Conjoined Events

Neil C. C. BROWN
School of Computing
University of Kent
UK
neil@twistedsquare.com

Abstract

Many existing synchronous message-passing systems support choice: engaging in one event XOR another. This paper introduces the AND operator that allows a process to engage in multiple events together (one AND one more AND another; all *conjoined*), engaging in each event only if it can atomically engage in all the conjoined events. We demonstrate using several examples that this operator supports new, more flexible models of programming. We show that the AND operator allows the behaviour of processes to be expressed in local rules rather than system-wide constructs. We give an optimised implementation of the AND operator and explore the performance effect on standard communications of supporting this new operator.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms

Keywords Multiway synchronisations, Choice, Conjoined Events

1. Introduction

Messages can be passed between concurrent processes either synchronously or asynchronously. Asynchronous systems allow a process to send a message and proceed without waiting for the receiver to accept the message, and thus necessarily involves the use of buffering. Synchronous systems require that the sender wait and synchronise with the receiver (and therefore no buffering is required). For basic communication, both asynchronous and synchronous systems can easily emulate the other (by sending acknowledgements, or introducing a buffering process, respectively).

Synchronous message-passing allows processes to interact in two senses: they pass data, and they establish a common point in time. Both pieces of information can be exploited in a system's design. Examples of synchronous systems include Concurrent ML [9] and all CSP-based [8, 10] systems (e.g. Go [1], occam-π [12], CHP [2]).

Many synchronous systems include a choice operator; processes are able to offer to engage in exactly one of a choice of communications. This operator facilitates the use of more powerful design patterns [2, 11]. The choice operator can be thought of as an "XOR" (exclusive-or) operator, allowing processes to synchronise on one event or another (but not both).

This paper introduces an "AND" operator for message-passing systems, that allows a process to engage in one event AND another (we call this conjoining the events). This is different from performing the two events in sequence or parallel: both events must happen, or neither. They are dynamically (and temporarily) fused into behaving like a single event. This enables the use of new design patterns.

In section 2 we explore uses of this new operator, and in section 3 we explore some of its properties. In section 4 we explain an algorithm for its implementation, built on top of Software Transactional Memory (STM), and in section 5 we benchmark this implementation against comparable message-passing systems to examine the cost of supporting conjunction.

2. Examples

To motivate the implementation of this new conjunction feature, we give several examples here of how it can be used.

2.1 Moving Agents

Consider a topology of sites – say, a one-dimensional line of sites – with a pair of communication channels (one in each direction) connecting adjacent sites. Each site may either be empty or contain a single agent. Each agent may move to an adjacent site within a time-step.

One way to implement this is to make the site behave as follows in each time-step:

- if it is empty, it offers an exclusive choice between receiving an agent from each adjacent site, whereas
- if it is full, it offers an exclusive choice between sending its agent on an outward channel (to any of the sites to which the agent is willing to move).

This simple implementation has two problems in the case where a site is full. Firstly, it prohibits an agent moving into a site on the same time-step as an agent moves out (the "slow" problem); this may not be a property we want. Secondly, if two sites adjacent to each other both try to send an agent to each other, neither will progress (the "blocking" problem). In a one-dimensional line of sites, the blocking problem is particularly problematic and can easily lead to deadlock.

An erroneous correction to this implementation would be to offer (when the site is full) to send the agent out in parallel with offering to receive a new agent. This could result in receiving a new agent without sending on the old agent; this would leave two agents in a site, breaking our design.

A correct solution involves the use of conjunction. When a site is full and has an agent wanting to move left, it offers to either:

AMP '10 June 6, Toronto, Canada

(send the agent left) or (send the agent left *and* receive a new agent from the left). This way, if the site to the left is empty, the agent will be sent. If the site to the left is full and its agent wants to move right into our site, the agents are swapped. This solves our blocking problem. If we also want to solve the slow problem, we offer to either: (send the agent left) or (send the agent left *and* receive a new agent from the left) or (send the agent left *and* receive a new agent from the right). This solution generalises to any number of adjacency connections (and thus to any topology, such as two- and three-dimensional regular grids). It allows the problem to be solved using a different local design rather than a system-wide solution involving extra events.

2.2 Platelet Pipeline

We consider a problem similar to that posed by Schneider et al. [11]. Platelets move (in a consistent direction) along a one-dimensional pipeline. On each time-step a platelet may move or not move, with the following rule: if there are platelets immediately before or immediately after it in the pipeline, it will only move forwards if they do so too. We also add the feature that each platelet will refuse to move on a time-step with probability p (typically 5%).

The CSP algorithm [11] requires knowledge of the platelets from more than the adjacent sites, and multiway synchronisations; a later solution in the occam-π language used a higher-level clot process to group together the movement decisions of adjacent platelets [13]. We can use conjunction to implement a solution wherein each platelet only requires (two-party) synchronisations with its immediate neighbours, and an optional global synchronisation.

2.2.1 Ticking

One methodology for designing concurrent simulations is the tick pattern; in each time-step, each process offers the choice between some actions, and a lower-priority tick event. If the actions are chosen, a further choice is offered between any remaining actions and the tick event (and so on). All the actions that can occur, will – followed by the tick event that signifies the end of this time-step (and thus the beginning of the next).

In our platelet simulation, each site may communicate two different things to each neighbour: a platelet (signalling a move) or an empty signal. An empty site begins by offering:

- to read a signal from the site behind it in the pipeline, *or*
- to send an empty signal to the site ahead of it, *or*
- to synchronise on the tick event.

If either of the former two happens, the remaining two of three are offered again, and so on, so that the process may do the first none or once, the second none or once, until the third happens – at which point it becomes full if it read a platelet from the site behind it (i.e. if it did perform the first event), and remains empty otherwise.

A full site generates a random probability for the time-step; with a 5% chance it simply waits for the tick event. In the other 95% of cases it offers a choice:

- to read a signal from the site behind it in the pipeline *and* send the current platelet forward in the pipeline, *or*
- to synchronise on the tick event.

If the former happens, the process subsequently commits to the latter. Once the tick event has happened, the process is empty if it read an empty signal from the site behind it, and remains full otherwise.

The pipeline can be terminated with a site that repeatedly reads in any signal, and started with a site that offers signals according to some pattern of platelet generation.

From these simple rules for behaviour we get emergent clotting behaviour; no site that is full can move unless its neighbours are empty (and thus willing to engage in events with it) or are full and willing to move – but not if they are unwilling to move. This becomes transitive; no clot (contiguous group of full sites) can move unless all are willing to move. This would not be possible so simply (and with entirely local rules such as these) without conjunction.

It is also possible to construct a solution that does not use a tick event, nor priority [3]. This is done by introducing two more events between processes beyond empty and move: a can-stay event and a must-stay event.

Implementing this system with a concurrent process per site may be inefficient due to the large number of synchronisations that would be required. It is possible to instead construct a hybrid approach that simulates a group of many contiguous sites sequentially, and uses the above design/protocol at the edges of each group to communicate between the concurrent processes that simulate entire groups of sites.

2.3 Hidden Process

Conjunction can also be useful in other situations, to "hide" processes. When two processes communicate with each other, they also synchronise. Placing a process in the middle of the two that forwards the messages introduces buffering, which may be unwanted. An added problem is that if the buffer process offers to read in a value, it is continually available – whereas the original receiver may not be, thus altering the synchronisation behaviour; if the writer only intends to write when the reader is available (and choose something else otherwise), its choice will be affected.

The solution to this problem of "hiding" a process is surprisingly complicated, and involves the use of conjunction and another feature: extended communications. An extended communication involves making the second party in a communication wait while the first performs an action. An extended write means that once both parties have agreed to synchronise, the writer performs an action before sending the value (typically, something that involves calculating the value). An extended read means that once the data has been passed, the reader can perform an additional action (typically a synchronisation) before the writer is freed. The other party in an extended transaction does not need to know (at the program level) that the other party has performed an extended communication, and both sides may extend the communication.

A process may be hidden as follows. It creates an internal communication channel. It then performs the conjunction of an extended input on its external input channel, and an extended output on its external output channel. This means that it maintains the synchronisation behaviour, because it fuses together its input and output; they behave as if they were the same event (i.e. as if the process was not present at all).

During the extended input phase of the input (i.e. while the outside writer is waiting), the value read from the external input is sent on the internal communication channel. During the extended output phase of the output (i.e. while the outside reader is waiting for a value), the value is read from the internal communication channel, and is used as the value for the external outward communication. Thus, the value received is passed across and communicated on the next channel. This implements a hidden identity process that does not modify the value; any processing on the value should take place during one of these two extended phases, or by having a more complex internal structure than a simple forwarding channel.

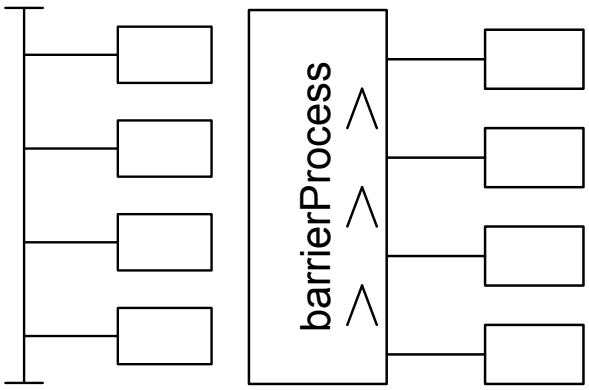

Figure 1. The duality between N processes enrolled on an N-party event (left), and N processes synchronising on 2-party events with a single "barrier process" that synchronises on the conjunction of all of them (right).

2.4 Event Duality

In addition to two-party synchronisations (as channel communications often are), many frameworks also support N-party synchronisations. There is a duality between N processes synchronising on one item together, and N processes each synchronising on a 2-party event with a further process that conjoins them all (this is illustrated in figure 1). This means that any system with only two-party synchronisations and conjunction can emulate N-party synchronisations.

Furthermore, it is possible to form a similar correspondence for partial events. Partial events are events where any X processes of N may synchronise together (where $X < N$; if $X = N$ this is a full event). To implement a partial event, the extra process must offer a choice between all subsets of size X of the N two-party events that it has been given. It is possible to further constrain the subsets to treat certain parties differently (e.g. the process could offer to synchronise with one of two special master processes and any three of ten worker processes).

These correspondences are useful in theory, and can be useful for thinking about expressive power; however, it is likely that the systems involving conjunctions of two-party events will be slower than implementing the full or partial events directly.

A hybrid approach is interesting – consider 201 processes wanting to synchronise on a barrier together. They could all enroll on one barrier and synchronise together – all 201 would contend for the same barrier. Consider instead two barriers – 100 processes would solely enroll on barrier and 100 solely on the other. One process would enroll on both and would conjoin the two. This has the same behaviour as the original barrier that was twice as large as these two new barriers, but (using the implementation described in this paper) the participants would contend on only their one barrier until it was ready, and only then would they examine the other barrier. Thus conjunction could be used to reduce the contention on large barriers.

3. Properties, Terminology and Notation

We use a notation borrowed from boolean logic, with $a \oplus b$ meaning that a process waits for either a or b but will only perform one of them, and $a \wedge b$ means that a process waits for both a and b. The operators are associative and commutative. The unit of $\oplus$ is the never-ready event STOP, and the unit of $\wedge$ is the always-ready event SKIP. Conversely, STOP is a zero of $\wedge$.

a	b	c	$a \wedge (b \oplus c)$	$(a \wedge b) \oplus (a \wedge c)$
✗	✗	✗	None	None
✗	✗	✓	None	None
✗	✓	✗	None	None
✗	✓	✓	None	None
✓	✗	✗	None	None
✓	✗	✓	$\{a, c\}$	$\{a, c\}$
✓	✓	✗	$\{a, b\}$	$\{a, b\}$
✓	✓	✓	$\{a, b\}$ or $\{a, c\}$	$\{a, b\}$ or $\{a, c\}$

Figure 2. The event table showing the equivalence $a \wedge (b \oplus c) \equiv (a \wedge b) \oplus (a \wedge c)$. The event table is akin to a boolean logic truth table; each event is either ready (indicated by ✓) or not ready (indicated by ✗), and the possible resolutions (sets of events that could complete together) are indicated for each compound choice.

a	b	c	$a \oplus (b \wedge c)$	$(a \oplus b) \wedge (a \oplus c)$
✗	✗	✗	None	None
✗	✗	✓	None	None
✗	✓	✗	None	None
✗	✓	✓	$\{b, c\}$	$\{b, c\}$
✓	✗	✗	$\{a\}$	$\{a\}$
✓	✗	✓	$\{a\}$	$\{a\}$ or $\{a, c\}$
✓	✓	✗	$\{a\}$	$\{a\}$ or $\{a, b\}$
✓	✓	✓	$\{a\}$ or $\{b, c\}$	$\{a\}$ or $\{a, b\}$ or $\{a, c\}$ or $\{b, c\}$

Figure 3. The event table showing that $a \oplus (b \wedge c)$ will not behave the same as $(a \oplus b) \wedge (a \oplus c)$. It can be seen that the right-hand formula admits more possible resolutions than the left-hand formula.

Conjunction does distribute over choice: $a \wedge (b \oplus c) \equiv (a \wedge b) \oplus (a \wedge c)$, as demonstrated in the event table in figure 2. However, choice does not distribute over conjunction: $a \oplus (b \wedge c)$ will not behave the same as $(a \oplus b) \wedge (a \oplus c)$. The difference is revealed in the event table in figure 3; alternatively one could say that the given semantics of the latter are indeterminate.

Our algorithms deal with a choice normal form (a choice of conjunctions). Due to the distributivity properties, it is easier to transform complex offers into this choice normal form than it would be into, say, conjunctive normal form (a conjunction of choices).

We define a synchronisation event as being a concurrency primitive that has a persistent membership count of processes that can dynamically be increased (by enrolling) or decreased (by resigning). An event with membership count N can only complete when all N members agree to synchronise on the event. Events with a static count are a special case of those with a dynamic count.

Each *offer* can be a single event or a conjunction of several events; we choose to represent an offer as a set of events, where multiple events indicate a conjunction, a singleton set is the contained single event, and the empty set is not permitted.

A process can make multiple offers, forming an *offer-set*, but will only complete exactly one of those offers. If one or more offers can complete immediately, an arbitrary choice will be made between them (see section 4.6 for discussion of priority). If no offers can currently complete, the new process must wait until another process later resolves the offers. We can envisage offer-sets, offers and events as a tripartite graph (see figure 4).

A resolution is a collection of offers (each from a distinct offer-set) such that each event that appears in any offer appears in exactly as many offers as it has enrollees.

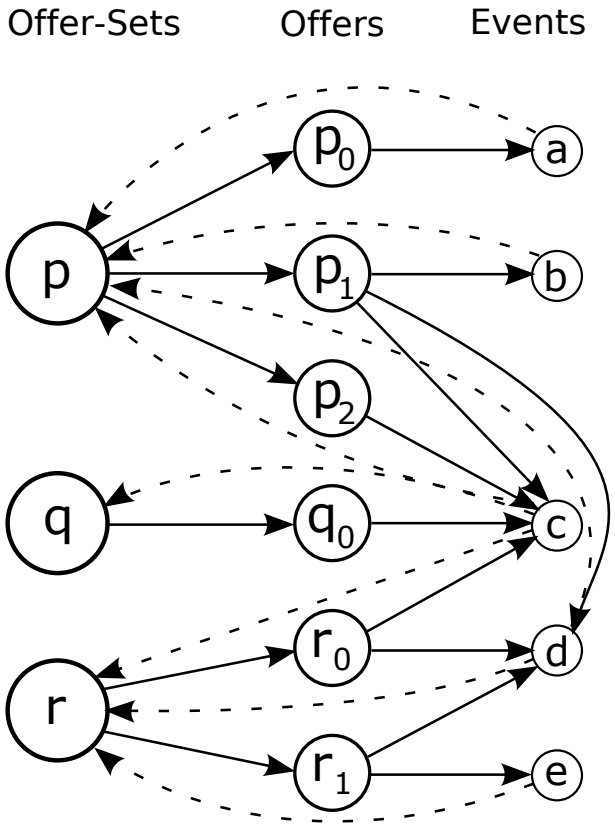

Figure 4. Tripartite graph of offer-sets (left), offers (middle) and events (right). The direction of the arrows indicates the direction of references in the system, and also the direction that the search follows. The line styles are for visual clarity and have no special meaning. Each offer is always referenced by exactly one offer-set. Events reference an offer-set if and only if the offer-set contains an offer featuring that event. Multiple edges between the same nodes are not permitted anywhere in the graph.

4. Algorithm

The algorithm for implementing conjunctive choice is at its heart a search for a solution to a constraint satisfaction problem. We have a collection of offer-sets and we need to search for a solution (a resolution) that satisfies some of the constraints. We can make use of specific knowledge about the problem to tailor an algorithm. We also have the interesting characteristic that investigating each new event incurs a time penalty for accessing it. Therefore we ideally want to locate a completion by accessing as few events as possible. By far the most common case (and the one we want to optimise for) is when each process offers one (or perhaps two to three) choices, each of which is a single two-party channel communication.

We use Software Transactional Memory (STM) [7] to provide atomicity in our algorithm, as this eases reasoning about the concurrent algorithm.

4.1 Basic Algorithm

One basic algorithm to find a resolution in the graph is as follows. Begin with the latest added offer-set. At each offer-set, choose an offer (this is a choice point). At each offer, visit all events. For each event, decrement an associated temporary count (that begins at the count of enrollees, if this is the first visit to the event). If the event was not previously visited in the search history, visit all its offer-sets that themselves have not been previously visited in the search history. At the end, if any visited event has an associated count that is not zero, back-track to the last choice-point (unwinding the history to that point) and search a different offer. If no offers remain unsearched at the last choice-point when back-tracking is required, back-track to the choice-point before that (i.e. perform a depth-first search).

In this algorithm, each visit to an event must come from a different offer-set (via some offer) because each offer-set is only traversed once, and each event can only appear in an offer once.

4.2 Actual Algorithm

The basic algorithm described above is inefficient; we may only notice that a search path is uncompletable quite late on when we discover that not enough processes have agreed to synchronise on a particular event. We can optimise this by noting that we can tell earlier if an event cannot complete: an event cannot complete if any offer-set featuring that event (transitively, via an offer) chooses an offer that does *not* feature that event. (For example, in the offer $a \oplus (b \wedge c) \oplus (a \wedge c)$, choosing the offer $b \wedge c$ prevents a from completing.) We can go so far as to prevent the offer-set choosing an event for which we are currently searching for a completion.

Our search begins with the SEARCH-NEW-OFFER-SET function, shown in figure 5. Via the SEARCH-OFFER-SET function, this tries each offer in the offer-set until one completes without backtracking.

The function for searching each offer, SEARCH-OFFER, is shown in figure 6. First, the events are all checked (lines 2–4), which affects the work queue. Once all the events have been checked, this work queue is examined using the PROCESS-WORK function. If the work queue is empty, a result can be returned based on all the offer-sets that have been visited in the search. If it is not empty, the next item from the work queue is processed using search-offer-set (with *new* set to false).

The final function is CHECK-EVENT, shown in figure 7. This begins by reading the current status of the event (line 1). An event is potentially ready to complete if it has as many offerers as enrollees, or it is part of the newly offered events and has one less offerers than enrollees. We use this knowledge to cut short our search and backtrack if neither of these conditions is met (lines 2 and 18).

We then consider all the offer-sets associated with the event (bar the one we are currently processing, which we know has chosen this event; line 3):

- If there are none to consider, the work queue is returned unchanged (line 5).
- If any have been previously visited, we backtrack (line 7). This relies on the knowledge that each event is only processed once, and the knowledge that each offer has all its events processed before a further offer-set is searched; so if we are processing this event, and one of the offer-sets has already been visited, it must not have chosen this event – which will prevent this event from ever completing and thus we backtrack.
- In the other case, the new work queue is formed from three groups (lines 9–15):
  1. *unrelated*: all the offer-sets in the work queue that are not featured in this event;
  2. *new*: all the offer-sets in the event not already in the work queue, filtered; and
  3. *filtered-old*: all the offer-sets in the event already in the work queue, filtered further.

The FILTER function removes all the offers not featuring the event (as choosing any offers later in the search that do not feature this event would be a fruitless path). If this leaves no offers remaining, we can backtrack straight away as this will only lead to

```
SEARCH-NEW-OFFER-SET(offer-set)
(1)  search-offer-set(true, offer-set, offers(offer-set), {}, {})

SEARCH-OFFER-SET(new, offer-set, offers, done, work)
(1)  case offers of
(2)  []:backtrack
(3)  [o:os]:search-offer(new, offer-set, o, done, work) ∨ search-
       offer-set(new, offer-set, os, done, work)
(4)  end case
```

Figure 5. The algorithms for searching an offer-set. search-offer-set tries the offer at the head of the list; if this backtracks, the rest of the list is processed. If the end of the list is reached, the algorithm backtracks further.

```
SEARCH-OFFER(new, offer-set, offer, done, work)
(1)  case offer of
(2)  {e} ∪ es:
(3)      work-mod ← check-event(new, offer-set, e, done, work)
(4)      search-offer(new, offer-set, es, done, work-mod)
(5)  {}:
(6)      let done-mod = done ∪ {offer-set ↦ pristine(offer)}
(7)      process-work(done-mod, work)
(8)  end case

PROCESS-WORK(done, work)
(1)  case work of
(2)  {}:return done
(3)  {(offer-set ↦ offers)} ∪ work-rest:
(4)      search-offer-set(false, offer-set, offers, done, work-rest)
(5)  end case
```

Figure 6. The algorithm for searching an offer. The full list of events is checked using check-event, and once all have been processed the not-visited map is examined; if there are no further offer-sets to visit, a result is returned based on the visited map (including the pristine version of the current offer) – otherwise the next unvisited offer-set is searched.

backtracking later when we come to process the empty offer-set in SEARCH-OFFER-SET (figure 5, line 2). We then remove the event from all the remaining offers so that we do not process any event twice.

If, after all of this search process, we are unable to find a resolution (i.e. we end up backtracking until we can backtrack no further), we instead record our new offer-set in all the events contained in that offer-set (transitively via offers) and then block on some private synchronisation object that is recorded with the offer-set. When a resolution is found, this synchronisation object will be triggered for all offer-sets involved in the resolution. Thus, this algorithm involves no polling or busy-waiting. Each process performs a single transaction featuring the above algorithm and then either blocks or has successfully found a resolution and proceeds.

4.2.1 Worked Example

We can explain a short example using figure 4. We will consider the case where p and r have already been offered, and q is the latest offer-set. The enrollment counts are: a-2, b-1, c-3, d-2 and e-2.

The newest offer-set is q, and thus this is passed to SEARCH-NEW-OFFER-SET. An offer is picked by SEARCH-OFFER-SET (q_0) and SEARCH-OFFER calls CHECK-EVENT on all the events ($\{c\}$). The length of the offer-sets in c is two (q is a new offer-set, so will not have been recorded, only p and r) against its enrollment count of three, and "offer-set-new" is true, so the block starting on line 3 of CHECK-EVENT will be entered. The "other-offer-sets" binding

```
CHECK-EVENT(offer-set-new, offer-set, event, done, work)
(1)  (enrolled-count, offer-set-list) ← read event
(2)  if (enrolled-count = length(offer-set-list) or (enrolled-count =
       length(offer-set-list) + 1 and offer-set-new)
(3)      let other-offer-sets = offer-set-list \ {offer-set}
(4)      if other-offer-sets = {}
(5)          return work
(6)      else if other-offer-sets ∩ done ≠ {}
(7)          backtrack
(8)      else
(9)          let unrelated = work \ other-offer-sets
(10)         let new = {offer-set ↦ filter(event, offers)
(11)                 | offer-set ↦ offers ∈ other-offer-sets \ work}
(12)         let filtered-old = {offer-set ↦ filter(event, offers)
(13)                 | offer-set ↦ offers ∈ work,
(14)         offer-set ∈ keys(other-offer-sets)}
(15)         return unrelated ∪ new ∪ filtered-old
(16)     end if
(17) else
(18)     backtrack
(19) end if

FILTER(event, offers)
(1)  if [] = [offer | offer ∈ offers, event ∈ offer]
(2)      backtrack
(3)  else
(4)      return [offer \ event | offer ∈ offers, event ∈ offer]
(5)  end if
```

Figure 7. The check-event function (and associated sub-function filter). Note that the filter function can be implemented as a single pass over each offer in offers. Similarly, the lines 9–15 in check-event can be implemented in a single simultaneous pass over other-offer-sets and not-visited.

will be $\{p, r\}$ and "done" is empty, so lines 9–15 will be executed. The "new"(-ly discovered) offer-sets, p and r, will be filtered down; p' will contain $p'_1 = \{b, d\}$ and $p'_2 = \{\}$, while r' will contain $r'_0 = \{d\}$.

PROCESS-WORK will then be called with "done" being $\{q \mapsto q_0\}$ and "work" being $\{p', r'\}$. PROCESS-WORK will arbitrarily pick an item from the work list – we will consider the case where p' is picked, and p'_2 is picked as the offer in SEARCH-OFFER-SET. SEARCH-OFFER will then be called with the empty offer p'_2, and will call PROCESS-WORK with "done" being $\{q \mapsto q_0, p \mapsto p_2\}$, and "work" being $\{r'\}$.

PROCESS-WORK will pick r' from the work list and SEARCH-OFFER will be called with r'_0. This will call CHECK-EVENT on the event d. d has two offer-sets associated (p and r) and an enrollment count of two, so can complete. The "other-offer-sets" binding will be p, which is non-empty – but it does intersect with the keys in "done". This will trigger a backtrack. To see why, we should look at figure 4. We are originally trying to complete offer q_0, which requires completing event c. To do so we need three offers; one each from p, q and r. Now that we are attempting to complete offer r_0 we must complete event d. This requires two offers: one from r (which is r_0) and one from p, which must be p_1 – but we have already chosen p_2 instead, so we have a contradiction (a dead-end in our search).

We therefore backtrack, to the latest choice point in SEARCH-OFFER-SET – but r' only contained r'_0, so we backtrack one further than that: to our choice of p'_2 from p. We instead choose p'_1, and thus call CHECK-EVENT on d. The event can complete, and we filter our work list (which is r' that contains $r'_0 = \{d\}$) down to r'' that contains $r''_0 = \{\}$. When we then pick r''_0 from r'', there is

nothing to be done, and our search is ultimately able to complete, with the returned value of done being $\{q \mapsto q_0, p \mapsto p_1, r \mapsto r_0\}$; events b, c and d will complete.

4.3 Scalability

One way to implement this complicated choice algorithm would be with a single central data structure with a lock. Each offering process would acquire the lock, then search for completions and either resolve successfully or record their choice and wait. The problem with this approach would be that it would cause independent processes in the system (one trying to complete event a, another trying to complete event z) to contend for the lock.

The advantage of the algorithm described above, using Software Transactional Memory, is that processes will only contend if they share an event in their offers. Processes trying to complete event a and event z will do so separately without causing any contention. Thus the algorithm can be said to scale well as more cores are added to the system; only connected processes will interact with each other, and processes that are separate in this regard can all executed simultaneously on many cores without issue.

4.4 Implementation without STM

It may be desired to implement this conjunction algorithm using basic atomic operations. This can be done by translating our use of STM through an implementation technique for STM [6] as follows.

When searching for a resolution, each read of an event is performed using an atomic read of the pointer that caches the event address and read-value in local storage. If the pointer is found to be null, the read must be retried.

Regardless of the outcome of the whole algorithm, after the search the algorithm will update the events (either to add our offer-set to the event or to clear all of them). At this point the events are claimed by performing an atomic swap (for null) on all the events in ascending memory-address order. If any results are null, the algorithm must retry the whole search (as another process must be modifying the event). Once all events are claimed, the values of the events must be compared against the cache. If all are equal, the new values can be atomically written to the pointers. If any are non-equal, the search must be retried (after the events are restored, unmodified, with atomic writes).

4.5 Partial Events

A partial event is one where only X out of N enrolled processes are required to complete the event ($X < N$). When $X \neq N$, a central assumption of our algorithm is broken. It is no longer the case that when an offer-set features an event but does not choose the offer containing that event, that event can no longer complete. So the way the offers in the work queue are pruned cannot work the same way. Additionally, our strategy that once an event is featured in one offer it must feature in all other offers that might be chosen is no longer valid.

Our existing pruning strategy can be formulated as the calculation that X participants are required for the event, and since there are N enrolled processes and for full events $X = N$, when at least one participant refuses to participate, the entire event cannot complete. For partial events, this means that when at least $N - X$ participants refuse to participate, the event can no longer occur – and in fact, if there are N' processes offering on the event ($N' \leq N$), when at least $N' - X$ participants refuse to participate, the event can no longer occur. So we would need to keep a count associated with each event of participants who had so far (on the search path) refused to engage in the event; when this count exceeds $N' - X$, we would backtrack.

SEARCH-NEW-OFFER-SET-PRI(offer-set)
(1) offers-sorted ← sort-by(offers(offer-set), λoffer → first(offer))
(2) search-offer-set(**true**, offer-set, offers-sorted, {}, {})

Figure 8. A slight modification to the SEARCH-NEW-OFFER-SET algorithm shown in figure 5 to support limited priority. The offers in the new offer-set are first sorted by an arbitrary event from the event-set; typically the head of the list or root entry in a tree.

4.6 Priority

One important feature not yet discussed is priority. Priority is useful when programming simulations: in particular to allow low-priority events (see section 2.2.1 for an example). Our discrete-time simulations invariably have the following pattern: optionally perform actions X and/or Y, until we synchronise on a global tick event with all other agents in the simulation. It is important here that the global tick is lower priority than the other actions. If it is not, nondeterminism can result because a process may tick rather than performing an action (which all other participants also wanted to perform). The intention in the design is that tick should happen only when nothing else can.

It is clear that local priorities are in general a poor solution: if one process offers a or b preferring a, and a second offers a or b preferring b, an event cannot be chosen that will satisfy both. We instead consider global priorities, where each event possesses an intrinsic constant priority. Without conjunction, this would be a simple matter, semantically: when offering a or b and both are ready, choose the event with the highest priority. With conjunction it is less clear; given the choice of completing (a and b) or (c and d), how should it be decided, given the priorities of a, b, c and d?

Note also that featuring priority makes the choice algorithm less optimal; if the priority of a resolution is decided based on all the events being completed, we must find all resolutions (which involves examining potentially many more events than finding the first resolution) and then calculate their priority before deciding on one. This is easily implementable with a small change to our current algorithm, but could have devastating consequences for the performance.

An alternative, which would require an even smaller change to the algorithm, is to guarantee that resolution A will be chosen over resolution B if all events in A have a higher priority than all of those in B: $(\forall a \in A. \forall b \in B. a > b) \implies A > B$. In any other case, an arbitrary choice will be made. This can then be implemented quite simply: by sorting the new offers by the priority of an arbitrary event (the first in the set). This will compare arbitrary events from the resolutions that would result from successfully searching this offer; this gives consistent priority if the above condition is satisfied. Note also that if no processes are using conjunction, each resolution will contain a single event, and thus this system is equivalent to global priorities on events without conjunction. An accordingly modified version of the algorithm is given in figure 8.

5. Benchmarks

Given that conjunction is a new feature, it is not possible to benchmark our implementation of conjunction against another. Instead, we give indications of the cost of supporting conjunction for standard channel communications, as this is the primary negative effect of adding conjunction to existing message-passing frameworks.

Currently, we have only implemented the conjunction algorithm in our Haskell library CHP (Communicating Haskell Processes) [2], due to Haskell's good support for Software Transactional Memory [7]. Therefore it seems appropriate to benchmark

System	Mean	95% CI
MVar	0.7188	0.7069 – 0.7319
STM	1.0392	1.0215 – 1.0610
Sync	1.9387	1.9166 – 1.9635
CML	34.2610	34.1744 – 34.3670
CHP	4.1625	4.1345 – 4.1927

Table 1. Times (means and 95% confidence intervals) for 100 pairs of writer and reader processes, communicating 100000 times (seconds, to 4 d.p.)

against other Haskell-based message-passing libraries. These include synchronous channel implementations based on Haskell's MVars and STM itself, Concurrent ML (a Haskell implementation of the ML library) and an STM implementation of synchronisations with choice (a de-centralised version of the Oracle algorithm [13]).

5.1 Methodology

All the benchmarks were carried out on the same 8-core Intel Xeon E5310 (1.60GHz) machine with 4GB RAM, running Debian GNU/Linux, kernel version "2.6.26-1-686-bigmem".

The benchmarks were timed using the Criterion benchmarking library by performing 100 iterations of each benchmark. Each benchmark recording included all startup and shutdown costs (particularly crucial in a garbage-collected language), with iterations set sufficiently high to make the fixed costs for an empty program ($\mu = 5.227$ms; 95% CI: 5.190ms, 5.287ms) immaterial.

All confidence intervals were calculated using a non-parametric (i.e. without assumption of underlying distribution) bootstrap with bias-corrected acceleration [5] using 100000 resamples.

5.2 Pairs of Communicators

To test the speed of channel communications in the various systems, we create a program with N pairs of communicators, where each pair consists of a writer and a reader, with the former repeatedly communicating to the latter (100000 times). The run-time system underlying all our systems use light-weight threads scheduled flexibly (i.e. with migration) across the different processor cores.

It can be the case that having the writer and reader on different cores, running without contention, is actually slower than having them on the same core, continually being switched out for each other. Due to the effects of scheduling and migration with small numbers of pairs (low values of N), we set N to be suitably high. Our benchmarks were carried out with $N = 100$ to avoid such problems.

The results are presented in table 1. It can be seen that the results for the implementation of conjunction in CHP are approximately a factor of four to six slower than those implementations without choice (MVar, STM) and a factor of two slower than the fastest implementation with choice (Sync).

This result shows that adding conjunction to existing systems does not introduce a major *factor* difference for basic channel communications; given that adding choice added a factor of two or three, a further factor of two for conjunction seems an acceptable parallel in exchange for the extra expressive power.

5.3 Conjunctive Pairs

To give an idea of the speed of conjunction itself, we compare it to that of standard channel communications. A benchmark similar to the one in the previous section was constructed, but each pair of communicators were communicating on a conjunction of two channels, instead of solely on one.

The results are presented in table 2. It can be seen that for CHP, communicating on the conjunction of two channels is a factor of three slower than communicating on a single channel; ideally, we might hope that this factor was less than two (the cost for the two channel communications separately). The time is around one order of magnitude worse than communicating on a single channel in the fastest systems (see table 1).

Type	Mean	95% CI
Single	4.1625	4.1345 – 4.1927
Conjunction	16.6746	16.5818 – 16.9561

Table 2. Times (means and 95% confidence intervals) for 100 pairs of writer and read processes communicating 10000 times on a pair of channels in conjunction versus a single pair in CHP (seconds, to 4 d.p.)

6. Related Work

The concept of joining multiple actions into an atomic item is present in Software Transactional Memory (STM) [7]. STM allows transactions to be executed that either happen in their entirety (without intermediate states being visible to other processes) or do not happen at all. STM transactions only read from and write to transactional variables; there is no synchronisation between processes during a transaction, so the standard API for STM has only a subset of the power of conjunction.

There has been work to build a transactional system where the primitives are synchronisation events [4]. A transaction can consist of a sequence of events, including choice between events, and the events in a transaction only take place at the end of a transaction, with all-or-nothing semantics. The transactional events work is slightly more general than conjunction, but because of this it lacks an efficient implementation (the implementation races threads against each other, with the first thread to finish killing its siblings).

7. Conclusions

This paper has introduced a conjunction operator that allows two events to be synchronised on together, so that each event can only occur with the other. These conjunctions can be combined with a standard choice to permit choice between conjunctions. The algorithm has been described, including extensions to support partial events and/or priority between events.

This paper has not explored the language/library binding for conjunction, because it is typically trivial; an AND operator or function is added that takes synchronisation events as its arguments, mirroring the design of the XOR operator in the given language/library.

We have begun some preliminary work to modify the traces model of the CSP process calculus [8, 10] to provide formal reasoning support for conjunction. A mapping to plain CSP is difficult as it requires analysis of the whole system at once, and conjoined events interact awkwardly with CSP's hiding operator. However, modifying the traces model itself is a more promising avenue of investigation.

One benefit of the CSP programming model is that it can easily be transferred to distributed systems. Channels can easily be "stretched" over the network to join two separate machines without any modification to the original program. There is no obvious way to implement conjunction in a distributed system – this is reserved for future work.

References

[1] The Go programming language. http://golang.org/. Visited March, 2010.

[2] N. C. C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, pages 67–83, Sept. 2008.

[3] N. C. C. Brown. Sticky platelet pipeline – finally tickless. http://chplib.wordpress.com/2010/01/25/sticky-platelet-pipeline-finally-tickless/. Visited March, 2010.

[4] K. Donnelly and M. Fluet. Transactional events. *SIGPLAN Not.*, 41 (9):124–135, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1160074.1159821.

[5] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1993.

[6] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1233307.1233309.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60. ACM, 2005. ISBN 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065952.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. URL http://www.usingcsp.com/.

[9] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[10] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. URL http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.

[11] S. Schneider, A. Cavalcanti, H. Treharne, and J. Woodcock. A Layered Behavioural Model of Platelets. In Michael G. Hinchey, editor, *ICECCS-2006*, pages 98–106, Stanford, California, Aug. 2006. IEEE.

[12] P. H. Welch and F. R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, 2005. ISBN 3-540-25813-2. URL http://www.cs.kent.ac.uk/pubs/2005/2162.

[13] P. H. Welch, F. R. M. Barnes, and F. A. C. Polack. Communicating complex systems. In M. G. Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford, California, August 2006. IEEE. URL http://www.cs.kent.ac.uk/pubs/2006/2398. ISBN: 0-7695-2530-X.