Unifying Framework for Message Passing*

Tomas Plachetka

Comenius University, Bratislava, Slovakia

Abstract. Theoretical models are difficult to apply for the analysis of practical message passing systems used today. We propose a model which can be used for such an analysis. Our framework for message passing is in many ways similar to the framework for transactional database systems. The abstract message passing system is defined in our framework independently of hardware, operating system and means of communication. The interface between the application and the message passing system consists of four basic abstract message passing operations. The application can be written in any programming language, provided that the application's communication primitives can be translated into semantically equivalent sequences of the basic message passing operations. We prove that a restricted version of our model is as powerful as the unbounded asynchronous channel model. We also prove that MPI, the Message Passing Interface, is in some sense weaker than our restricted model and therefore also than the unbounded asynchronous channel model.

1 Introduction

The reason for the introduction of a unifying framework is that we know of no theoretical message passing model which can be directly mapped onto contemporary practical systems. For instance, the abstract channel model [1], [7] has been used in computer languages and software libraries which support parallel computation, e.g. [9], [4], [7]. Nevertheless, the mapping of the abstract channel model onto a computer network is not apparent for the following reason. A *channel* is an unbounded first-in-first-out data structure which stores messages sent to the channel by a sender process; a receiver process removes the messages from the channel, or blocks if the channel is empty. The wires in computer networks have no capacity—either the receiver or the sender processes can store messages, but not the wire between them. Therefore channels cannot be directly mapped onto wires and vice versa. We propose a model which uses neither channels nor wires. It uses an abstraction of communication which can be efficiently mapped onto different communication mechanisms provided by contemporary networking and shared-memory systems. Mutual simulations of various abstract models are summarised in the *invariance thesis: "'Reasonable' machines can simulate each other with a polynomially bounded overhead in time and a constant overhead in space."* [12]. We will show that our model is 'reasonable' in this sense.

* This work was partially supported by the grant APVT-20-018902.

Our framework fits into the framework for transactional database systems which is well-accepted among academic researchers and implementors of the systems [3], [2], [5]. The latter defines a clean interface between a database transaction and a database system. This interface consists of only four basic operations which operate on database records: READ, WRITE, INSERT and DELETE (the last two operations are often omitted in database textbooks which silently assume that the database is non-empty and its cardinality does not change). The semantics of these basic operations is defined independently of the actual database programming language (e.g. SQL). It is only required that the application's language primitives can be automatically translated into equivalent sequences of the basic operations. This allows for the programming of database transactions without any knowledge as to how the four basic operations are implemented, independently of whether the database system is centralised or distributed and independently of the hardware or the operating system used to run the database system. This also gives rise to the development of important abstract theories such as serialisability and recovery which help the implementors of database systems to optimise their systems by reordering the basic operations in the system, while adhering to the semantics of the basic operations. Altogether, the framework for transactional database systems is a standard with a solid scientific background which helps to make complex database systems robust and reliable. In our opinion, this all holds for our message passing framework—only the set of the basic operations is different. The basic database operations work with database records, whereas the basic message passing operations work with messages.

This paper is organised as follows. Section 2 describes the components of our framework and formally defines its main component, the message passing system. This definition induces the semantics of basic message passing operations. We prove in Section 3 that a restricted version of our model can simulate the unbounded asynchronous channel model and vice versa within the bounds of the invariance thesis. This means that our restricted model is as powerful as other abstract models. We then prove that MPI, the Message Passing Interface [11], [10] cannot simulate our restricted model within the bounds of the invariance thesis and is therefore weaker than the asynchronous channel model. The same holds for other practical systems—we chose MPI as it is becoming a de facto industrial standard for programming parallel applications. Section 4 concludes the paper.

2 Components of the Message Passing Framework

This section describes the roles of the components used in our framework. Fig. 1 depicts the relationships between the components. *Process* communicates with other processes only by submitting basic message passing operations to the message passing system. (We will assume throughout this paper that the set of processes does not change with time; we only make this restriction for the sake of simplicity.) A process can be a process in the POSIX sense but our framework does not require

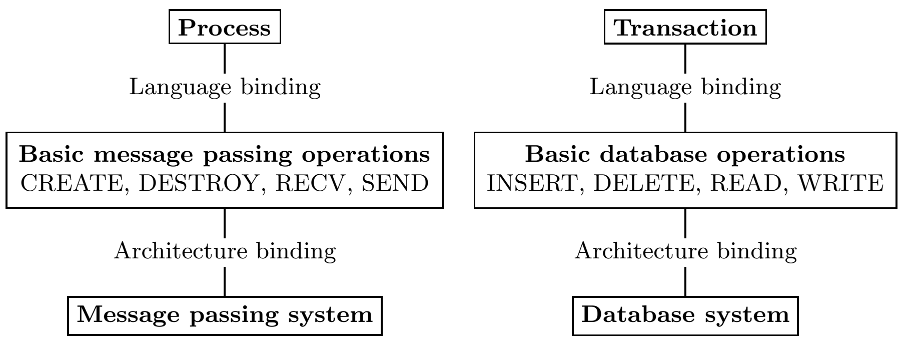

Fig. 1. Left: Components of the message passing framework; Right: Components of the database framework

that. The system regards a process as a single entity with a unique identifier from which it reads a stream of basic message passing operations. A process corresponds to a transaction in the database framework. *Language binding* translates communication primitives used in processes (e.g. a broadcasting primitive, a barrier primitive etc.) into semantically equivalent sequences of basic message passing operations. The use of synchronous and asynchronous communication primitives in processes does not influence the semantics of basic message passing operations. The interface between the processes and the message passing system consists of four types of *basic message passing operations* which work with messages: CREATE, DESTROY, RECV, SEND. The semantics of the basic message passing operations is induced by the definition of the message passing system. The representation and contents of messages is arbitrary and does not influence the semantics of the basic operations. *Architecture binding* maps the semantics of the basic message passing operations onto a specific architecture of the message passing system. This mapping may for example include routing algorithms for distributed architectures with different network topologies, algorithms which guarantee faults tolerance; etc. Architecture binding hides similar mechanisms from processes and guarantees that the semantics of the basic operations does not depend on the actual implementation of the system. *Message passing system* is an abstract component which reads basic message passing operations and executes them as it is defined in the rest of this section.

Definition 1 (Submission of a basic message passing operation). *Submission of a basic operation denotes the act of passing the operation from a process (or the language binding layer) to the message passing system.*

Definition 2 (Representation of basic message passing operations). *All basic message passing operations are tuples* $[op, x, Y, m, f, s, t]$*, where* $op \in \{\text{CREATE}, \text{DESTROY}, \text{SEND}, \text{RECV}\}$*;* x *is the identifier of the process which submits the operation;* Y *is a set of process identifiers;* m *is a message;* f *is a boolean function defined on messages (a filter);* s *is either a reference to* *a semaphore object which can be accessed by the message passing system, or* NULL*; t is the time stamp of the submission of the operation (i.e. the time when the operation has been read by the message passing system).*

Definition 3 (Scope of a process). *The* scope *of a process is a memory space where* messages *relating to the process are stored. A message can only be accessed (i.e. read from, written into, shrunk or expanded) by the process in the scope of which the message is stored. A process can create and destroy messages only by submitting* CREATE *and* DESTROY *operations. The system creates, destroys and accesses messages in scopes of processes only as it is defined in this section. In addition, the system stores operations which it has read in the scope of the process which submitted the operation. $SC(x)$ will denote the scope of the process x and $SC(*)$ will denote the union of the scopes of all the processes.*

To keep things simple, we will deliberately mix messages with pointers to messages in the field m. It is obvious where in the text m denotes a message and where it denotes a reference to a message.

Definition 4 (Processing of submitted operations). *The system may at any one time either read one operation or execute one operation. The system only reads those operations which have been submitted. Every submitted operation is only read once by the system. When the system reads an operation, then it updates the operation's time stamp and stores the operation in the scope of the process which submitted the operation. At any time t, the system can only execute an operation which is stored in $SC(*)$ at the time t. The system may postpone the execution of a submitted operation (i.e. operations are not necessarily executed by the system in the order in which they are submitted).*

Definition 5 (Matching operations). *We will say that two basic message passing operations*
$BO_1 = [op_1, x_1, Y_1, m_1, f_1, s_1, t_1]$, $BO_2 = [op_2, x_2, Y_2, m_2, f_2, s_2, t_2]$, (or $BO_2 = [op_1, x_1, Y_1, m_1, f_1, s_1, t_1]$, $BO_1 = [op_2, x_2, Y_2, m_2, f_2, s_2, t_2]$, respectively) are a matching pair (we will also say that BO_1 is an operation matching the operation BO_2 and vice versa) iff

$$(op_1 = \mathrm{SEND} \wedge op_2 = \mathrm{RECV} \wedge x_1 \in Y_2 \wedge x_2 \in Y_1 \wedge f_2(m_1) \wedge$$

$$\begin{gathered}(\forall BO_1' = [op_1', x_1', Y_1', m_1', f_1', s_1', t_1'] \in SC(*) : (BO_1' \equiv BO_1 \vee \\ op_1' \neq \mathrm{SEND} \vee x_1' \notin Y_2 \vee x_2 \notin Y_1' \vee \neg f_2(m_1') \vee t_1' \geq t_1)) \wedge \\ (\forall BO_2' = [op_2', x_2', Y_2', m_2', f_2', s_2', t_2'] \in SC(*) : (BO_2' \equiv BO_2 \vee \\ op_2' \neq \mathrm{RECV} \vee x_1 \notin Y_2' \vee x_2' \notin Y_1 \vee \neg f_2'(m_1) \vee t_2' \geq t_2)))\end{gathered}$$

Informally, a send operation $BO_1 = [SEND, x_1, Y_1, m_1, f_1, s_1, t_1]$ matches a receive operation $BO_2 = [RECV, x_2, Y_2, m_2, f_2, s_2, t_2]$ iff the set of recipients Y_1 contains x_2, the set of senders Y_2 contains x_1, the filtering function f_2 accepts the message m_1 and neither BO_1 nor BO_2 can be replaced with an older operation so that all the previous properties hold.

The definition of matching operations can be weakened if we do not require the messages sent from a process to another process to be received in the same order by the latter process. In such a case, the time-stamps are ignored and the predicate in the definition 5 becomes

$$(op_1 = \text{SEND} \wedge op_2 = \text{RECV} \wedge x_1 \in Y_2 \wedge x_2 \in Y_1 \wedge f_2(m_1))$$

We will use the definition with message ordering (i.e. Definition 5) in the sequel.

Definition 6 (Execution of CREATE **operations).** *The execution of an operation* $[\text{CREATE}, x, Y, m, f, s, t]$ *consists of the following actions performed in an atomic step:*

1. *The system creates a new message m in $SC(x)$.*
2. *If $s \neq$ NULL then the system performs* semaphore_signal(s).
3. *The system removes this operation from $SC(x)$.*

Definition 7 (Execution of DESTROY **operations).** *The execution of an operation* $[\text{DESTROY}, x, Y, m, f, s, t]$. *consists of the following actions performed in an atomic step:*

1. *The system removes m from $SC(x)$.*
2. *If $s \neq$ NULL then the system performs* semaphore_signal(s).
3. *The system removes this operation from $SC(x)$.*

Definition 8 (Execution of RECV **and** SEND **operations).** *The system may execute an operation $BR = [\text{RECV}, x, Y, m, f, s, t]$ at time t only if a matching operation $BS = [\text{SEND}, x', Y', m', f', s', t']$ exists in $SC(*)$ at the time t. If the system decides to execute BR then it must also execute the matching BS in the same atomic step which consists of the following actions:*

1. *The system creates a new message m in $SC(x)$.*
2. *The system copies the contents of the message m' into the contents of the message m.*
3. *The system removes the message m' from $SC(x')$.*
4. *If $s \neq$ NULL then the system performs* semaphore_signal(s).
5. *If $s' \neq$ NULL then the system performs* semaphore_signal(s').
6. *The system removes BS from $SC(x')$.*
7. *The system removes BR from $SC(x)$.*

Definition 9 (Progress of processing). *The system will eventually read every submitted operation and it will eventually execute all* CREATE *and* DESTROY *operations which have been read. Moreover, if a matching operation pair exists in $SC(*)$ at any time t then the system will eventually execute at least one of the operations of that matching pair.*

The last part of Definition 9 is expressed cautiously in order to support alternative definitions of matching operations. For example, replace Definition 5 with the definition which uses no time-stamps and consider the following scenario. A pair of matching operations BS and BR exists in $SC(*)$ at time t. Before either of these operations is executed, another send operation BS' which also matches BR, is stored into $SC(*)$. Then the "at least one" part of Definition 9 allows the system to execute either the pair BR, BS or the pair BR, BS'.

3 Computational Power of Models

We will say that a program which uses primitive statements of a model A is *correct* iff all possible executions of the program with the same input yield the same output. We will say that a *model B can simulate a model A* iff an arbitrary correct program which uses primitive statements of the model A can be written as a functionally equivalent program which uses primitive statements of the model B. The functional equivalence means that for any input, B computes the same output as A; and B terminates iff A terminates (a program terminates iff all its processes terminate). We will only consider imperative programs with a single thread of control in every process and we will use the C-like notation to write the programs. We will say that two models are *of the same computational power* iff they can simulate each other within the bounds of the invariance thesis from Section 1. If a model B can simulate a model A within these bounds but not the other way around then we will say that B is *computationally stronger* than A (or that A is *computationally weaker* than B).

3.1 Asynchronous Unbounded Channel Model

We will shortly describe the asynchronous unbounded channel model (a more formal definition can be found e.g. in [7]). A channel is an unbounded FIFO queue which stores messages. An application program consists of a constant number of parallel processes which communicate exclusively via channels. The number of channels can be arbitrary but the set of channels does not change in run-time. The processes use only two communication primitives with the usual semantics: PUT(ch, m) inserts the message m into the channel ch. GET(CH, m) (where CH denotes a set of channels) atomically reads a message from a channel $ch \in CH$, stores the message into the variable m and then it removes the message from the channel ch. Each channel can be accessed by any number of processes but only one process can access a channel at any time. PUT never blocks; GET blocks until it has not read a message. It is guaranteed that if a channel ch is non-empty at some time and some GET(CH, m) with $ch \in CH$ is blocked at that time then some (not necessarily the same) blocked GET(CH', m') with $ch \in CH'$ will eventually read a message and unblock.

3.2 Our Restricted Model

We will make the following restrictions in our model from Section 2. All the messages will be tuples $[c, m]$, where c (context) belongs to some finite set C and m is of an arbitrary data type. If $M \equiv [c, m]$ then $M[1]$ will denote c and $M[2]$ will denote m. The only filtering functions in basic message passing operations will be $f_{CH}(M) =$ TRUE iff $M[1] \in CH$, $CH \subset C$ (i.e. only testing a context prefix of messages for a membership in CH, $CH \subset C$, will be allowed). All processes will submit CREATE operations only in the following context (i.e. only a blocking CREATE will be allowed): {new(s); semaphore_init(s, 0); [CREATE, x, NULL, M, NULL, s, t]; semaphore_wait(s); delete(s);}. All the processes will submit RECV operations only in the following context (i.e. only a blocking RECV will be allowed): {new(s); semaphore_init(s, 0); [$RECV, x, Y$, M, f, s, t]; semaphore_wait(s); delete(s);}. All irrelevant fields in the 7-tuples representing basic message passing operations will be NULL.

3.3 The MPI Model

The MPI model [11] uses many primitives, but we will only describe those which are relevant for the comparison with the two previous models. MPI does not use the channel abstraction. It uses point-to-point message addressing which is similar to the one of our restricted model. MPI has a primitive MPI_Recv which blocks until it receives a message and it has a nonblocking send primitive, MPI_Isend. Unlike our model, MPI requires the process to free the memory occupied by the message sent used in MPI_Isend. However, the process must not free this memory before the recipient has received the message—in MPI's terminology, before the MPI_Isend completes. In order to detect this completion, each MPI_Isend must be paired either with MPI_Wait which blocks until the MPI_Isend completes, or with nonblocking MPI_Test which returns a value indicating whether the MPI_Isend has completed. As we will show, this pairing requirement makes the MPI model weaker than the previous two models. Note that our model allows for this kind of synchronisation (deferred synchronisation), as a process may include a semaphore s in a send operation and perform semaphore_wait(s) later. Nevertheless, our model does not require the process to do this.

3.4 Mutual Simulations of Models

Theorem 1. *Our restricted model can simulate the channel model and vice versa with a constant overhead factor in both time and space.*

Proof. We will show that our restricted model can simulate the channel model, with a constant overhead factor in both time and space. Consider a program $PROG_1$ which uses the PUT and GET communication primitives of the channel model. We will construct a program $PROG_2$ which is functionally equivalent with $PROG_1$ but only uses the basic message passing operations of our restricted model. Messages in $PROG_2$ will be tuples $[ch, m]$, where ch is a channel identifier in $PROG_1$ and m is a message in $PROG_1$. The program $PROG_2$ will consist of the same processes as $PROG_1$. Let P_* denote the union of all the processes. Replace in each process x in $PROG_2$ each occurrence of PUT(ch, m); with {new(s); semaphore_init(s, 0); [CREATE, x, NULL, m', NULL, s, t]; semaphore_wait(s); delete(s); $m' = [ch, m]$; [SEND, x, P_*, m', NULL, NULL, t];}. Replace in each process x in $PROG_2$ each occurrence of GET(CH, m); with {new(s); semaphore_init(s, 0); [RECV, x, P_*, m', f_{CH}, s, t]; semaphore_wait(s); delete(s); $m = m'[2]$; [DESTROY, x, NULL, m', NULL, NULL, t];}. It follows directly from the definitions of the models that the programs $PROG_1$ and $PROG_2$ are functionally equivalent and that the replacements only incur a constant overhead in both time and space.

We will now prove that the channel model can simulate our restricted model within the bound. Consider a program $PROG_1$ which uses the basic message passing operations of our restricted model. We will construct a program $PROG_2$ which only uses the channel communication primitives PUT and GET. The program $PROG_2$ will consist of the same processes as $PROG_1$. The channel identifiers in $PROG_2$ will be tuples $[c, Y]$ where $c \in C$ and Y is a set of processes (this tuple can be encoded as an integer if the channel model requires it). Replace in each process x in $PROG_2$ each occurrence of the sequence {new(s); semaphore_init(s, 0); $[\mathrm{CREATE}, x, \mathrm{NULL}, m, \mathrm{NULL}, s, t]$; semaphore_wait($s$); delete($s$);} with new($m$);. Replace in each process x in $PROG_2$ each occurrence of $[\mathrm{DESTROY}, x, Y, m, \mathrm{NULL}, \mathrm{NULL}, t]$; with delete($m$);. Replace in each process x in $PROG_2$ each occurrence of $[\mathrm{SEND}, x, Y, m, \mathrm{NULL}, \mathrm{NULL}, t]$; with {PUT($[m[1], Y]$, m); delete(m);}. Replace in each process x in $PROG_2$ each occurrence of the sequence {new(s); semaphore_init(s, 0); $[\mathrm{RECV}, x, Y, m, f, s, t]$; semaphore_wait($s$); delete($s$);} with GET($CH, m$);, where CH is the set of all the channels $ch = [c, m]$ for which $f(ch) = \mathrm{TRUE}$. Note that the set CH can be computed in constant time as the set of first message components c is known and finite and $f([c, m])$ only depends on c. It follows directly from the definitions of the models that the programs $PROG_1$ and $PROG_2$ are functionally equivalent and that the replacements only incur a constant overhead in time and space. □

Theorem 2. *The MPI model cannot simulate our restricted model within the bounds of the invariance thesis.*

Proof. Consider the following program in our model which consists of processes $p0$ and $p1$ ($P0$ will denote the set containing $p0$, $P1$ will denote the set containing $p1$ and f_{TRUE} will denote a function which always returns TRUE).

```
p0(FILE *inp0)
{
  while (! feof(inp0))
  {
    new(s);
    semaphore_init(s, 0);
    [CREATE, $p0$, NULL, $m$, NULL, $s$, $t$];
    semaphore_wait(s);
    delete(s);
    m=fgetc(inp0);
    [SEND, $p0$, $P1$, $m$, NULL, NULL, $t$];
    printf("sent");
  }
}
```

```
p1(FILE *inp1)
{
  while (! feof(inp1))
  {
    new(s);
    semaphore_init(s, 0);
    [RECV, $p1$, $P0$, $m$, $f_{\mathrm{TRUE}}$, $s$, $t$];
    semaphore_wait(s);
    delete(s);
    printf("received %c", m);
    [DESTROY, $p1$, NULL, $m$, NULL,
    NULL, $t$];
    fgetc(inp1);
  }
}
```

It is easy to verify that this program is correct. We will now prove that it cannot be simulated by a program which uses MPI primitives MPI_Recv, MPI_Isend, MPI_Wait and MPI_Test without breaching the bounds of the invariance thesis. (The rest of MPI's primitives apparently does not help in the simulation of the program above.) The process $p1$ receives $n1$ messages from the process $p0$, where $n1$ is the number of characters in the input stream $inp1$ (this number is unknown until the entire stream $inp1$ has been read). This can only be accomplished by calling MPI_Recv $n1$ times in the $p1$. In the process $p0$, MPI_Isend must obviously be called $n0$ times, where $n0$ is the number of characters in the input stream $inp0$. These $n0$ calls must be paired with $n0$ either MPI_Wait or MPI_Test calls in $p0$, otherwise the memory overhead of the MPI program for $n0 = n1$ would depend on $n1$ and would therefore exceed a constant factor. (We recall that even if $p0$ submits SEND operations faster than $p1$ or vice versa, the system is allowed to postpone the reading of these operations until the previous operations of that process have been executed—therefore the program above can be executed in constant memory for $n0 = n1$. Generally, the space complexity of the program above is $c + |n0 - n1|$, where the constant c depends on neither $n0$ nor $n1$.) MPI_Wait cannot be used in any of the $n1$ pairings because the MPI program would not terminate for $n0 > 0$ and $n1 = 0$, whereas the program above would. This implies that MPI_Test must be used in all the $n1$ pairings. In each of these pairings, the nonblocking MPI_Test must be repeatedly called until the corresponding MPI_Isend completes, otherwise the memory overhead would exceed a constant factor. However, in this case the MPI program would not terminate for $n0 > 0$ and $n1 = 0$, whereas the program above would. $\square$

4 Conclusions

We presented a framework for message passing which defines an interface between message passing applications and message passing systems. We proved that its restricted version is as powerful as the unbounded asynchronous channel model (our unrestricted model is apparently at least as powerful). We also proved that the MPI model is less powerful than these models. The substantial difference between the models is that the MPI standard only supports so-called deferred synchronous communication [8]. Statements such as "MPI has full asynchronous communication" [6] are false. This deviation of the MPI standard from theoretical models has negative consequences for efficiency and portability of parallel applications which build on the MPI standard.

Our framework can serve as a well-founded specification of message passing systems. We stress that this specification only defines the semantics of the basic message passing operations (which should be provided by any message passing system), not the means of their implementation. For instance, the implementation of the operations for distributed architectures does not require a global clock despite of the time-stamps in Definition 5. We implemented the framework as a message passing library for several operating systems and network types. Our implementation is thread-safe and polling-free (it uses no busy waiting).

References

1. Andrews, G.A.: Concurrent Programming, Principles and Practice. Benjamin/ Cummings Publishing Company (1991)
2. Bacon, J.: Concurrent Systems (Operating Systems, Database and Distributed Systems: An Integrated Approach). Addison-Wesley-Longman (1998)
3. Bernstein, A.J., and Lewis, P.M.: Concurrency in Programming and Database Systems. Jones and Bartlett Publishers (1993)

4. Galletly, J.: Occam 2. Pitman Publishing (1990)
5. Gray, J., and Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
6. L.P. Hewlett-Packard Development Company: HP MPI User's Guide, eight edition (2003)
7. Peyton Jones, S.L., Gordon, A., and Finne, S.: Concurrent Haskell. In 23rd ACM Symposium on Principles of Programming Languages (1996) 295–308
8. Liebig, C., and Tai, S.: Middleware-Mediated Transactions. In G. Blair, D. Schmidt, and Z. Tari (eds), Proc. of the 5th International Symposium on Distributed Objects and Applications (DOA'01), IEEE Computer Society (2001) 340–350
9. Mitchell, D.A.P., Thompson, J.A., Manson, G.A., and Brookes, G.R.: Inside The Transputer. Blackwell Scientific Publications (1990)
10. MPI Forum. MPI-2: Extensions to the Message Passing Interface (1997)
11. MPI Forum. MPI: Message Passing Interface, Version 1.1 (1998)
12. van Emde Boas, P.: Machine Models and Simulation. In Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), Elsevier and MIT Press (1990) 1–66

Appendix

Semantics of Semaphores

Throughout the paper, we assume the standard semaphore semantics as it is defined in [ISO/IEC 9945-1: 1990 Information Technology. Portable Operating System Interface (POSIX), Part 1: System Application Program Interface, C language]. Although we are convinced that most readers are familiar with the notion of semaphores, we provide this appendix in order to avoid misunderstandings concerning the notation.

A semaphore object is an abstract synchronisation object which keeps an internal variable *count*. (The formal definition of semaphores allows for an arbitrary representation of *count*, provided that the semantics of the semaphore's operations remains unchanged.) The call new(s) creates a semaphore object referred to as s and semaphore_init(s, c) sets the variable *count* belonging to s to c. The call delete(s) destroys the semaphore s.

The call semaphore_wait(s) acquires the semaphore. Its semantics corresponds to the semantics of Dijkstra's operation $P(s)$: if the variable *count* of s is 0 then the calling process blocks in the call, otherwise the variable *count* of s is decreased by 1 and the calling process continues. The testing and decreasing of the variable *count* is an atomic operation.

The call semaphore_signal(s) signals the semaphore and its semantics corresponds to the semantics of Dijkstra's operation $V(s)$: the *count* of s is increased by 1. Moreover, if there is at least one process which is blocked on the semaphore s then one of these blocked processes unblocks and attempts to acquire the semaphore as if it has called that semaphore_wait(s) once again.