Transaction processing primitives and CSP

by J. C. P. Woodcock

Several primitives for transaction processing systems are developed using the notations of Communicating Sequential Processes. The approach taken is to capture each requirement separately, in the simplest possible context: The specification is then the conjunction of all these requirements. As each is developed as a predicate over traces of the observable events in the system, it is also implemented as a simple communicating process; the implementation of the entire system is then merely the parallel composition of these processes. The laws of CSP are then used to transform the system to achieve the required degree of concurrency, to make it suitable for execution in a multiple-tasking system, for example. Finally, there is a discussion of how state-based systems may be developed using this approach together with some appropriate notation for specifying and refining data structures and operations upon them and of how the system may be implemented. This work is intended as a case study in the use of CSP.

1. Introduction

We describe below several primitives for transaction processing systems using Communicating Sequential Processes [1, 2]: the rather trivial case in which there is just a single user; multiple users with a simple locking protocol to achieve mutual exclusion; the same but with queueing for busy resources; and finally, discarding the locking of resources and instead taking the rather optimistic view that conflicts probably will not occur anyway.

As we consider each system we separate its requirements and capture them as individual predicates on the history of the system. In the style that we are exploring in this paper, this is usually done by describing the *firing condition* for an event: If an event occurs, some predicate must hold on the history of events up to that moment. The specification of each system is simply the conjunction of its requirements. This is indeed a powerful and natural way to capture the formal specification of a system. Less familiar perhaps is the approach to implementation in CSP's process algebra. As the specification proceeds, we implement each requirement as a simple communicating process. Keeping both predicate and process as small as possible reduces the task of proving the implementation correct. In CSP, parallel combination of processes corresponds to conjunction of their specifications; thus the implementation of the system is just the parallel composition of the processes implementing each requirement.

The result of the development process that we are describing is an implementation consisting of a highly distributed collection of communicating processes. The laws of CSP are then used to transform the implementation. We

could, if we wished, transform it into a form which is readily translated into occam [3] and have it run on a collection of transputers.

Appendices contain a summary of the notation used in this paper for the benefit of those unfamiliar with CSP.

2. A single-user system

In the transaction processing systems that we are considering, there is a shared data structure that takes values from the set C. We make a gross simplification—for the moment—that there is only one user in the system, who accesses the data by reading and writing values.

In this system, if a and b are drawn from the set of values C, then the event read.a corresponds to the user reading the value a; write.b corresponds to the user writing the value b. Let

$$R \triangleq \{read.c \mid c \in C\}$$

$$W \triangleq \{write.c \mid c \in C\}$$

It is not difficult to see that if there is just a single user, the data behave as we would expect a programming variable would: If the user reads their contents, then she discovers the most recently written value. We can specify this: We want a process VAR with alphabet

$$\alpha VAR \triangleq R \cup W$$

and any trace tr of VAR must satisfy the predicate

$$VARSPEC \triangleq (\forall c \in C \cdot \overline{tr}_0 = read.c \Rightarrow \overline{tr \upharpoonright W}_0 = write.c)$$

That is, if the last thing that happened in the system is that the user read a particular value c, then the most recently written value is also c. For this predicate to make sense, there must be at least one write before the first read. This specification is really describing the firing conditions for events in R. The events in W are left unconstrained.

An implementation is well-known (see, for example, [2, p. 137]):

$$VAR \triangleq (write?x \rightarrow VAR_x)$$

$$VAR_x \triangleq (read!x \rightarrow VAR_x \mid write?y \rightarrow VAR_y)$$

The process VAR is only willing to participate in a write event; having done so, it proceeds as VAR_x. In this behavior, VAR is rather like a nice sort of uninitialized variable: It does not permit a read before the first write. VAR_x, on the other hand, behaves like a variable currently holding the value x. If the user tries to read the variable, she finds that it has the value x; this does not change the value (the variable continues to behave like VAR_x). However, the user can write a new value—say y—replacing the old one (the variable now behaves like VAR_y).

Note that we could have produced an implementation that avoids the use of a state variable, but it would have appeared rather more complicated.

We have adopted the convention that the specification for a process P is called PSPEC. In this paper we have not included any proofs that implementations actually satisfy their specifications; this is for reasons of space, rather than difficulty: In a development of this kind, it is usual to do proofs in a routine manner. Write down the predicate on traces; write down the behavior in the process algebra; write down the proof of satisfaction. Developing all three together *does* offer valuable insights.

3. Multiple-user systems

• *A simple locking protocol*

We now remove the restriction to a single user and specify a multiple-tasking system. Let T be the set of task names. The protocol we wish to describe involves tasks *locking* the data structure before accessing it. We add two new events to the interface: lock and unlock. Define

$$LOCKED \triangleq (tr\downarrow lock - tr\downarrow unlock = 1)$$

$$\overline{LOCKED} \triangleq (tr\downarrow lock - tr\downarrow unlock = 0)$$

LOCKED holds for the trace tr—which is free in the definition—just when there is one more lock in tr than unlock. Similarly, UNLOCKED holds just when there is an equal number of lock and unlock events in tr. A task may have either locked or not locked the data structure:

$$\alpha LOCK \triangleq \{lock, unlock\}$$

$$LOCKSPEC \triangleq (LOCKED \vee \overline{LOCKED})$$

This is implemented by a process which alternates between lock and unlock events:

$$LOCK \triangleq \mu X \cdot (lock \rightarrow unlock \rightarrow X)$$

The definition of the process LOCK is recursive: It is the process X which first engages in the event lock, followed by unlock, and then it behaves like the process X.

Each task guarantees only to access information which it has previously locked. If a task reads a value from the data structure, the data structure must be locked:

$$\alpha READ \triangleq R \cup \alpha LOCK$$

$$READSPEC \triangleq (\overline{tr}_0 \in R \Rightarrow LOCKED)$$

Similarly, if a task writes a value from the data structure, the data structure must be locked:

$$\alpha WRITE \triangleq W \cup \alpha LOCK$$

$$WRITESPEC \triangleq (\overline{tr}_0 \in W \Rightarrow LOCKED)$$

These two requirements may be implemented by separate, but similar, processes: READ and WRITE. Instead, we offer their combination: the single process that permits reading and writing only after a lock, but before the next unlock. Let

$$\alpha\text{READWRITE} \triangleq \alpha\text{READ} \cup \alpha\text{WRITE}$$

$$= \alpha\text{VAR} \cup \alpha\text{LOCK}$$

$$\text{READWRITE} \triangleq \mu X \cdot (\text{lock} \rightarrow \mu Y \cdot (x : \alpha\text{VAR} \rightarrow Y \mid \text{unlock} \rightarrow X))$$

Notice that, because of the way in which we have chosen to implement READWRITE, with alternating locks and unlocks, it also satisfies LOCKSPEC. Thus, when we put LOCK and READWRITE in parallel, we do not constrain the behavior of READWRITE at all. Formally,

$$(\text{LOCK} \parallel \text{READWRITE}) = \text{READWRITE}$$

Each task must behave in the way that we have specified, and guarantee to follow the locking protocol

$$\text{USESPEC} \triangleq (\forall t \in T \cdot t.\text{LOCKSPEC} \wedge t.\text{READSPEC} \wedge t.\text{WRITESPEC})$$

USESPEC is satisfied by

$$\text{USE} \triangleq \|_{t \in T} t : \text{READWRITE}$$

Since conjunction in the specification corresponds to concurrency in the implementation, the universal quantification in USESPEC becomes parallel composition over a set in the process USE. Each task relies on having *exclusive* access to locked information. So, when a task t acquires a lock, no other task may already have it:

$$\text{MUTEXSPEC} \triangleq (\forall t \in T \cdot \overline{tr}_0 = t.\text{lock} \Rightarrow (\forall u \in T \cdot u \neq t \Rightarrow u.\overline{\text{LOCKED}}))$$

An alternative specification—and perhaps a more obvious one—is that, for every pair of tasks t and u owning the lock, t and u are one and the same:

$$\forall t, u \in T \cdot t.\text{LOCKED} \wedge u.\text{LOCKED} \Rightarrow t = u$$

MUTEXSPEC is implemented by a process that guarantees that a new lock cannot be acquired between a lock/unlock pair. Notice that the double parallelism corresponds nicely with the double universal quantification:

$$\text{MUTEX} \triangleq \|_{t \in T} \|_{u \in T - \{t\}} \mu X \cdot (u.\text{lock} \rightarrow u.\text{unlock} \rightarrow X \mid t.\text{lock} \rightarrow X)$$

Also notice that the recursive processes in this definition each have precisely *three* events in their alphabet.

Although MUTEX is truly an implementation, the large amount of concurrency actually makes it rather difficult to use. After some straightforward manipulations, we discover it really is a simple, sequential process:

$$\text{MUTEX} = \mu X \cdot (\Box_{t \in T} t.\text{lock} \rightarrow t.\text{unlock} \rightarrow X)$$

This is probably the implementation that we would have obtained directly from our alternative to MUTEXSPEC. However, we have had the choice of discarding the concurrency that MUTEX originally had; it is always easier to remove concurrency from a process if you do not want it, than to create it if you do.

MUTEX is now simple enough to combine with what we already have:

$$\text{USE} \parallel \text{MUTEX} = \|_{t \in T} \mu X \cdot (t.\text{lock} \rightarrow \mu Y \cdot (x : \alpha(t : \text{VAR}) \rightarrow Y \mid t.\text{unlock} \rightarrow X)) \parallel \mu X \cdot (\Box_{t \in T} t.\text{lock} \rightarrow t.\text{unlock} \rightarrow X)$$

$$= \mu X \cdot (\Box_{t \in T} t.\text{lock} \rightarrow \mu Y \cdot (x : \alpha(t : \text{VAR}) \rightarrow Y \mid t.\text{unlock} \rightarrow X))$$

So far, all that we have done is to describe the interference that may be caused and that can be tolerated in the system. Now we must say how information changes or persists in the system. In fact, our multiple-user system behaves not unlike a *single-user* system: Anyone reading the contents of a data structure discovers the most recently written value. Define a function which removes *any* task name from an event:

$$\text{tstrip} \triangleq \cup_{t \in T} \text{strip}_t$$

strip_t is the function that removes the particular label t from an event [1] (see also Appendix A). tstrip then is the function that removes *any* label t from an event. Now, if we consider just the sequence of reads and writes and ignore which tasks initiated them, then our structure behaves just like a variable does. Since we do not care who initiates read or write events, we can use our forgetful function tstrip to disregard who does what:

$$\text{MVARSPEC} \triangleq \text{VARSPEC}[(\text{tstrip}^* \text{ tr}) \upharpoonright \alpha\text{VAR}/\text{tr}]$$

tstrip* tr is the trace formed by applying tstrip to each element of tr. VARSPEC is a predicate on the free variable tr, and is in terms of the events in αVAR; therefore, the definition of MVARSPEC which substitutes an expression for tr carefully restricts that expression to use events only from αVAR.

Of course, we already know how to implement a variable, and we can reuse this implementation, with a suitable relabeling of event names. Since we are unconcerned with the identity of tasks, the inverse image under tstrip of VAR will give us a promiscuous version of VAR: It doesn't care who reads its values, nor who writes new ones:

$$\text{MVAR} \triangleq \text{tstrip}^{-1} \text{ VAR}$$

Expanding this, we obtain

$$MVAR \triangleq (\Box_{t \in T}\ t.write?x \rightarrow MVAR_x)$$

where

$$MVAR_x \triangleq (\Box_{t \in T}\ t.read!x \rightarrow MVAR_x \mid \Box_{t \in T}\ t.write?y \rightarrow MVAR_y)$$

We have now completed our specification of how many users can share the data structure. We have specified that tasks may lock the data structure, and that they may only read and write while they have the lock; we have specified that while a task has the lock it has exclusive access; and we have specified that the shared data behave like a variable. Our multiple-user system must satisfy all three requirements:

$$MUSPEC \triangleq (USESPEC \wedge MUTEXSPEC \wedge MVARSPEC)$$

Our system is implemented by

$$MU \triangleq (USE \parallel MUTEX \parallel MVAR)$$

We already have a simplified version of USE ‖ MUTEX. Substituting this and our definition of MVAR into the definition of MU, we obtain

$$MU = \mu X \cdot (\Box_{t \in T}\ t.lock \rightarrow \mu Y \cdot (x : \alpha(t : VAR) \rightarrow Y \mid t.unlock \rightarrow X)) \parallel (\Box_{t \in T}\ t.write?x \rightarrow MVA\ R_x)$$

where

$$MVAR_x \triangleq (\Box_{t \in T}\ t.read!x \rightarrow MVAR_x \mid \Box_{t \in T}\ t.write?y \rightarrow MVAR_y)$$

Now, we can rewrite this to simplify it and eliminate the remaining concurrency, obtaining

$$MU = (\Box_{t \in T}\ t.lock \rightarrow (t.unlock \rightarrow MU \mid t.write?x \rightarrow MU_{t.x}))$$

where

$$MU_x \triangleq (\Box_{t \in T}\ t.lock \rightarrow MU_{t.x})$$

$$MU_{t.x} \triangleq (t.unlock \rightarrow MU_x \mid t.read!x \rightarrow MU_{t.x} \mid t.write?y \rightarrow MU_{t.y})$$

So MU is a process that allows an external choice as to which task gains the lock; only *that* task may read or write values to the data structure, until that same task yields the lock. It also ensures that a value is written to the data structure before a value can be read.

• *Queueing for busy resources*

The system described in the last section suffers from the dangers of infinite overtaking: An unlucky task wanting a lock may *always* be unsuccessful and be continually preempted by faster tasks. We shall try to solve this by serving requests for locks in order. We introduce a new event: request. Let

$$REQ1 \triangleq (tr{\downarrow}request - tr{\downarrow}lock = 1)$$

$$\overline{REQ1} \triangleq (tr{\downarrow}request - tr{\downarrow}lock = 0)$$

$$REQ2 \triangleq (tr{\downarrow}request - tr{\downarrow}unlock = 1)$$

$$\overline{REQ2} \triangleq (tr{\downarrow}request - tr{\downarrow}unlock = 0)$$

Each task may have at most one outstanding request:

$$\alpha REQUEST1 \triangleq \{request, lock\}$$

$$REQUEST1SPEC \triangleq (REQ1\ V\ \overline{REQ1})$$

$$\alpha REQUEST2 \triangleq \{request, unlock\}$$

$$REQUEST2SPEC \triangleq (REQ2\ V\ \overline{REQ2})$$

These specifications should by now be quite familiar; they have the implementations

$$REQUEST1 \triangleq \mu X \cdot (request \rightarrow lock \rightarrow X)$$

$$REQUEST2 \triangleq \mu X \cdot (request \rightarrow unlock \rightarrow X)$$

We need to say how these requests get serviced. Define, for each task t and event e, a projection function which tells us which task initiated an event:

$$task\ t.e = t$$

Also, define the sets of all t.lock events and t.request events, for all possible t:

$$Tlock \triangleq tstrip^{-1}\ lock$$

$$Treq \triangleq tstrip^{-1}\ request$$

Our requirement is that a task obtaining a lock must be the next one deserving it; that is, the longest outstanding request should be served next:

$$QSPEC \triangleq task^*\ (tr \upharpoonright Tlock) \leq task^*\ (tr \upharpoonright Treq)$$

$tr \upharpoonright Tlock$ is the sequence of t.lock events in the trace tr. $task^*\ (tr \upharpoonright Tlock)$ is just the sequence of the names of those tasks which gained the lock. Similarly, $task^*\ (tr \upharpoonright Treq)$ is just the sequence of the names of those tasks which issued requests for the lock. QSPEC says that the sequence of names of tasks gaining the lock is a *prefix* of the sequence of names of tasks requesting the lock. It is reminiscent of the specification of a buffer: What comes out is a *prefix* of what goes in. This suggests an implementation which is similar to that of a buffer:

$$Q \triangleq Q_{\langle\rangle}$$

$$Q_{\langle\rangle} \triangleq (\Box_{r \in T}\ r.request \rightarrow Q_{\langle r\rangle})$$

$$Q_{\langle t\rangle^\frown s} \triangleq (t.lock \rightarrow Q_s \mid \Box_{r \in T}\ r.request \rightarrow Q_{\langle t\rangle^\frown s^\frown\langle r\rangle})$$

Initially, the queue of requests is empty, and the process is only willing to accept a request. When there is at least one request, the task at the front of the queue may obtain the lock, or further requests may be added to the *end* of the queue. This is where the queueing discipline is encoded.

A "fair" multiple-user system behaves like our earlier multiple-user system, allows at most one outstanding request per task, and has the queueing discipline that we have described:

$$FAIRMUSPEC \triangleq (MUSPEC \wedge QSPEC \wedge \forall t \in T \cdot t.REQUEST1SPEC \wedge t.REQUEST2SPEC)$$

$$FAIRMU \triangleq (MU \parallel Q \parallel\mid_{t \in T} (t:REQUEST1 \parallel t:REQUEST2))$$

Of course, in this section we have only been fooling ourselves: We have pushed the problem back from getting the lock to requesting one. As before, a fast task might get into the queue, acquire the lock, release it, and get into the queue again before a slower one gets its act together. Thus it is slightly misleading—in fact downright lying—to call this solution "fair." As pointed out in [2], the correct solution to this problem is probably to regard it as insoluble, because if any task is particularly determined on having so much access to a data structure, then someone—this task or another requiring access—will inevitably be disappointed. In CSP, we cannot distinguish between a task that takes an infinite amount of time to require access to a particular data structure, and one which does require access but is being discriminated against by our transaction processing system. It seems that in our Kafkaesque world paranoia is indistinguishable from genuine persecution. However, in practical terms, we have merely decided to delegate to the implementor the responsibility of ensuring that any desired event that is possible takes place within an acceptable period of time. So we ask that the implementation ensure that requests are serviced in an even-handed way.

4. An optimistic approach

The last section dealt with a system which allows multiple users to gain mutually exclusive access to shared data by *locking*. It can handle contention for resources by allocating them on a first-come, first-served basis. In this section we consider a different strategy: Each task rather optimistically assumes that there will be no interference from other tasks, and so may go blithely about its transaction. But there must always be a day of reckoning: Upon completion of a transaction, the system examines whether, with hindsight, the case for optimism was justified or not. If indeed there has been no interference, then the transaction is committed; if interference was possible, then the offending transaction is deemed not to have occurred. Clearly, the suitability of this approach depends on the character of the individual application.

We introduce some new events: start, comnull, comread, comwrite, and fail. We shall have a different structure for our transactions than before. A transaction has a start point, and may be finalized in one of four ways: It might be the null transaction; it might be a read-only transaction; it might also have written to the data structure; or it might fail in some way. Which of the options are available to a transaction at any time will depend on what events are comprising the transaction, and the interference that the transaction might cause, or might have to tolerate.

Our specification starts in a familiar way. Let

$$Commit \triangleq \{comnull, comread, comwrite\}$$

$$Final \triangleq Commit \cup \{fail\}$$

and define

$$ST \triangleq (tr\downarrow start - tr \upharpoonright Final = 1)$$

$$\overline{ST} \triangleq (tr\downarrow start - tr \upharpoonright Final = 0)$$

We shall require that transactions have unique names: A transaction will only be started *once*:

$$\alpha UNIQUE \triangleq \{start\}$$

$$UNIQUESPEC \triangleq (tr\downarrow start \leq 1)$$

$$UNIQUE \triangleq (start \rightarrow STOP)$$

Transactions start and then they are finalized either by being committed or by failing:

$$\alpha TRANS \triangleq \{start\} \cup Final$$

$$TRANSSPEC \triangleq (ST \vee \overline{ST})$$

This specification is rather like LOCKSPEC; not surprisingly, its implementation is similar to that of LOCK:

$$TRANS \triangleq \mu X \cdot (start \rightarrow x : Final \rightarrow X)$$

Reading and writing may only be done within transactions:

$$\alpha RWTRANS \triangleq \alpha TRANS \cup \alpha VAR$$

$$RWTRANSSPEC \triangleq (\overline{tr}_0 \in \alpha VAR \Rightarrow ST)$$

This specification is again familiar: It is similar to READSPEC and to WRITESPEC. Its implementation is correspondingly straightforward:

$$RWTRANS \triangleq \mu X \cdot (start \rightarrow \mu Y \cdot (x : \alpha VAR \rightarrow Y \mid x : Final \rightarrow X))$$

A transaction must satisfy all three requirements: It may only be started once, and may only end by being committed or by failing, and reading and writing may only be carried out during transactions:

$$\mathsf{TRANSACTSPEC} \triangleq (\mathsf{UNIQUESPEC} \wedge \mathsf{TRANSSPEC} \wedge \mathsf{RWTRANSSPEC})$$

This is implemented as

$$\mathsf{TRANSACT} \triangleq (\mathsf{UNIQUE} \parallel \mathsf{TRANS} \parallel \mathsf{RWTRANS})$$

Simplifying this, we obtain

$$\mathsf{TRANSACT} = (\mathsf{start} \rightarrow \mu X \cdot (x : \alpha\mathsf{VAR} \rightarrow X \mid x : \mathsf{Final} \rightarrow \mathsf{STOP}))$$

This shows quite clearly that a transaction can only occur once, is either committed or fails, and that reading and writing are only permitted during the transaction.

The three commit events for a particular transaction t are each labeled by t taken from T, which we now regard as the set of *transaction names*:

$$\mathsf{Commit}_t \triangleq \mathsf{strip}_t^{-1}\ \mathsf{Commit}$$

Let committed s denote the set of names of *successfully completed* transactions in some trace s:

$$\mathsf{committed}\ s \triangleq \{t \in T \mid s \upharpoonright \mathsf{Commit}_t \neq \langle\rangle\}$$

Of interest at the start of each transaction is the most recently committed value—if it exists. The sequence of write events made by successfully committed transactions in a trace s is

$$\mathsf{succwrite}\ s \triangleq s \upharpoonright \{t.\mathsf{write}.c \mid t \in \mathsf{committed}\ s \wedge c \in C\}$$

If this is not empty, then lastwrite s is its last element, where

$$\mathsf{lastwrite}\ s \triangleq \overline{\mathsf{succwrites}\ s_0}$$

The view that each transaction has of the shared data structure simply consists of the lastwritten value—if it exists—followed by the reads and writes of the transaction itself. From each of these viewpoints the data structure appears as though it were a variable, possibly with an initial value. If we have

$$W_t \triangleq \mathsf{strip}_t^{-1}\ W$$

then the requirement is

$$\mathsf{OVARSPEC} \triangleq \forall t \in T, c \in C \cdot \overline{tr}_0 = t.\mathsf{read}.c \Rightarrow (tr \upharpoonright W_t = \langle\rangle \wedge \exists u \in T \cdot \mathsf{lastwrite}\ tr = u.\mathsf{write}.c) \vee (tr \upharpoonright W_t \neq \langle\rangle \wedge \overline{tr \upharpoonright W_{t_0}} = t.\mathsf{write}.c)$$

This should be reminiscent of the specification of a variable, but with a few extra bits and pieces. If a transaction t reads the value c from the data structure, then one of two cases must hold:

1. Transaction t has not previously written a value, in which case c is equal to the last successfully committed written value.
2. Transaction t has written a value, in which case the last value was also c.

This is implemented by a process that maintains a state containing the last successfully committed value, and the last written value for each transaction,

$$\mathsf{OVAR} \triangleq \mathsf{OVAR}(\bot, \{\})$$

$$\mathsf{OVAR}(v, f) \triangleq (\Box_{t \in T}\ t.\mathsf{start} \rightarrow \mathsf{OVAR}(v, f \oplus \{t \mapsto v\}) \mid \Box_{t \in T \mid (ft) \neq \bot}\ t.\mathsf{read}!(f\ t) \rightarrow \mathsf{OVAR}(v, f) \mid \Box_{t \in T}\ t.\mathsf{write}?x \rightarrow \mathsf{OVAR}(v, f \oplus \{t \mapsto x\}) \mid \Box_{t \in T}\ x : \mathsf{Commit}_t \rightarrow \mathsf{OVAR}((f\ t), f))$$

This *optimistic variable* OVAR initially behaves like OVAR(⊥, {}), for some distinguished value ⊥. The second definition describes the behavior of OVAR (v, f) for some value of the shared data structure $v \in C$, and some function $f : T \rightarrow C$. When a transaction t starts, the function f is updated with the maplet $\{t \mapsto v\}$. Values may be read or written by transaction t; these are operations on t's copy of the data structure in the mapping f. However, OVAR never engages in the event t.read.⊥, for any t. Finally, when transaction t is successfully committed, the shared value of the data structure is updated with the final value computed by t.

We can make the intuitive link between OVAR and VAR precise by being more explicit about the "view" that each transaction has of the shared data structure. If

$$\mathsf{initial}_t \triangleq \begin{cases} \mathsf{lastwrite}\ (tr\ \underline{\mathsf{before}}\ t.\mathsf{start}) & \text{if succwrites } tr \neq \langle\rangle \\ & \text{and } tr \upharpoonright W_t = \langle\rangle \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$\mathsf{view}_t \triangleq \mathsf{initial}_t\ \hat{}\ (tr \upharpoonright \alpha(t : \mathsf{VAR}))$$

then we can prove that

$$\mathsf{OVARSPEC} \Rightarrow \forall t \in T \cdot \mathsf{VARSPEC}[\mathsf{tstrip}^*\ \mathsf{view}_t/tr]$$

That is, each view of the shared data structure reveals it to be just like a variable—no interference, no nasty surprises.

Now consider the various commit events. comnull corresponds to finalizing the null transaction, so, if a transaction says that it made no access to a data structure, then this must be the case:

$$\alpha\mathsf{NULL} \triangleq \{\mathsf{comnull}\} \cup \alpha\mathsf{VAR}$$

$$\mathsf{NULLSPEC} \triangleq (\overline{tr}_0 = \mathsf{comnull} \Rightarrow tr \upharpoonright \alpha\mathsf{VAR} = \langle\rangle)$$

Reading or writing *disables* the comnull event:

$$\mathsf{NULL} \triangleq \mu X \cdot (\mathsf{comnull} \rightarrow X \mid x : \alpha\mathsf{VAR} \rightarrow \mathsf{STOP}_{\{|\mathsf{comnull}|\}} \parallel \mathsf{RUN}_{\alpha\mathsf{VAR}})$$

A transaction finalized with a comread event must have read something:

$$\alpha CR1 \triangleq \{comread\} \cup R$$

$$CR1SPEC \triangleq (\overline{tr}_0 = comread \Rightarrow tr \upharpoonright R \neq \langle\rangle)$$

but not written anything:

$$\alpha CR2 \triangleq \{comread\} \cup W$$

$$CR2SPEC \triangleq (\overline{tr}_0 = comread \Rightarrow tr \upharpoonright W = \langle\rangle)$$

So reading *enables* the comread event:

$$CR1 \triangleq (x : R \rightarrow RUN_{|comread|\cup R})$$

and writing *disables* it:

$$CR2 \triangleq \mu X \cdot (comread \rightarrow X \mid x : W \rightarrow STOP_{|comread|} \parallel RUN_W)$$

Putting these two together, we get

$$(CR1 \parallel CR2) = (x : R \rightarrow \mu X \cdot (x : R \rightarrow X \mid comread \rightarrow X \mid x : W \rightarrow RUN_{\alpha VAR}) \mid x : W \rightarrow RUN_{\alpha VAR})$$

If a transaction says that it has written to the data structure, then it must not be lying:

$$\alpha CW1 \triangleq \{comwrite\} \cup W$$

$$CW1SPEC \triangleq (\overline{tr}_0 = comwrite \Rightarrow tr \upharpoonright W \neq \langle\rangle)$$

Writing *enables* the comwrite event:

$$CW1 \triangleq (x : W \rightarrow RUN_{|comwrite|\cup W})$$

Adding this to (CR1 ‖ CR2), we obtain

$$(CR1 \parallel CR2 \parallel CW1) = (x : R \rightarrow \mu X \cdot (x : R \rightarrow X \mid comread \rightarrow X \mid x : W \rightarrow RUN_{|comwrite|\cup\alpha VAR}) \mid x : W \rightarrow RUN_{|comwrite|\cup\alpha VAR})$$

If we now add to this the process NULL, we get a description of how processes may be finalized:

$$FINAL \triangleq (NULL \parallel CR1 \parallel CR2 \parallel CW1)$$

$$= \mu X \cdot (comnull \rightarrow X \mid x : R \rightarrow \mu Y \cdot (x : R \rightarrow Y \mid comread \rightarrow Y \mid x : W \rightarrow RUN_{|comwrite|\cup\alpha VAR}) \mid x : W \rightarrow RUN_{|comwrite|\cup\alpha VAR})$$

A transaction t cannot be finalized with a t.comread or t.comwrite event if there has been an update of the data structure during t's lifetime. The simplest way of ensuring this is to say that no other transaction can have been finalized with a comwrite since t started. No interference has been caused to t by u if

$$\alpha NOINT_{t,u} \triangleq \{t.start, t.comread, t.comwrite, u.comwrite\}$$

$$NOINT_{t,u}SPEC \triangleq (\overline{tr}_0 \in \{t.comread, t.comwrite\} \Rightarrow \overline{tr}'_0 = t.start)$$

The implementation of this requirement must ensure that u.comwrite *disables* t.comread and t.comwrite events:

$$NOINT_{t,u} \triangleq \mu X \cdot (t.start \rightarrow (x : \{t.comread, t.comwrite\} \rightarrow u.comwrite \rightarrow STOP \mid u.comwrite \rightarrow STOP) \mid u.comwrite \rightarrow t.start \rightarrow x : \{t.comread, t.comwrite\} \rightarrow STOP)$$

We have now completed the description of the optimistic transaction processing primitives. Our full specification is:

$$OPTSPEC \triangleq (OVARSPEC \wedge \forall t \in T \cdot t.TRANSACTSPEC \wedge t.FINALSPEC \wedge \forall u \in T \cdot u \neq t \Rightarrow NOINT_{t,u}SPEC)$$

That is, the shared data behave like an optimistic variable, reading and writing can only be done within transactions which have unique names, transactions must be finalized in the manner described, and the success of a transaction depends on the interference which has been caused or which can be tolerated. The implementation puts together the components we have developed:

$$OPT \triangleq OVAR \parallel |||_{t \in T} (t : TRANSACT \parallel t : FINAL \parallel |||_{u \in T \setminus \{t\}} NOINT_{t,u})$$

The generally accepted correctness criterion for maintaining the consistency of a database is called *serializability* [4]. A sequence of atomic reads and writes is called *serializable* essentially if its overall effect is as though the users took turns, in some order, each executing their entire transaction indivisibly. The reader may be wondering how the optimistic transaction processing described above relates to this notion of serializability.

Define the function f_s for each trace s which, when applied to transaction t, returns the sequence of reads and writes performed by t in s:

$$f_s\, t \triangleq s \upharpoonright \alpha(t : VAR)$$

Clearly, f_s t is t's entire transaction in s. Now define the function success which, when applied to a trace s, returns

the sequence of names of successfully committed transactions

$$\mathsf{success}\ s \mathrel{\hat{=}} \mathsf{trans}^*\ (s \upharpoonright \bigcup_{t \in T} \mathsf{Commit}_t)$$

where trans merely projects the transaction name from an event

$$\mathsf{trans}\ t.e = t$$

Given a trace of OPT, tr, we can find the sequence of entire transactions in the order of their successful commitment as follows:

$$\mathsf{serial}\ tr \mathrel{\hat{=}} {}^\frown/(f^*_{tr}\ (\mathsf{success}\ tr))$$

If tr is a trace of our optimistic transaction processing system, then tr and serial tr have the same effect. The proof of this fact follows from each transaction's view of the shared data and the freedom from interference that each successfully committed transaction enjoys.

5. Discussion

The transaction processing primitives that we have presented in this paper offer particular interfaces to the user. Those which involve locking items of data are inspired by a very successful, but fairly primitive, kind of system with which we are familiar. Here we are not providing a *robust* interface: The system can suffer certain deadlocks if users do not obey the protocol required to use the shared data. A mischievous user can deadlock the entire system by progressively gaining all the locks and refusing to yield them. Presumably everyone then dies of boredom. A careless user can obtain the same result by gaining all the locks and then becoming livelocked and not getting around to yielding the locks. Pairs of users can deadlock each other by each waiting for a lock owned by the other. However, there are well-known techniques that cooperative users can employ to get round these problems, so we do not pursue the matter further (but see, for example, [5]).

The optimistic transaction processing system should be able to avoid these tiresome outcomes: Transactions need not wait upon other transactions to finish before they can start. Of course, users should be warned that the possibility of deadlock has been traded for the possibility of starvation.

This paper documents a case study in the practical application of a mathematically precise notation—CSP—to an interesting problem: that of transaction processing. The usefulness of case studies can hardly be overemphasized: They help to establish confidence in the practicality of the notation and ideas, especially when applied to realistic, industrial-scale problems; they help to explore the areas of application of CSP; they help to establish a convenient style for the use of CSP; and they provide information and motivation for further research.

This case study does indeed show that CSP is a practical tool. However, as with other formal methods that have been introduced into industry, such as Z [6, 7] or VDM [8], education is essential before any degree of fluency in using CSP is achieved, or even before a paper such as this may be read. The use of CSP allows a designer the opportunity to specify systems in a concise fashion. For example, the optimistic transaction processing system has a very short and simple specification, even though it is a lot more sophisticated than the other systems considered, as is borne out by its design and implementation.

The style adopted in this paper seems quite successful: Specify each requirement separately, in the simplest context that seems appropriate; implement each requirement as a simple process; form the specification from the conjunction of requirements, and the implementation from the parallel combination of the processes. The development of two complementary descriptions—a predicate and a piece of process algebra—helped us to understand what we were describing much better than a single description would have done. Our confidence was bolstered by performing the usually simple proof that the process was indeed an implementation of the specification: that the two descriptions were of the same thing.

Many of the specifications and implementations in the systems that we have presented in this paper are really the same predicates and processes in different guises. We could obtain an economy of expression by the widespread use of relabeling functions, but it is felt that this often leads to rather obscure descriptions. The first reaction of the reader is often to try to do all the substitutions in his head, to see what the definition really means. So we have limited such relabeling to situations where it is easy to see what is going on. For example, in promoting a property of a process to being a property of a labeled process, for any label in some set, relabeling is a powerful technique which actually makes it easier to understand the system. Drawing a rather tenuous link between disparate system properties, on the other hand, seems to obscure the issues. The insight about the connection is more valuable as a way of reducing the burden of proof than as a way of making the description more comprehensible. We still get the economy of an easy implementation and its proof, the strategy being merely to exhibit a relabeling scheme to establish the connection with an existing satisfaction proof.

The style of writing the predicate as a firing condition for an event was also helpful. Often, rather complicated predicates—with plenty of existential quantifiers—which we *thought* captured a requirement were replaced by several predicates describing firing conditions which matched our intuition for the problem. Also, such simple predicates often have really very simple implementations, and some pleasing patterns have emerged in this and other case studies: conjunctions of firing conditions as parallel processes, as usual; disjunctions of firing conditions with disjoint alphabets as interleaved processes; disjunctions of firing

conditions with overlapping alphabets as parallel processes with certain new internal events; and simple processes describing predicates in which events enable or disable other events.

It would be a fairly straightforward matter to translate the CSP implementations of the systems that we have described to occam [3]. This would be a good idea because occam has direct language support for many of the concepts of CSP: It was designed with this in mind. It is also a simple language with a relatively simple semantics; a proof of the translation would not be too difficult. For many reasons occam is not *yet* everyone's first choice for the implementation of concurrent systems. Companies have in-house standards: They support some languages and not others; they have concerns of compatibility, and of running systems on a large variety of different computers. In a companion paper, we shall address ourselves to the problems of implementing CSP descriptions in low-level languages with only meager synchronization facilities. The idea is not to ape the synchronization mechanisms that may be found in occam, but rather to find a semantics in CSP for whatever synchronization mechanism happens to be available in the chosen language. An equivalence can then be demonstrated between the CSP implementation and the actual program. More case studies are required to demonstrate that this is a *practical* technique that can be accomplished in a development laboratory.

One of our declared aims is to combine CSP with Z in some appropriate way; as a first step toward this, we can imagine the style that we have used in this paper extending to a development method incorporating both notations and ideas (cf. [9]):

- Specify the system as a conjunction of simple predicates over traces. Perhaps make design steps to get a suitable description.
- Implement the system as a highly distributed collection of communicating processes.
- Transform the system using the laws of CSP until the required degree of concurrency is obtained. At this stage the system is described as a collection of parallel processes with implicit state.
- Transform the system to make all the state explicit. The state can be described using a notation such as Z [6, 7]. The result is a system described as a collection of parallel state-based processes.
- Since the descriptions of states have been derived from the structure of processes, it will probably be necessary to refine the data structures.
- An implementation in a programming language should now be straightforward.

In this paper we have omitted all the proofs that we conducted in the development of each system. There are three sorts of proof that we have found: proofs of theorems about predicates over traces; proofs that processes satisfy their specifications; and proofs of equivalence between processes—process transformations. None of the proofs that we have carried out seem particularly difficult; however, they are often long and tedious, and we have made many a slip. Now that we understand how each proof may be made, we would like to check it with mechanical assistance, and we propose to conduct some research in this area. *Appropriate* mechanical assistance will have a large impact on the acceptance of a notation such as CSP in industry; we must get it right.

Acknowledgments

This work has been carried out under a contract with IBM United Kingdom Laboratories, Hursley Park, Winchester, to whom we are most grateful for their continuing support and interest. The problem of describing these kinds of transaction processing systems in the notations of CSP was suggested by Peter Lupton, who also made some very helpful comments on an earlier draft of this paper, as did Geoff Barrett, Jeremy Jacob, and Steve King. The inclusion of a potentially more difficult system—using an optimistic strategy—was suggested by reading a description of the Amoeba file service written in Z [10]. Paul Gardiner provided many important insights into this and other problems. Some elegant solutions to problems in transaction processing—developed independently and entirely in CSP's process algebra, without trace specifications—may be found in [11]. The referees gave some extremely useful comments on an earlier draft of the paper. Finally, thanks—as usual—to Jock McDoowi.

Appendix A: Glossary of symbols

This glossary of symbols was taken from [2], except that we have included substitution for free variables in predicates, and we do not require relabeling functions to be injections, but find the definition given in [3] to be more convenient.

Definitions

Notation	*Meaning*	*Example*
$\hat{=}$	is equal to by definition	$R \mathrel{\hat{=}} \{read.c \mid c \in C\}$

Predicates

Notation	*Meaning*	*Example*
$=$	equals	$x = x$
$\neq$	is distinct from	$x \neq x + 1$
$P \wedge Q$	P and Q	$x \leq x + 1 \wedge x \neq x + 1$
$P \vee Q$	P or Q	$x \leq y \vee y \leq x$
$\neg P$	not P	$\neg 3 > 5$
$P \Rightarrow Q$	P implies Q	$x < y \Rightarrow x \leq y$
$P \equiv Q$	P if and only if Q	$x < y \equiv y > x$

$\exists x \in A \cdot P$	there exists an x in set A such that P
$\forall x \in A \cdot P$	for all x in set A, P
$P[a/b]$	P with a substituted for b $(x < 9)[3/x] \equiv (3 < 9)$

Sets

Notation	*Meaning*	*Example*
$\in$	is a member of	$2 \in \{1, 2, 3\}$
$\notin$	is a member of	$4 \notin \{1, 2, 3\}$
$\{a\}$	the singleton set containing a	{start}
$\{a, b, c\}$	the set with members a, b, and c	{request,lock,unlock}
$\{x \mid P\ x\}$	the set of all x such that P x	$\{read.c \mid c \in C\}$
$A \cup B$	A union B	$\{1\} \cup \{2, 3\} = \{1, 2, 3\}$
$A \setminus B$	A minus B	$\{1, 2, 3\} \setminus \{2\} = \{1, 3\}$
$\cup_{i \in I} S_i$	the union of a family of sets	

Functions

Notation	*Meaning*	*Example*
f x	function application, f of x	succ tr
$strip_l$	the function which removes the label l	$strip_t$ t.lock = lock
$strip_l^{-1}$	the function which adds the label l	$strip_t^{-1}$request = t.request
f^{-1} S	the inverse image under f of S	$strip_t^{-1}$ R = $\{t.read.c \mid c \in C\}$
$a \mapsto 1$	a maps to 1	$f \triangleq \{a \mapsto 1, b \mapsto 2\}$
$f \oplus g$	function override	$f \oplus \{a \mapsto 3\}$

Traces

Notation	*Meaning*	*Example*
$\langle\rangle$	the empty trace	
$\langle a \rangle$	the trace containing only a	$\langle$t.commit$\rangle$
$\frown$	one trace followed by another	$\langle t \rangle \frown s$
$\frown/$	distributed catenation	$\frown/\langle\langle a \rangle, \langle b, c \rangle\rangle = \langle a, b, c \rangle$
$s \upharpoonright A$	s restricted to A	tr $\upharpoonright$ W
$s \leq t$	s is a prefix of t	$\langle a, b \rangle \leq \langle a, b, c \rangle$
s in t	s is in t	$\langle b, c \rangle$ in $\langle a, b, c, d \rangle$
$s \downarrow a$	the number of a's in s	$\langle a, a, b, a, c \rangle \downarrow a = 3$
s_0	the head of s	$\langle a, b, c \rangle_0 = a$
s'	the tail of s	$\langle a, b, c \rangle' = \langle b, c \rangle$
$\bar{s}$	the reverse of s	$\overline{\langle a, b, c \rangle} = \langle c, b, a \rangle$
$\bar{s}_0$	the last element of s	$\overline{\langle a, b, c \rangle}_0 = c$
$\bar{s}_0'$	the penultimate element of s	$\langle a, b, c \rangle_0' = b$
f* s	f applied to every element of s	$f^* \langle a, b, c \rangle = \langle fa, fb, fc \rangle$

Events

Notation	*Meaning*	*Example*
l.a	participation in event a by process named l	t.lock
c.v	communication of value v on channel c	read.b
l.c.v	communication of value v on channel l.c	t.read.b

Processes

Notation	*Meaning*
αP	the alphabet of process P
$(a \rightarrow P)$	a then P
$(a \rightarrow P \mid b \rightarrow Q)$	a then P choice b then Q
$(x : A \rightarrow P\ x)$	choose x from A then P x
$\mu X \cdot F\ X$	the process X which satisfies X = F X
P $\parallel$ Q	P in parallel with Q
l : P	P with name l
P $\Box$ Q	P choice Q
b!e	on channel b output the value of e
b?x	from channel b input to x
f^{-1} P	the inverse image under f of the process P
tr	an arbitrary trace of the specified process
ref	an arbitrary refusal of the specified process
P **sat** S	process P satisfies specification S

Appendix B: Additional notation

The notations used in this paper are all drawn from [1, 2], with the following exceptions, which either are derived or are notational conveniences.

Given a sequence of events s containing an event e, then

s before e

is the largest prefix of s not containing e. That is,

$\neg(\langle e \rangle$ in (s before e))

and

(s before e) $\frown \langle e \rangle \leq s \frown \langle e \rangle$

Given a predicate on traces PSPEC,

l.PSPEC

denotes a new predicate that may be satisfied by a process named by l:

l.PSPEC $\triangleq$ PSPEC[$strip_l^*$tr / tr]

In CSP we have the proof rule (taken from [2, p. 91])

if P **sat** PSPEC

then f^{-1} P **sat** PSPEC[f* tr / tr]

We can therefore derive the following proof rule:

if P **sat** PSPEC
then l : P **sat** l.PSPEC

since

$$l : P \triangleq \text{strip}_l^{-1}\ P$$

Also, since

if P **sat** S
and Q **sat** T
then (P ‖ Q) **sat** (S[tr ↾ αP / tr] ∧ T[tr ↾ αQ / tr])

we can derive

if P **sat** PSPEC
then $\|_{l \in L}\ l : P$ **sat** $\forall l \in L \cdot l.\text{PSPEC}$

References

1. C. A. R. Hoare, "Notes on Communicating Sequential Processes," *Technical Monograph No. PRG-33*, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, August 1983.
2. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, London, 1985.
3. INMOS Ltd., *occam Programming Manual*, Prentice-Hall International, London, 1984.
4. C. H. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM* **26**, No. 4, 631–653 (1979).
5. M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, London, 1982.
6. I. Hayes, Ed., *Specification Case Studies*, Prentice-Hall International, London, 1987.
7. B. A. Sufrin, "Notes for a Z Handbook," Programming Research Group, Oxford University Computing Laboratory, Oxford, England, March 1986.
8. C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall International, London, 1986.
9. M. A. Jackson, *System Development*, Prentice-Hall International, London, 1983.
10. T. Gleeson, "The Amoeba File Service," The Distributed Computing Software Project, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, September 1986.
11. M. Arcus and J. Jacob, "Flagship Synchronisation Problems in CSP," *Industrial Software Engineering Unit Report No. 1*, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, March 1986.

Received December 5, 1986; accepted for publication May 26, 1987

James C. P. Woodcock *Oxford University Computing Laboratory Programming Research Group, 8–11 Keble Road, Oxford OX1 3QD, England.* Dr. Woodcock was awarded the Ph.D. degree in 1980 for his work in computation at the University of Liverpool. He then joined the research laboratories of the General Electric Company of England, where he worked first on telecommunications software, and then founded the Formal Methods Research Group. After four years at General Electric, he took a lectureship at the University of Surrey, and soon thereafter joined the Programming Research Group at Oxford University. For the past two years, he has done research on the application of mathematically based notations for the specification and design of computer systems, in particular, Z and CSP. Throughout this time, he has been working in collaboration with IBM United Kingdom Laboratories at Hursley Park, near Winchester. He has recently been elected a Fellow of Pembroke College, Oxford.