Assured Selection — A Relaxed Concurrency Control Mechanism

© **Cinzia Foglietta**

LINKÖPINGS UNIVERSITET

Supervisor: Esa Falkenroth, Nancy E. Reed, Anders Törne
Examiner: Nancy E. Reed, Anders Törne

ISBN 91-xxxx-xxx-x   ISSN 0280-7971
Printed in linköping, Sweden
by linköping university

Ai
Miei Cari Genitori
Gino e Lorenza

Univerisita' degli Studi di Pisa
Facolta' di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

Anno Accademico 1998/99

Tesi di laurea

Assured Selection — A Relaxed Concurrency Control Mechanism

Il Candidato
Cinzia Foglietta

Il Relatore
Ch.mo prof. **Maurizio Bonuccelli**

Il Controrelatore
Ch.mo/ma prof.ssa

Sommario

Le tecniche di mutua esclusione sono tradizionalmente applicate in ambienti a dati condivisi per evitare inconsistenze quando processi in esecuzione concorrente accedono simultaneamente alle risorse comuni. Benche' tali tecniche siano efficaci, riducono il parallelismo durante l'esecuzione concorrente.

Assured Selection e' un meccanismo alternativo e ottimistico per il controllo della concorrenza basato sulla gestione di eccezioni. Questo meccanismo non mira ad evitare inconsistenze dei dati condivisi, invece rileva e risolve eventuali conflitti solo dopo l'occorrenza.

Quando la probabilita' di conflitto e' ridotta e il costo per il rilevamento e la gestione delle inconsistenze e' contenuto, Assured Selection aumenta il parallelismo rispetto alle tecniche di mutua esclusione tradizionali.

Questa tesi studia la semantica di Assured Selection. In aggiunta, prova la validita' del nuovo meccanismo attraverso risultati sperimentali e analisi teorica.

Abstract

Mutual exclusion techniques are traditionally applied in shared data environments to avoid inconsistencies when concurrent executing tasks simultaneously access common data. Although these techniques are effective, they reduce the parallelism of concurrent execution.

Assured Selection is an alternative optimistic mechanism for concurrency control based on exception handling. This mechanism is not aimed at avoiding inconsistencies on the shared data, instead detects and solves any corruptions after they have occurred.

When the probability of conflict is low and the overhead for the corruption detection and handling is small, Assured Selection improves parallelism compared with traditional mutual exclusion techniques.

This thesis investigates the semantics of Assured Selection. Furthermore, it proves the validity of the new mechanism through experimental results and a theoretical analysis.

Acknowledgements

I wish to express my deep and sincere gratitude to all who have provided invaluable discussions, advice and support throughout the genesis of the thesis.

I am especially thankful to my supervisor Esa Falkenroth for the idea of the thesis and for his many valuable comments, discussions, arguments and criticism.

Next, I wish to thank Nancy E. Reed and Asmus Pandikow for their valuable suggestions, comments and corrections while writing the report.

Thanks also to Anders Törne and all the other members of Real Time System Laboratory for providing me with a collegial atmosphere and a cooperative research environment in which to work.

Finally, I wish to thank my family and my dear friends Marco, Mustapha, Algirdas, Sonia, Anna, Ulf, Emanuela, Deborah and Mahmood for their moral suppor and the many encouraging words.

Thanks a lot to all of you!

Cinzia Foglietta

Contents

Chapter 1
Introduction

This chapter gives a short summary of the thesis and provides a description of the task. Next, it describes the state before the investigation and the audience intended for the report. Finally, it explains the report outline.

1.1 Summary of the Thesis

Monitors, semaphores and conditional critical regions are mutual exclusion techniques traditionally used in shared data systems for concurrency control. They avoid corruptions of shared data during concurrent access by serializing the task's access to the data. Although this serialization prevents corruptions, it also restricts parallelism in the concurrent execution since it forces the tasks into unproductive waiting.

An alternative approach for concurrency control is to allow tasks to operate simultaneously on the shared data, avoiding serialization during the concurrent access. However, since consistency might not be preserved, the idea is to detect possible corruptions afterwards and recover from them through exception handling.

Under the optimistic assumption of rare corruptions and having low overhead for corruption detection and handling, the result is an improvement of parallelism.

This Master's thesis investigates a concurrency control technique based on the approach mentioned above. The thesis studies issues related to the semantic definition of the new technique, analysing semantic choices and alternative solutions. Moreover, the mechanism studied is tested and validated through practical implementation. Lastly, it is compared with monitors through a theoretical analysis.

The results of the analysis prove that the new mechanism improves parallelism compared to traditional mutual exclusion techniques. However, this only holds when the probability of corruptions on the shared data is low. As the number of corruptions grows the benefits are lost and then traditional mutual exclusion techniques provide better performance. This suggests the combination of both the new technique and traditional ones for a more flexible and complete solution. An accurate evaluation of the possible conflicts on the shared data is needed to determine which technique is most efficient for each situation.

1.2 The Task and its Genesis

The idea for this Master's thesis emerged from a case study for a manufacturing control system investigated in the RTSLAB (Real Time Systems Laboratory) at Linköping University in Sweden.

The case study showed the necessity of avoiding concurrency problems when concurrent tasks simultaneously access shared resources. Due to the reduced probability of corruption on shared data, the use of mutual exclusion techniques could introduce unproductive delays in the concurrent execution reducing the parallelism of the tasks. This lead to the idea of Assured Selection (AS) an alternative approach for concurrency control based on exception handling.

The task of this Master's thesis is to further develop and define the idea of AS as a mechanism for concurrency control.

The proposal includes the following work:

- Investigation of choices, strategies and alternative solutions to define the semantics of AS.
- Validation of the approach through implementation and simulation.
- Evaluation of parallelism with AS as compared to mutual exclusion techniques.
- Specification of the benefits and limitations of the new mechanism.

A more detailed description of the task and the organization of the work done is presented in Chapter 5.

1.3 State Before the Investigation

AS was conceived to improve the parallelism in common resources systems during concurrent access to shared data. The idea is to reduce the strictness of locking during the concurrent access, allowing multiple processes to read and update the common data simultaneously. Although the parallelism is improved, conflicts can occur because of the relaxed locking, corrupting the common data. Inconsistencies are automatically monitored and subsequently solved using exception handling. The reader should refer to Sections 4.1 and 4.2 for further details about the current state of AS.

1.4 Audience

This report is aimed at designers and implementers of programming languages and concurrent systems where time is a critical issue. The report is intended for this audience because it defines the semantics for a new concurrency control technique. Moreover, it discusses the benefits of using the new technique when the number of corruptions on the shared data is limited. Finally, the experimental results with an implementation show the validity of the new mechanism and its easy integration in a concurrent programming language.

It is assumed that readers are familiar with high-level sequential and concurrent programming and concurrency issues. The specific background needed to understand the report is presented in Chapter 2.

1.5 Report Outline

This section gives information about the organization of the report.

Chapter 1 gives a short summary of the work done for the thesis. It also describes the task of the thesis, the state before the investigation and the audience for the report. Finally, it explains how the report is organized.

Chapter 2 introduces the background information needed to understand the thesis. At first, a short description of sequential and concurrent processes is given. Next, the mutual exclusion problem and the techniques traditionally applied to solve it are described. Then, concepts of exception handling and error recovery are introduced. Finally, the relation with the thesis is discussed.

Chapter 3 describes the problem of reduced parallelism when semaphores, conditional critical regions and monitors are applied for concurrency control. Moreover, it defines the measures used to evaluate delays and parallelism in concurrent systems.

Chapter 4 introduces Assured Selection (AS) as a new optimistic approach for concurrency control and describes the current state of its semantics. In addition, it mentions issues to be investigated for a complete definition of AS.

Chapter 5 presents and analyses the task of the Master's thesis project. Moreover, it explains choices made during the organization of the work and it introduces the requirements to be fulfilled during the investigation of AS. Lastly, it describes the restrictions made to the task.

Chapter 6 discusses possible semantics of AS. It provides a detailed classification of alternative solutions for AS and explains their semantics. In addition, it compares these solutions, selecting the one which best meets the requirements introduced in chapter 5.

Chapter 7 investigates further semantic issues of the solution selected in chapter 6, completing its definition.

Chapters 8 reports the experience of implementing AS. In particular, first it introduces the CAMOS system. CAMOS is the implementation environment in which the solution studied for AS has been integrated and tested. Next, it describes how AS has been integrated in CAMOS and presents some details of the implementation. Finally, it explains how the implemented solution has been tested and its performance evaluated in a case study.

Chapter 9 provides a theoretical evaluation of the performance of AS, not related with the implementation. The evaluation compares AS and monitors through a theoretical analysis.

Chapter 10 discusses the advantages and limitations of AS. Next, it explains the contribution given by the thesis to the existing idea of AS. Finally, it provides a summary of the thesis.

Chapter 12 examines unresolved issues and suggests future investigations.

Chapter 2 Background and Related Work

This chapter presents the theory and the background information necessary to understand the following chapters. First, it introduces the concept of sequential processes. Next, it explains the use of mutual exclusion techniques in shared data systems to coordinate the activities of concurrent cooperating processes. Moreover, it introduces exception handling and error recovery. Readers familiar with these background concepts may continue reading the last section of this chapter where the relation between the background information and the thesis is explained.

2.1 The Notion of Process

A sequential process is the thread of control of the program during execution. It starts when the first instruction of the program is processed and continues in sequential fashion executing one instruction at the time.

Two or more sequential processes may be associated with the same program, since they are considered two separate execution sequences. The path through the program may differ due to variations in input data but for any particular execution of the program there is only one path.

As a process executes, it changes *state*. The *state* of a process is defined by the current activity of that process. The diagram in Figure 2.1 shows the states that a sequential process reaches during its execution.

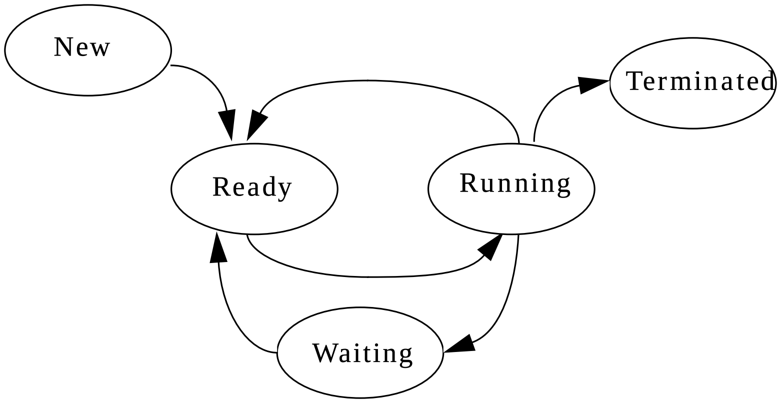


Figure 2.1: Diagram of process states [Sil94]

- New: the process is being created.
- Running: Instructions are being executed.
- Waiting: the process is waiting for some event to occur.
- Ready: The process is waiting to be assigned to a processor.
- Terminated: The process has finished execution.

During its execution, a process can create new processes. The creating process is called the parent process, whereas the new processes are called children of that process. When a process creates child processes, two possibilities exist in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all its children have terminated.

Figure 2.2 shows a tree structure of parent processes and their children.

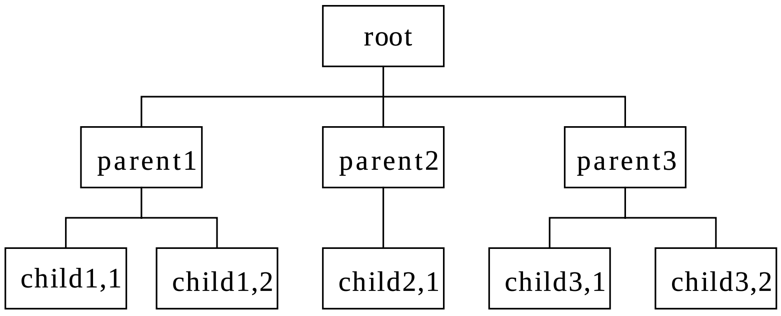


Figure 2.2: Tree of processes

A process terminates under the following circumstances:

- Completion of execution of the process body.
- Suicide, by execution of a "self-terminate" statement.
- Abortion, through the explicit action of another process.
- Occurrence of an unrecoverable error condition.
- When no longer needed.

Processes assumed to execute non-terminating loops, never terminate.

A parent process may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some resources it has

been allocated.
- The task assigned to the child is no longer required.

In addition when a parent process terminates, all its children are forced to terminate as well, or alternatively a parent cannot terminate before its child processes have terminated. The reason is that a parent process has usually a supervisor role during the execution of the children.

2.2 Concurrent Executing Processes

Concurrent programming languages all incorporate the notion of processes. A concurrent program can be seen as a collection of sequential processes that logically execute in parallel. The term concurrent refers to the potential parallelism.

Concurrent processes may be either independent or cooperating processes. Any independent process executes without affecting or being affected by other processes. In contrast, a cooperating process interacts with other concurrent processes. A process that shares data with other processes is an example of a cooperating process.

Cooperating processes need to communicate with each other during the concurrent execution. This can be done, using either shared variables or message passing. Shared variables are objects that more than one process has access to. Communication can therefore take place through these variables. Message passing is another means of communication. It involves the explicit exchange of data between two processes by means of a message that passes from one process to the other via some agency. In this thesis, only communication through shared variables will be considered.

2.3 Mutual Exclusion

Shared variables are a straightforward way of passing information between concurrent cooperating processes. However, their unrestricted use is unreliable and unsafe due to the multiple update problem. This section describes the mutual exclusion problem, consequence of the simultaneous access of processes to shared variables. Moreover, it presents the mechanisms traditionally applied to solve it.

Consider two processes updating a shared variable, X, with the assignment:

```
X := X + 1
```

On most hardware this will not be executed as an indivisible (atomic) operation but will be implemented in three distinct instructions:

(1) Load the value of X into some register
(2) Increment the value in the register by 1
(3) Store the value in the register back to X

As the three operations are not indivisible, two processes simultaneously updating the variable could follow an interleaving that would produce an incorrect result. For example, if X was originally 5, the two processes could each load 5 into their registers, increment and then store 6.

A sequence of statements that must be executed indivisibly to prevent incorrect interleaving is called critical section. The synchronization method required to protect a critical section is known as mutual exclusion.

Critical regions, semaphores and monitors are examples of mutual exclusion mechanisms that guarantee the indivisible execution of a critical section and thereby the consistency of shared data. The following subsections give a short overview of these mechanisms, further details can be found in [Sil94], [Bur97], [Law92] and [Tho98].

2.3.1 SEMAPHORES

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. The semantics for these operations is as follows:

- Wait(S): If the value of the semaphore S is greater than zero, then decrement its value by one; otherwise delay the process until S is greater than zero and then decrement its value.
- Signal(S): Increment the value of the semaphore S by one.

Wait and Signal are atomic operations. Therefore, two processes executing at the same time wait or signal on the same semaphore, cannot interfere with each other.

Mutual exclusion can be easily programmed using semaphores as shown with the following example:

```
(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)

process P1;
 loop
  wait (mutex);
    <critical section>
  signal (mutex);
  <non-critical section>
 end
end P1;

process P2;
 loop
  wait (mutex);
    <critical section>
  signal (mutex);
  <non-critical section>
 end
end P2;
```

If P1 and P2 are in contention, then they will execute their wait statements simultaneously. However, as wait is atomic, one process will complete execution of this statement before the other begins. One process will execute a wait(mutex) with mutex=1, which will allow that process to proceed into its critical section and set mutex to 0; the other process will execute wait(mutex) with mutex=0, and be delayed. Once the first process has exited its critical section, it will signal(mutex). This will cause the semaphore to become 1 again and allow the second process to enter its critical section (and once more set mutex to 0).

With a wait/signal bracket around a section of code, the initial value of the semaphore will restrict the maximum amount of concurrent execution of the code. If the initial value is 0, no processes will ever enter; if it is 1 then a single process may enter (that is, mutual exclusion); for values greater than 1, the number of allowed process corresponds to the value.

2.3.2 Conditional Critical Regions

A critical region is a section of code that needs guaranteed mutual exclusion. Variables that must be protected from concurrent usage are grouped together into named regions and are tagged as resources. Processes are prohibited from entering a critical region in which another process is already active. A Boolean guard governs the access to a region. When a process wishes to enter a critical region, it evaluates the guard (under mutual exclusion); if the guard evaluates true, it may enter, but if it is false, the process is suspended and the execution is delayed. Suspended processes must then re-evaluate their guard every time a process naming that resource leaves the critical region.

To explain the use of conditional critical regions, an example follows with two processes, one writing and the other reading characters from a bounded buffer.

```
Program buffer_example;
  type buffer_t is record
    slots      : array (1..N) of characters;
    size       : integer range 0..N;
  end record;

  buffer : buffer_t

  resource buf : buffer;

  process writer;
   ...
   loop
    region buf when buffer.size < N do
     --- place char in buffer ---
     end region;
    ...
    end loop;
  end;

  process reader;
   ...
   loop
    region buf when buffer.size > 0 do
     --- take char from buffer ---
    end region;
   ...
   end loop;
  end;
end;
```

The writer or the reader process accesses the shared resource **buf** in mutual exclusion and only if the correspondent guard to the critical region is satisfied. If it is false, the process is suspended (releasing the mutual exclusion on the shared resource **buf**) until this condition becomes true.

2.3.3 MONITORS

A monitor is a high-level data structure that collects critical regions in the form of procedures. Processes invoke these procedures, with appropriate arguments, when they wish to gain access to a shared resource. Only one process is granted access to the shared resource at a time; thus the procedures of the monitor are executed in mutual exclusion. The following code shows the general structure of a monitor.

```
Monitor MONITORNAME;
(* declaration of local data *)

  procedure PROCNAME (parameter list);
begin
 (* procedure body *)
end

  (* declaration of other local procedures *)

begin

  (* initialization data *)

end;
```

The monitor contains a declaration of local data, which is the data to be shared or control information concerning access to a shared resource. The procedures that can manipulate shared resources are then declared with appropriate parameter lists. The body of the monitor is executed at the beginning and provides any necessary initialization of the shared resource. Once initialized, processes can invoke the procedures (that are so-called entry points) of the monitor, passing actual parameters. The invocation of a monitor procedure can be done as follows:

```
MONITORNAME.PROCNAME (actual parameters)
```

The use of monitors makes the programming of mutual exclusion more flexible and less errors prone compared to semaphores or conditional critical regions.

2.4 Exception Handling and Exception Handlers

Reliability, safety and fault tolerance are requirements becoming more and more important in computer applications. For this purpose, exception handling is used to recover from the abnormal conditions that arise when exceptions occur.

After an error has occurred, the exception is first detected. Next, the process executing the operation that caused the error is notified. Finally, the notified process tries to solve the problem. The notification of exceptions is usually referred to as the raising or signalling of exceptions and the resolution of exceptions is referred to as exception handling.

Programming languages supporting exception handling have additional features in their structure for detection and handling of exceptions. First, these programming languages declare a new type of variable called an exception. Moreover, they allow the raising of exceptions from the points where the exception situation is detected. Finally, they group the set of statements to be executed when an exception is raised into modules known as exception handlers.

The following example shows how exceptions can be declared, raised and handled in a simple program that calculates the solution to a quadratic equation:

```
Procedure quadratic (a, b, c, r1, r2)
imaginary: exception

d:= b*b - 4*a*c
if d < 0 then raise imaginary endif
r1:= (-b + sqrt (d))/(2*a)
r2:= (-b - sqrt (d))/(2*a)

end procedure

exception handler

imaginary : print 'imaginary roots'

others : print 'fatal error'
        raise failure

end exception handler
```

As soon as an exception is raised, control is transferred to the corresponding place in the exception handler and the handling code is executed. The exception handler looks like a case statement, in which each exception is listed. Note that exceptions can also arise within exception handlers. In the previous example the exception **others** rises the exception **failure**.

During the execution of a program, exceptions can be signalled from the program itself, from other programs, from the operating system, or the hardware. Their recovery within the exception handler may be possible or not.

2.4.1 EXCEPTION PROPAGATION

After the occurrence of an exception, the process which caused the error is signalled and is in charge of handling the exception. A particular situation arises when no local handler to the signalled process is found to handle the exception.

One possible solution to this problem is to look for handlers higher in the chain of caller processes. This is called exception propagation. A potential problem with exception propagation occurs when the language requires exceptions to be declared within a given scope. Under some circumstances, it is possible for an exception to be propagated outside its scope. However, there are problems with variable visibility and parameter passing.

This section provided a short overview about exceptions and exception handling, further details can be found in [Bur97] and [Rom97].

2.5 Error Recovery

After an exception is detected and signalled, it is handled in the exception handler associated with the operation which caused the exception. Two strategies can be applied for the exception recovery: backward or forward error recovery.

Forward error recovery attempts to continue from an erroneous state by making selective corrections that lead to a new consistent state. Although forward error recovery can be efficient, it is system specific and depends on the accurate identification of the location and cause to errors. Examples of forward recovery techniques include redundant pointers in data structures and the use of self-correcting codes. An abort facility may also be required during the recovery process.

Backward error recovery restores the system to a safe state and the then executes an alternative section of the program. The point to which a process is restored is called a recovery point. To establish a recovery point it is necessary to save appropriate system-state information at run-time.

Further details about forward and backward error recovery can be found in [And81], [Cam86] and [Rom97].

2.6 Resumption, Termination and Signal Models

After an exception has been handled, an important consideration is whether the process that caused the exception should continue its execution. Three models have been studied to cope with this problem: the resumption, termination and signal models. They are briefly explained next.

2.6.1 THE RESUMPTION MODEL

This model can be applied when the exception handler is able to solve the problem that caused the exception. Once the exception has recovered, the execution of the operation that caused the exception can be resumed. The problem with this model is that errors raised by the run-time environment are often difficult to repair.

2.6.2 THE TERMINATION MODEL

In this model, when an exception has been raised and the handler called, control does not return to the point where the exception occurred. Instead, the block or procedure containing the handler is terminated and control passed to the caller block or procedure. This usually happens when the exception is not recoverable in the exception handler.

2.6.3 THE SIGNAL MODEL

This is a hybrid model in which the handler can decide whether to resume the execution of the operation that caused the exception or to terminate it. If the handler can recover the exception, the semantics is the same as in the resumption model. If the error is not recoverable, the execution of the operation that caused the exception is terminated and control passes to the caller block or procedure.

2.7 Relation of the Background Notions to the Thesis

The previous sections of this chapter presented the background information needed to understand the work presented in this work. This section explains the connection of these topics to the thesis.

Notions about processes and their concurrent execution, the mutual exclusion problem and techniques applied to solve it, are needed to understand the problem of reduced parallelism with concurrency control mechanisms (Chapter 3). Moreover, they are also needed to understand the solutions and semantic choices for the new mechanism presented in the Chapters 4, 6 and 7 and the comparison with monitors presented in Chapter 9.

Exception handling and error recovery notions are required to understand how the concurrency control mechanism investigated in this thesis detects and compensates for corruption on shared data.

Furthermore, semaphores, conditional critical regions and monitors are also related to the thesis. These techniques and the new mechanism follow different approaches but all of them are solutions for the concurrency control problem.

Chapter 3
The Problem

This chapter describes the problem of reduced parallelism when semaphores, conditional critical regions and monitors are applied for the concurrency control problem. Furthermore, it defines measures to estimate the parallelism and the delays which affect the concurrent execution.

3.1 Reduced Parallelism in Concurrent Systems Sharing Data

In concurrent systems sharing resources, several processes simultaneously access common data. To ensure both logical and timing correctness, a concurrency control mechanism is needed to synchronize the accesses.

Mutual exclusion techniques (monitors, semaphores and critical regions) are the mechanisms traditionally applied to preserve consistency. They ensure correctness by serializing the access to the common data. However, they can represent a bottleneck in the system since they limit the parallelism in the concurrent execution.

The problem with these techniques is the strictness of the lock which, during the mutual exclusive access, forces processes to perform unproductive waiting. In the worst case all the concurrent processes try to access the shared data at the same time. However, only one process at a time obtains the mutual exclusion. The others that are competing to access the resource are forced to wait.

Delays in the processes' activities and reduction of the *throughput* are the consequences. The delay for each process is directly proportional to the waiting time before accessing common data. The decrease of the *throughput* is inversely proportional to the total delay. When a process locks out other processes for a long time, the benefits of the concurrency are lost: concurrent execution becomes serial execution.

Traditional mutual exclusion techniques are pessimistic approaches to the concurrency control. They avoid conflicts by serializing the processes access to the shared data even when several processes could work in parallel without risks of collisions. For example, when the modifications produced by one or more processes do not introduce corruptions to the others. In these cases, the use of mutual exclusion techniques is inadequate since it delays the concurrent execution even if consistency is preserved. Moreover, the use of these techniques can introduce problems in systems that cannot tolerate delays in the processes' activities or in those ones for which the *throughput* has to be guaranteed to stay over a certain limit.

Two examples are introduced in the next sections to illustrate the reduction of parallelism in the concurrent execution when mutual exclusion techniques are applied.

3.1.1 EXAMPLE NR.1

An integer variable *w* is shared between a number *n* of concurrent processes $p_1 \dots p_n$. The program executed by process p_1 contains the portion of code in Figure 3.1.

```
          .
          .
          .

    if w>8 then begin
                  operation 1;
                  operation 2;
                  .
                  .
                  operation m;
                 end

          .
          .
          .
```

Figure 3.1: Portion of code for p_1

Furthermore, if the condition $w > 8$ is satisfied, the *m* operations in the *then* branch of the *if* statement should be executed under the guarantee that the value of *w* remains > 8. To avoid corruptions of the shared variable *w* during the execution of the *m* operations, the access to the variable *w* has to be synchronized. Using traditional mutual exclusion techniques process p_1 locks the access to the variable *w* has shown in Figure 3.2.

Consider a situation in where process p_1 starts the execution of the code in Figure 3.2 while the value of *w*=10 and the access to *w* is not locked by other processes. p_1 locks the access to *w* and starts the execution of the *if* statement. Assume now that another process p_2 tries to update the variable *w* to 9 while p_1 is executing the *if* statement. Process p_2 is blocked until p_1

unlocks the access to *w*. In this case, the blocking of p_2 is not needed. Both p_1 and p_2 could execute concurrently since the condition w>8 is satisfied even if the value of *w* is changed to 9 during the execution of the *if* statement.

```
                .
                .
                .
  ────────►  lock access to w
            if w>8 then begin
                        operation 1;
                        operation 2;
                        .
                        .
                        operation m;
                       end
  ────────►  unlock access to w
                .
                .
                .
                .
                .
```

Figure 3.2: Locking the access to *w*

3.1.2 EXAMPLE NR. 2

A long-running task collects statistical information about all customers and phone calls in a telecom database. To guarantee consistency, the long-running task must lock all phone-call records. This lock will prevent customers from making any calls while the long-running task executes. To improve the response time the long-running task may lock the phone-call record one by one, but then the long-running task may give an incorrect result since the phone-records may be modified under the duration of the long-running task.

Therefore, using a coarse-level lock will produce correct statistics but the response time will be unacceptable. Instead, the alternative where the records are locked one by one will give acceptable response time but the consistency of the long-running task will be compromised. Since incorrect data may lead to serious problems and global locks drastically reduce the inherent parallelism of the control application, none of these solutions is acceptable.

3.2 Problem Analysis

As explained in the previous section, when mutual exclusion techniques are applied, they can delay the concurrent execution and reduce the parallelism. In this section at first the delays which affect the execution of each concurrent process during the access to shared resources are estimated. Moreover, the *blocking time* which affects the concurrent execution is defined. Finally, the *throughput* is defined as a measure of the parallelism.

3.2.1 SINGLE-ACCESS DELAY, SINGLE-RESOURCE DELAY AND TOTAL DELAY OF CONCURRENT EXECUTING PROCESSES

Critical regions, semaphores and monitors serialize the access to shared data. Only the process which accesses the common resource first, obtain the mutual exclusion and is allowed to use the resource. All the others which try to access later are instead queued and wait till the mutual exclusion is released. Furthermore, when the resource is unlocked one queued process at a time in FIFO order obtains the mutual exclusive access on the common resource. The delay which affects each concurrent process during the access to common data can be estimated and expressed in term of the *Single-Access Delay, Single-Resource Delay* and *Total Delay.*

Consider a concurrent system with n_r common resources and n_p processes, and assume that the time for the processes' scheduling is infinitely fast compared to the time needed to use the shared resources. The *Single-Access Delay (SAD)* of a concurrent process p_i is the time p_i has to wait before it obtains the mutual exclusive access to the resource r_j during the access s_k. This time is zero if no other processes are using the resource, neither processes are queuing to use it at the moment of the access s_k . Otherwise, the *SAD* depends on:

- the time $t_{res}(r_j,s_k)$ needed at the moment of the access s_k to the process which is using the resource r_j to finish its access
- the number $n_q(r_j,s_k)$ of processes which are queued to obtain the mutual exclusion on the shared resource r_j at the moment of the access s_k
- the time $t_{use}(r_j)$ needed for each queuing process to use the shared resource r_j

Assuming $t_{use}(r_j)$ *is* constant and known for each access to the shared resource r_j, the *SAD* of a process p_i during the access s_k to the shared resource r_j can be expressed as:

$$SAD(p_i, r_j, s_k) = t_{res}(r_j, s_k) + n_q(r_j,s_k) \times t_{use}(r_j)$$

The SAD is maximum when p_i tries to access the common resource r_j and all the other concurrent processes are queued to use it. In this case, $n_q(r,s_k) = n_p - 1$ and $t_{res}(r_j,s_k) = 0$. The *SAD* of p_i during the access s_k to r_j becomes:

$$SAD_{max}(p_i, r_j, s_k) = (n_p - 1) \times t_{use}(r_j)$$

Moreover, knowing the number of accesses n_{acc} of process p_i to the resource r_j during the execution of its program, it is possible to estimate the total delay of process p_i during the n_{acc}

accesses to the resource r_j. This time is called *Single-Resource Delay (SRD)* and it is obtained by adding the *SAD* for each of the n_{acc} accesses of p_i to r_j as follows:

$$SRD(p_i, r_j) = \sum_{k=1}^{n_{acc}} SAD(p_i, r_j, s_k)$$

Finally, the *Total Delay* (TD) of a process p_i is defined as the total time p_i has to wait during its execution while accessing the n_r shared resources of the concurrent system. This time is obtained by calculating and adding together the SRD of process p_i for each of the n_r shared resources of the concurrent system.

$$TD(p_i) = \sum_{j=1}^{n_r} SRD(p_i, r_j)$$

SAD, SRD and TD have been defined in this section as measures of the delays which affect concurrent executing process when mutual exclusion techniques are applied for the concurrency control. In particular, SAD estimates the delay of a concurrent process during a single access to a shared resource. SRD gives a measure of the total delay of a concurrent process during its execution for the access to a single shared resource. Finally, TD is a measure of the total time a concurrent process is delayed during its execution because of the access to all shared resources in the system.

3.2.2 BLOCKING TIME DURING THE CONCURRENT EXECUTION

This section explains how the time needed for the execution of a concurrent program is delayed when mutual exclusion techniques are used for concurrency control. Moreover, it defines the *blocking time* as measure of the delay which affect the concurrent execution.

A concurrent program can be seen as a collection of processes which logically execute in parallel. Therefore, the time of execution of the concurrent program depends on the time of execution of the single processes. In particular, assuming that all concurrent processes start executing at the same time, the time to execute the concurrent program correspond to the time needed for the execution of the longest process. Figure 3.3 shows an example with three concurrent processes p_1, p_2, p_3 which start their execution at the time t_0. The time required for the execution of processes p_1, p_2 and p_3 are respectively t_1, t_2 and t_3. As the three processes execute in parallel, the concurrent execution starts at t_0 and finishes at t_2 when the execution of the longest process p_2 terminates.

When mutual exclusion techniques are applied, they can delay the execution of the concurrently executing processes even more. In particular, for each concurrent process, the time strictly needed to execute is extended by the waiting time spent during the access to shared resources. In the previous section, the TD (Total Delay) has been defined to estimate the total delay which affects each concurrent process during its execution.

Consider again the example in Figure 3.3 and assume that t_1, t_2 and t_3 are the times strictly needed for the three processes p_1, p_2 and p_3 to execute. Moreover, assume that p_1, p_2 and p_3 share resources and that mutual exclusion techniques are used to control the access to them. The total delays which affect processes p_1, p_2 and p_3 during the access to shared resources can be expressed with TD(p_1), TD(p_2) and TD(p_3). Figure 3.4 shows the new time of execution for p_1, p_2 and p_3.

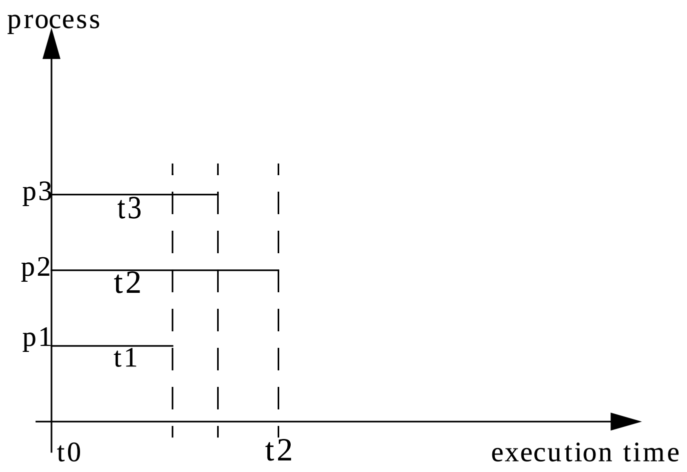


Figure 3.3: Time of execution of a concurrent program

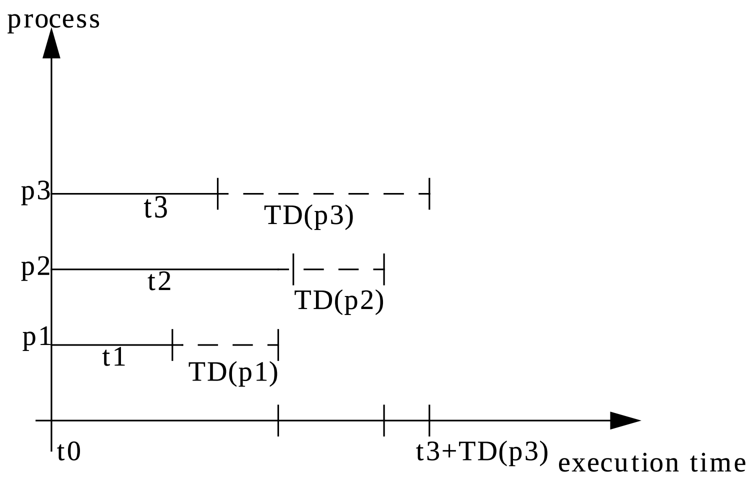


Figure 3.4: New time of execution for p_1, p_2 and p_3

As a consequence of the delay for each concurrent process, the time needed to execute the concurrent program becomes longer. In the example in Figure 3.4, the execution of the concurrent program starts at t_0 and ends at t_3 + TD(p_3), when the longest concurrent process terminates.

In the general case of a concurrent system with $p_1 \dots p_n$ concurrent executing processes, the time needed for the concurrent execution (namely the time needed for the concurrent program to execute) can be expressed as follows:

$$T_{conc} = max(t_i + TD(p_i))$$

and corresponds to the time needed for the process p_i with the largest execution plus waiting time sum to terminate its execution.

Furthermore, a measure of the delay which affects the execution of a concurrent program can be given and will be referred to as *blocking time* of the concurrent execution.

$$BlockingTime = T_{conc} - max(t_j)$$

In particular, in the formula above, t_j is the time strictly needed for a process p_j to execute.

The T_{conc} and the *BlockingTime* defined in this section will be used in Chapter 9 to estimate the decrease of parallelism in concurrently executing systems when concurrency control techniques are applied.

3.2.3 THROUGHPUT AS A MEASURE OF PARALLELISM

In this section the *throughput* is defined as measure of parallelism.

The throughput of a system is a ratio which measures the speed at which the system works. This ratio can be expressed in instructions per second or jobs per hour or some other units of performance. In this thesis the throughput will be expressed in term of work produced per time unit.

$$Troughput = \frac{WorkProduced}{TimeUnit}$$

In particular, the throughput of a concurrent system will be calculated dividing the units of work produced during the execution of a concurrent program by the time needed to execute the program.

$$Troughput = \frac{UnitsOfProducedWork}{TimeNeededToExecuteTheProgram}$$

As the time to execute a concurrent program can be delayed when mechanisms for the concurrency control are used, the throughput decreases in a proportional way.

The throughput of a concurrent system will be used in Chapters 8 and 9 to evaluate the parallelism of concurrent systems when concurrency control techniques are used.

3.3 Summary

Concurrency control is needed in concurrent sharing data systems to ensure integrity of the common data. Using traditional mutual exclusion techniques, processes achieve mutual exclusion on the shared resources by locking the access to them. Although integrity is preserved, tasks may be forced to wait for others to finish. The problem is that strictness of the lock blocking the concurrent execution may be more than is actually necessary. Consequences of this blocking are delays for the tasks' activities and therefore reduced parallelism. In this chapter the Single-Access Delay, the Single-Resource Delay, the Total Delay and the *blocking-time* have been defined as measures of the delays which affect the execution of concurrent tasks. In addition, the *throughput* has been defined as a measure of parallelism.

Chapter 4
Approach and Current State of Assured Selection

This chapter presents Assured Selection (AS) as an alternative solution for concurrency control. At first, a description of the approach use by AS is given. Then, the terminology, the syntax and the issues related to a semantic definition of AS are introduced.

4.1 Fine Grained Locking and Exception Handling

As explained in Chapter 3, traditional mutual exclusion techniques reduce the parallelism of concurrent processes sharing data. The problem is that the strictness of the lock that forces processes to perform unproductive waiting. AS is an alternative approach to concurrency control that solves this problem by relaxing restrictions during concurrent access.

The approach combines fine grained predicate locks with exception handling. The fine grain of the locks allows more tasks to execute in parallel on the shared data, reducing the blocking during concurrent access. However, the relaxed blocking does not ensure integrity and can lead temporarily to inconsistent data.

To avoid error propagation, corruptions must be detected and resolved afterwards. Error detection can be done by monitoring the asynchronous access to the shared data. Once detected, a corruption can be handled within exception handling using backward or forward error recovery.

The idea of AS is to ensure consistency and at the same time to reduce the blocking without incurring unbounded or excessive run-time overhead when monitoring and solving corruptions. Under the assumption of low probability of corruption, an improvement of the throughput is expected.

A more detailed description of the approach, the terminology and the syntax for AS are given in the following sections.

4.1.1 GUARD PREDICATES, ASSURED REGIONS AND EXCEPTION HANDLERS

AS combines *guard predicates*, *assured regions* and *exception handlers*:

- A *guard predicate* is a Boolean expression which contains shared variables. If the Boolean value of the guard predicate is satisfied the corresponding assured region is entered.
- An *assured region* is a section of code executed under the assumption that the associated guard predicate remains satisfied. However, no restrictions are specified on the concurrent access when entering this region and simultaneous updates of shared variables can corrupt the value of the

guard. During execution within this area, corruptions are automatically monitored and notified.

- An *exception handler* is a portion of code associated with an assured region and executed to restore integrity when a corruption of the predicate guard is notified within the assured region.

Later, an assured region and the corresponding predicate guard will be referred to as AS block.

In Figure 4.1, the sequence of execution of the described parts of AS is shown:

1. The guard predicate is evaluated.
2. If the Boolean value of the guard predicate is true, the execution of the assured region starts. Updates of the shared variables contained in the guard predicate are automatically monitored within the assured region and corruptions are signalled.
3. If a corruption of the monitored data is signalled, execution continues in the exception handler.

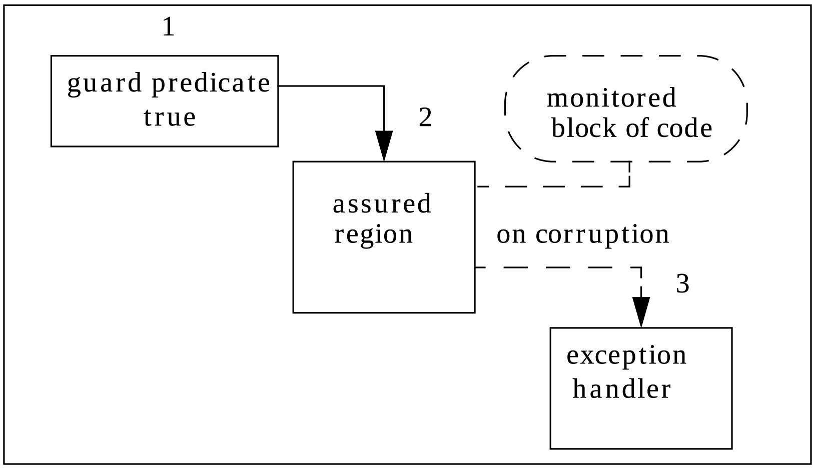


Figure 4.1: Sequence of execution of AS.

4.1.2 THE SYNTAX OF AS

The syntax of AS is similar to an *if* statement with an exception handler associated to the *then*-branch.

The Boolean expression in the test is the guard predicate, the portion of code in the then branch is the assured region. If the evaluation of the guard predicate is true, the assured region is executed. If it is false, the execution continues in the else branch (if any) as in ordinary if statements.

An asterisk following the *if* term is used to distinguish the AS statement from an *if*. Figure 4.2 shows the parts of the AS statement.

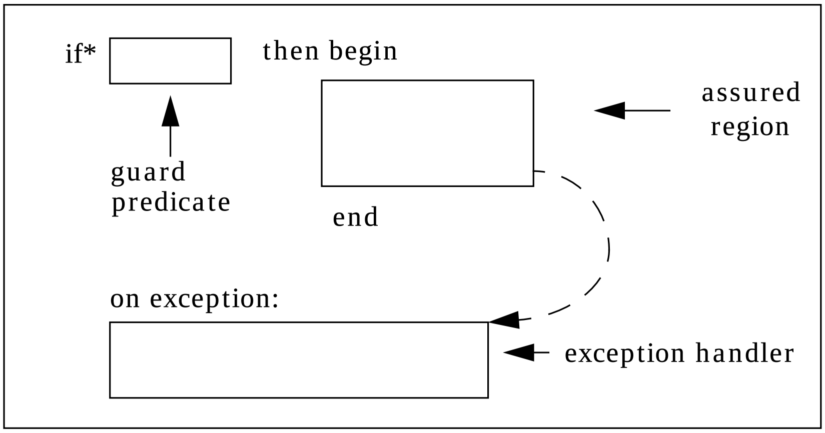


Figure 4.2: AS statement.

With referring to the example NR.1 in Section 3.1, the program of process p_1 using AS can be written as in Figure 4.3.

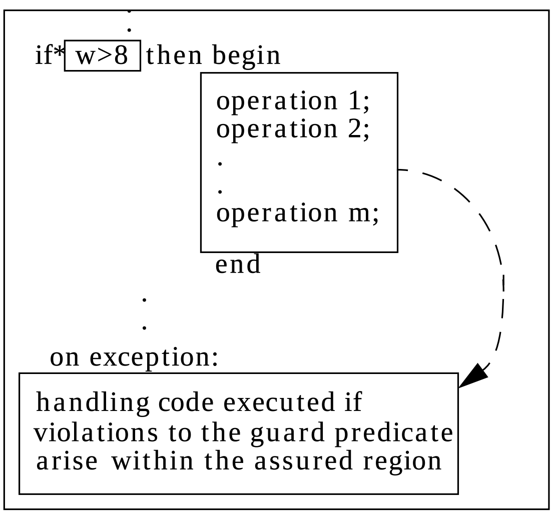

.
.
if* w>8 then begin
operation 1;
operation 2;
.
.
operation m;
end
.
.
on exception:
handling code executed if violations to the guard predicate arise within the assured region

Figure 4.3: Section of program for process p_1 using AS

4.1.3 VIOLATIONS, VIOLATING AND VIOLATED PROCESSES

Within an assured region, corruptions on shared data can occur due to the relaxed restrictions on the concurrent access.

A process that corrupts the value of an assured guard is referred to as *violating process*. The process that executes within the assured region during the corruption is called *violated process*. The corruption is referred to as *violation* and the notification to the violated process is called *signal*.

Figure 4.4 shows two processes A and B in concurrent execution. While A is executing an AS block, B violates the guard predicate of the AS block that A is executing. The violation is signalled to process A.

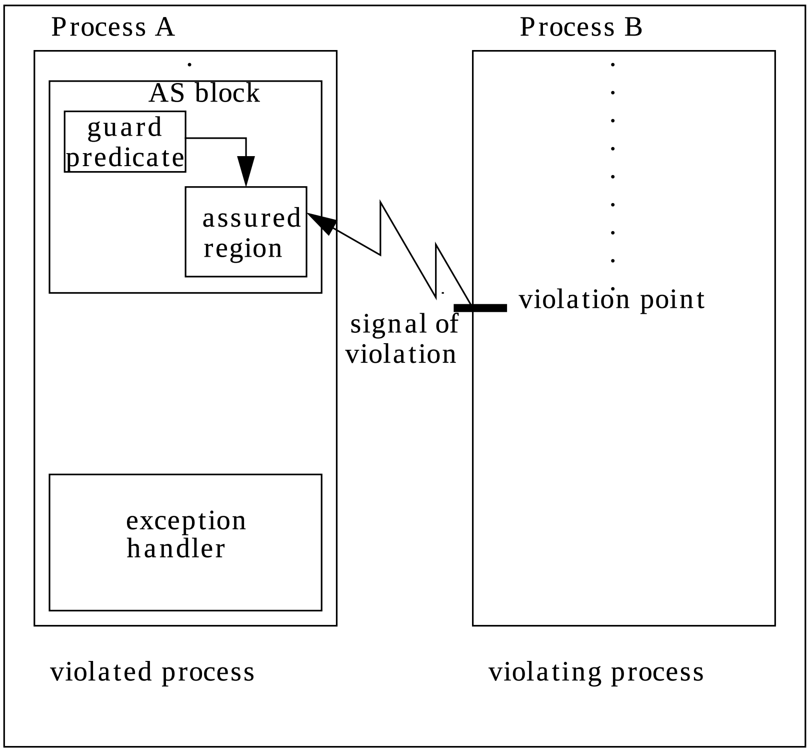


Figure 4.4: Signal of violation to the violated process

4.2 Current State of the Semantics of AS

This section explains the current state of the semantics of AS, pointing out, with examples, semantic choices to be studied.

As mentioned in the previous section, purpose of AS is to relax restrictions on the concurrent access. Assured regions are entered without locking the access on the shared variables. If during the execution within this region the guard predicate remains satisfied, all processes can concurrently read and update the shared variables of the guard. Lock operations are applied for the time strictly needed to update a shared variable.

In the example presented in Figure 4.3, while process p_1 is executing in the assured region other processes can access and update the value of *w*. As long as the updates do not change the value of the guard predicate, the concurrent access to *w* is allowed. Figure 4.5 shows a situation in which process p_1 is executing operation 2 in its assured region (the value of *w* = 10) and a process p_2 updates the shared variable *w*. Since the update does not violate the value of the guard predicate, conflicts do not arise. The two processes continue their execution without limitation on their parallelism.

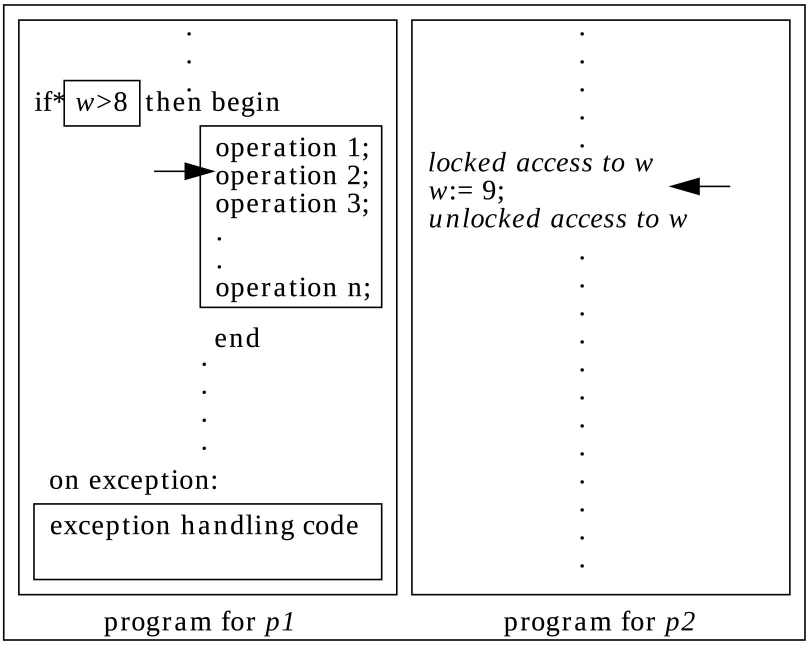


Figure 4.5: Concurrent execution of p_1 and p_2 where both processes simultaneously operate on the shared variable *w*

Instead, if the update of p_2 to w changed the Boolean value of the guard predicate (ex. w:= 3), a violation would have occurred.

After a violation of a guard predicate has occurred, the execution of the violated process in the assured region is stopped and continues in the exception handler.

Figure 4.6 shows the state before the violation between two concurrent processes A and B, when A is executing within an assured region. The dashed lines represent the point of execution in the two processes.

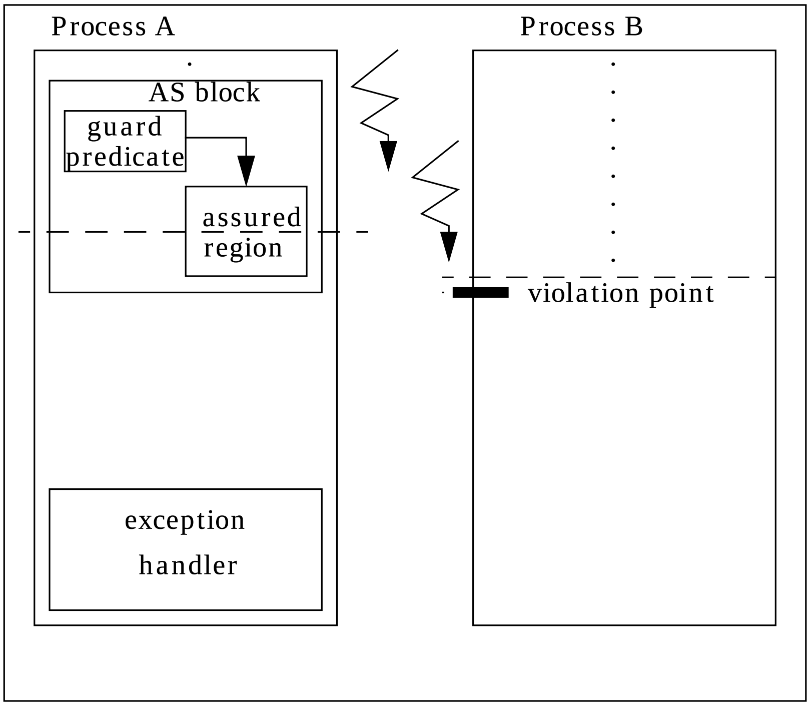


Figure 4.6: State of execution of A and B before the violation

Figure 4.7 summarizes the situation after the violation. After it is detected, the violation is signalled to the violated process (1) which continues execution in the exception handler (2).

Both the solutions to continue or stop execution in the violating process can be valid and need to be studied.

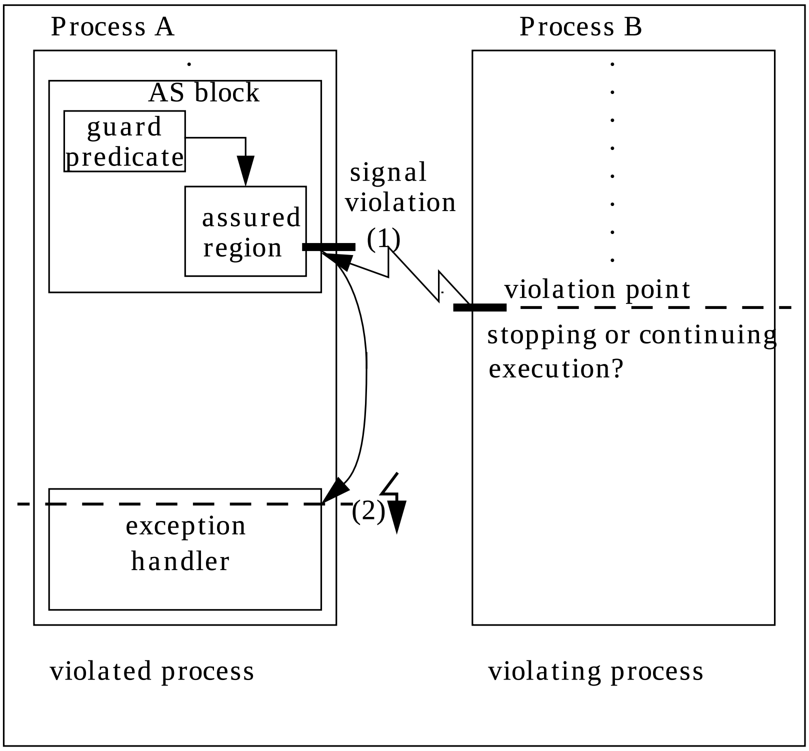


Figure 4.7: State of execution of A and B after the violation

As mentioned before in this chapter, in the exception handler the violation is compensated for using backwards or forwards error recovery. After the violation has been handled, aborting, restarting or resuming execution in the violated process are valid subsequent alternatives. Furthermore, the violated process can be resumed from (3) different resumption points: from the beginning, from the AS block or from the break point within the assured region as shown in Figure 4.8.

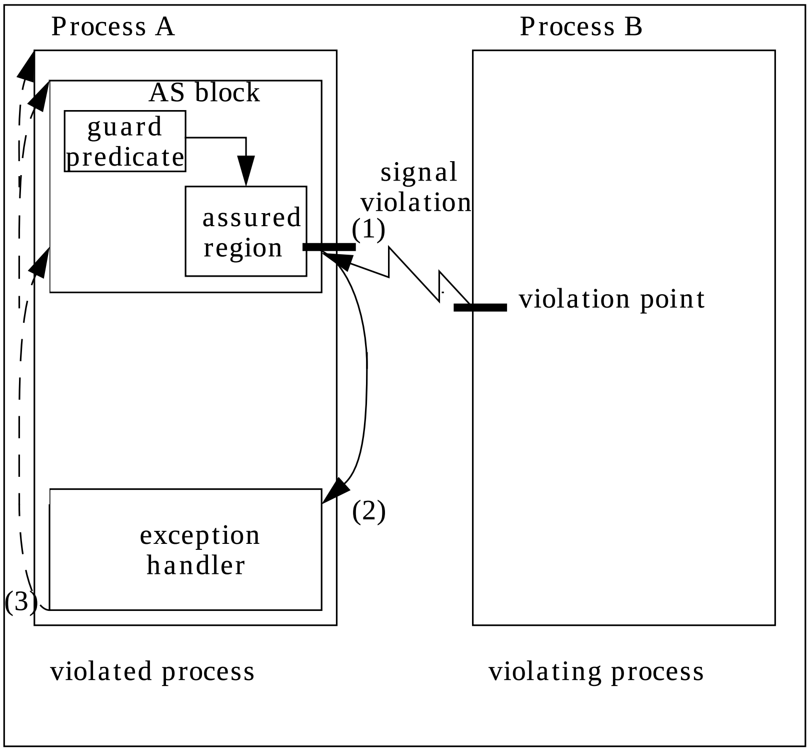


Figure 4.8: Alternative resumption points for the violated process: from the beginning, from AS block, from the break point within the assured region

Whether to block or not block the violating process, restart, abort or resume the violated process and the resumption points are choices to be studied to define the semantics of AS. Moreover, in the solution presented for AS it is the violated process that gets and handles the violation. The possibility that the violating or another process handles the violation instead of the violated one needs to be analysed as well. A more complete list of semantic issues to be discussed for AS is presented in the next section.

4.3 Semantic Issues of AS

To define AS as a mechanism for the concurrency control its semantics needs to be defined and the following choices need to be investigated:

- What processes should be blocked after a violation?
- When are blocked processes unblocked?
- When is a violation signalled and to which processes?
- What happens if the violated process has child processes? Should they be stopped or aborted or can they continue their execution?
- Which process should handle the violation? Is it the violated, the violating or another process?
- What happens if a process violates itself?
- How is a violation handled?
- What happens to the signalled processes after the violation has been handled? Are they aborted, restarted or resumed?
- From which point are the signalled processes resumed? Is it from the beginning, from the AS block or from the break point within AS?
- How should nested violations of a guard predicate be dealt with?
- What happens if a violating process gets violated?

All these choices are discussed in details in Chapter 6 and 7 when solutions for AS are analysed.

Chapter 5 The Task

This chapter analyses the work to be done to solve the task of the thesis introduced in Section 1.2. In addition, it mentions the restrictions made to the task.

5.1 Analysis of the Task

The task of this Master's thesis is to develop the idea of AS as presented in Chapter 4. This section presents choices and decisions made during the organization of the work for the thesis. Moreover, it presents the requirements to be fulfilled during the investigation and the implementation of AS. Finally, it explains how the work to solve the task has been organized.

5.1.1 THE CHOICES AND THE DECISIONS

Four choices have been made during the organization of the work to this thesis:

1. The theoretical and experimental scope.
2. The generality of the investigation.
3. The environment for the investigation.

4. The evaluation of performance for AS.

The first choice concerned the scope of the thesis. Two alternatives have been considered: to develop a totally theoretical investigation or to combine experimental results along with the theoretical studies. The second choice was about the generality of the investigation. AS could be specialized for a particular system or studied at more abstract level, finding a more general solution adaptable to specific systems. Once the scope of the thesis and the generality of the investigation were decided, the next choice was to select a testing environment. The final choice concerned the evaluation of the performance of the new mechanism, through simulation or theoretical analysis.

The decisions taken and the motivations are explained as follows:

1. **Theoretical investigation versus implementation**. The decision for the scope of the thesis, was to combine a theoretical investigation with experimental results. The reason for this was to validate the studies of the theory with practical experience and also to explore issues of AS not pointed out from the theory.
2. **General solution versus system specific**. The decision for the generality of the investigation was to study the semantics for AS at a general level and subsequentially to model and implement it in a specific system. The motivation was to find a solution for AS of general interest, not linked to any specific system and easy to adapt in different concurrent environments.
3. **Concurrent shared data systems versus CAMOS**. As a consequence of the desired generality of the solution, the theoretical investigation has been thought to be suitable for all concurrent sharing data environments. However, the environment chosen for the implementation work was the CAMOS system for control of manufacturing plants (see Chapter 8 for a description of CAMOS). The motivation for select-

ing CAMOS was the natural suitability of its environment to AS, that made the integration of the solution fairly easy. Moreover, an existing case study programmed in CAMOS could be used to test and evaluate the performance of the new mechanism implemented.

4. **Theoretical analysis versus simulations.** Concerning how to evaluate the performance, the decision was to use a theoretical analysis for the comparison of AS and traditional mutual exclusion techniques. Furthermore, a case study was chosen to analyse the performance of the mechanism implemented in CAMOS through graphical simulations. The motivation for combining theoretical analysis and simulations was to provide a more accurate and complete analysis of the results.

5.1.2 REQUIREMENTS ON THE SEMANTIC DEFINITION OF AS AND ON THE IMPLEMENTATION IN CAMOS

As a consequence of the decisions mentioned above, the task of the thesis has been organized in two parts: the theoretical investigation and the implementation. To define AS as a mechanism for concurrency control, the theoretical investigation includes studies of semantic issues of AS. The implementation work includes the integration of one solution for AS studied in the theory in CAMOS. This section introduces the requirements to be met during the semantic definition of AS and during the implementation.

Requirements on the Semantic Definition of AS

The solution investigated to define AS as a mechanism for the concurrency control must satisfy the following requirements:

- Clear and easy semantics. The semantics of the solution must be clearly defined and easy to understand.

- Generality. The solution must be adaptable in different environments and situations.
- Completeness. The solution must be as complete as possible.
- Maximum parallelism. The solution should allow maximum parallelism by reducing the delays in the concurrent execution.

These requirements are discussed in Chapter 6 when alternative solutions for AS will be analysed.

Requirements on the Implementation of AS in CAMOS

The following requirements must be fulfilled during the integration of AS with the CAMOS system:

- Few lines of added/changed code.
- Separation of exception handling code from normal code.
- Improvement of parallelism.

The requirements on the implementation of AS in CAMOS are discussed in Chapter 8 when the implementation experience is described.

5.1.3 ORGANIZATION OF THE WORK

According to the choices mentioned in Section 5.1.1 and to the requirements on the semantics and on the implementation of AS described above, the task has been organized as follows.

The theoretical investigation includes the following activities:

- Definition and analysis of the problem of reduced parallelism with traditional mutual exclusion techniques.
- Investigation of alternative solutions to the concurrency control following the approach for AS introduced in Chapter 4.
- Classification, analysis and comparison of the alternatives, with respect to the requirements introduced in Section 5.1.2.
- Selection of the solution that matches these requirements best.

- Definition of semantic issues for the selected solution.
- Evaluation of the performance of the resulting mechanism with comparison to traditional mutual exclusion techniques.
- Specification of the benefits and limitations of the new mechanism.

The implementation consists of the following work:

- Implementation of AS and integration with the CAMOS system.
- Experimental tests on the correctness of the implemented AS mechanism.
- Evaluation of the performance of the implemented AS mechanism in a case study.

5.2 Scope of the Task

This section presents the restrictions made on the task during the investigation and the implementation of AS.

At first, since the intention was to keep the focus on the many semantic issues of AS, formal definitions have not been used to express the semantics of AS. Nor have proof rules been provided to verify the correctness of AS.

The second restriction concerns the comparison between the performance of AS and traditional mutual exclusion techniques. Semaphores, conditional critical regions and monitors realize the concurrency control following the same approach. They lock access to shared resources serializing the concurrent access to them. In this sense, their performances has been considered to be equivalent. As a consequence, the comparison has been done only between AS and monitors as the main representative of the class of mutual exclusion techniques. Moreover, the kind of analysis done for the comparison of the performances is a worst case analysis. Average case analysis is beyond the scope.

A further restriction has been made during the implementation. Since the purpose of the implementation was only to validate or refute the approach of AS and also due to restrictions in time, not all the issues investigated for the theoretical solution have been implemented and tested.

5.3 Summary

In this Chapter the task of this master thesis has been presented and analysed. The task is to study the idea of AS presented in Chapter 4 and includes the following activities:

- Definition and analysis of the problem of reduced parallelism when traditional mutual exclusion techniques are applied for the concurrency control in shared data systems.
- Investigation of alternative solutions for AS following the approach described in Chapter 4.
- Analysis, classification and comparison of the alternatives.
- Selection of the solution that meets the requirements presented in Section 5.1.2 best.
- Definition of semantic issues for the selected solution.
- Implementation of the resulting mechanism in the CAMOS system.
- Experimental tests on the correctness of the implemented mechanism.
- Evaluation of the performance of the implemented mechanism through simulations.
- Comparison of the performance of AS and monitors through a worst case analysis.
- Specification of the benefits and limitations of the studied mechanism.

Chapter 6
Solutions for AS

This chapter classifies, explains and compares solutions for AS. First, the classification criteria are introduced and a classification of solutions is presented with respect to these criteria. Next, the semantics of the classified solutions is explained. Lastly, the solutions are compared to select the one that meets the requirements on the semantic definition of AS best.

6.1 Classification Criteria

This section introduces the exception handler process and the blocking, quasi-blocking, non-blocking behaviour of the violating process as criteria to classify solutions for AS.

6.1.1 EXCEPTION HANDLER PROCESS

According to the current state of AS introduced in Section 4.2, the violated process is in charge of handling violations of the guard predicate of an AS statement. When an exception is signalled, the violated process stops its execution in the assured region and continues in the exception handler associated with the AS block.

However, other processes could recover from a signalled violation. For example, the violation could be handled by the violating process, the parent of the violated process, both the violated process and its parent or another process. Depending on the exception handler process, alternative solutions for AS can be studied.

6.1.2 BLOCKING, NON-BLOCKING, QUASI-BLOCKING AS

Referring to the semantics of AS presented in the Section 4.2, after a violation of an AS statement is signalled, both alternatives to stop or to continue execution in the violating process, can be valid. The blocking, non-blocking, quasi-blocking semantics of AS refer to the behaviour of the violating process after the violation.

Using a non-blocking semantics the violating process continues execution after it has violated a guard predicate. The violated process and the exception handler process, if different from the violated, are the only processes affected by the violation. Their execution is delayed for the time necessary for the recovery.

With a blocking semantics instead, in addition to the violated and the exception handler process, also the violating process gets delayed. After the violation the execution in the violating process is blocked and resumed only after exception handling. The blocking semantics is needed for example when the violating process continues reading the corrupted data or would update it generating other violations. However, it is not really clear when to apply the non-blocking or the blocking semantics since it depends on the situation and the environment of execution.

An alternative solution is to delay the decision to stop or not stop a violating process at run-time, combining the blocking and the non-blocking semantics within the quasi-blocking semantics. The violating process suspends itself after the violation and the conditions to resume its execution are evaluated within the exception handler. The violating process is resumed as soon as possible, after it is verified that its execution does not interfere with the error recovery.

The exception handler process, the blocking, non-blocking, quasi-blocking semantics are used in the next section as criteria to classify alternative solutions for AS.

6.2 Classification and Analysis of Solutions for AS

This section presents a classification of solutions for AS and explains their semantics. Initially, the solutions are grouped in five categories based on whether the exception handler process is the violated, the violating, the parent of the violating, both the violating and its parent or another process. Moreover, for each category three subcategories are provided based on the blocking, non-blocking, quasi-blocking semantics for the violating process. Table 6.1 shows the resulting classification. The idea for the classification has been found in [Buh95] where, similarly, different types of monitors have been classified.

Table 6.1: Classification of solutions for AS

Exception handler process	**blocking violating (B)**	**non-blocking violating (NB)**	**quasi-blocking violating (QB)**
violated process (VD)	VDB	VDNB	VDQB
violating process (VG)	-----	VGNB	-----
parent of violated process (P)	PB	PNB	PQB
violated and its parent (VP)	VPB	VPNB	VPQB
other process (O)	OB	ONB	OQB

The semantics of the solutions classified above will be explained through an example with two processes B and C in concurrent execution. The program executed by B contains an AS block and the one executed by C a critical instruction that, depending on the moment of execution, could violate the guard predicate of the AS block in B. See Figure 6.1.

If process C executes the critical instruction before or after process B executed the AS block, the violation of the guard predicate does not occur. Therefore, all the solutions in table 6.1 have the same behaviour. The concurrent execution of B and C continues without limitation of parallelism.

Instead, the semantics of the classified solutions are different if the violation is signalled. This happens if the critical instruction in C is executed while process B is executing in the assured region.

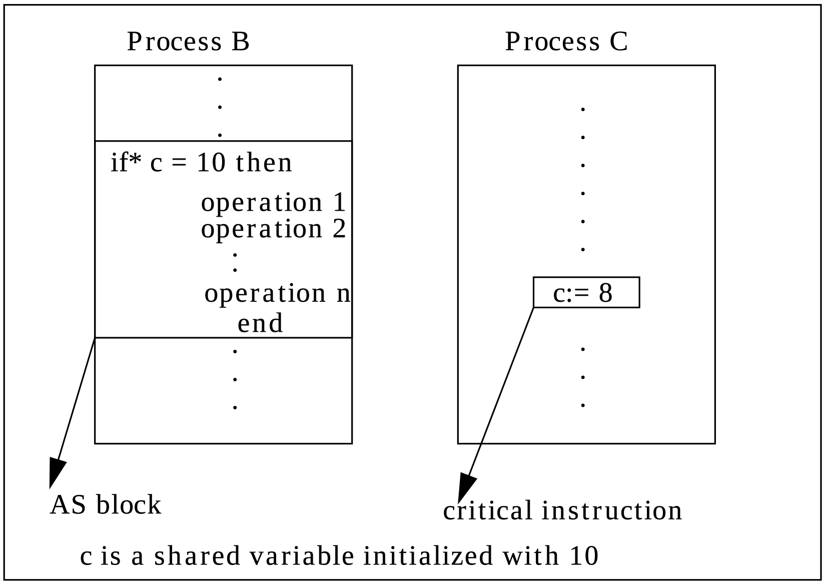


Figure 6.1: Sections of programs executed by processes B and C

To focus the differences, for each category of solutions in the table 6.1 two moments of the concurrent execution need to be analysed: before and after the violation. This analysis is presented in the following five sections. However, for the categories of solutions VD, P, VP and O only the QB subcategory is described in detail. Assuming the same state of execution before the violation, the subcategories B, NB can be easily derived from QB as explained in the next sections. For the category VG instead, only the NB subcategory is presented since it is the only one defined.

6.2.1 THE VIOLATED PROCESS IS THE EXCEPTION HANDLER (VD)

In this category of solutions the exception handler is part of the program of the process that executes the AS statement.

Referring to the example in Figure 6.1, Figure 6.2 shows the state before the violation for the subcategories of solutions VDB, VDNB and VDQB. Process B is executing in the AS block and C is before the critical instruction. The dashed line shows the point of execution.

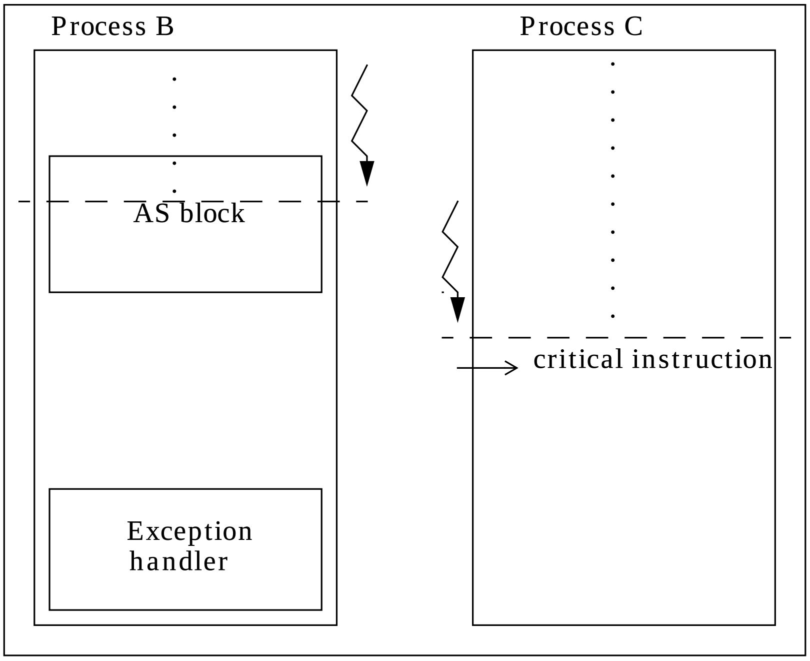


Figure 6.2: VDB, VDNB and VDQB: state of execution before the violation

If process C executes the critical instruction before B exits the AS block, the violation occurs.

Figure 6.3 shows the state after the violation for the subcategory of solutions VDQB. After the violation has occurred, process C temporary suspends its execution (1). The exception is signalled to the violated process B (2) that continues execution in the exception handler (3). Before the exception handling takes place, process B checks conditions to resume the violating process. The violating process is resumed immediately if its execution does not interfere with the violation recovery, otherwise the violating process is unblocked as soon as possible during the exception handling (4). After the error recovery and the unblocking of the violating process, process B aborts, restarts or resumes its own execution. The resumption point is decided within exception handling. Possible resumption points (at the beginning of the program, at the beginning of the AS statement or at the break point within AS) are shown in the figure labelled as (5).

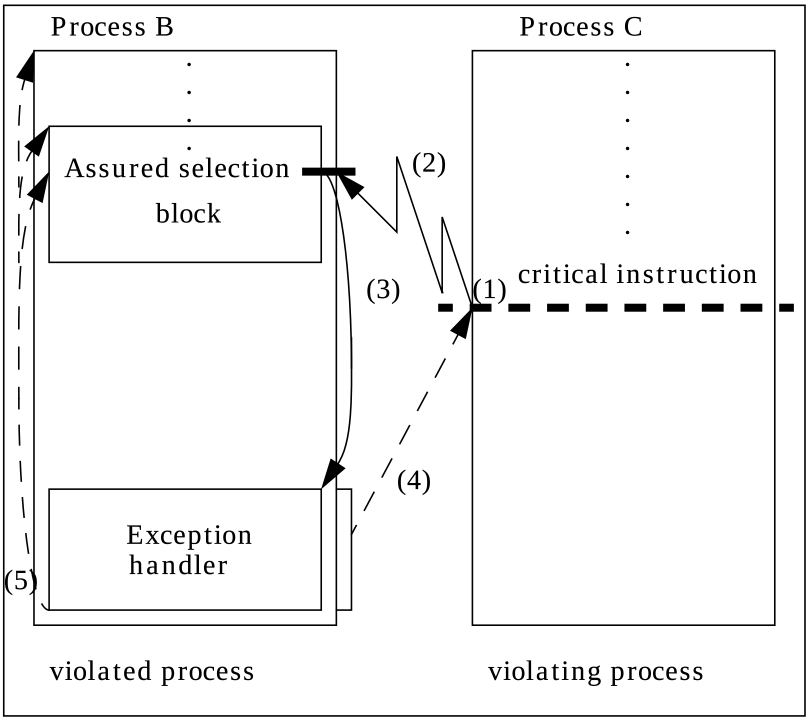


Figure 6.3: VDQB: state after the violation

For the subcategories of solutions VDB and VDNB the state after the violation appears similar to the one shown in Figure 6.3. The difference for the category VDB is that the violating process gets blocked after the violation and is unblocked by the violated process only after the violation recovery. For the subcategory VDNB instead, the difference is that the violating process continues execution after the violation. Therefore, the violated process does not handle unblocking the violating process.

6.2.2 THE VIOLATING PROCESS IS THE EXCEPTION HANDLER (VG)

In this category of solutions the exception handler associated with the AS block is part of the program of the violating process. After a violation of the guard predicate is signalled, the violating process stops its execution after the violating instruction and continues in the exception handler. As the violating process is in charge of recovery from the violation, its execution cannot be blocked after a violation. Therefore, only the non-blocking subcategory of solutions is defined within the category VG.

Referring again to the example in Figure 6.1, the state before the violation for the subcategory of solutions VGNB is shown in Figure 6.4. Process B is executing within the AS block and process C is before the critical instruction. The dashed line shows the point of execution in the two processes.

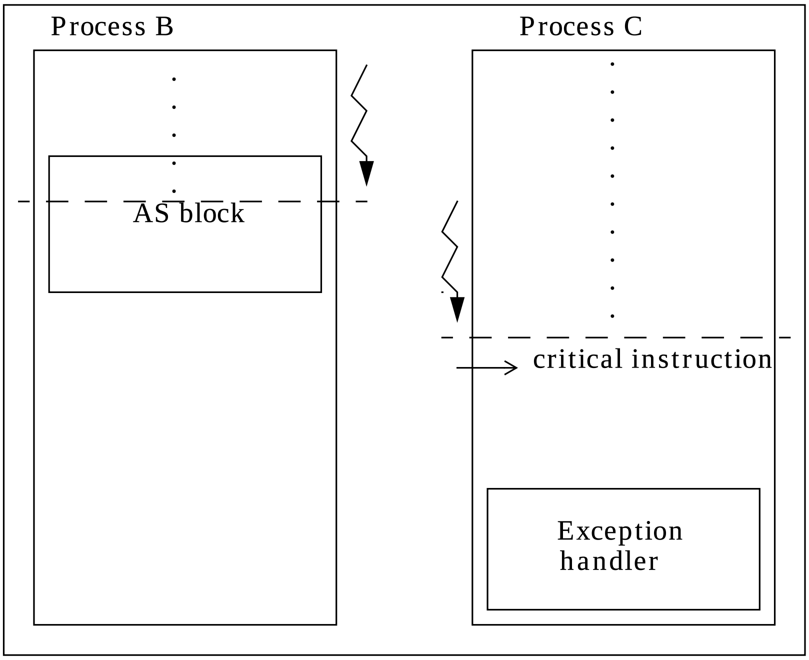


Figure 6.4: VGNB: state before the violation

If process C executes the critical instruction when B is executing in the assured region the violation occurs.

The state after the violation for the subcategory of solutions VGNB is shown in Figure 6.5. The violation is signalled to the violated process B (1). B stops its execution to avoid error propagation (2). Instead, the violating process C continues in the exception handler (3) and recovers the violation. Furthermore, process C resumes process B from the point decided within exception handling (possible resuming points are shown labelled (4)). Finally, process C resumes its own execution after the critical instruction (5).

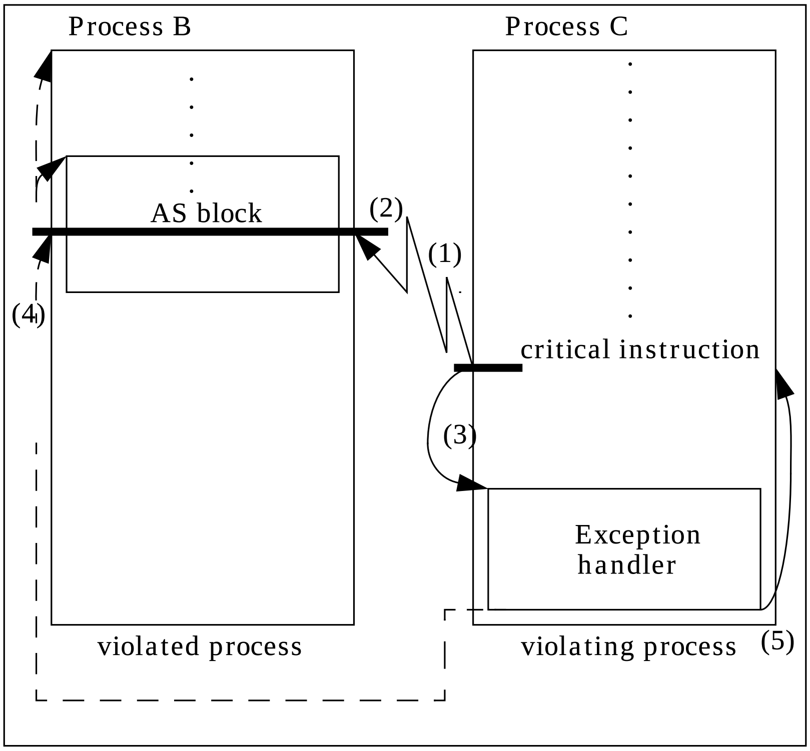


Figure 6.5: : VGNB: state after the violation

6.2.3 THE PARENT OF THE VIOLATED PROCESS IS THE EXCEPTION HANDLER (P)

In this category of solutions the exception handler is in the program of the parent of the process that contains the AS block.

Referring to the example with processes B and C introduced in Section 6.2, now it is assumed that a process A is the parent of process B. The state before the violation between B and C for the subcategories of solutions PB, PNB and PQB is shown in Figure 6.6.

Process A is executing within its program after having created by process B, B is executing within the AS block and process C is before the critical instruction. The dashed line shows the point of execution in processes A, B and C.

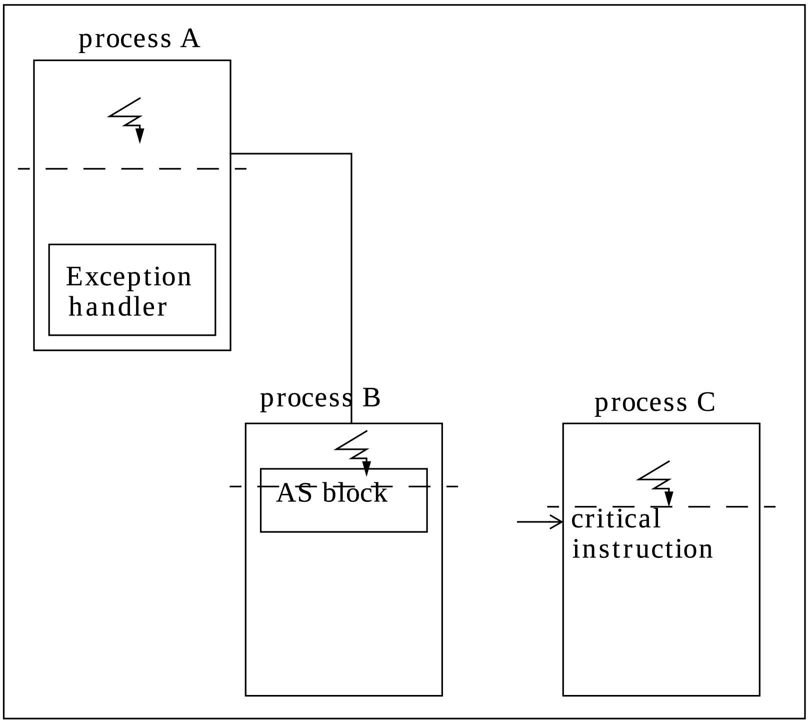


Figure 6.6: PB, PNB and PQB: state before the violation

If the critical instruction in process C is executed when B is executing in the assured region the violation occurs.

Figure 6.7 shows the state of execution after the violation for the subcategory of solutions PQB. After the violation has occurred, the execution in the violating process C is temporarily suspended (1) and the violation is signalled to the violated process B (2) and to process A (3). The execution in the violated process is stopped (4) and process A continues in the exception handler (5). Before the violation recovery, conditions to resume the violating process are checked. The violating process is resumed immediately if its execution does not interfere with the recovery, otherwise the violating process is resumed as soon as possible during exception handling (6). After the violation is recovered, process A aborts, restarts or resumes process B. Possible resumption points are shown in (7). Finally, process A resumes its own execution (8).

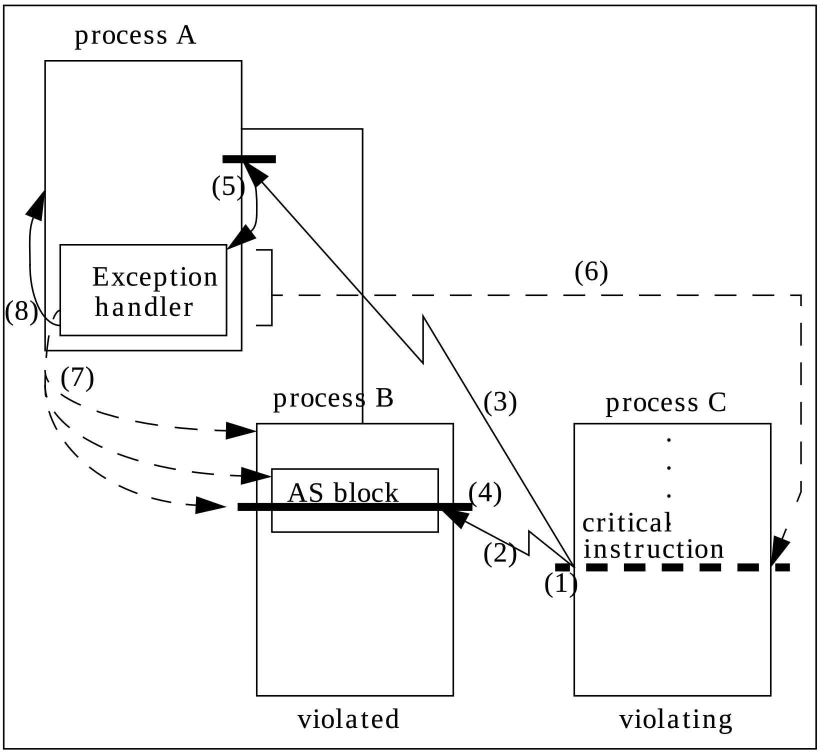


Figure 6.7: PQB: state after the violation

The state after the violation for the subcategories of solutions PB and PNB is similar to the one shown in Figure 6.7. The difference for the subcategory PB is that the violating process gets blocked after the violation and is unblocked by the parent of the violated process only after the exception recovery. For the subcategory PNB instead, the difference is that the violating process continues its execution after the violation. Therefore, the parent of the violated process does not have to handle the unblocking of the violating process.

6.2.4 THE VIOLATED PROCESS AND ITS PARENT ARE THE EXCEPTION HANDLERS (VP)

In this category of solutions the violation can be handled by two different exception handlers: one is in the program of the violated process and the other is in the program of the parent of the violated process. After the violation is signalled, the violated process tries to recover from within its handler. If the violation is recoverable the violated process handles it and then resumes its own execution. Instead, if the violation is not recoverable, it is forwarded to the parent of the violated process. Afterwards, the parent process handles the forwarded violation by aborting or restarting the execution of the violated child with the same or different parameters.

Consider again the example with processes B and C and assume A is the parent of B. The state before the violation for the subcategories of solutions VPB, VPNB and VPQB is shown in Figure 6.8. Process A is executing within its program after having created process B. B is executing in its AS block and process C is before the critical instruction. The dashed line shows the point of execution in A, B and C.

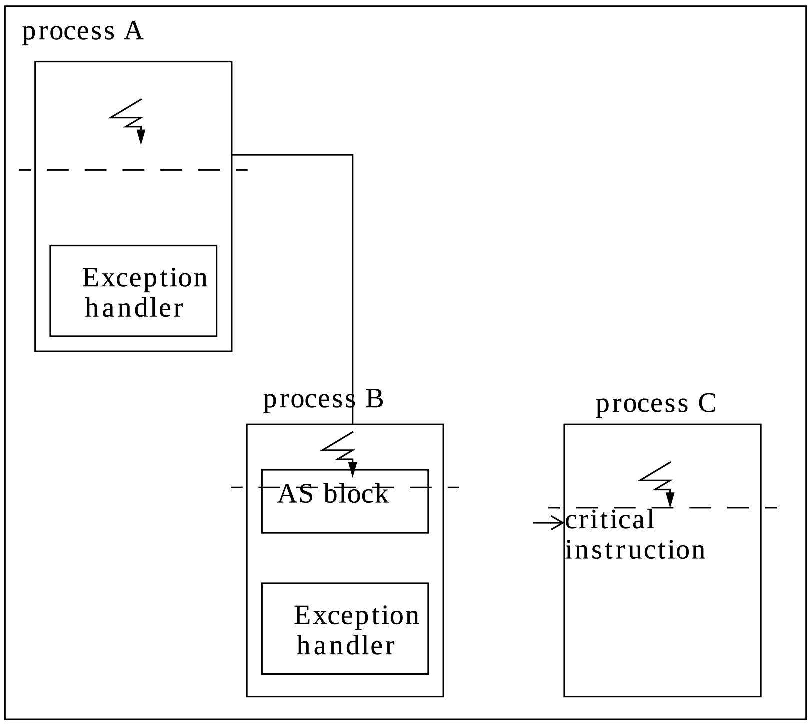


Figure 6.8: VPB, VPNB and VPQB: state before the violation

If the critical instruction in process C is executed and B is executing in the assured region a violation occurs.

Figure 6.9 shows the state after the violation for the subcategory of solutions VPQB. After the violation has occurred, the violating process C temporarily suspends its execution (1) and the exception is signalled to the violated process B (2). Process B continues execution in its exception handler (3). Here, the violation can be recoverable or not. If the violation is recoverable process B checks conditions to resume the violating process. The violating process is unblocked immediately if its execution does not interfere with the recovery, otherwise the violating process is unblocked during exception handling (4). After violation recovery, process B resumes its own execution from one of the resumption points in (5).

Instead, if the violation is not recoverable, the violated process forwards the signal to its parent A (6). A handles the forwarded violation in its exception handler (7) by aborting or restarting, with the same or different parameters, process B. Furthermore, process A unblocks the violating process C (8) and finally resumes its own execution (9).

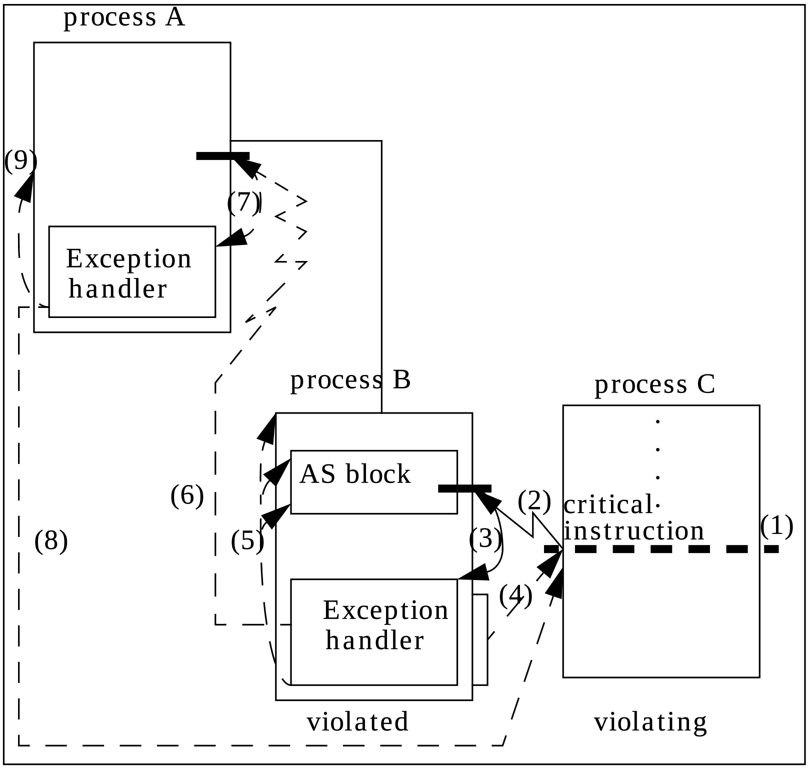


Figure 6.9: VPQB: state after the violation

The state after the violation for the subcategories of solutions VPB and VPNB is similar to the one in Figure 6.9. The difference for the subcategory VPB is that the violating process gets blocked after the violation signal and it is unblocked by the violated process or its parent only after the exception has been handled. Instead, for the subcategory VPNB, the difference is that the violating process continues its execution after the violation. As a consequence, neither the violated process nor its parent handle unblocking the violating process.

6.2.5 ANOTHER PROCESS IS THE EXCEPTION HANDLER (O)

In this category of solutions the exception handler is a process different from the violating, the violated and the parent of the violated. The task of the handler process is to recover from violations of AS statements that possibly occur during concurrent execution.

Consider again the example with processes B and C and assume now that O is the handler process. The state before the violation for the subcategories of solutions OB, ONB and OQB is shown in Figure 6.10. B is executing the AS block and C is before the critical instruction. Process O is suspended, waiting to handle possible violations.

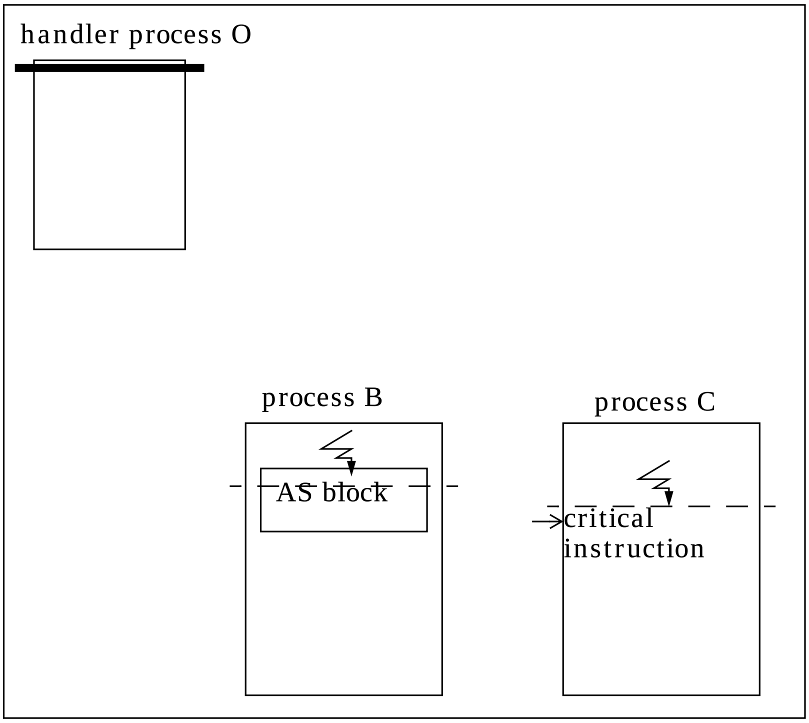


Figure 6.10: OB, ONB, OQB: state before the violation

If C executes the critical instruction while process B is executing in the AS block the violation occurs.

The state of execution for the subcategory OQB after the violation is shown in Figure 6.11. The execution in the violating process C is temporarily suspended (1). The violation is signalled to the violated process B (2) and to the handler process O (3). The execution of the violated process is stopped (4) and process O starts the recovery of the signalled violation (5). Before violation recovery, O checks the conditions necessary to resume the violating process C. C is resumed immediately if its execution does not interfere with the recovery. Otherwise, the violating process is resumed as soon as possible during exception handling (6). After the violation has been recovered, process O aborts, restarts or resumes process B. Possible resumption points are shown in (7). Finally, process O again suspends its execution waiting to recover from other violations.

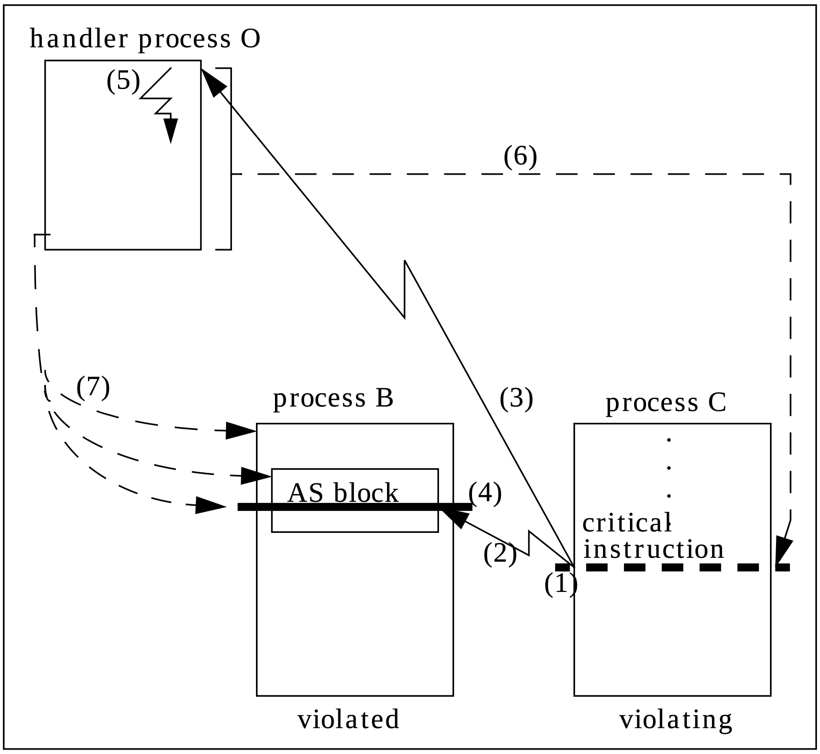


Figure 6.11: OQB: state after the violation

The state after the violation for the subcategories of solutions OB and ONB is similar to the one in Figure 6.11. For the subcategory OB the difference is that the violating process is blocked after the violation signal and it is unblocked by the handler process O after the exception has been recovered. For the subcategory ONB, the difference is instead that the violating process continues its execution after the violation. Consequently, process O does not handle unblocking the violating process.

6.3 Comparison of Solutions

In this section the solutions for AS classified and explained in the previous section are compared. The comparison is made to select the solution that best meets the requirements on the semantic definition of AS (clear and easy semantics, completeness, generality and maximum parallelism) introduced in Section 5.1.2. First the convenience of using the blocking, non-blocking and quasi-blocking semantics of AS is discussed. The purpose is to identify the subcategory of solutions within B, NB and QB, that meets the requirements referenced above best. Next, the solutions within the selected subcategory will be compared.

The blocking, non-blocking and quasi-blocking semantics for AS describes the behaviour of a violating process after it violates an AS statement. The blocking semantics forces the violating process to stop during violation recovery. Instead, the non-blocking semantics allows the violating process to continue execution. The quasi-blocking semantics is a combination of the blocking and the non-blocking semantics. After a violation is signalled, the violating process is suspended and is resumed as soon as it is verified that its execution does not interfere with the violation recovery. In the best case the violating process is resumed immediately and it is not delayed during the recovery (like in the non-blocking semantics). Instead, in the worst case the violating process gets blocked for the time of the recovery (like in the blocking semantics). Therefore, the quasi-blocking semantics includes (simulates) both of the other semantics. As a consequence, the quasi-blocking semantics is more complete than the blocking and non-blocking ones.

Moreover, it is not really clear when to apply the blocking and when the non-blocking semantics. The choice depends on the situation and the moment when a violation occurs. In this sense, the quasi-blocking semantics is also more clear and general than the blocking and non-blocking ones. The decision whether to continue or to stop the execution of the violating process is automatically taken at run-time, after a violation has occurred. As a consequence, the quasi-blocking semantics can be used for all the situations in which the blocking and non-blocking semantics are applied.

From the parallelism point of view, the blocking semantics is the one that delays the concurrent execution most. Both the violated and the violating processes (and eventually the exception handler process if different from them) are delayed during the violation recovery. Instead, with the non-blocking semantics the violating process is never delayed after a violation. However, due to its lack of generality, this semantics cannot be applied in situations in which the execution of the violating process must be stopped. This suggests using the quasi-blocking semantics that delay the violating process only when really needed and only for the time strictly necessary to avoid interference during exception handling.

As a consequence of the reasoning given above, the subcategories of solutions B and NB will be excluded from the comparison. The comparison with respect to the requirements on the semantic definition of AS concerns the following:

- clear and easy semantics
- generality
- completeness
- maximum parallelism

is then restricted to the following solutions: VDQB, PQB, VPQB and OQB.

Clear and Easy Semantics

For the solutions VDQB, PQB, VPQB and OQB the following semantic issues have been clearly defined:

- Which processes are signalled after the violation.
- Which processes compensate for the violation.
- Which processes are blocked after the violation.
- When the blocked processes are resumed.

Nevertheless, for none of these solutions have the semantics been completely specified. However, the following comment can be made. For the solutions PQB and OQB the exception handler process is not the process that gets violated. A semantic issue difficult to define during the violation recovery is how to handle problems of parameter passing and variable visibility between the exception handler process and the violated process. This problem is avoided by the solutions VDQB and VPQB. In the solution VDQB the violation is always recovered by the violated process itself. In the solution VPQB the violation can be handled alternatively by the violated process or its parent process. However, if the parent of the violated process compensates for the violation, it can only abort or restart the violated child. Therefore, problems of parameter passing and variable visibility do not occur. Hence, the semantics for the solutions VDQB and VPQB are easier to define.

Generality

The following decisions have been made to keep the solutions for AS as general as possible:

- To use the quasi-blocking semantics for unblocking the violating process.
- To allow both continuation and abortion semantics for the violated process during exception handling.

- Three alternative resumption points for the violated process have been defined if a violation is recovered with continuation semantics.

However, the issues mentioned above are common for the solutions VDQB, PQB, VPQB and OQB. Consequently, the generality of all these solutions can be considered equal.

Completeness

After an AS statement is violated, the most convenient process to compensate for the violation is usually the one that gets violated, because it has complete visibility of the variables to be recovered. The solutions PQB, OQB do not cover this possibility since the violation is handled by the parent of the violated process or another process (different from the violating and the violated ones). This makes the solutions PQB and OQB non-complete.

Concerning the completeness for the solutions VDQB and VPQB, the solution VDQB only considers the case that the violation is handled by the violated process. Instead, the solution VPQB allows a more flexible handling of a violation, considering the possibility of recovery of the violated process. With this solution the parent of the violated process supervises the violation recovery and decides whether to abort or restart the excepted child process when the exception is not recoverable. Moreover, the solution VPQB allows execution of the violated process to restart with different parameters. This possibility is not considered by the solution VDQB. Hence, the solution VPQB is more complete than VDQB.

Maximum Parallelism

The solution which maximizes the parallelism is the one that delays the concurrent execution the least during the recovery of violations. As the time necessary to compensate for violations is not known, the delays in the concurrent execution will instead be expressed as the number of processes delayed during the violation recovery.

For all the solutions VDQB, PQB, VPQB and OQB, the violating process can be delayed or not during the exception handling according to QB semantics. Furthermore, the following processes are delayed:

- In the solutions PQB and OQB two processes are delayed: the violated process (which is stopped to avoid error propagation), and the process in charge of handling the violation (respectively the parent of the violated process or another process different from the violated or the violating ones).
- In the solution VDQB only the violated process is delayed since it is in charge to handle the violation.
- In the solution VPQB the delayed processes can be one or two. If the violation is recovered by the violated process, only this process is delayed. Instead, if the violation is not recoverable, it is handled by the parent of the violated process. Hence, both the violated process and its parent are delayed during the recovery. However, a parent process can only handle a forwarded violation by aborting or restarting the excepted child. Consequently, the delay which affects the parent process is fairly contained since it is limited to the time strictly necessary for the abortion or the restarting of the violated child.

Therefore, the solutions that delay the concurrent execution the least during recovery of violations, are VDQB and VPQB.

Conclusion

From the reasoning above it follows that all solutions are general. Nevertheless, the semantics for the solutions PQB and OQB are potentially more difficult to define than those for the solutions VDQB and VPQB. Moreover, the solutions PQB and OQB are not complete and during the handling of violations they delay the concurrent execution more than the solutions VDQB and VPQB. Hence, the solutions VDQB and VPQB meet the requirements on the semantics definition of AS best. Furthermore, concerning these two solutions, they fulfill the requirements of clear semantics, generality and maximum parallelism in a similar way. However, VPQB is a more complete solution since it allows more flexible handling of violations. Therefore, the solution VPQB is selected to be defined in more detail.

6.4 Summary

In this chapter alternative solutions for AS have been presented. At first, they have been classified with respect to the blocking, non-blocking and quasi-blocking behaviour of a violating process and to the exception handler processes. Moreover, the classified solutions have been analysed and compared. From the comparison the solution VPQB has been selected and its semantics will be investigated in more detail. In the rest of the thesis, this solution will be referred to as VPQB AS.

Chapter 7 Semantic Issues of VPQB AS

This chapter studies semantic issues of VPQB AS related to the semantic definition of AS as a mechanism for concurrency control.

7.1 Current State of VPQB AS

Two main semantic choices have been made so far for VPQB AS: the processes in charge of handling violations of an AS statement are the violated process and its parent and the quasi-blocking semantics for the violating process. In this section a short summary of VPQB AS is given.

A process that has an AS statement in its program starts executing in the assured region if the Boolean value of the guard predicate is satisfied. Therefore, it continues within the assured region under the assumption that the guard predicate remains verified. However, no restrictions are specified for the concurrent access to any shared variables. As a consequence, other processes in concurrent execution can update the common variables tested in the guard predicate and change the Boolean value of the guard.

When a guard predicate is no longer satisfied, a violation is signalled to the process which is executing the AS statement. The execution of the violating process is temporary suspended. The violated process tries to recover from the signalled violation. Therefore, if the violation is recoverable, the violated process handles it and resumes the violating process according to the quasi-blocking semantics. Moreover, depending on how the exception is handled, the violated process resumes its own execution either from the beginning of the program, from the beginning the AS block or from the break point within the AS block. If the violation is not recoverable, the violated process forwards the violation signal to its parent process. This process will handle the forwarded violation by restarting or aborting the excepted child. Afterwards, the parent process resumes the execution of the violating process and finally its own execution.

7.2 Dealing with Child Processes

The semantics of VPQB AS has been presented for a simple case with three processes in concurrent execution. What happens if the process which executes an AS statement has child processes, created before or within the AS block? Furthermore, what happens if these processes have child processes themselves?

Figure 7.1 shows an example of a hierarchy of child processes. Process B, child of A, has an AS block and child processes created before and within the assured region. In particular, the set of all the processes created within the assured region is referred to with the expression *assured children*.

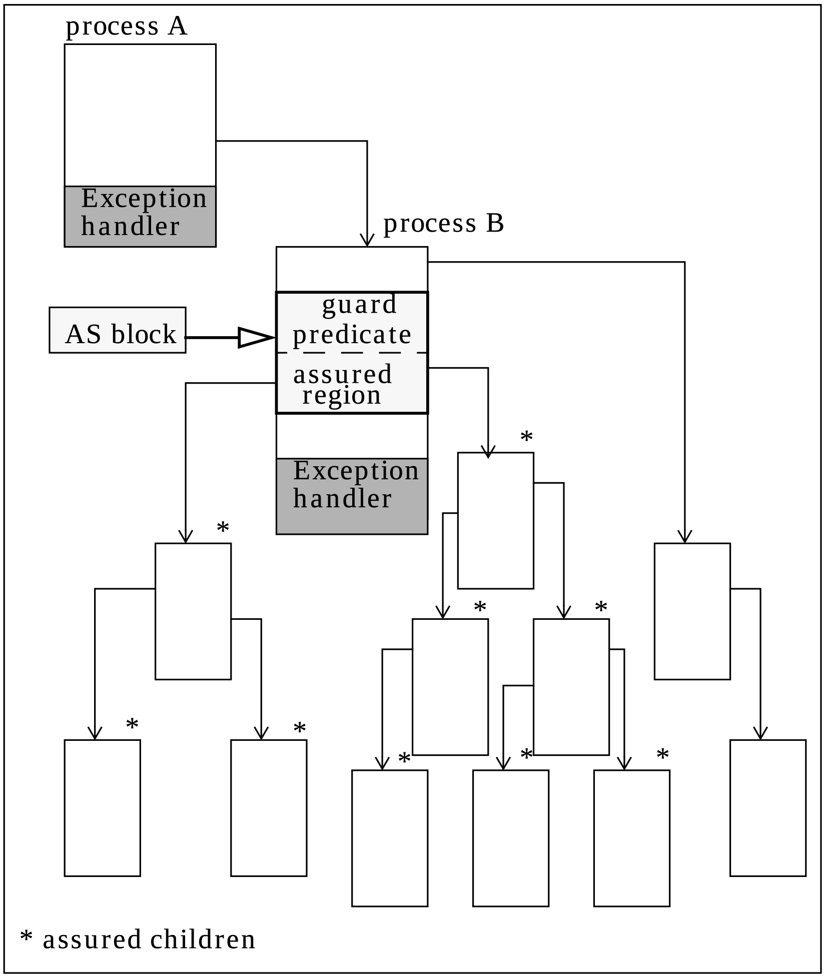


Figure 7.1: Hierarchy of child processes for A and B

The assured children are created within an assured region. Consequently, their execution proceeds under the assumption that the guard predicate associated with the assured region remains satisfied. If the guard predicate gets violated, the assured children could be affected by an error and their execution must be stopped to avoid error propagation. Moreover, the recovery of the assured children could be necessary to restore consistency in the concurrent execution.

In the presented VPQB AS, after a violation is signalled, the violated process and its parent are in charge of compensating for it. This solution can be extended allowing the violated process and its parent to recover from any violation in the assured children also. However, problems of variable visibility and parameter passing could arise during the compensation. Furthermore, the exception could be handled for one process at a time. This is a centralized solution that could introduce a bottleneck in the concurrent execution. The execution of the violated process, its parent, its assured children and the violating process can be delayed and a subsequent reduction of the throughput is possible. An alternative solution, is to allow the assured children to handle violations in their programs as explained in the following section.

7.3 Inheriting VPQB AS

Inheriting VPQB AS allows assured children to recover a signalled violation concurrently, each one in its own exception handler.

When an assured child is created, it inherits from its parent process an exception handler for the recovery of violations of the AS statement in which the assured child has been created. Figure 7.2 shows an example of a process B which has four assured children. Each assured child inherits the exception handler from its parent process.

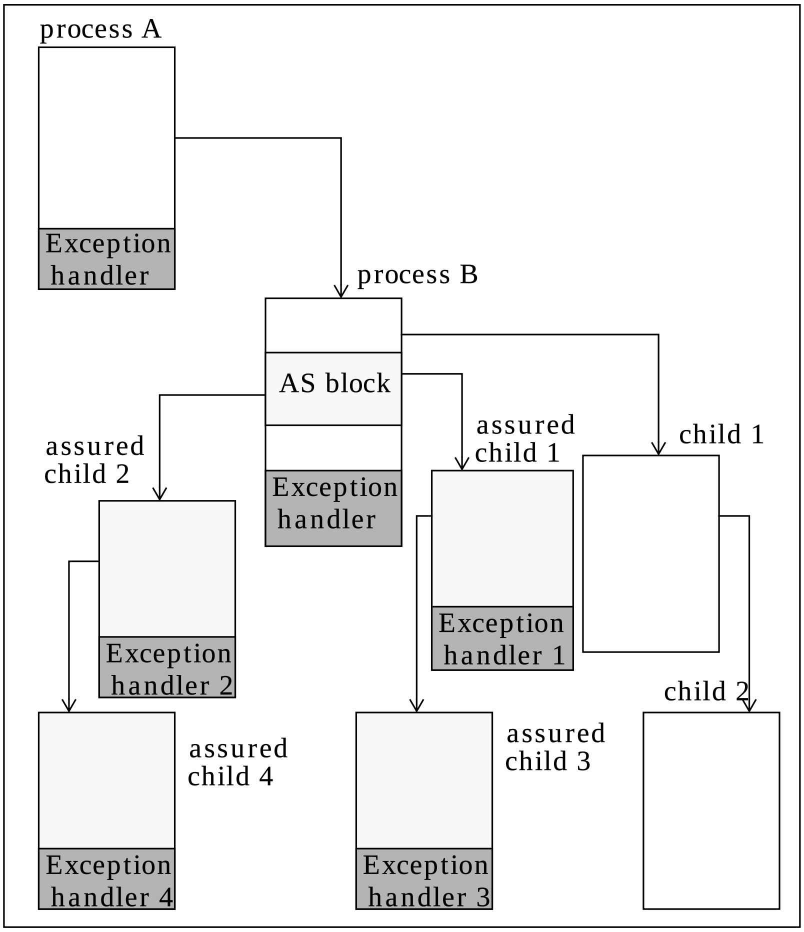


Figure 7.2: Inheriting the exception handler from parent process

When an AS statement gets violated the execution of the violating process is temporarily suspended. Moreover, the violation is signalled to the violated process and to its assured children. To avoid error propagation the signalled assured children pause their execution. Instead, the violated process continues in its exception handler and starts handling the violation in its program as explained in Section 6.2.4. See the example in Figure 7.3.

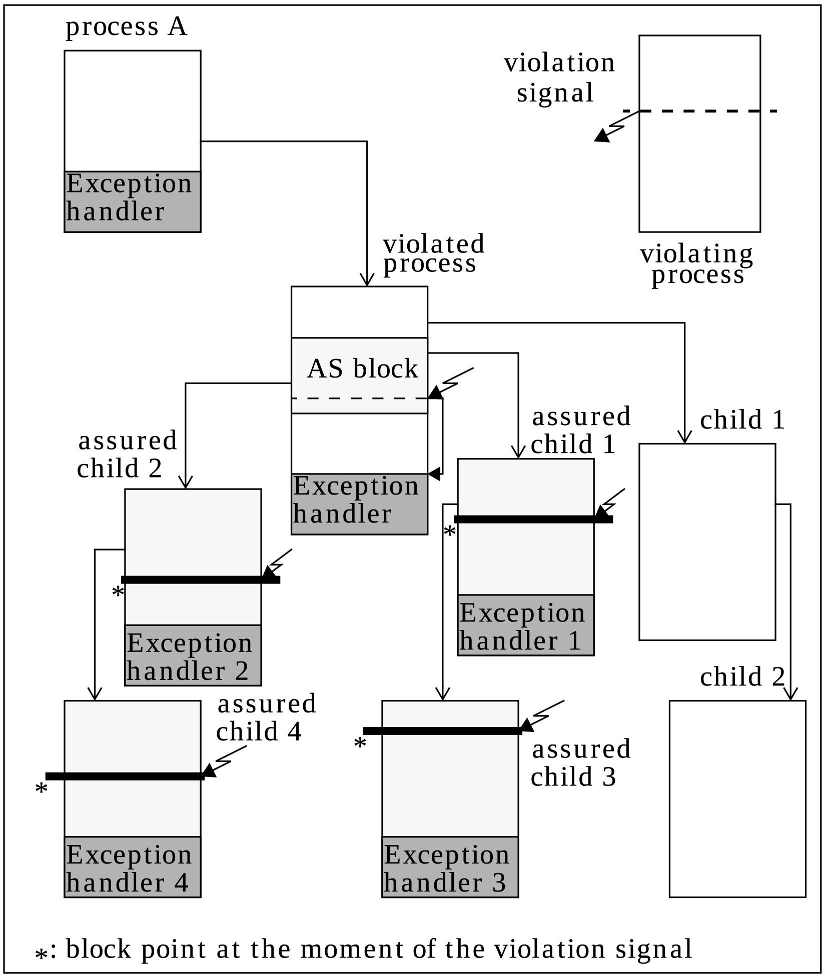


Figure 7.3: Inheriting VPQB AS: state of execution of the violating, the violated and the assured child processes after a violation signal

Furthermore, depending on how the violation is handled, the violated process decides whether the recovery of the assured children is necessary or not.

In particular, according to the semantics of VPQB AS the following four alternatives are possible for the handling of the violation in the violated process:

1. The violation is not recoverable and the violation signal is forwarded to the parent of the violated process.
2. The violation is recovered and the resumption point in the violated process is the beginning of the program.
3. The violation is recovered and the resumption point in the violated process is the beginning of the AS block.
4. The violation is recovered and the resumption point in the violated process is the break point within the AS block.

In the first case, the recovery of the assured children is not needed. The violation will be handled by the parent of the violated process by restarting or aborting the excepted child. Consequently, the execution of the child processes of the violated, both assured and not, serves no purpose. Instead, it might cause problems. If the violated process is aborted, its child processes do not have a parent process left to return the results of their tasks. If instead the violated process is restarted, it will execute the same or different paths of its program, restarting and/or creating new assured children. The execution of the old assured children could interfere with the new execution. Consequently, before the violation signal is forwarded to its parent, the violated process must abort all its child processes.

Furthermore, the recovery of the assured children is not needed in the second and third cases above. In case three, although the violation is recovered, the violated process resumes its execution from the beginning of the AS block. Therefore, it will execute again the AS block restarting or creating new assured children. The execution of the old assured children serves no purpose. Then, the violated process aborts these processes before resuming its own execution.

In case two instead, the violated process will resume its execution from the beginning of the program. Consequently, the execution of all its child processes created before or within the AS block is no longer needed. The violated process aborts all these processes before resuming its own execution.

Lastly, in case four the violated process will resume its execution from the break point within the AS block. The computations done by the child processes of the violated process may still be useful. However, the assured children could be affected by the error and their recovery is necessary. The recovery of the assured children proceeds as explained next.

First, the violated process unblocks the assured children blocked after the violation signal. Next, the unblocked processes continue execution in their exception handlers and start the recovery of the violation asynchronously from each other. See the example in Figure 7.4.

Each assured child, that is not able to recover the violation locally, aborts its subset of child processes. Next, it forwards the violation signal to the violated process notifying it also which processes have been aborted. The unrecovable assured children will be later restarted or aborted by the violated process. This explains why the subset of child processes of each unrecoverable assured child has to be aborted. As an unrecoverable assured child is restarted or aborted, the computations done by its child processes are no longer needed.

Moreover, the recoverable assured children that have not been aborted handle the violation locally and define the resumption point in their programs. Afterwards, they give a notification of the recovery to the violated process.

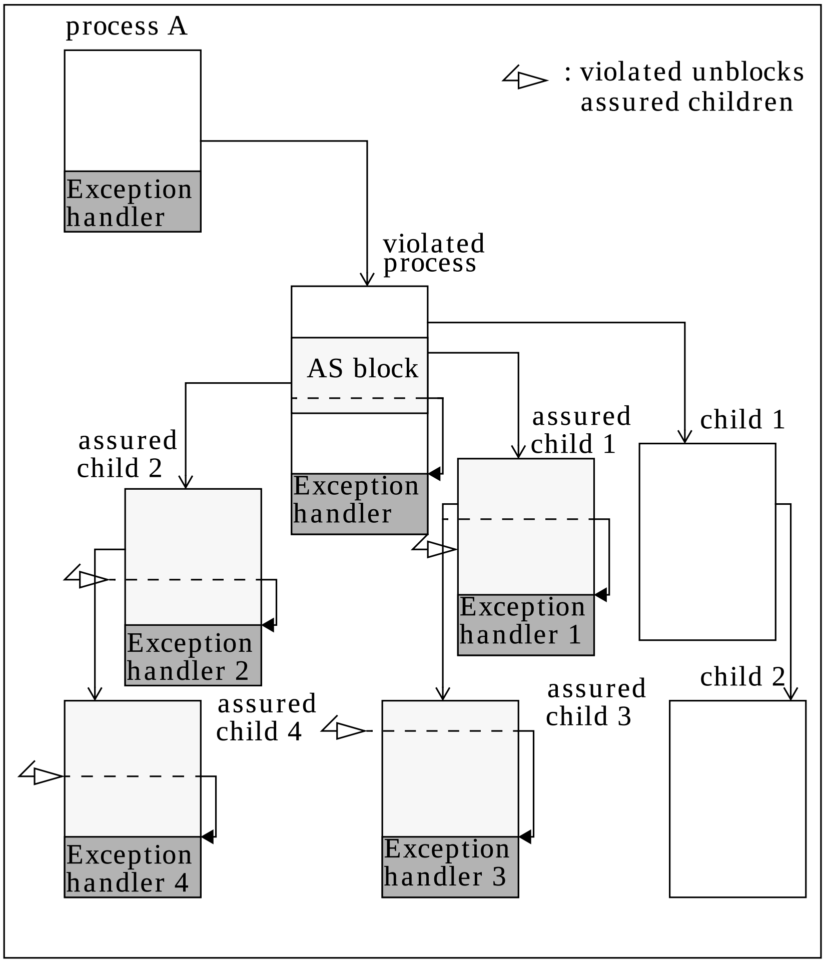


Figure 7.4: Inheriting VPQB AS: recovery of a violation in the assured children

The violated process waits to receive notifications of recovery or abortion and violation signals forwarded from all the assured children. Then, it resumes, restarts or aborts the execution in these processes. In particular, each recovered assured child is resumed from the point decided within the local handler if its resumption does not interfere with the resumption of the violated process. Otherwise, the recovered assured child is restarted or aborted. Moreover, all the assured children that forwarded the violation signal to the violated process are aborted or restarted. The decision to restart or abort an assured child depends on how the exception has been handled in the violated process. Finally, when all the assured children have been resumed, aborted or restarted, the violated process resumes its own execution.

Note that the quasi-blocking semantics for the violating process is still valid for the inheriting VPQB AS. In addition, for a more efficient unblocking of the violating process a further extension can be made. The violating process can be unblocked not only from the violated process or its parent, as in the original semantics, but also from an assured child during the local recovery of the violation. In this way, the violating process is resumed as soon as possible, after it is verified that its execution does not interfere with exception handling.

The inheriting VPQB AS gives autonomy to the assured children to recover from the violation in their programs and to decide locally the most appropriate resumption point. Nevertheless, the violated process always maintains control of the assured children since it synchronizes their execution before and after the recovery. Moreover, it decides to abort, restart or resume the assured children according to decisions made in its exception handler. Consequently, the violated process has a supervisory role during the exception handling and confirms or refutes the decisions taken locally by the assured children. A similar idea can be found in Java (threads and beans) [Fla97].

One restriction for the inheriting VPQB AS

What happens if the process which executes an AS statement finishes the execution of the assured region before all the assured children have terminated?

In this case, violations of the guard predicate should not be signalled to the process which executed the AS statement but only to the assured children created within the assured region and not yet terminated. In addition, the signalled assured children should be able to recover from the violation in their programs independently from the process which executed the AS statement. The inheriting VPQB AS does not cover this case.

To avoid the situation described above the following restriction has been made. The process which executes the AS statement is forced to wait until all the assured children have terminated before exiting its AS block.

7.3.1 CONTINUATION AND ABORTION SEMANTICS DURING EXCEPTION HANDLING FOR THE INHERITING VPQB AS

This section points out the actions executed during the recovery from a violation by the violated process, its parent and its assured children within their exception handlers.

The exception handler in the program of the violated process and the ones in the assured children are specific for an AS statement. The code contained in these handlers is executed to recover from violations of the particular AS statement to which the handlers are associated. The exception handler of the parent of the violated process is instead more general. Different types of exceptions can be handled in it, either detected locally in the program of the parent of the violated process (like overflow or division by zero) or signalled by other processes (like calls to procedures with invalid parameters or a forward of unrecoverable violations of AS statements).

After a violation of the guard predicate of an AS statement is signalled, the violated process continues execution in its exception handler. According to the possibility of recovery the violated process handles the violation as described below.

If the violation is recoverable, the violated process executes the following actions:

- It recovers the violation in its program.
- It tries to unblock the violating process according to the quasi-blocking semantics.
- It defines the resumption point in its program, deciding between the beginning of its program, the beginning of the AS block or the break point within the AS block.
- If the resumption point chosen is at the beginning of the program, the violated process aborts all its children, assured and non-assured. Next, it resumes its own execution.
- If the resumption point is at the beginning of the AS block, the violated process aborts all its assured children. Afterwards, it resumes its own execution.
- If the resumption point is at the break point within the AS block, the violated process unblocks the assured children blocked by the violation signal. Moreover, it waits to receive notification of recovery or abortion and violation forward of all the assured children. Therefore, it resumes, aborts or restarts the assured children as explained in the previous section. Finally, it resumes its own execution.

If the violation is not recoverable, the violated process executes the following actions:

- It aborts all its child processes.
- It forwards the exception to its parent.
- It waits to be aborted or restarted by the parent process.

The exception handlers in the assured children are simplified versions of the handler in the violated process. Within its exception handler an assured child handles a signalled violation as follows.

If the violation is recoverable, the assured child executes the following actions:

- It recovers the violation locally to its program.
- It tries to unblock the violating process, if still blocked, during the local recovery of the violation.

- It defines the resumption point in its program.
- It notifies its recovery to the violated process.
- It waits to be resumed, aborted or restarted by the violated process.

If the violation is not recoverable locally, the assured child executes the following actions:

- It aborts its subset of child processes.
- It forwards the violation signal to the violated process, giving also notification of the aborted processes.
- It waits for the violated process to abort or restart its execution.

In the case where the violation is not recoverable in the program of the violated process, the violation signal is forwarded to the parent of the violated process. The parent process handles the forwarded violation within its exception handler by executing the following actions:

- It aborts or restarts the violated process.
- It unblocks the violating process.
- It resumes its own execution.

During the recovery of a violation within the exception handlers, temporary locking of the corrupted assured data could be necessary to avoid error propagation. Consequently, all the concurrent processes that try to access the inconsistent data could be delayed for the duration of the lock.

To keep the solution of inheriting VPQB AS as general as possible, both continuation and abortion semantics have been allowed in the exception handlers of the violated process, of its parent and of its assured children. The decision to resume or abort the violated process and its assured children after a violation is taken at run-time, depending on the situation and the moment of the violation.

7.3.2 THE VIOLATING PROCESS IS THE VIOLATED PROCESS ITSELF OR ONE OF ITS ASSURED CHILDREN

The semantics of the inheriting VPQB AS has not been defined for the following two cases:

1. The violating process is the violated process itself.
2. The violating process is one of the assured children of the violated process.

The first case happens when the process which contains the AS statement in its program, during execution within the assured region, executes an instruction which violated the guard predicate of the AS statement. The second case occurs when one of the assured children, created within the assured region of the AS statement, executes an instruction which violates the guard predicate of the AS statement. The inheriting VPQB AS is adapted to these cases as explained next.

1. **The violating process is the violated process itself**. After the execution of the violating instruction the violating process suspends its execution. Next, the violation, which is automatically detected, is signalled to the violated process itself and to its assured children. The assured children block their execution. The violated process instead, which suspended its execution after the violation, is unblocked by the violation signal and continues execution in its exception handler. The exception is then recovered from by the violated process, by its assured children and by the parent of the violated process as explained in the previous sections. However, the following simplification can be made. As the violating process is unblocked by the violation signal, the handler processes do not need to handle its unblocking during exception handling.
2. **The violating process is an assured child of the violated process.** After the execution of the violating instruction, the violating assured child process blocks its execution. The

violation is signalled to all the other assured children and to the violated process. The signalled assured children block their execution. Afterwards, the violated process starts the recovery of the violation continuing execution in its exception handler. The violation is then recovered in the violated process and in its assured children (including the violating one) as explained in the previous section. As the violating process is also an assured child of the violated process, its execution is aborted, resumed or restarted by the violated process at the end of the recovery. Therefore, the assured children of the violated process do not need to handle the unblocking of the violating process during exception handling.

7.3.3 VIOLATIONS TO VIOLATING PROCESSES

According to the quasi-blocking semantics, when a process violates a guard predicate of an AS statement its execution is blocked until the violated process, or one of its assured children, unblock it. However, the violating process could be blocked while executing an AS statement itself. What happens if the violating process gets violated itself while it is blocked?

A simple solution to this problem is to delay the violation signal until the violating process is unblocked. However, this solution would not be consistent when the violated process has assured children. When delaying the violation signal, these processes would continue their normal execution and error propagation could result.

A revised and consistent solution is that the violation is signalled to the blocked violating process and its assured children as soon as it has occurred. At the violation signal, the signalled assured children block their execution avoiding in this way error propagation. Nevertheless, the violation is handled as explained in Section 7.2.1 only when the violating-violated process gets unblocked.

7.3.4 VIOLATIONS ARISING DURING THE EXECUTION OF PROCEDURES CALLED FROM AN ASSURED REGION

This section explains how violations of the guard predicate arising during the execution of a procedure called from an AS statement can be handled.

A procedure does not have its own thread of execution. Instead, the procedure is executed in the thread of the caller process. Consider the example of Figure 7.8 in which process A calls the procedure P() from its assured region AR. At first, process A calls the procedure P() (1), then it executes the called procedure (2) and finally it continues the execution in the assured region after the procedure call point (3).

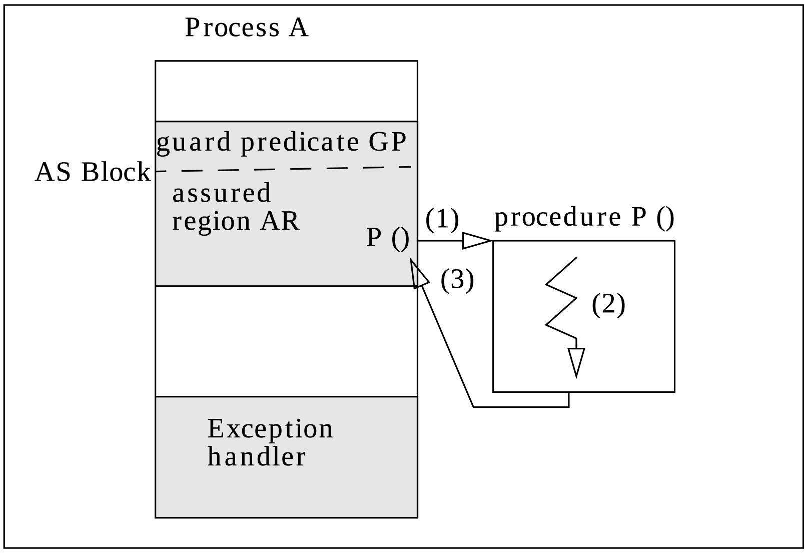


Figure 7.5: Thread of execution of A during the call and the execution of the procedure P()

Consequently, a procedure called from the assured region of an AS statement, is executed from the same process which executes the AS statement. This fact simplifies the handling of violations signalled during the execution of procedures called from an assured region.

When a violation is signalled, the violated process suspends its execution within the procedure. Next, it continues in its exception handler and the violation is handled as explained in Section 7.2.1. After exception handling the execution of the procedure can be alternatively aborted or resumed from the break point at the moment of the violation signal.

The first case occurs when the execution of the procedure is no longer necessary after the exception handling. This happens if the violation is not recoverable or if the resumption point for violation recovery is at the beginning of the AS block or at the beginning of the program of the violated process. The execution of the procedure is instead resumed if the resumption point for the violation recovery is at the exact break point within the AS block.

7.4 Nested AS Statements

As introduced in chapter 4 a single AS statement consists of an AS block and an associated exception handler. A nested structure of AS statements can be instead represented as a number of nested AS blocks and an exception handler. The exception handler has as many exception handling sections as the number of the AS blocks. Each section is associated with an AS block.

An example, with two nested AS statements in the program of a process A, is shown in Figure 7.5. The AS block AB2 is contained in the assured region of the AS block AB1. The exception handler consists of two sections of code, one for each AS block. Each section contains the handling code for the recovery of violations signalled in the corresponding AS block.

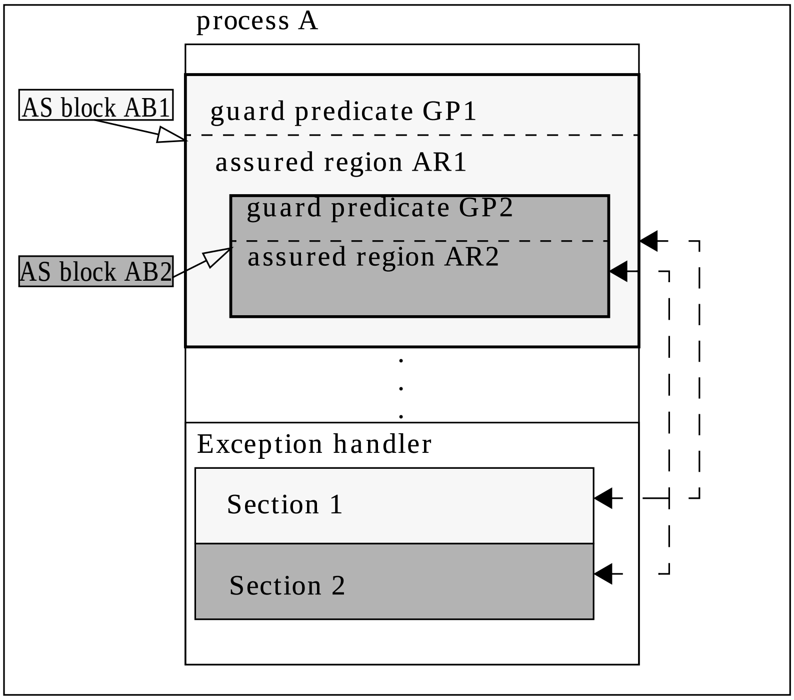


Figure 7.6: Two nested AS statements

As for a single AS statement the assured region of one of the nested AS statements is executed if the corresponding guard predicate is satisfied. Moreover, if during execution within the assured region the Boolean value of the guard predicate changes, a violation is signalled. In the previous example, violations of the guard predicate GP1 can occur during execution within the assured region AR1 (then also during execution of the AS block AB2 which is contained in the assured region AR1). Analogously, the guard predicate GP2 can be violated in the assured region AR2.

When a violation of a guard predicate is signalled, execution continues in the section of the exception handler associated with the violated AS block. Referring to Figure 7.5, if during the execution in the assured region AR1 a violation of the guard predicate GP1 is signalled the execution continues in the section one of the exception handler.

It is assumed that during the handling of a violation nested violations of the same or different guard predicates of an AS statement cannot occur. Consequently, only one violation at a time is allowed to be signalled and handled during the execution of an AS statement.

7.4.1 INHERITING VPQB AS WITH NESTED AS STATEMENTS

The semantics of the inheriting VPQB AS needs to be examined for the case where assured children are created within a nested structure of AS statements.

In Figure 7.7 an example of a process A which has three nested AS statements and four assured children ac1, ac2, ac3, ac4 is presented. The assured blocks of the three AS statements are labelled AB1, AB2 and AB3. Moreover, the sections of handling code associated with the assured blocks AB1, AB2 and AB3 are labelled respectively Ex1, Ex2, Ex3.

Violations occurring in one of the nested AS statements should not be notified to all the assured children. Instead, only the assured children created within the assured region associated with the violated guard predicate must be signalled. For example, refer to Figure 7.7 and assume that a violation of the assured predicate associated with the assured block AB2 occurs when process A is executing in the assured block AB3, after having created the assured child ac4. The violation must be signalled to process A and only to the assured children ac3 and ac4.

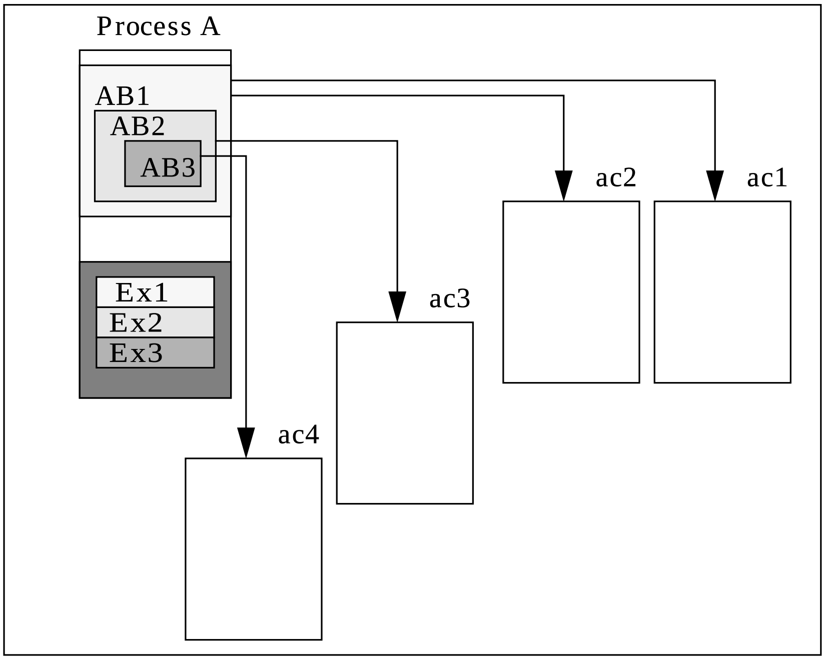


Figure 7.7: Assured children created within a nested structure of AS statements

The inheriting VPQB AS can be extended as follows to allow nested AS statements. When an assured child is created, it inherits from the parent process an exception handler. This exception handler consists of one section of handling code for each of the AS blocks in which the assured child is nested. Referring to the example in Figure 7.7, when the assured children ac1 and ac2 are created, they inherit from process A the exception handling sections Ex1.1 and Ex1.2. Moreover, when the ac3 is created, it inherits from the parent process the exception handling sections Ex1.3 and Ex2.1. Finally, when the assured child ac4 is created, it inherits from process A the exception handling sections Ex1.4, Ex2.2 and Ex3.1. See Figure 7.8.

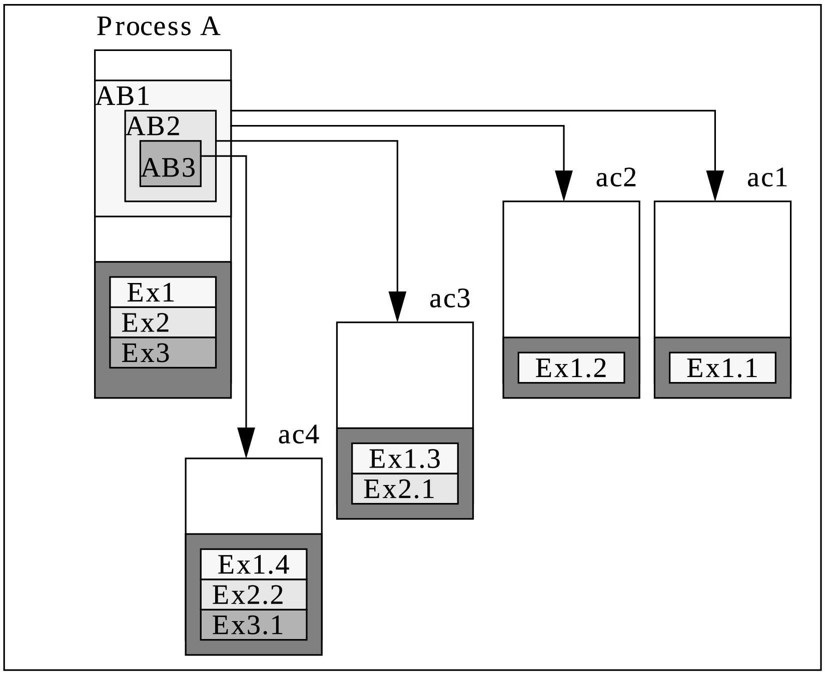


Figure 7.8: Inherence of exception handling sections

When a violation of a guard predicate occurs the violated process and the assured children created within the assured region associated with the violated guard predicate are signalled. Therefore, the violation is handled. The signalled assured children block their execution. All the other assured children are instead not affected by the violation and continue their execution. The violated process continues executing in the section of the exception handler associated with the violated AS block. For example, referring to Figure 7.8, it is assumed that the guard predicate of the assured block AB2 is violated during the execution of the assured block AB3, after the assured child ac4 has been created. The solution is signalled to the assured children ac3 and ac4 and to process A. The signalled assured children block their execution. Process A continues with section Ex2 of the exception handler. The assured children ac1 and ac2 continue their execution. The signalled violation is then handled in the violated process and in the signalled assured children. The following four alternatives are possible for handling the violation:

1. The violation is not recoverable and it is forwarded to the parent of the violated process.
2. The violation is recovered and the resumption point is at the beginning of the program of the violated process.
3. The violation is recovered and the resumption point in the program of the violated process is at the beginning of the violated AS statement.
4. The violation is recovered and the resumption point in the program of the violated process is at the break point within the violated AS statement.

In cases one and two the violation is handled as explained in Section 7.2.1. In case three the only difference with the solution presented in Section 7.2.1 is that the assured children which are aborted before the violated process resumes its execution, are the only ones that get the violation signal. In case four, the violated process unblocks all the signalled assured children blocked by the violation signal. The unblocked assured children continue execution in the section of their exception handlers associated with the violated AS statement. Subsequently, the violation is recovered locally in the programs of these processes or forwarded to the violated process as described in Section 7.2.1.

7.5 Summary

In this chapter further semantic issues for the solution VPQB AS have been defined. In particular, VPQB AS has been extended to allow local handling of violations in processes created within the assured region of an AS statement resulting in inheriting VPQB AS. Furthermore, the semantics of the inheriting VPQB AS have been studied in the case where violations of an AS statement arise from the violated process itself or from processes created within the assured region. Next, a solution has been presented for handling violations occurring during the execution of procedures called from AS statements. Finally, the semantics of nested AS statements have been defined.

Chapter 8 Implementation and Experimental Results

This chapter describes an implementation of AS and presents experimental results obtained using the implementation. The first two sections provide the notions needed to understand the work done during the implementation. In particular, a short description of CAMOS, the environment chosen for the implementation, is given. Next, *operations* in CAMOS are described. Subsequently, it is explained how the solution VPQB AS has been modelled and implemented to define *assured operations* in CAMOS. Finally, the performance of the implemented solution is evaluated using a case study.

8.1 CAMOS: the Implementation Environment

CAMOS (Control Application Mediator Object System) is a prototype for a Manufacturing Control System (MCS) developed as part of a licentiate thesis [Fal96] in the Real Time Systems Laboratory (RTSLAB) at Linköping University. An MCS is used for the automatic control of machinery in a production process. The purpose of an MCS is to provide easier maintenance and more flexibility during the production process and to allow fast changes of the control software.

CAMOS is a database centred MCS and solves the data management problems in the production process by using an active object-relational database management system. A short description of the CAMOS architecture is given in the next sections.

8.1.1 THE CAMOS ARCHITECTURE

Four elements characterize the CAMOS architecture: the active database system AMOS (Active Mediator Object System), the high level language CAMOS(L), the operation manager and the real-time kernel. Figure 8.1 shows an overview of the CAMOS architecture. A short description of the four components mentioned above is then provided. Further details can be found in [Fal96] and [Die97].

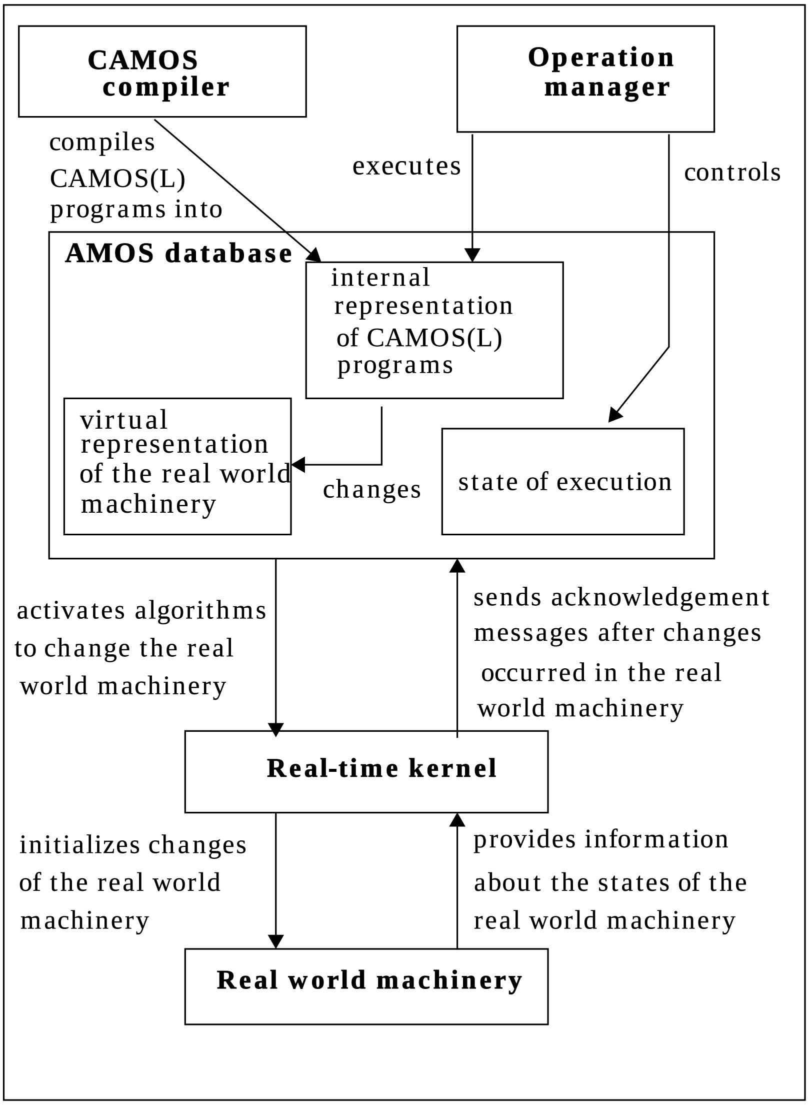

CAMOS compiler
Operation manager
compiles CAMOS(L) programs into
executes
controls
AMOS database
internal representation of CAMOS(L) programs
virtual representation of the real world machinery
changes
state of execution
activates algorithms to change the real world machinery
sends acknowledgement messages after changes occurred in the real world machinery
Real-time kernel
initializes changes of the real world machinery
provides information about the states of the real world machinery
Real world machinery

Figure 8.1: Overview of the CAMOS architecture

The Active Database Management System AMOS

The CAMOS architecture uses the database AMOS to store the information for the production process. AMOS is an example of an Active Database Management System (ADMS) developed in the Engineering Databases and Systems Laboratory (EDSLAB) at Linköping University. AMOS supports mechanisms to store data using object-oriented techniques. Moreover, it provides active rules, a new sort of database notion introduced by ADBMS, that makes the database active and not only a passive data repository. Active rules monitor the state of data in the database and automatically execute one or more user-programmed database operations when a user-programmed condition is satisfied. Further details about ADMS and active rules can be found in [Wid96]. For more information about AMOS refer to [Kar94].

The CAMOS(L) Language

CAMOS(L) is a high-level language used to program the machinery behaviour for MCS in the CAMOS system. It provides mechanisms to coordinate the machinery's activities and supports data management. CAMOS(L) has been developped to simplify the programming of production processes. Therefore, it consists of only a few syntactical constructs.

When a CAMOS(L) program is used to control a manufacturing production process, at first an internal representation of the real-world machinery and its behaviour is provided within the database AMOS. Therefore, the machinery can be controlled via software using CAMOS(L) operations. Operations are the most important syntactical CAMOS(L) constructs and are described in the Section 8.3. The compiler CAMOS is used to compile CAMOS(L) programs into database queries, active rules and run-time code. The object code resulting from the compilation is stored in the AMOS database and is executed by the operation manager.

The Operation Manager

The operation manager organizes the execution and executes the operations stored in the database. The operation manager consists of three parts: the interpreter, the scheduler and the rule coordinator. The functionality of these parts can be summarized as follows:

- **The interpreter** reads the run-time code resulting from the compilation of CAMOS(L) programs and executes operations according to their execution order. It also performs changes in the virtual representation of the real word machinery internal to the database, during the execution of the operations. Moreover, it works on the acknowledgement messages from the real-time kernel.
- **The scheduler** selects an operation that is allowed to execute. If there is more than one operation to execute at the same time, the scheduler uses the round-robin policy.
- **The rule coordinator** activates and deactivates operations by the activation and deactivation of active rules.

The Real-time Kernel

The real-time kernel is the connection between the virtual representation of the real world machinery in AMOS and the machinery in the real word. It supervises movements of the real word machinery by initializing the changes after the virtual representation of the machinery changed. Moreover, when changes in the real world are done, the real-time kernel, sends acknowledgement messages back to the operation manager.

8.2 Structure and High-Level Representation of CAMOS(L) Operations

Operations are the most important element of CAMOS(L) and are used to program and control machinery in a production process. in this section at first the general structure of CAMOS(L) operations is described, since this is necessary to understand the work done during the implementation. Moreover, an example is used to introduce the high-level representation of operations in CAMOS(L).

8.2.1 THE STRUCTURE OF CAMOS(L) OPERATIONS

A CAMOS(L) operation consists of a Boolean condition and a body which contains a number of suboperations. The Boolean condition must be satisfied before the operation is allowed to execute. This condition is called the operation's *wait condition* and its purpose is to synchronize the operation with other operations. Furthermore, an operation consists of several suboperations, grouped into guarded sets. The guard is a database query returning a Boolean value that decides if the guarded set of suboperations will be executed or not.

Suboperations are threads of execution of the process executing the main operation. All suboperations within one operation may execute in parallel. However, dependencies can be used to serialize suboperations that must not be executed in parallel.

When all the suboperations with fulfilled guards have finished execution, the operation ends as well, unless it is an iterative operation. An iterative operation starts again by checking the wait condition and then executing again all the suboperations with fulfilled conditions.

8.2.2 HIGH-LEVEL REPRESENTATION OF CAMOS(L) OPERATIONS

The following example shows what an operation in CAMOS(L) looks like. Moreover, the different parts of the operation are described.

```
CREATE OPERATION MoveCrane (crane .c)
AS WHEN
       numberOfLifts (.c) < 50
DO
IF position (.c) = 0 THEN
  1: PUT .c: h_position = 1;
  2: magnetOff (((cranemagnet (.c))));
  DEPENDENCIES: (1,2);
ENDIF
IF position (.c) = 1 THEN
  2: PUT .c: h_position = 0;
  3: magnetOn(((cranemagnet (.c))));
  DEPENDENCIES: (2,3);
ENDIF
IF numberOfLifts (.c) = 40 THEN
  3: SET .c: maintenance = "TRUE";
ENDIF
```

MoveCrane is the name of the operation. A list of formal parameters, enclosed in parenthesis, follows the name of the operation. A formal parameter consists of the name of a CAMOS(L) class, previously defined in the database AMOS, and an identifier to which the object is referred to within the operation. The identifier begins with a "." to recognize it in database queries.

The *wait condition* is enclosed between the keywords AS WHEN and DO. It is a database query returning a Boolean result.

Finally, the body of the operation *MoveCrane* contains a non-empty set of if-statements. Each if-statement has an if clause, called the *if-guard*, enclosed between IF and THEN. According to the value of the if-guard the body of the if-statement is executed or not.

The body of the if statement contains one or more suboperations, preceded by a label, which is a positive integer number followed by a colon. A DEPENDENCIES: statement can be used to order the execution of the suboperations.

The following two types of operations are possible:

- Primitive operations.
- Call operations.

Primitive operations are requests for change in the real world machinery. They can be recognized since they always begin by PUT or SET.

Call operations are instead calls to other operations. In the operation *MoveCrane* examples of call operations are **`magnetOff (((cranemagnet (.c))))`** and **`magnetOn(((cranemagnet (.c))))`**.

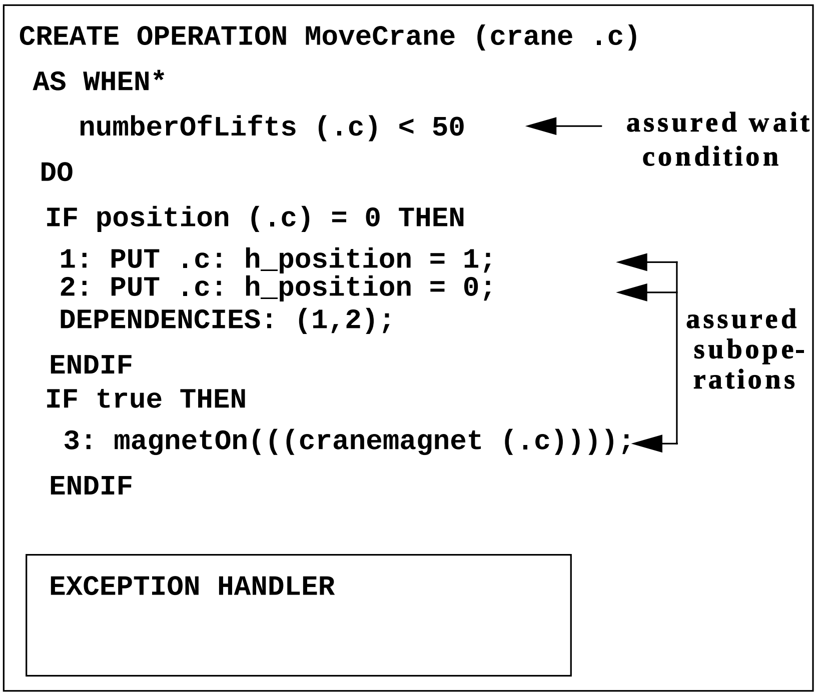
8.2.3 EXECUTION OF CAMOS(L) OPERATIONS

Before the execution, CAMOS(L) operations are first compiled from the high-level to the internal representation which is the stored into the AMOS database. Therefore, operations must be activated. The operation manager handles the activation of an operation, which means the activation of an active rule associated with the operation. This active rule is generated by the CAMOS(L) compiler during the compilation and is used to check automatically when the wait condition of the operation is satisfied. Only when this happens, is the active rule deactivated and the execution of the body of the operation starts.

8.3 Applying AS to Assure CAMOS(L) Operations

Operations have been presented as the basic constructs of CAMOS(L). AS has been integrated in CAMOS to assure CAMOS(L) operations.

When an operation is assured, the operation is called an *assured operation* and its wait condition is called an *assured wait condition*. Moreover, the suboperations called or executed within the body of the assured operation are called *assured sub-operations*. An asterisk is used to distinguish an assured operation from an ordinary operation. Furthermore, an exception handler is associated with the assured operation. The example of a CAMOS(L) assured operation is shown in Figure 8.2.



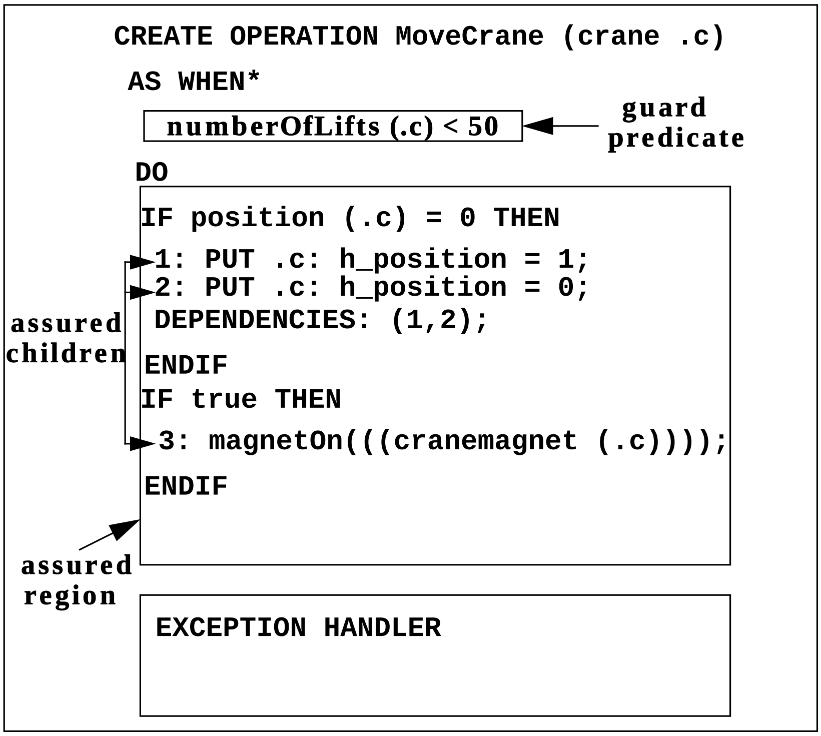
```
CREATE OPERATION MoveCrane (crane .c)
 AS WHEN*
    numberOfLifts (.c) < 50
  DO
  IF position (.c) = 0 THEN
   1: PUT .c: h_position = 1;
   2: PUT .c: h_position = 0;
   DEPENDENCIES: (1,2);
  ENDIF
  IF true THEN
   3: magnetOn(((cranemagnet (.c))));
  ENDIF

  EXCEPTION HANDLER
```

Figure 8.2: An example of an assured operation in CAMOS(L)

The execution of an assured operation starts when its assured wait condition is satisfied. Next, the body of the operation is executed under the assumption that the Boolean value of the assured wait condition remains satisfied. Changes to the assured wait condition are automatically monitored and if its Boolean value changes a violation is signalled. Consequently, the code in the exception handler is executed to compensate for the violation.

The guard predicate, the assured region and the assured children described for an AS statement are recognized within an assured operation as follows. The assured wait condition corresponds to the guard predicate. The body of the operation is recognized as the assured region. Moreover, the suboperations executed or called within the body of the assured operation correspond to the assured children of the process which executes the assured operation. See the example in the Figure 8.3.

When a violation of an assured wait condition occurs, the process which is executing the violated assured operation will be referred to as violated process. The process which violates the assured wait condition will be referred to as violating process.



```
CREATE OPERATION MoveCrane (crane .c)
 AS WHEN*
  numberOfLifts (.c) < 50
 DO
  IF position (.c) = 0 THEN
   1: PUT .c: h_position = 1;
   2: PUT .c: h_position = 0;
   DEPENDENCIES: (1,2);
  ENDIF
  IF true THEN
   3: magnetOn(((cranemagnet (.c))));
  ENDIF

  EXCEPTION HANDLER
```

Figure 8.3: Guard predicate, assured region and assured children of an assured operation

8.4 Integrating the VPQB AS to Assure Operations in CAMOS

This section explains how VPQB AS has been integrated within CAMOS. In addition, some issues of the implementation and the requirements fulfilled during the implementation are described. Finally, possible extensions to the integrated solution are suggested.

8.4.1 THE MODELLED SOLUTION

As the implementation was performed to valid or refute the approach of AS, a simplified version of VPQB AS has been modelled and implemented. This section explains how VPQB AS has been modelled to define assured operations in CAMOS.

When an assured operation is created, an exception handler is associated with the assured operation in the program of the process which executes the operation. In this exception handler, violations of the assured wait condition occurring during the execution of the body of the operation are handled. The assured suboperations executed or called from the body of the assured operation, are instead not provided with their own exception handlers. The reason for this choice derives from the assumption made that the process which executes the assured operation is always the most convenient process to handle violations in its assured suboperations.

When a violation of the assured wait condition occurs during execution of an assured operation, the violating process blocks its execution. Moreover, the violated process and its assured suboperations are signalled. The execution of the signalled assured suboperations is suspended to avoid error propagation. Instead, the violated process executes the code in the exception handler associated with the violated assured operation and compensates for the violation. In particular, the following three alternatives have been considered:

1. The violation is recoverable in the violated process and in the suspended assured suboperations.
2. The violation is recoverable in the violated process but not in the suspended assured suboperations.
3. The violation cannot be recovered in the violated process.

In the first case, the violation is recovered in the violated process and in the suspended assured suboperations. Moreover, the violating process is unblocked as soon as possible during the violation recovery (quasi-blocking semantics). Finally, the suspended assured suboperations and the violated process are resumed.

In the second case, the suspended assured suboperations are aborted. Next, the violation is recovered in the program of the violated process and the violating process is unblocked according to the quasi-blocking semantics. Finally, the violated process is resumed and the aborted assured suboperations are restarted.

In the third case, both the suspended assured suboperations and the violated process are aborted. Next, the violation signal is forwarded to the process that called the assured operation. The caller process will handle the unblocking of the violating process. Finally, it will decide whether to restart or not the assured operation.

Nested Assured Operations

The solution for AS modelled in CAMOS allows nested assured operations.

An assured operation is nested when it is called from the body of another operation that is assured itself. Each nested assured operation is provided with its own exception handler. Figure 8.4 shows an example in which the assured operation Y is called from the body of assured operation X.

Violations of a nested assured operation can occur during the execution of its body. These violations are handled in a way similar to those described in the previous section.

After a violation has occurred, it is signalled to the process which is executing the nested assured operation (violated process) and to its subset of assured suboperations. The signalled assured suboperations are suspended. Instead, the violated process executes the code in the exception handler associated with the violated assured operation. Therefore, it compensates for the violation as explained in the previous section.

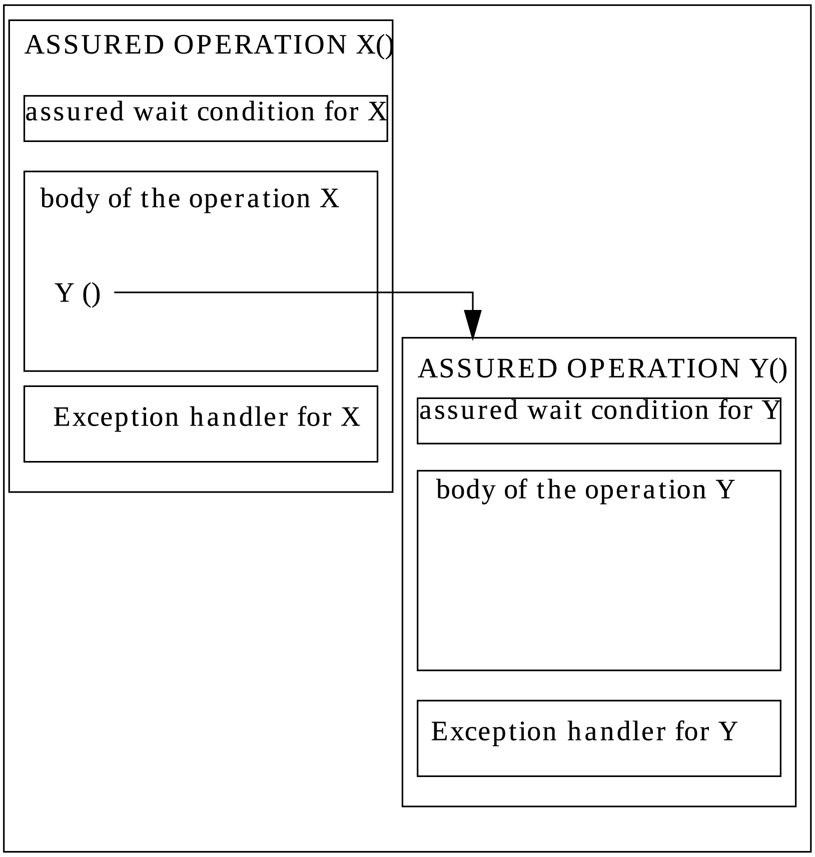


Figure 8.4: Structure of two nested assured operations

Restrictions

The following restrictions have been made when integrating VPQB AS with CAMOS. First, nested violations of an assured wait condition arising during the handling of a previous violation have not been allowed. Therefore, only one violation at a time can be signalled and handled by a violated process. Moreover, nested violations of different assured wait conditions when nested assured operations are used have not been allowed either.

8.4.2 THE IMPLEMENTATION

Since they are not relevant to the purpose of this thesis, the modifications made during the implementation and integration with the existing code for CAMOS will not be explained in detail. Instead, in the next sections, it is first described at an abstract level how active rules have been used for the monitoring of assured operations. Then, it is explained how exception handling has been realized in CAMOS. Finally, the requirements fulfilled during the implementation are discussed.

Automatic monitoring of Assured Wait Conditions using Active Rules

Within the CAMOS system, active rules represent a natural way to automatically detect the state of the data stored in the AMOS database. Active rules have been used to automatically monitor changes of assured wait conditions.

When an assured operation is created, an active rule is generated and associated with the operation. This rule is designed to check, automatically, when the assured wait condition of the operation is satisfied. Before execution, the assured operation is then activated. The operation manager handles the activation of the operation, meaning the activation of the associated rule. Afterwards, the activated rule signals when the assured wait condition is satisfied. When this happens the active rule is deactivated and the execution of the assured operation begins. However, before the execution of the body of the assured operation, a new active rule is activated. The task of this new rule is to automatically detect changes occurring to the assured wait condition during the execution of the operation and to signal possible violations.

If and when the Boolean value of the assured wait condition becomes false, the new activated rule signals a violation to the process which is executing the assured operation and to its assured suboperations. Therefore, the violation is handled by the signalled processes as explained before. The active rule used for the monitoring of violations is finally deactivated when the execution of the assured operation terminates. In particular, this happens in the following situations:

- The assured operation is aborted during the recovery of a signalled violation.
- The execution of the body of the assured operation is successfully terminated.

The two active rules described above are generated during the compilation of the assured operation from high-level to internal representation. The activation and deactivation of active rules are handled by the operation manager.

Defining Exception Handling in CAMOS

The earlier CAMOS system did not support exception handling. Therefore, a solution has been implemented to allow the handling of violations of assured operations.

When an assured operation is created, an exception handler is also specified. The exception handler consists of a section of code executed to recover from violations occurring from the assured operation. In particular, the code in the exception handler can only be executed if violations are signalled during the execution of the body of the assured operation.

Syntactically, the exception handler has the same structure as the body of an operation and consists of guarded sets of suboperations. Moreover, the execution of the code in the exception handler proceeds in the same way as explained for the body of the operation. After a violation has been signalled, the if-guards in the exception handler are evaluated. Then the suboperations with fulfilled guard are processed. The suboperations in the exception handler are executed to compensate for the signalled violation. These suboperations, like the ones in the body of an operation, can be calls to other operations or primitive operations (SET or PUT). Furthermore, five additional primitive operations can be called from an exception handler. These primitive operations are used to abort, restart or resume execution in the violating process, in the violated assured operation and in the assured suboperations. They are listed below and described as follows:

- **ResumePC**. This primitive operation resumes execution in a violated assured operation and in its suboperations suspended after a violation. The primitive ResumePC is called after the violation has been recovered in the violated assured operation and in the suspended suboperations.
- **ResumeP**. This primitive operation restarts the execution of a violated assured operation after the violation has been recovered.
- **ResumeV**. This primitive operation unblocks the violating process during the recovery of a violation.
- **AbortChildren**. This primitive operation aborts execution in the assured suboperations started from an assured operation and suspended after a violation signal. The primitive AbortChildren is called when the recovery of the suspended assured suboperations is not possible.
- **AbortAll**. This primitive operation aborts the violated assured operation and its assured suboperations. Moreover, it forwards the violation signal to the caller process of the aborted assured operation. The primitive AbortAll is called when the recovery of the violated assured operation is not possible.

The primitive operations described above have been integrated in CAMOS and can only been used within exception handlers.

Requirements fulfilled during the Implementation

During the implementation of assured operations, the following requirements have been fulfilled:

- **Few lines of added/changed code**. One target of the implementation was to show the easy integrability of AS into CAMOS. Therefore, few extensions and changes have been made to the existing code for CAMOS. Only 213 LOC (Lines Of Code) have been added. The time strictly needed to implement the solution for AS described in Section 8.4.1 required two weeks of programming work. Most of the time during implementation was spent to become familiar with the existing code for CAMOS, to understand where to modify the code and to test the implemented solutions.
- **Separation of handling code from the normal code**. To avoid errors during the typing and the execution of CAMOS(L) programs, the code in the body of an assured operation has not been mixed with the compensation code in the exception handler. In particular, calls to the new primitive operations ResumePC, ResumeP, ResumeV, AbortChildren, and AbortAll have been allowed only within the exception handler of assured operations. In addition, the code in the exception handler is only executed if violations of an assured operation occur during the execution of its body. Afterwards, the execution of the normal code in the body of the assured operation can be resumed or not, depending on the choices of violation recovery.
- **Improvement of parallelism**. The implementation of assured operations was performed to provide CAMOS with a mechanism for the concurrency control that could improve parallelism when applied in CAMOS(L) applications.

8.4.3 FURTHER EXTENSIONS OF THE INTEGRATED SOLUTION

The solution for AS integrated into CAMOS is not the most efficient and complete. Further extensions and improvements are possible.

A first extension could be to allow the assurance of single sets of suboperations within an operation by defining assured if statements within the body of the operation. This solution would be more complete and flexible. In this way, when writing an assured operation, it could be possible to choose whether to assure the complete operation or only some parts of it.

Moreover, during the implementation, the compiler CAMOS(L) has not been modified to recognize and translate automatically assured operations from the high-level to the internal representation. Instead, manual modifications in the object code, after the compilation of a program, are required. This can introduce errors since the object code is not very easy to read and modify. Consequently, for a more efficient use of assured operations the compiler should be extended.

In the integrated solution few alternatives have been considered for the recovery of the assured suboperations called from the body of a violated assured operation. In particular, the recovery is possible either for all the assured suboperations or for none of them. A further extension could be to consider a more flexible recovery of the assured suboperations. For example, allowing inheritance of exception handlers and local recovery in the assured suboperations like the inheriting VPQB AS.

8.5 Testing and Evaluating the Implemented Solution in a Case Study

Several tests have been made in a case study to verify the correctness of the implemented solution.

The case study is a production cell which exists in a metal plant in a factory in Karlsruhe, Germany. See [FZI93] for a detailed description of the production cell.

The task of the production cell is to forge metal pieces in a press. Figure 8.5 shows the production cell.

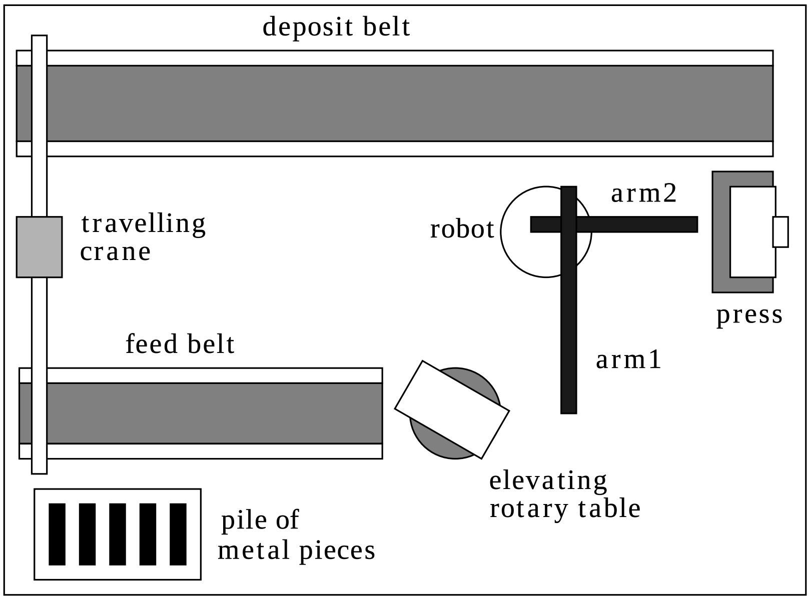


Figure 8.5: The production cell

The following gives a short overview of the work of the production cell:

1. A metal piece is put from the pile onto the feed belt.
2. The feed belt transports the piece to the elevating rotary table.
3. The table goes in its upper position and rotates clockwise 50 degrees, so that arm1 of the robot can take the piece.
4. The robot rotates, so that arm1 points to the table and picks up the pieces. Again, the robot rotates, so that arm1 can deposit the piece in the press, which is in the middle position. After that, the robot turns to a safe position.
5. The press goes up and forges the piece. Afterwards, it moves to a lower position, so that arm2 of the robot can reach the press.
6. The robot rotates arm2 to the press and picks up the processed piece. Then, it rotates to the deposit belt and releases the piece on the belt.
7. The deposit belt transports the piece to the crane and stops here.
8. The crane fetches the piece from the deposit belt and transports it to the feed belt, so that the entire process can start again.

A MCS (Manufacturing Control System) for the production cell has been previously developed at Linköping University as part of a Master's thesis [Die97]. The MCS is modelled on the CAMOS architecture and the software used for the control of the machinery is programmed in CAMOS(L).

Since CAMOS(L) did not support concurrency control mechanisms, the previous existing implementation of this case study showed the necessity to avoid consistency problems when concurrent execution threads access shared manufacturing equipment.

A new implementation for the production cell has been developed using assured operations to control the movements of machinery during the access to common resources. In particular, assured operations have been applied to control the access to the following shared equipment:

- The elevating rotating table. Consistency problems could occur when metal pieces are fetched from the feed belt to the elevating rotating table while the arm1 of the robot tries to reach the table.
- The feed belt. Concurrency control is needed because metal pieces could be fetched simultaneously from the pile and from the travelling crane.

During the implementation, an existing graphical simulation for the case study has been used to check if the control system of the production cell worked properly when applying assured operations.

Moreover, the graphical simulation has been used also to evaluate the performance of the new implemented solution with respect to the previous one. The method used for the evaluation was to compare the throughput during the execution of the two programs. In particular, first the previous existing program for the production cell and then the new version using assured operations have been executed in the same environment. During the execution, the graphical simulation has been observed and for both executions, the time needed to forge the five pieces in the pile (executing for each piece the operations 1-8 described above) has been calculated.

Experimental results found that 297 seconds are needed to forge five metal pieces using the old program of the production cell. Using the new version only 274 seconds are needed to process the same five pieces.

According to the definition given in Section 3.2.3 the *Throughput* of a concurrent system is calculated by dividing the units of work produced during the execution of a concurrent program by the time needed to execute the program.

$$Troughput = \frac{UnitsOfProducedWork}{TimeNeededToExecuteTheProgram}$$

Applying the formula above, the throughput of the production cell using the old implementation was:

$$Throughput_{Old} = \frac{5}{297} \cdot \frac{Units}{Secs} = 0,016835 \cdot \frac{Units}{Secs}$$

Instead, with the new implementation, the throughput of the production cell was:

$$Throughput_{New} = \frac{5}{274} \cdot \frac{Units}{Secs} = 0,018248 \cdot \frac{Units}{Secs}$$

Therefore, the solution using assured operations improved the throughput of the production cell when compared with the previous implementation.

A further result obtained by rewriting the program of the production cell using assured operations, is a greater compactness of code. In the previous implementation, to guarantee that the wait condition of an operation remained verified during the execution of its body, the condition had to be checked many times. Checks were necessary for example, in all if-guards and again in the suboperations called from the body of the operation. Instead, using assured operations, changes to the assured wait condition are automatically monitored during the execution of the body and additional checks are not required. As a consequence, the code appears more compact and is easier to use, read and maintain. The maintenance of code is also an important issue, since the main reason for controlling production processes via computers is to quickly adapt the production machinery to new products required for the market.

In conclusion, two benefits resulted from applying assured operations to control the movements of machinery during the access to common resources: the greater compactness of code and the improvement of the throughput.

8.6 Conclusions

Experimental results showed that AS is particularly suitable for use of the CAMOS environment. First, the overhead spent on monitoring violations of assured shared data does not delay the movements of the machinery controlled by CAMOS(L) programs. In fact, the time needed for the machinery to move is considerable longer compared with the time needed to control them via software. Therefore, the time needed for monitoring violations delays only the execution of the control software but not the machinery in the physical world. Moreover, CAMOS provides active rules, which represent a natural way to detect automatically violations of assured data. The use of active rules made the integration of AS in CAMOS fairly easy, requiring only few modifications to the existing code. The evaluations of performance done in the case study, showed the improvement of parallelism when the implemented mechanism AS is used for concurrency control. However, since CAMOS does not support monitors, semaphores nor conditional critical regions, comparisons of performance have not been possible between these techniques and the solution for AS implemented in CAMOS.

Chapter 9 Evaluations of Performance

In this chapter the performance of VPQB AS and monitors are compared through a worst case analysis.

9.1 Comparing the Performance of VPQB AS and Monitors

A worst case analysis is used to compare the performance of VPQB AS and monitors. First, the maximum blocking time *Max-Block*, which affects the concurrent execution when the two techniques are applied for concurrency control is estimated. Next, the throughput in the worst case of execution for monitors and VPQB AS is compared.

MaxBlock is the *BlockingTime* that affects the concurrent execution in the worst case of execution using a concurrency control technique. According to the definition of *BlockingTime* given in Chapter 3, the *MaxBlock* will be calculated as follows:

$$MaxBlock = T_{conc} \text{ in the worst case of execution - } t_{max}$$

Where, T_{conc} is the time needed for the execution of a concurrent program and t_{max} is the time strictly necessary for the longest concurrent process to execute without considering the delays during the access to shared data. See Section 3.2.2 for further details.

The comparison of the performance of VPQB AS and monitors will be done for a concurrent system with N_p concurrent processes $p_1 \dots p_n$ sharing a resource *s*. The task of each process during its execution is to use the resource *s*. The following assumptions are made:

- All the concurrent processes start executing at the time $t_0 = 0$
- The time t_{use} to use the resource *s* is equal for each concurrent process
- The time strictly necessary for each process to execute is exactly the time t_{use} needed to use the resource *s*
- The concurrent execution terminates when all the processes have used the resource *s*
- The time needed for the scheduling of the processes is considered infinitely fast compared with the time needed to use the resource *s* and will be ignored

In the hypothetical case that all the N_p processes can execute in parallel without consistency problems on the shared resource *s*, concurrency control is not necessary. Therefore, as shown in Figure 9.1, the time needed for the concurrent execution to terminate is exactly $T_{conc} = t_{use}$.

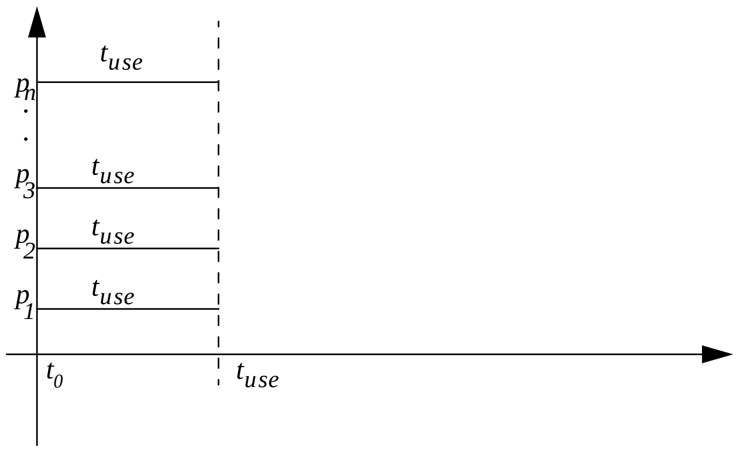


Figure 9.1: T_{conc} for $p_1 ... p_n$ when concurrency control during the access to s is not necessary

However, corruptions on the shared data may occur and a concurrency control technique must be used to preserve consistency.

In the next two sections, the maximum blocking time that affects the concurrent execution of $p_1 ... p_n$ when monitors and VPQB AS are applied to control the access to s is estimated. The time necessary to access and release the shared resource when monitors are used, as well as the time needed to detect possible corruptions when using VFQB AS are considered infinitely fast compared to t_{use} and will be ignored.

9.1.1 Maximum Blocking Time for Monitors

The maximum blocking time for monitors, $MaxBlock_{mon}$, is the maximum delay that affects concurrent execution when monitors are used to control the access to the resource s. This delay occurs in the worst case that all the N_p concurrent processes try to access the shared resource at the same time. Since monitors serialize the access to the resource s, only one process at a time is allowed to use the resource. As shown in Figure 9.2, the time necessary for the concurrent execution of $p_1 \ldots p_n$ is $T_{conc} = N_p t_{use}$ which is the time needed for the process p_n to finish its execution.

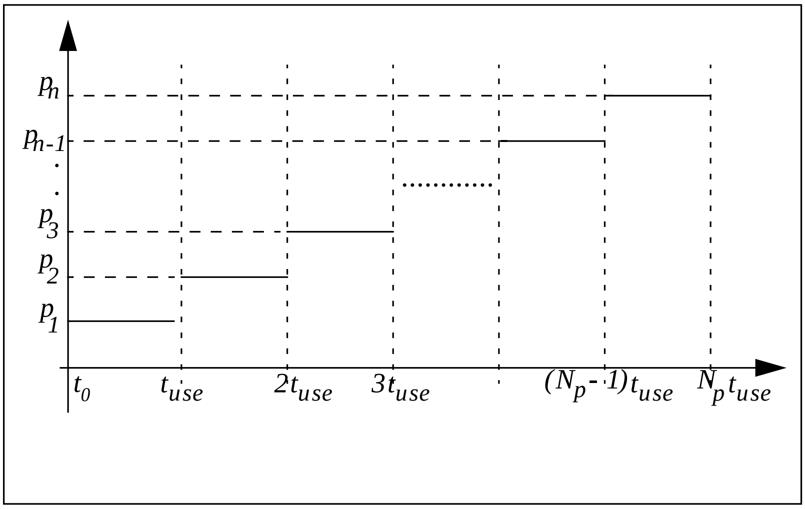


Figure 9.2: T_{conc} in the worst case of execution for monitors

According to the definition of $MaxBlock$, the $MaxBlock_{mon}$ is obtained as follows:

$$MaxBlock_{mon} = Npt_{use} - t_{use} = (N_p - 1)t_{use}$$

9.1.2 MAXIMUM BLOCKING TIME FOR VDQB AS

The maximum blocking time for VDFB AS, $MaxBlock_{AS}$, is the maximum delay that affects the concurrent execution when VPQB AS is used for the concurrency control. This delay depends on the number of violations v that occurs during the concurrent access to the common resource s. Assuming the time t_{rec} necessary to recover from a violation is equal for each signalled violation, the $MaxBlock_{AS}$ is estimated as follows.

The number of violations is v = 0. All the N_p concurrent processes use the resource s in parallel. As shown in Figure 9.3, the time needed for the concurrent execution of $p_1 \dots p_n$ is $T_{conc} = t_{use}$. In this case, the worst case and the best case for VPQB AS coincide. Since delays do not affect the concurrent execution, the $MaxBlock_{AS} = t_{use} - t_{use} = 0$.

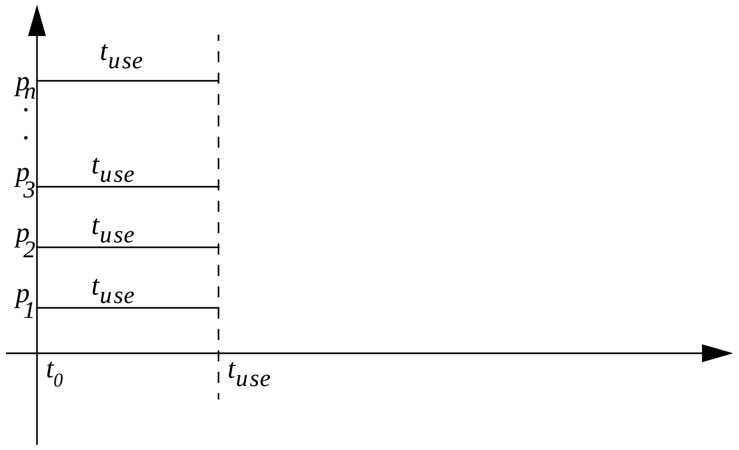


Figure 9.3: T_{conc} in the worst case for VPQB AS when $v = 0$

The number of violations is v = 1. The $MaxBlock_{AS}$ that affects the concurrent execution of $p_1 \ldots p_n$ occurs in the following worst case:

- The violation is signalled to a concurrent process just before this process finishes using the resource.
- To avoid error propagation, the access to the shared resource is locked by the violated process for the duration of the recovery.
- After recovery from the violation, the violated process must restart its access to *s*.
- The violating process is resumed at the end of the violation recovery.

For example, referring to Figure 9.4, process p_2 violates process p_1 at the time $t_{use} - \varepsilon$. p_2 suspends its execution, p_1 recovers from the violation. During the recovery p_1 locks the access to the resource *s*, consequently, the access of $p_3 \ldots p_n$ to *s* is delayed for the duration of the recovery. After the violation has been recovered, p_2 is resumed and finishes its execution, p_1 is restarted, and finally $p_3 \ldots p_n$ can continue their access to *s*.

However, the violating process could be the violated process itself. In this situation the worst case is when the violation occurs just before the violated process finishes its access to *s*. To avoid error propagation the violated process locks the access to *s* during the violation recovery, delaying the execution of the other concurrent processes. Then, after recovery from the violation, the violated process restarts using the resource *s* while the other concurrent processes finish their access. This situation is shown in Figure 9.5. In the Figure, p_1 is the process that violates itself and v_1 indicates the violation.

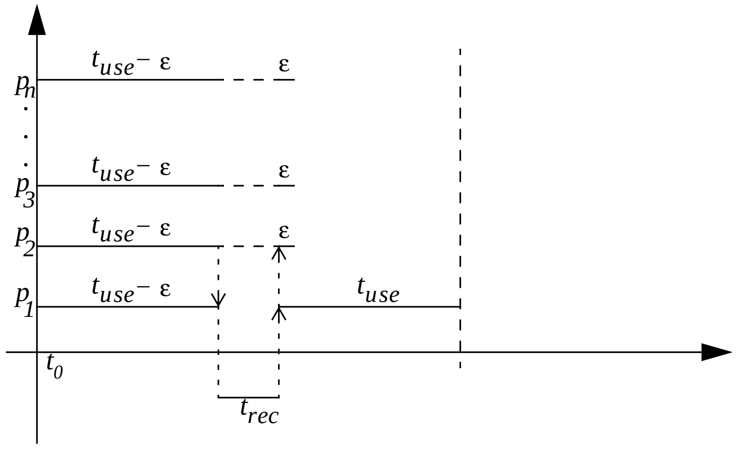


Figure 9.4: T_{conc} in the worst case for VPQB AS when v = 1

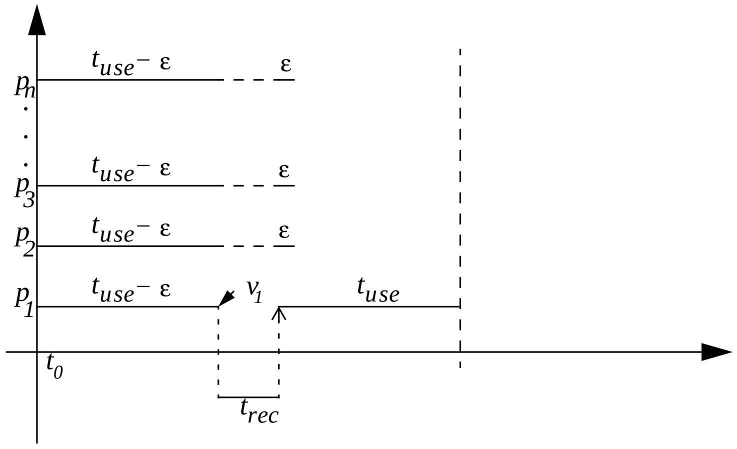


Figure 9.5: T_{conc} in the worst case for VPQB AS when v = 1 and the violating process violates itself

In both the cases described above and shown in Figures 9.4 and 9.5, the time needed for the concurrent execution is the time needed for the most delayed process to terminate, which is:

$$T_{conc} = (t_{use} - \varepsilon) + t_{rec} + t_{use}$$

Assuming ε small enough to be ignored, this time can be approximated as $T_{conc} = 2t_{use} + t_{rec}$. As a consequence, the $MaxBlock_{AS}$ when $v = 1$ is:

$$MaxBlock_{AS} = 2t_{use} + t_{rec} - t_{use} = t_{use} + t_{rec}$$

More generally, if the **number of violations is v =** n_v (n_v is a non negative integer), the worst case of execution for VPQB AS is when all the n_v violations delay a single process p_i. In particular, each violation rises when p_i has almost finished its access to the resource *s*. To avoid error propagation during the recovery of each signalled violation, the access to the shared resource is locked. Moreover, after the violation recovery, p_i must restarts the access to *s* from the beginning.

As nested violations have not been allowed in the semantics of VPQB AS, the situation described above happens for example in the following extreme case:

- p_i violates itself n_v times.
- p_i recovers from each violation and locks the access to *s* during the recovery.
- After a violation has been recovered, p_i restarts its access to *s*.
- Only after the n_v violations happened process p_i succeed in using *s*.
- All the processes $p_1 \dots p_n <> p_i$ are delayed during the recovery of the first violation.

Figure 9.6 shows such a situation. In particular, $p_i = p_1$ and the labels $v_1 \ldots v_n$ indicate the violations that occur during the execution.

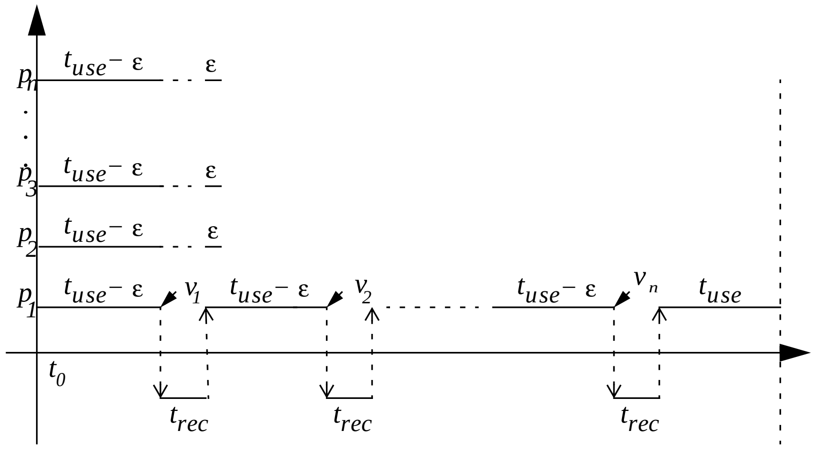


Figure 9.6: T_{conc} in the worst case for VPQB AS when $v = n_v$

In the case illustrated above, the time needed for the concurrent execution is the time needed for p_1 to terminate which is:

$$T_{conc} = n_v(t_{use} - \varepsilon) + n_v t_{rec} + t_{use}$$

Assuming ε small enough to be ignored, this time can be approximated as $T_{conc} = (n_v + 1)t_{use} + n_v t_{rec}$. Therefore, the $MaxBlock_{AS}$ when $v = n_v$ is the following:

$$MaxBlock_{AS} = (n_v + 1)t_{use} + n_v t_{rec} - t_{use} = n_v(t_{use} + t_{rec})$$

9.1.3 COMPARISON OF THROUGHPUT AND ANALYSIS OF THE NUMBER OF VIOLATIONS

Now that the $MaxBlock_{AS}$ and $MaxBlock_{mon}$ have been estimated, a comparison of the worst case throughput for monitors and VPQB AS is possible.

As defined in Section 3.2.3 the *Throughput* of a concurrent system is calculated by dividing the units of work produced during the execution of a concurrent program by the time needed to execute the program. According to this definition and assuming that one unit of work is produced for each successful use of the resource *s*, the worst case throughput for monitors is given with the following formula:

$$Throuthgput_{mon} = \frac{N_p UnitsOfWork}{N_p t_{use}}$$

The worst case throughput for VPQB AS is instead the following:

$$Throughput_{AS} = \frac{N_p UnitsOfWork}{(n_v + 1)t_{use} + n_v t_{rec}}$$

As the throughput is inversely proportional to the time needed for the concurrent processes to use the resource *s*, VPQB AS has a better performance than monitors when:

$$(n_v + 1)t_{use} + n_v t_{rec} < N_p t_{use}$$

namely, when:

$$MaxBlock_{AS} < MaxBlock_{Mon}$$

when

$$n_v(t_{use} + t_{rec}) < (N_p - 1)t_{use}$$

From the formula above, an upper limit on the number of violations can be found. If the number of violations is maintained under this limit, VPQB AS has a better throughput compared with monitors. This happens when:

$$n_v < \frac{(N_p - 1)}{1 + \frac{t_{rec}}{t_{use}}}$$

Setting the ratio $t_{rec}/t_{use} = r$, it is possible to rewrite the relation above as follows:

$$n_v < \frac{(N_p - 1)}{1 + r}$$

This relation can be studied as a first degree function in three variables n_v, N_p and *r*. Moreover, the function can be represented fixing alternatively *r* or N_p as parametric constants.

Keeping first **the number of processes N_p as a parametric constant *k***, the number of violations n_v can be studied in function of the ratio *r*. Since it makes sense to have a mechanism for the concurrency control when at least two concurrent processes share a common resource, *k* must be $k \geq 2$.

Figure 9.7 shows the function $n_v = (k - 1)/(1 + r)$. As this function is parametric in *k*, Figure 9.7 shows how the curve associated with the function gets closer to the origin of the axis for smaller values of *k*.

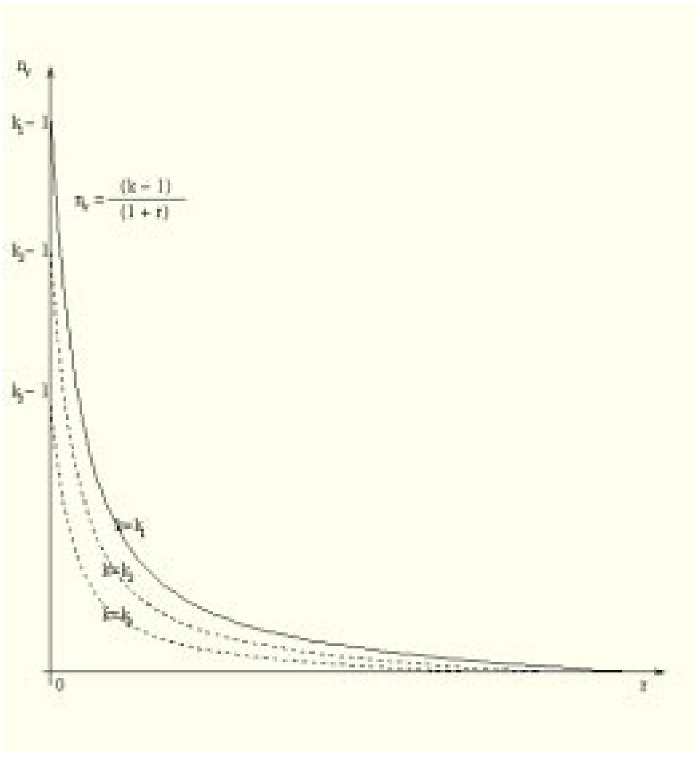


Figure 9.7: The function $n_v = (k-1)/(1+r)$ for $k = k_1$, $k = k_2$, $k = k_3$ and $k_1 > k_2 > k_3 \geq 2$

With fixed k and varying the ratio r, VPQB AS is better than monitors if the number of violations n_v is maintained in the region below the curve associated with the function $n_v = (k-1)/(1+r)$.

For example, with a constant number of processes $N_p = k = 5$, Figure 9.8 shows the curve associated with the function $n_v = 4/(1+r)$ when the ratio r is $0 \leq r \leq 6$.

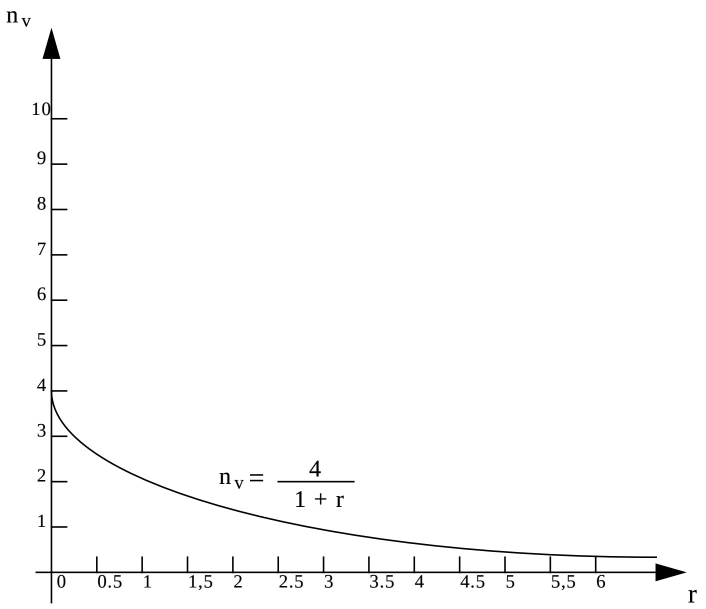


Figure 9.8: Course of the function $n_v = 4/(1+r)$ when $0 \le r \le 6$

In the figure above, the ratio *r* is $r = 0$ if the time t_{rec} necessary to recover from a violation is zero or if the time t_{use} to use the shared resource is infinitely fast compared with the time t_{rec}. In this case, VFQB AS has a better performance than monitors if during the concurrent access of the five processes to the common resource, the number of violations is maintained strictly less then four. In particular, since it makes sense to have an integer number of violations, the maximum number of tolerated violations is $n_v = 3$.

In the same way, if the ratio *r*=1 (namely if $t_{rec} = t_{use}$) VFQB AS has a better performance than monitors when the number of violations $n_v < 2$. Therefore, only one violation is tolerated in this case during the concurrent access of the five concurrent processes to the shared resource.

Note that when the ratio $r \geq 3$, the number of tolerated violations n_v becomes less than one. This means that VFQB AS is better than monitors only if violations do not occur during the concurrent execution.

Analogously, keeping now **the ratio *r* as parametric constant *k*** ($k \geq 0$), the number of violations n_v can be studied as a function of the number of process N_p.

The course of the function $n_v = (N_p - 1)/(1 + k)$ is shown in Figure 9.9. This function is linear and is closer to the abscissa axis as k increases.

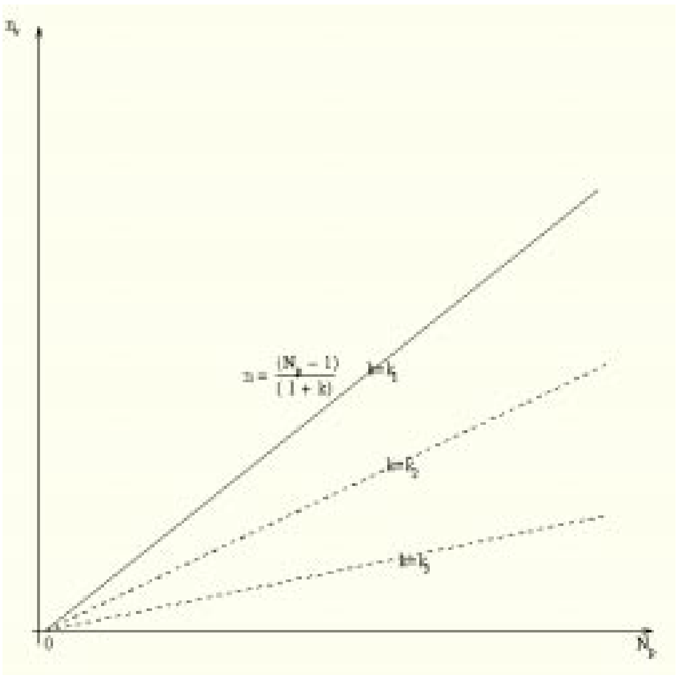


Figure 9.9: The function $n_v = (N_p - 1)/(1 + k)$ for $k = k_1$, $k = k_2$, $k = k_3$ and $k_3 > k_2 > k_1 \geq 0$

To justify using VPQB AS, with fixed k and varying the number of processes N_p, the number of violations should be maintained in the region below the line associated with the function $n_v = (N_p - 1)/(1 + r)$.

For example, fixing the ratio $r = k = 1$ as a constant, Figure 9.10 shows the line associated with the function $n_v = (N_p - 1)/2$ when the number of processes N_p is $1 \le N_p \le 10$.

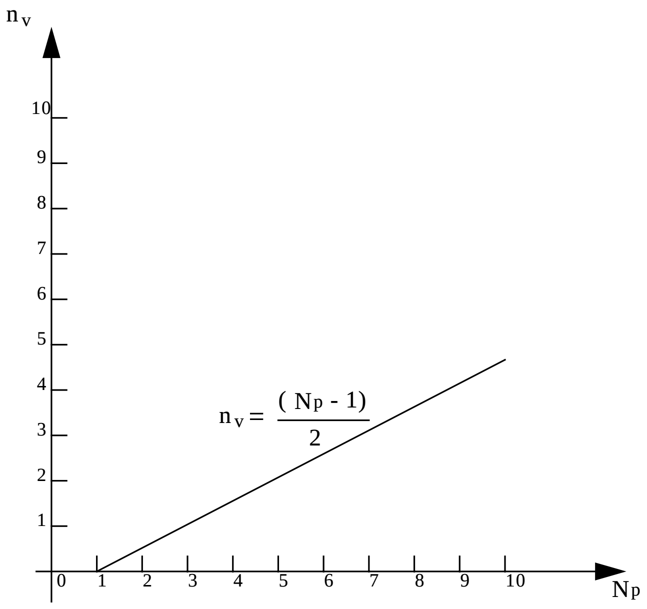


Figure 9.10: Course of the function $n_v = (N_p - 1)/2$ when $1 \le N_p \le 10$

In particular, if the N_p=10, the number of violations n_v occurring during the concurrent execution must be $n_v < 4.5$. Consequently, as there must be an integer number of violations, the maximum number of tolerated violations for VFQB AS to have a better performance than monitors is $n_v = 4$. Analogously, if N_p=5, only one violation is tolerated during the concurrent access of the five processes to the shared resource (since n_v must be strictly less than two), and so on for other values of N_p.

9.2 Conclusions

The analysis presented in the previous section proved that using VPQB AS to control the concurrent access to a shared resource can improve the parallelism of a concurrent system. In particular, the worst case performances of VPQB AS and monitors have been compared. From the comparison it emerged that VPQB AS is better than monitors when the number of violations occurring during the concurrent access to the shared resource is below some limit. Furthermore, an upper bound of the number of violations tolerated in the worst case has been found. If the number of violations is maintained under this bound, VPQB AS provides a better throughput than monitors. However, as the number of violations grows, the benefits on parallelism are lost and monitors perform better.

Chapter 10 Concluding remarks

This chapter describes the advantages and limitations of VPQB AS. Next, it discusses the contribution of this thesis to the existing idea of AS. Finally, it presents a summary of the thesis.

10.1 Advantages and Limitations of VPQB AS

What are the advantages and limitations of the new mechanism for the concurrency control?

The advantages of VPQB AS include:

- The simple integration of the mechanism in concurrent programming languages. As presented, the idea of the new mechanism is simple since it combines a guard predicate, a way of monitoring the assured shared data with exception handling. As a result, the practical integration in concurrent programming languages should be fairly easy. Especially, if

exception handling facilities are already supported
- Applying VPQB AS for concurrency control allows concurrent processes to simultaneous accesses the assured shared data, reducing delays during the concurrent execution

However, to really have improvement of parallelism when VPQB AS is applied, the following limitations must also be considered:

- The number of corruptions on the assured shared data must be less than an upper bound
- The overhead necessary to monitor the assured shared data must not delay the concurrent execution very much if at all
- The possibility of recovery from corruptions on the assured shared data is required
- Since during the recovery concurrent execution is delayed, the time necessary for the exception handling must be upper bounded

Furthermore, the limitations listed above can mean that VPQB AS is difficult to apply and limit the situations in which VPQB AS can be used with benefits to some region of the possible situations.

This suggests a more flexible and complete solution to the concurrency control problem: to combine the use of both VPQB AS and monitors (or one of the other traditional mutual exclusion techniques) in a system. However, a careful analysis of the possible corruptions occurring on the shared data, the possibility of recovery and the time necessary to detect and handle the inconsistencies is required to decide when to apply one or the other technique.

10.2 Contribution of the Work

This thesis extends the existing idea of AS to describe the VDQB AS with the following contributions:

- It clearly defines, after a violation of the assured data has occurred, which processes are signalled, which processes are in charge of handling the violation, which processes are blocked and when these processes are unblocked.

- It introduces the quasi-blocking semantics for the violating process that allows a flexible unblocking of a violating process after it violates assured shared data.
- It provides the processes created within the assured region of an AS statement with their own exception handlers (inheriting VPQB AS). This extension allows a flexible and fast handling of violations occurring during the execution of an AS statement, without incurring problems such as variable visibility and parameter passing.
- It allows the forwarding of violation signals from a violated process to its parent process. In this way, violations that are not recoverable in the violated process can be handled by its parent process by aborting or restarting (with the same or different parameters) the excepted child. Consequently, the parent process can supervise the execution of its children during the recovery from violations.
- It defines the decisions to be taken in the exception handlers of the process which executes the AS statement, of its parent process and of any processes created within the AS statement. Both continuation and abortion semantics have been allowed during exception handling.
- It defines the semantics in the cases that a violating process is the violated process itself or one of its child processes created within the AS statement.
- It defines how to handle violations of assured data occurring during the execution of procedures called from the violated AS statement.
- It specifies the semantics for nested AS statements.

A further contribution derives from the implementation of VPQB AS in CAMOS, since it proved the validity of the new concurrency control technique. Moreover, the experimental results from the implementation showed the benefits in parallelism when the new mechanism is applied. Finally, from the comparison of the performance of VPQB AS and monitors, the given contribution is an estimation of the number of violations under which VPQB AS improves parallelism compared to monitors.

10.3 Summary of the Thesis

In shared data environments concurrent processes simultaneously access common resources. Mutual exclusion techniques are traditionally applied for concurrency control. To prevent data corruption they allow concurrent processes to operate in mutual exclusion on the shared data by locking access to it. However, a problem with mutual exclusion techniques is the strictness of locking that introduces delays in the concurrent execution and reduces the throughput.

Assured Selection (AS) is an alternative approach for concurrency control. The idea of AS is to relax the strictness of the lock and to allow several processes to simultaneously access the shared data. However, due to the relaxed restrictions allowing concurrent access, the consistency of the shared data might not be preserved. Any corruptions that occur are detected and solved afterwards using exception handling.

When the number of corruptions is low and the overhead to detect them is less than an upper bound, AS has better performance than traditional mutual exclusion techniques. Since processes simultaneously access the shared data, delays in concurrent execution are reduced and an improvement of the throughput is the result.

This thesis investigated a concurrency control technique based on the approach of AS. During the investigation alternative solutions for AS have been studied and classified. Moreover, semantic issues for the solution VPQB have been defined in detail. Next, the new concurrency control technique VPQB AS has been defined, tested and evaluated through implementation in CAMOS. Finally, the performances of VPQB AS and monitors have been compared through a worst case theoretical analysis.

The implementation demonstrated that it was fairly easy to integrate VPQB AS with the CAMOS system. Moreover, two benefits emerged from the implementation in CAMOS applications: an improvement of the throughput and a greater compactness of code. Furthermore, an analysis of the performance of VPQB AS and monitors found that VPQB AS has a better throughput than monitors. However, this result only holds when the number of corruptions on the shared data remains under a certain bound. As the number of corruptions grows, monitors provide better performance. This leads to the conclusion that either VPQB AS and monitors are needed. For a more flexible and complete solution for the concurrency control problem, the use of VPQB AS and monitors or another of the traditional mutual exclusion techniques could be combined. Therefore, an accurate evaluation of the frequency of corruptions on shared data and the overhead cost of recovery is needed to decide which technique is most adequate for each situation.

Chapter 11
Future work

As part of the thesis was to investigate the approach of AS, Chapters 6 and 7 presented the semantics of VPQB AS. A first suggestion for future work is to further extend the semantics of the studied mechanism. During the investigation, nested violations of AS statements have not been allowed. However, the problem of nested violations arising and their recovery should be adequately dealt with. As this problem is similar to nested exceptions arising within exception handling, a suggestion is to study how nested exceptions are solved in concurrent programming languages supporting exception handling. Therefore, a similar solution could be adapted and defined for VPQB AS.

Next future work concerns the precise specification of the semantics of VPQB AS. Since the intention was to keep the focus on the many semantic issues of VPQB AS, a formal specification of the semantics has not been provided within the thesis. Consequently, the semantics should be formally defined. Moreover, proof rules should be provided to verify the correctness of the semantics.

Concerning the implementation of VPQB AS, not all the semantic issues studied in the theory have been implemented. In particular, the inheriting VPQB AS, studied to reduce delays during exception handling, should be tested experimentally to verify if it really improves the parallelism as expected.

Another area for future work concerns the comparison of performance between VPQB AS and monitors. A worst case analysis has been presented for this purpose in Chapter 9. However, evaluations of performance have not been provided for the average case. For a more complete and precise comparison of the two mechanisms, an average case analysis should also be carried out. Furthermore, to verify the results of the analysis, VPQB AS could be integrated in a concurrent programming language that supports monitors. Therefore, experimental tests could be done to estimate the parallelism achieved with the two mechanisms in practical situations.

Appendix A

Abbreviations

AS:	Assured Selection
ADMS:	Active Database Management Systems
AMOS:	Active Mediator Object System
CAMOS:	Control Application Mediator Object System
CAMOS(L):	Control Application Mediator Object System Language
EDSLAB:	Engineering Databases and Systems Laboratory
LOC:	Lines Of Code
MaxBlock:	Maximum Blocking Time
MCS:	Manufacturing Control Systems
OB:	Other Blocking
OB:	Other Blocking
OQB:	Other Quasi-Blocking

PB: Parent Blocking

PNB: Parent Non-Blocking

PQB: Parent Quasi-Blocking

RTSLAB: Real Time Systems Laboratory

SAD: Single Access Delay

SRD: Single Resource Delay

T_{conc}**:** Time for concurrent execution

TD: Total Delay

VDB: Violated Blocking

VDNB: Violated Non-Blocking

VDQB: Violated Quasi-Blocking

VPB: Violated Parent Blocking

VPNB: Violated Parent Non-Blocking

VPQB: Violated Parent Quasi-Blocking

References

[And81] Anders, T., Lee, P. A., *Fault Tolerance: Principles and practice*, Prentice-Hall International, U.S.A.

[Sil94] Silberschatz, A., Galvin, P., B., *Operating System Concepts*, Addison-Wesley, Fourth Edition, U.S.A.

[Buh95] Buhr, P., A., Fortier, M., Coffin M., H., *Monitor Classification*, ACM Computing Survey, March 1995, pp. 63-109

[Bur97] Burns, A., Wellings, A., *Real-Time Systems and Their Programming Languages*, Addison-Wesley, Second Edition, U.S.A.

[Cam86] Campbell, R., H., *Error Recovery in Asynchronous Systems*, IEEE Transactions on Software Engineering, August 1986, pp. 811-826

[Die97] Diederich, J., *Modelling a Production Cell Using the CAMOS Language for Manufacturing Control*, Master's thesis LiTH-IDA-Ex-97/46, Dept. Computer and Information Science, Linköping University, Sweden

[Fla97] Flanagan, D., *Java in a Nutshell*, O'Reilly, Second Edition, U.S.A.

[Fal96] Falkenroth, E., *Data Management in Control Applications - A Proposal Based on Active Database Systems*, Lic thesis 589, Dept. Computer and Information Science, Linköping University, Sweden

[FZI93] http://www.fzi.de/prost/projects/production_cell/ProductionCell.html

[Karl94] Karlsson, J., S., Larsson, S., Risch, T., Sköld, M., W., M., *AMOS User's Guide*, Linköping University, Sweden

[Law92] Lawson, H., W., *Parallel Processing in Industrial Real-Time Applications*, Prentice-Hall International, U.S.A.

[Rom97] Romanovsky, A., *Practical Exception Handling and Resolution in Concurrent Programs*, Computer Languages, April 97, pp. 43-58

[Tho98] Thomasian, A., *Concurrency Control: Methods, Performance, and Analysis*, ACM Computing Survey, March 98, pp. 71-117

[Wid96] Widom, J., Ceri, S., *Active Database Systems: Triggers and Rules For Advanced Database Processing*, Morgan Kaufmann Publishers, San Francisco, U.S.A.