), i.e., P resumes execution without being signaled. Notice, however, that even if every step of the derivation is correct, this does not imply a fault-free situation.

Although this post-checking is less accurate due to the fact that the monitor states between the checking points are not recorded, it can still detect most concurrency control faults identified in the previous section, except the ones at the user process level which need to be checked against in real-time. By properly defining the checking frequency T, the checking can be made more accurate. When $T = 1$, the checking becomes real-time.

3.3.1. Data structures

In keeping with the above strategy, we need to modify the way of recording the scheduling events as follows. Since we want to avoid tracing back the previous events, whenever a blocked process (on either *EQ* or *CQ*) is resumed, its time and flag will not be changed. Actually, we even do not have to use the flag for Wait, as well as the time for all events. Therefore, the EVENTset becomes:

EVENTset = { Enter(Pid, Pname, flag),
Wait (Pid, Pname, Cond),
Signal-Exit(Pid, Pname, Cond, flag) }

In addition to the monitoring state, we also record the active process in the monitor at checking time and denote

it as *Running*. The scheduling event sequence recorded will be used to construct the following checking lists at each checking time.

1) *Enter-0-List*: This list records the in and out of processes awaiting on the *EQ*; it is a list of elements of the form *Pid(Pr)*, where *Pid* denotes the calling process and *Pr* the calling monitor procedure. Initially, Enter-0-List is set to *EQ*. During the execution, its value is updated as follows:
   - Whenever l_k =*Enter(Pid, Pr,* 0) is encountered, *Pid(Pr)* is appended to *Enter-0-list*;
   - Whenever l_k = *Wait* or l_k = *Signal-Exit* is encountered, the first element (at the head of the list) of *Enter-0-List* is deleted.
2) *Wait-Cond-List*: One for each condition, the list records the in and out of processes awaiting on *CQ[Cond]*. It is a list of elements of the form *Pid(Pr)*, where *Pid* denotes the calling process and *Pr* the calling monitor procedure. Initially, *Wait-Cond-list* is set to *CQ[Cond]*; During the execution, its value is updated as follow:
   - Whenever l_k = *Wait(P, Pr, Cond)* is encountered, *Pid(Pr)* is appended to *Wait-Cond-List*;
   - Wheneve l_k = *Signal-Exit(Pid, Pr, Cond, 1)*, the first element of *Wait-Cond-List* is deleted.
3) *Running-List:* This list records the processes currently inside the monitor without waiting on any condition queue. It is a list of elements of the form *Pid*. Initially, the list is set to s_t *Running;* during execution, its value is updated as follows.
   - Whenever l_k = *Enter(Pid, Pr, 1)* is encountered, *Pid* is appended to *Running-List*;
   - Whenever an element is deleted from *Enter-0-List*, the element is appended into *Running-List*;
   - Whenever an element is deleted from *Wait-Cond-List*, the element is appended into *Running-List*;
   - Whenever l_k = *Wait(Pid, Pr, Cond)* is encountered, the first element with *Pid* in *Running-List* is deleted;
   - When l_k = *Signal-Exit(Pid, Pr, Cond, 0/1)*, the first element with *Pid* in *Running-List* is deleted.
4) *Resource-No*: This number indicates the changes of the resource status of the communication coordinator type monitors. Its value is the number of the available resources. Initially, *Resource-No* is set to s_p.*R#*; during execution, the value changes as follows:
   - Whenever l_k = *Signal-Exit(Pid, Send, empty, 0/1)* is encountered, *Resource-No* is decreased by one;
   - Whenever l_k = *Signal-Exit(Pid, Receive, full, 0/1)* is encountered, *Resource-No* is increased b y one.
5) *Request-List*: This list records the calling sequence of the monitor procedures `Request` and `Release` of the Resource-Access-Right-Allocator type monitors. It is a list of elements of the form *Pid*. Initially, it is set to empty and, during execution, its value is updated as follows:
   - When l_k = *Enter(Pid, Acquire, 0/1)*, *Pid* is appended to Acquire-List;
   - When l_k = *Signal-Exit(Pid, Release, 0/1)*, the first element with *Pid* is deleted from *Request-List*.

3.3.2. Fault detection algorithms

The design of the fault detection algorithms is based on a set of state transition rules. Let s_p and s_t be the monitor states at the last checking time t_p and the current checking time t, respectively; $L = l_1 \ldots l_n$ be the given scheduling event sequence generated during the time period between t_p and t. The following state transition rules must hold.

ST-Rule 1 : Up to l_n, $s_t.EQ$ = *Enter-0-List*.

ST-Rule 2 : Up to l_n, $s_t.CQ[Cond]$ = *Wait-Cond-List*.

ST-Rule 3 : At any time only one process can be inside a monitor:

a) At any time, $|Running\text{-}List| \leq 1$.
b) If l_k = *Wait(Pid)* or l_k = *Signal-Exit(Pid)*, then upto l_{k-1}, *Running-List* = {*Pid*}.
c) If l_k = *Enter(Pid, Pr, 1)*, then upto l_k, *Running-List* = {*Pid*}.
d) If l_k = *Enter*(0), then upto l_k, $|Running\text{-}List| = 1$

ST-Rule 4 : For any event l_i, up to l_{i-1}, Pid cannot be in either *Enter-0-List* or any of the *Wait-Cond-Lists*.

ST-Rule 5 : $\forall$ *Pid* $\in$ (*Wait-Cond-Lists* $\cup$ *Running-List*), *Timer(Pid)* $\leq$ *Tmax*.

ST-Rule 6 : $\forall$ *Pid* $\in$ *Enter-0-List*, *Timer(Pid)* $\leq$ *Tio*.

ST-Rule 7 : This rule concerns the data integrity of the Communication-Coordinator type monitors. Four sub-rules are induced:

a) $0 \leq r \leq s \leq (r + Rmax)$
b) $|s_t.R\#| = |s_p.R\#| + r - s$
c) If l_k = *Wait(Pid, Send, Cond, full)* then *Resource-No* = 0.
d) If l_k = *Wait(Pid, Receive, Cond, empty)* then *Resource-No* = *Rmax*.

ST-Rule 8 : This rule concerns the calling orders of the monitor procedures "Request" and "Release" of the Resource-Access-Right-Allocator type monitors.

a) No *Pid* is identical to another *Pid* in *Request-List*.
b) If l_k = *Enter(Pid, Release,* 0/1), then *Pid* must be in *Request-List*.
c) No *Pid* can be in *Request-List* forever.

It can be proved that any violation of the FD-Rules 1-7 defined in Section 3.2 will lead to a violation of the ST-Rules. The FD-Rules and the ST-Rules are defined in terms of different history information. Actually, the checking lists are pseudo-historical since they are generated at checking points. If the actual intermediate

monitor states are recorded and in each derivation step the checking lists are checked against these states then any violation of FD-Rules will lead to a violation of the correct state transitions. In this way, the FD-Rules are equivalent to the state transition rules.

Based on the state transformation rules stated in the previous section, three fault-detection algorithms have been developed, respectively, for checking (a) general monitor concurrency control operations, (b) the consistency of resource states that should be preserved by monitor procedures, and (c) the partial ordering of monitor procedure calls. See Algorithm-1, Algorithm-2, and Algorithm-3. Input to these algorithms are event sequences generated up to the checking time. Also, the monitor name and type are used as parameters to the algorithms.

The checking lists are initialized once to empty before any invocation of the detection algorithms.

Algorithm-1: General Concurrency-Control Checking
Input : Monitor state s_p at the last checking time t_p ;
Monitor state s_t at the current checking time t;
Scheduling event sequence $L=l_1...l_n$ generated from t_p to t;
Begin {
Step 1:
Initialize the checking lists;
For each l_1 in L **Do**
If *Pid* in l_1 is in Enter-0-List or any *Wait-0-list* **Then** report an error;
If l_1 = *Wait*(*Pid*) or *Signal*-Exit(Pid) **Then**
If {*Pid*} ≠ *Running-list* **Then** report an error;
Adjust *Enter-0-list* accordingly;
Adjust *Wait-Cond-lists* accordingly;
Adjust *Running-list* accordingly;
If |*Running-List*| > 1 **Then** report an error;
If l_i = *Enter*(*Pid*, *Pr*, *1*) **Then**
If *Running-List* ≠ {*Pid*} **Then** report an error;
If l_i = *Enter*(0) **Then If** |*Running-List*| ≠ 1 **Then** report an error;
Step 2:
If l_n is encountered **Then**
If *Enter-0-list* ≠ s_t.*EQ* **Then** report an error;
If *Running-List* ≠ s_t.*Running*; **Then** report an error;
For all Cond **Do**
If *Wait-Cond-list*=s_t.*CQ*[*Cond*] **Then** report an error;
For all *Pid* in *Running-List* and *Wait-Cond-Lists* **Do**
If *Timer*(*Pid*) ≥ *Tmax* **Then** report an error;
For all *Pid* in *Enter-0-List* **Do**
If *Timer*(*Pid*) ≥ *Tio* **Then** report an error;
} **End**.

Algorithm-2:Consistency-Of-Resource-States Checking
Input : Monitor state s_p at the last checking time t_p ;
Monitor state s_t at the current checking time t;
Scheduling event sequence $L=l_1...l_n$ generated from t_p to t;
Begin {
Step 1:
Initialize the checking lists;
For each l_1 in L **Do**
Adjust *r, s* accordingly;
Adjust *resource-No* accordingly;
If $0 \le r \le s \le r + Rmax$ does not hold **Then** report an error;
If l_1 = *Wait*(*Pid, Receive, Cond.full*) **Then**
If *Resource-No* ≠ 0 **Then** report an error;
If l_1 = *Wait*(*Pid, Receive, Cond.empty*) **Then**
If *Resource-No* ≠ *Rmax* **Then** report an error;
Step 2:
If $|s_t.R\#| \ne |s_p.R\#| + r - s$ **Then** report an error;
} **End**.

Algorithm-3: Calling Orders Checking
Input : Monitor state s_p at the last checking time t_p ;
Monitor state s_t at the current checking time t;
Scheduling event sequence $L=l_1...l_n$ generated from t_p to t;
Begin {
Step 1:
Initialize *Resource-No*;
For each l_1 in L **Do**
Adjust *Request-list* accordingly;
If l_1 = *Entert*(*Pid, Release, 0/1*) **Then**
If *Pid* is not in *Request-list* up to that time **Then** report an error;
If there exists identical *Pids* in Request-*list* **Then** report an error;
Step 2:
For each *Pid* in *Request-list* **Do**
If Timer(*Pid*) ≥ Tlimit **Then** report an error
} **End**.

4. A prototype implementation in Java

The monitor construct is now augmented with the specification of the information necessary for run-time fault detection. The information includes procedure-calling orders, monitor types, etc. The general form of the monitor specification is shown as follows.

```
MonitorName: Monitor (type);
  Declarations of local variables;
  Declarations of condition variables;
  Specification of procedure call orders;
  Declarations of monitor procedures;
  Declarations of local procedures;
  Initialization section;
End MonitorName.
```

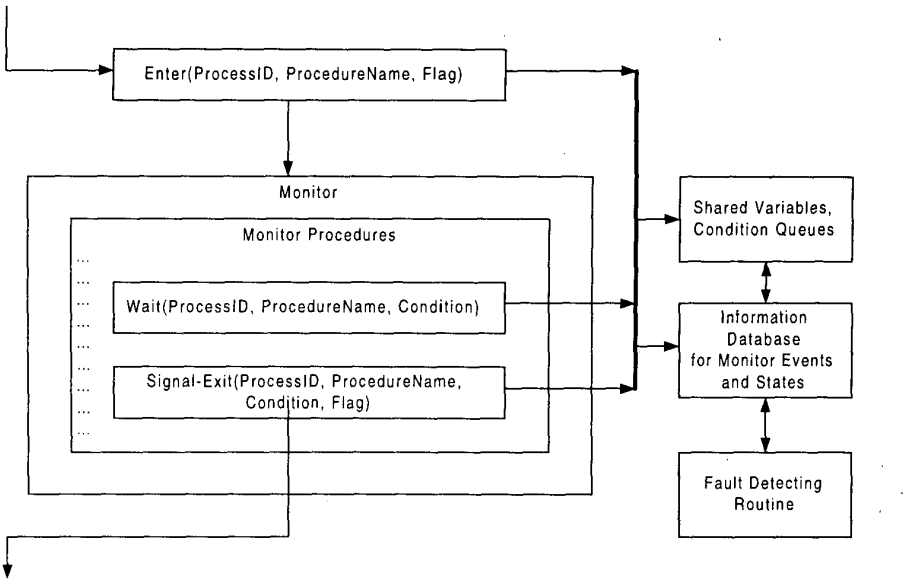

Figure 1. Fault Detection Model

Table 1. Overhead induced by run-time fault detection

	Communication Coordinator		Resource Access Right Allocator		Resource Operation Manager	
Checking Time	Time used with checkings	Time used without checkings	Time used with checkings	Time used without checkings	Time used with checkings	Time used without checkings
0.5 second	52.285	7.52	50.93	7.405	52.067	7.735
	Ratio for overheads	6.953	Ratio for overheads	6.878	Ratio for overheads	6.731
1.0 second	38.681	7.752	39.119	7.561	37.504	7.473
	Ratio for overheads	4.990	Ratio for overheads	5.174	Ratio for overheads	5.019
2.0 second	34.221	7.622	32.424	7.219	32.67	7.043
	Ratio for overheads	4.490	Ratio for overheads	4.491	Ratio for overheads	4.639
3.0 second	30.168	7.491	30.823	7.371	31.41	7.57
	Ratio for overheads	4.027	Ratio for overheads	4.182	Ratio for overheads	4.150

To facilitate fault detection, the system maintains a history information database, which consists of the scheduling event sequence recorded during monitor operation and the checking lists generated at the checking points. Coupled with this database the system would also allow for control of accesses to the database. Therefore, two types of routines are needed: data gathering routines which collect the information and record them into the database, and checking routines which operate on the data structures and report on their error states. The data gathering routines run in real-time and are invoked by the three monitor implementation procedures `Enter`, `Wait`, and `Signal-Exit`. The checking routines, which actually implement the fault detecting algorithms, are invoked periodically. Upon detection, all other running processes are suspended and are resumed only after the checking has finished.

The overall structure of the augmented monitor construct is shown in Figure 1. It consists of four main functional units: the monitor, the shared resources, the data gathering routine, and the fault detection routine. The data gathering routine collects historical information on monitor usage and states while the fault detection routine uses this collected information to

analyze any violations of concurrency control rules. The two routines can be implemented outside the monitors so that any changes to the routines will not affect the monitors and vice versa.

To evaluate the proposed robust monitor construct, we have developed a software prototype in Java [4]. Two measures are used for the evaluation: robustness and performance. The former is concerned with whether the proposed extension is effective in detecting run-time faults, while the latter measures the overhead incurred by the extension. Faults of different kinds as classified in Section 3.2 are injected randomly for evaluating the coverage of the fault detection algorithms. The results show that all injected faults are detected.

To evaluate the overhead imposed by history information recording and fault detection, statistics of elapsed times spent on recordings of history information and on checking of concurrency control faults are collected with different checking time intervals. Table 1 shows the overhead calculated as the average ratio between the time spent on executing monitor operations with the extension and that without the extension. The results show that, as expected, when the checking time interval increases, the overhead decreases. The performance of the fault detection model can also be determined. For example, when the time interval for invoking the fault detection routine of the communication coordinator type monitors is set to 0.5 second, the performance of the augmented monitor construct is decreased by nearly seven times of that without fault detection. Therefore, in order to keep checking of concurrency control faults without scarifying too much performance, the checking time interval must be carefully decided.

5. Conclusions and Future Work

In this paper, we have introduced a framework for detecting concurrency control faults in a multiprogramming system based on the monitor construct. We proposed an augmented monitor construct with run time assertion checking and underlying fault detection, which allows the integrated detection of concurrency control faults inside the monitor mechanism. A software prototype of the proposed robust monitor construct has been developed in Java.

Extensions can be made to allow predefined and user-supplied assertions to be specified as part of monitor declarations and used for checking the functional operations and external use of the monitors. The validity of this checking is based upon the assumption that the implementation of the monitor primitives is correct. The underlying fault detection, applied to run time execution of the monitor primitives, facilitates the dynamic verification of the behavior of the implemented monitor mechanism. Improvements in these two aspects provide a more reliable monitor construct.

The proposed extensions to the monitor constructs only enable the monitors to detect faults. A fault tolerant system detects errors created as the effect of a fault and in addition, applies error recovery techniques to restore and continue the normal operations. Therefore, in order to make the monitor construct to be fault-tolerant, error recovery mechanisms should be incorporated into the model to handle the faults detected by recovering the errors.

Acknowledgement

Nick K.C. Cheung is partially supported by Hong Kong Polytechnic University Postgraduate Scholarship under Grant G-V598. The authors wish to thank Prof. Tharam S. Dillon for reading and editing the manuscript and providing valuable comments.

References

[1] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, Vol. 15, No. 1, Mar 1983. pp.3-43.

[2] P.A. Buhr and M. Fortier, "Monitor Classification", *ACM Computing Surveys*, Vol, 27, No. 1, Mar 1995. pp.63-107.

[3] R.H. Campbell and R.B. Kolstd, "A Practical Implementation of Path Expressions", *Tech Report CS-R-80-1008, Dept. of Computer Sci, UIUC*, June 1980.

[4] J. Cao, Nick. Cheung, Alvin. Chan, "Implementing a Robust Monitor Construct in Java", *submitted for publication.*

[5] Thomas W. Doeppner Jr. AND Alan J. Gebele, "C++ on a Parallel Machine", *Tech. Report, CS-87-26*, Dept of Computer Sci, Brown Univ., Nov. 17, 1987

[6] P.B. Hansen, *Operating Systems Principles* Prentice Hall, 1973.

[7] P.B. Hansen, "The Programming Language Concurrent Pascal", *IEEE Trans. Software Engineering*, SE-1 (2) 1975. pp.199-206.

[8] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *CACM*, Vol. 17 No 10, Oct 1974, pp 549-557

[9] John H. Howard, "PROVING MONITORS", The University of Texa at Austin, May 1976.

[10] Butler W. Lampson AND David D. Redell, "Experiences with Processes and Monitors in Mesa", *CACM*, Vol. 23, No 2, February 1980, pp 105-117

[11] A.M. Lister and K.J. Mayland, "An implementation of Monitors", *Software - Practice and Experience*, Vol. 6, 1976. pp.375-385.

[12] Susan Owicki, "Verifying Concurrent Programs with Shared Data Classes" *Digital Systems Laboratory, Stanford University*

[13] Ashok R. Saxena and Thomas H. Bredt, "Verification of A Monitor Specification" University of Colorado, Hewlett-Packard Company, Stanford University

[14] P.D. Terry, "A Modula-2 Kernel for Supporting Monitors", *Software - Practice and Experience*,Vol. 16(5), May 1986, pp.457-472