ELSEVIER Computer Communications 19 (1996) 1151–1160 computer communications

Designing a multiway synchronization protocol

Joachim Parrow[a,*], Peter Sjödin[b,†]

[a]*KTH/Teleinfomatics, Electrum 204, S-164 40 Kista, Sweden*
[b]*SICS, Box 1263, S-164 28 Kista, Sweden*

Abstract

A multiway synchronization protocol makes it possible for several processes to synchronize in an environment where communication is asynchronous. We present the design of such a protocol. The design methodology is based on formulating the behaviour of the entities as transition systems. This admits a correctness proof: we show that the protocol is correct relatively an 'ideal' non-distributed algorithm, in the sense that the protocol and the ideal algorithm cannot be separated by any amount of testing. The proof method is based on cs-equivalence.

Keywords: Synchronisation protocol; Design methodology; Correctness proof

1. Introduction

A multiway synchronization is a mechanism that allows multiple parallel processes in a distributed environment to communicate and coordinate their actions in a synchronous way. Consider, for example, a distributed database where data are replicated over several processes. In order to guarantee that different processes have a consistent view of the contents of the database, they need to synchronize their interactions, e.g. several processes may not update the same database item at the same time. Multiway synchronization was introduced in Hoare's CSP [1], and has thereafter been used as a programming construct in several other languages [2–6].

The purpose of a *multiway synchronization protocol* is to implement multiway synchronizations in an environment providing only binary, asynchronous communication where messages arrive after a finite but unpredictable delay. This paper presents such a protocol, focusing on the high level design methodology and on the verification that no mistakes have been made in the design.

In order to describe and reason about the entities in our protocols, we shall present them as *transition systems*. This means that we define the states that can be reached when the entities execute, and the possible transitions between states. Such systems can be represented as directed graphs, where the nodes are states and the edges are transitions. They admit precise definitions of behaviours on a high level of abstraction and general conclusions about the design, such as deadlock freedom or conformance with intended behaviour, before considering problems related to low-level implementation. Several specification languages such as LOTOS [7], SDL [8] and Statecharts [9] employ this idea, and have automated tools for checking some of the correctness properties. The protocol in the present paper can easily be formulated in any of these languages but we found it more straightforward to work directly with transition systems. Our correctness proof that the protocol works well with an arbitrary number of participating processes is beyond automatic verification in the available tools.

A multiway synchronization protocol communicates with several *client* processes, each of which is prepared to participate in a subset of the predefined multiway synchronization *actions*. The task of the protocol is to select one such action for execution. Our design approach is to begin with an almost trivial, *ideal*, protocol that can always schedule any enabled multiway synchronization. This is formulated as a single transition system; we do not require that it is distributed over a network. Although the ideal protocol is therefore inapplicable in an asynchronous distributed environment, its behaviour serves as a reference for other, more involved solutions. We will proceed to give a *distributed* multiway synchronization protocol where it is assumed that processes can only communicate asynchronously. Again, this solution is formulated using transition systems. We thus have two solutions to the same problem: one ideal (and, hopefully, self-evidently correct) but

* Email: joachim@it.kth.se
† peter@sics.se

inapplicable; and one more complex but implementable. Our verification effort is to show that the two solutions are indistinguishable from the point of view of the clients. Therefore, if the simpler solution is correct, so is the complicated solution.

The transition systems will contain nondeterminism, in the sense that if it is possible to schedule several actions then there will be one transition corresponding to each choice. The verification shows that no such possibility is lost, or an additional (and faulty) one is gained, in the distributed implementation. Of course, this will not rule out inadmissible behaviour on account of low-level design decisions involving, e.g. the mechanism choosing between several enabled decisions, or the mechanism for allocating memory and processor capacity. The verification merely says that in going from the ideal to the distributed solution we do not introduce any design errors.

In an earlier paper [10] we have used the same design method; the present paper is different in that the distributed protocol uses less communication overhead, and in that the exposition is less technical. Many published verifications of multiway synchronization algorithms assert that an algorithm satisfies certain correctness properties [4, 11–15]; our verification methods yields a more general result: *all* properties of the ideal protocol, formulated in terms of communication events with the clients, are also properties of the distributed protocol.

The following section explains in more detail what we mean by multiway synchronization. Section 3 gives a description of the ideal multiway synchronization protocol. We proceed by presenting the distributed algorithm in Section 4, and then explain our verification method in Section 5, followed by an outline of the correctness proof. Section 6 concludes the paper.

2. A model of multiway synchronization

We assume that there are n processes, in the following called *client processes*, communicating over a communi-

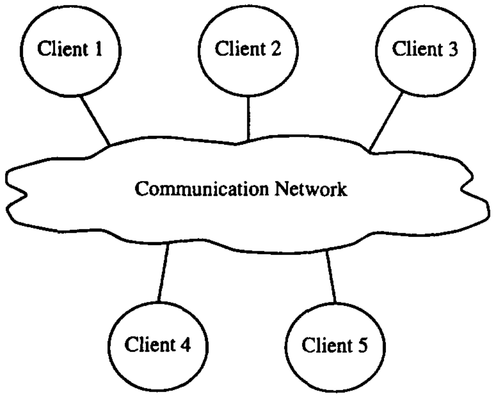


Fig. 1. System model: a set of client processes (1 to 5) communicating over a network.

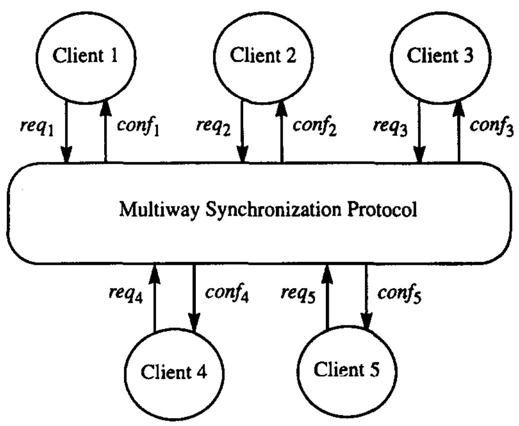


Fig. 2. Model of communication between client processes and the MSP.

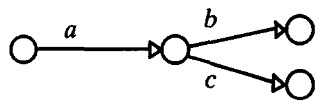
cation network providing asynchronous point-to-point connections (see Fig. 1). Thus, messages will arrive after an unknown and variable delay, and the network never loses messages.

We further assume that there are several multiway synchronization *actions*, denoted $a, b, \ldots$, in which client processes may wish to engage. Each such action a involves a fixed subset $P(a)$ of the client processes, all of which must agree to participate for the action to take place. For example, we may have three actions a, b and c with

$$P(a) = \{1,2\} \quad P(b) = \{2,4,5\} \quad P(c) = \{1,2,3,4,5\}.$$

So action a will happen only if client processes 1 and 2 agree to do so, b will happen only if client processes 2, 4 and 5 agree, and c if all client processes agree. The different actions compete for resources at the client processes, and therefore each client process can only engage in one action at a time. So in this example only one action can take place, because the participation of client process 2 is required in all three actions.

For a given client process i, at any given moment there is a set of actions in which it is prepared to participate. This set may vary over time. For example, client process 2 may first want to do only action a, refusing to participate in b or c. When a has been completed, client process 2 may change state and be prepared to do either b or c. In this way a client process can be thought of as a transition system where the



actions label the edges:

The multiway synchronization problem is to design a part of the communication network, called the *Multiway Synchronization Protocol* (MSP), communicating with the client processes and helping them to achieve the multiway synchronization actions. The client processes will tell the MSP in which actions they are prepared to participate; based

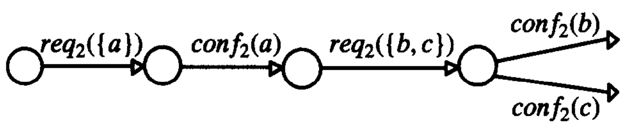
Table 1
State variables of the ideal MSP

RQ	A set of pending requests from client processes. A *request* is a pair (i, a), where $i \in P(a)$, and means that process i has said that it is prepared to engage in action a.
CF	A set of requests which should be granted because the corresponding actions have been selected by the MSP. If $(i, a) \in CF$ then it behooves the MSP to notify process i, along $conf_i$, that the action a has been scheduled.

on this information, the MSP will tell the client processes which action it schedules. This action must be one to which all processes in $P(a)$ agree. For this to work in a distributed system we need to impose a stability condition on client processes: whenever a client has told the MSP that it is offers to engage in a set of actions, it may not withdraw that offer until an action in the set is scheduled by the MSP. The reason is that when the MSP has informed one client process that a is scheduled, the decision to do a cannot be revoked by another client.

Since the network provides asynchronous communication, we cannot rely on a global clock available to the client processes. It is therefore meaningless to require that for an action a all processes in $P(a)$ execute their parts simultaneously. But we do require consistency in the sense that if one process is informed that a has taken place, then all processes in $P(a)$ will be so informed, and all these processes have actually agreed to participate in a.

We can now make the model of the client processes more precise. For the purposes of the MSP, each client process i has two communication channels, called req_i and $conf_i$ (see Fig. 2). Along req_i, it will send to the MSP a set of action names, namely the actions in which it is currently prepared to participate. Along $conf_i$ it will receive the action scheduled by the MSP. Because of the stability condition, a request (along req_i) is valid until the next confirmation (along $conf_i$). An example of the client process 2 in this model:



3. An ideal MSP

A top-down design of an MSP might begin by listing all properties that the MSP should possess, perhaps formulated in a (temporal) logic over the communication events req_i and $conf_i$, but that turns out to be a complicated task. It is difficult to be certain that no property has been forgotten. Instead, we will formulate an *ideal* MSP to act as a reference for all implementations. The idea behind the ideal MSP is that it functions in the best possible way: it is always prepared to receive requests from client processes, and it can schedule any action to which all participants have agreed. But it exhibits no internal structure in the form of different entities distributed over a network. Therefore, it is not applicable in a distributed environment, where different clients may reside at different places. Fortunately, it is possible to give distributed implementations that behave in exactly the same way as the ideal MSP. This fact, together with the simplicity of the ideal MSP, motivates our approach.

To formulate the ideal MSP we must define the states it can reach, and the transitions between states. A state of the ideal MSP is determined by two parameters, or *variables*, given in Table 1.

Even in the simple example of the previous section there are 2^{10} possible different combinations of requests, so the corresponding ideal MSP has at least that many states. Rather than drawing the ideal MSP as a transition graph, we shall formulate the transitions in a table (Table 2). Here, and in following similar tables, we group the transitions into classes and give each class on a separate row in the table. The first column gives the number of the transition class. The 'precondition' column says from which states the transitions go. The 'event' column says which communication (here req_i or $conf_i$), if any, is involved. The remaining columns define the target state of the transitions in terms of how it affects CF and RQ ('–' means that the variable is unchanged). Free symbols in the precondition or event (like A and i in the first row) are implicitly universally quantified in the whole row. Thus, there is one transition in the class for each instantiation of these symbols.

The ideal MSP has three classes of transitions. First, it must be able to receive requests to participate in multiway actions along req_i. Second, it must be able to schedule such actions. Third, it must inform the processes (along $conf_i$)

Table 2
Transitions of the ideal MSP

Class	Precondition	Event	RQ'	CF'
1	$\neg\exists l : (i, l) \in RQ \cup CF$	$req_i(A)$	$RQ \cup \{(i, a) : a \in A\}$	–
2	$\forall i : i \in P(a) \rightarrow$ $(i, a) \in RQ \wedge \neg\exists b : (i, b) \in CF$	–	$RQ - \{(i, b) : i \in P(a)\}$	$CF \cup \{(i, a) : i \in P(a)\}$
3	$(i, a) \in CF$	$conf_i(a)$	–	$CF - \{(i, a)\}$

Table 3
Channels for communication among ports and mediators

Channel	*Sender*	*Receiver*	*Meaning*
$ready_{i,a}$	Mediator i	Port a	Client i is ready to do action a
$yes_{i,a}$	Mediator i	Port a	Action a has been scheduled
$no_{i,a}$	Mediator i	Port a	Action a could not be scheduled
$query_{a,i}$	Port a	Mediator i	Request to start negotiating action a
$lock_{i,j}(a)$	Mediator i	Mediator j	Request to lock mediator j for action a
$commit_{i,j}$	Mediator i	Mediator j	Current action* has been scheduled
$abort_{i,j}$	Mediator i	Mediator j	Current action* could not be scheduled

* The action in the last lock request from j.

of this. These classes are represented in the three rows of the table. The first row says that a communication $req_i(A)$, where A is a set of actions, means that client t is prepared to engage in any of the actions in A. As a consequence, the requests (i, a) for all $a \in A$ will be added to RQ. A client may only issue one set of requests at a time; it must wait for one of the requested actions to be scheduled before it may issue the next set of requests. Thus, this communication is admissable as long as there are no requests from i in RQ or CF.

The second class represents the scheduling of an action a. This transition is internal to the MSP (so it does not require a communication with a client), and it is available in states where all processes in $P(a)$ have requested the action a, and none of them is yet scheduled for an action. As a consequence, all requests from all those processes are removed from RQ (ensuring that these processes cannot simultaneously be scheduled for another action), and the requests for a are added to CF, meaning that the MSP will notify all processes in $P(a)$. Finally, the third line says that in any state where a request (i, a) is in CF, the MSP can emit a $conf_i(a)$ communication, thereby confirming action a to process i. That request is thereby removed from CF to prevent it from being confirmed a second time.

It should be obvious from this description that the ideal MSP possesses all properties one could ever want from any MSP. It can communicate with the client processes except when such a communication would violate a principle behind multiway synchronization. It can schedule any action that all involved processes have requested, and it can schedule several (independent) actions in parallel. If there is a choice of which action to schedule, that choice is nondeterministic in the ideal MSP, meaning that in this model we do not include the factors that determine the outcome of the choice. Such factors could be that the first request should have priority, or that a certain process or a certain action should have priority. This must eventually be included in any concrete implementation of an MSP, but it permits a clearer description and more general correctness proofs to delay the introduction of these factors until they are necessary.

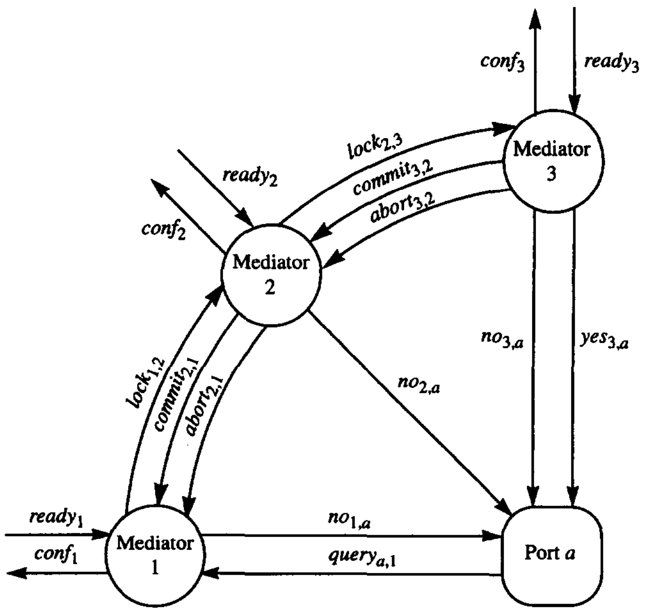


Fig. 3. The part of the distributed MSP for action a where $P(a) = \{1, 2, 3\}$.

4. A distributed MSP

In principle, it is possible to derive an implementation directly from the description of the ideal MSP. Such an implementation would consist of a single, centralized process, and might work well in a small system with few processes and short distances, but it would be less suitable in a large distributed environment. In particular, since this requires that all multiway synchronizations are established by a single process, this single process is likely to become a communication bottleneck.

We therefore design our system so that the task of establishing multiway synchronizations is performed by a group of cooperating processes, a *distributed MSP.* In the distributed MSP, scheduling of actions is negotiated in several steps. This negotiation is carried out between two kinds of processes: *mediators* and *ports.* There is one mediator per client process; a mediator negotiates for actions on behalf of its client. In a similar way, there is one port process per action, and a port is responsible for taking the initiative to

Table 4
State variables of mediator i

E	The set of actions client i currently has requested to engage in.
R	A set of requested actions that i will forward, along *ready* channels, to ports.
L	A set of actions that i has received lock requests for, but not yet processed.
C	A set of actions that have been scheduled, and i will confirm along *commit* channels.
A	A set of requests that i will deny, along *abort* channels.
w	An action that is being negotiated, and i is waiting for the negotiation to complete.
c	A scheduled action that will be confirmed to client i along $conf_i$.

start a negotiation. Ports and mediators communicate along a set of channels summarized in Table 3. We use the notation $chan_{x,y}$ to represent a communication channel *chan* from x to y, where x and y can be integers or actions to signify the corresponding mediator or port. When the channel carries a value v from x to y it is written $chan_{x,y}(v)$. See Fig. 3 for an illustration of the distributed MSP.

Mediator i receives synchronization requests from its client process along req_i. The mediator forwards the requests to the ports; mediator i uses channel $ready_{i,a}$ to forward requests to port a. When a port has received sufficiently many requests to determine that the action is possible, it can initiate negotiation for scheduling the action.

A negotiation for an action is carried out by requesting to *lock* each of the mediators involved in the action. Port a starts a negotiation by requesting to lock mediator i by a communication along channel $query_{a,i}$. If mediator i still is prepared to participate in action a, it can accept to be locked by a and sends in turn a lock request $lock_{i,j}(a)$ to mediator j (assuming that j is the next mediator involved in action a). When the final mediator m accepts the lock, it sends an indication along $yes_{m,a}$ to port a that the action has been scheduled. Mediator m also confirms the action to the mediator from which it received the lock request (mediator k). This confirmation is sent along $commit_{m,k}$. A locked mediator that receives a confirmation along a *commit* channel forwards the confirmation to the mediator from which it received the lock request. This is repeated until the confirmation reaches the mediator that sent the first lock request. In addition, a mediator that learns that an action has been scheduled confirms the action to its client process (mediator i does this along $conf_i$).

It can happen, however, that when mediator i receives a lock request ($lock_{j,i}(a)$ or $query_{a,i}$), it has already participated in some other action and therefore cannot be locked. The mediator then informs port a that the action could not be scheduled, through a communication along $no_{i,a}$, and aborts the ongoing negotiation through a communication along $abort_{i,j}$. Like a confirmation, an abort signal is forwarded between mediators until it reaches the first mediator that sent the lock request.

Finally, while a mediator is locked it neither accepts nor rejects to be locked for other actions – it waits for the outcome of the ongoing negotiation before it makes any more decisions. This means that lock requests received while waiting are stored and processed later. But for this to work, we must define carefully in what order mediators can be locked. Otherwise the system may end up in a deadlock situation. Consider, for example, a system with two ports a and b, and two mediators 1 and 2. Port a requests to lock mediator 1, which accepts this and forwards the lock request to mediator 2. At the same time, port b locks mediator 2, which forwards the lock request to mediator 1. Both mediators are waiting for the outcome of negotiations

Table 5
Transitions of mediator i

Class	Precondition	Action	C'	A'	R'	L'	E'	w'	c'
1	$E = \emptyset \wedge c = \perp$	$req_i(B)$	–	–	B	–	B	–	–
2	$a \in R$	$ready_{i,a}$	–	–	$R - \{a\}$	–	–	–	–
3	–	$query_{a,i}$	–	–	–	$L \cup \{a\}$	–	–	–
4	–	$lock_{j,i}(a)$	–	–	–	$L \cup \{a\}$	–	–	–
5	$a \in L \wedge a \in E \wedge w = \perp$ $\wedge\ c = \perp \wedge j = next(i,a)$	$lock_{i,j}(a)$	–	–	–	$L - \{a\}$	–	a	–
6	$a \in L \wedge a \in E \wedge w = \perp$ $\wedge\ c = \perp \wedge \perp = next(i,a)$	$yes_{i,a}$	$C \cup \{a\}$	–	–	$L - \{a\}$	$\emptyset$	–	a
7	$a \in L \wedge a \notin E$	$no_{i,a}$	–	$A \cup \{a\}$	–	$L - \{a\}$	–	–	–
8	$w = a \wedge prev(i,a) = \perp$	$commit_{j,i}(a)$	–	–	–	–	$\emptyset$	$\perp$	a
9	$w = a \wedge prev(i,a) \neq \perp$	$commit_{j,i}(a)$	$C \cup \{a\}$	–	–	–	$\emptyset$	$\perp$	a
10	$a \in C \wedge j = prev(i,a)$	$commit_{i,j}(a)$	$C - \{a\}$	–	–	–	–	–	–
11	$w = a \wedge prev(i,a) = \perp$	$abort_{j,i}(a)$	–	–	–	–	–	$\perp$	–
12	$w = a \wedge prev(i,a) \neq \perp$	$abort_{j,i}(a)$	–	$A \cup \{a\}$	–	–	–	$\perp$	–
13	$a \in A \wedge j = prev(i,a)$	$abort_{i,j}(a)$	–	$A - \{a\}$	–	–	–	–	–
14	$c \neq \perp$	$conf_i(c)$	–	–	–	–	–	–	$\perp$
	Initial value	–	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\perp$	$\perp$

so none of them will respond to the second lock request, and a deadlock has occurred.

Therefore, we define a total ordering relation for clients, and require that mediators are always locked according to that order. Thus, mediators can be thought of as forming a directed chain of processes, where lock requests are sent in the forward direction of the chain, and abort and commit messages are returned in the opposite direction. Since we use positive integers to enumerate client and mediator processes, we can use the relation '$>$' (strictly greater than) as our ordering relation. For example, mediator 1 comes before mediator 2, so both port a and b would request to lock mediator 1 first, which would then forward one of the lock requests to mediator 2 and delay the other. Thus, there cannot be any deadlocks. In our description, we use three functions on clients and their mediators: $first(a)$ is the client in $P(a)$ that comes first in order; $next(i,a)$ gives the process in $P(a)$ that comes after client i; and $prev(i,a)$ gives the client in $P(a)$ that comes before client i. If the value of a function is undefined, it is represented by the symbol $\perp$.

Each mediator has seven variables, described in Table 4, and Table 5 specifies the transitions of mediator i. Transition class 1 says that as long as a mediator does not have pending requests from its client, a mediator is prepared to receive a synchronization request from its client and assign the requested actions to the R and E set of actions. The second transition class specifies that the mediator can, for each action in R, communicate along *ready* with the corresponding port and then remove the action from the R set.

Transition classes 3 and 4 are receptions of lock requests: 3 is a lock request from a port (along a *query* channel), and 4 is a lock request from another mediator (along a *lock* channel). A mediator is always prepared to receive a lock request, and adds the corresponding action to the L set.

Transition classes 5 and 6 describe how mediators accept to be locked for an action. A mediator can decide that it is prepared to participate in an action provided that the action is present in E, and that the mediator is not waiting for the outcome of a negotiation or is about to confirm a scheduled action to its client. If the mediator is not at the end of the action chain (transition class 5), it forwards the lock request to the next mediator. If it is at the end of the chain (transition class 6), it signals to the port that the action has been scheduled (along $yes_{i,a}$), moves the action from the L to the C set (for later communication along a *commit* channel), and stores it in c for confirmation to the client. In addition, the mediator should now reject all locks (until the next request appears from the client), and therefore clears the E variable.

Table 6
State variables of port a

T	The set of processes that have advertised that they are prepared to do action a.
n	A flag indicating whether there is an ongoing negotiation for action a.

Table 7
Transitions of port a

Class	Precondition	Action	T'	n'
1	–	$ready_{i,a}$	$T \cup \{i\}$	–
2	$\neg n \wedge P(a) = T \wedge j = first(a)$	$query_{a,j}$	–	true
3	$n \wedge j \in P(a)$	$yes_{j,a}$	$\emptyset$	false
4	$n \wedge j \in P(a)$	$no_{j,a}$	$P(a) - \{j\}$	false
	Initial value	–	$\emptyset$	false

The transitions in class 7 say that lock requests for actions not present in E are always rejected, by signaling to the port along $no_{i,a}$ that the action cannot be scheduled, and by moving the lock reqpest from the L set to the A set (for later communication along *abort*).

Transition classes 8 and 9 specify that a mediator that is waiting for a negotiation (its w variable is not $\perp$) is always prepared to receive a positive acknowledgement of the action along the corresponding *commit* channel. Upon reception of a commit, the mediator knows that the scheduling is completed, and therefore clears its E and w variables, and assigns the c variable the value of the action (in analogy with transition class 6 above). If the mediator is not first in the chain (transition class 9), it should forward the acknowledgement to the previous mediator, and therefore also stores the action in the C set. Transition class 10 describes this forwarding of acknowledgements.

Similarly, transition classes 11, 12 and 13 describe reception and forwarding of negative confirmations along *abort* channels. Transition class 14, finally, defines how a mediator confirms an action to its client along a *conf* channel.

The specification of a port process is much simpler. Its state is determined by two variables, described in Table 6. The transitions of port a are given in Table 7. Transition class 1 specifies that a port is always prepared to receive a ready announcement. Transition class 2 describes how a port decides to initiate the negotiation for scheduling of an action: if all client processes in $P(a)$ have announced that they are willing to do the action, and there is no ongoing negotiation for it, then a negotiation can be started.

Transition class 3 says that when a port receives a positive acknowledgement of an action, through a communication along a *yes* channel, it 'forgets' all ready announcements it has received by clearing the T variable, and then awaits to receive new announcements. If the port receives a negative acknowledgement instead, along a *no* channel, it restores the client processes into T *except* the process that rejected the confirmation (since this process no longer is prepared to participate in the action).

Since a distributed MSP consists of port and mediator processes running in parallel, a state of the distributed MSP is given by the all the state variables of the ports and mediators together. We write C_i for state variable C of mediator i, etc. That is, for a system with n mediators and m ports, the state is defined by the mediator variables $C_1, \ldots, c_1$ through $C_n, \ldots, c_n$, and the port variables T_1, n_1

through T_m, n_m. As a shorthand for this, we will represent a state of the distributed MSP as $(C_1, \ldots, c_n, T_1, \ldots, n_m)$.

We define the parallel composition of port and mediator processes in such a way that communications among these processes yield internal transitions in the transition system of the distributed MSP. That is, communications along *ready*, *query*, *lock*, *yes*, *no*, *abort* and *commit* are internal transitions to the distributed MSP, and do not involve (and cannot be influenced by) the environment. These specifications are made in terms of synchronous communication events along channels. Since the system being specified is asynchronous, we specify our processes in such a way that a process never refuses to participate in events that correspond to receptions along a channel. Furthermore, a process does not act immediately on a communication event. Instead it stores information about the event in state variables, and processes this information later.

5. Correctness of the distributed MSP

The distributed MSP is significantly more complex than the ideal MSP, and it is not obvious from just studying the specification that it actually works. But to demonstrate that the distributed MSP is correct, we must first define what we mean by *correct*, that is, we must define more precisely what it means that the ideal and distributed MSP are indistinguishable from a client process' point of view.

To obtain a method for establishing correctness, it turns out to be convenient to formulate the correctness criteria in terms of *simulation* relations between the states of the two transition systems. In the most simple variety of simulation, we say that a state S_1 simulates another state S_2 if whenever there is a transition $S_2 \xrightarrow{a} S_2'$, there is a corresponding transition $S_1 \xrightarrow{a} S_1'$ such that S_1' continues to simulate S_2'. This implies that S_1 can do all sequences of transitions that S_2 can. But this simulation variety is too discriminating for our purposes – the ideal and distributed MSPs cannot simulate each other in this way. Consider, for example, a system with two clients, client 1 and client 2, which request to synchronize through an action a. The ideal MSP can when it has received the two requests $req_1(\{a\})$ and $req_2(\{a\})$ do a single internal transition and then be prepared to confirm the action through $conf_1(a)$ and $conf_2(a)$. The distributed MSP, on the other hand, must after $req_1(\{a\})$ and $req_2(\{a\})$ do five internal communications ($ready_{1,a}$, $ready_{2,a}$, $query_{a,1}$, $lock_{1,2}$ and $yes_{2,a}$) before it is prepared to do $conf_2(a)$. It has to do yet another internal communication ($commit_{2,1}$) before it can do $conf_1(a)$. Because of this difference in the amount of internal communication between requests and confirmations, the ideal and distributed MSP cannot simulate each other.

We therefore need a simulation relation that is less strict with respect to internal communications. This variety is called a *weak* simulation relation, written w-simulation, which allows the simulating transition to be preceded and succeeded by an arbitrary number of internal transitions. More formally, a state S_1 w-simulates a state S_2 if whenever there is a transition $S_2 \xrightarrow{a} S_2'$, there is a sequence of transitions from S_1 consisting of zero or more internal transitions, followed by a, followed by zero or more internal transition, leading to a state S_1' such that S_1' w-simulates S_2'. We will write $S_2 \overset{a}{\Rightarrow} S_1'$ to represent such sequences of actions. Furthermore, if a itself is an internal transition, we allow S_1 and S_1' to be the same state.

If we could base our correctness criterion on w-simulations, we would say that two transition systems are indistinguishable if their initial states can w-simulate each other. This is, however, a too weak criterion, since it does not take non-determinism into account. Non-determinism appears for instance in states where there are alternative internal transitions: internal transitions can then be thought of as representing decisions made internally by the system. For example, a state in which a system decides whether to do action a or action b is a state where two internal transitions are possible, the first leading to a state where only a is possible and the second to a state where only b is possible. However, such a state w-simulates (and is w-simulated by) a state in which both a and b are directly possible, that is, a state in which the system allows its environment to control which action should occur.

We want the ideal and distributed MSP to do their decision in the same way; a decision that is internal to one of them should be internal also to the other. We achieve this by requiring that some states in the two transition systems, in addition to their initial states, are *coupled* to each other, which means that the two states w-simulate each other.

The next question is: for what states do we require coupling? The strongest requirement would be to say that if a state S_1 w-simulates a state S_2, then also S_2 should w-simulate S_1 (the equivalence relation we then get is Milner's observation equivalence [16]). This is, however, too strong a requirement for our purposes; there exists no such relation between the states of the ideal and the distributed MSP. We therefore only require coupling for states from which there are no internal transitions, that is, states in which no internal decisions remain to resolve. Such states are said to be *stable*.

We can now formulate the *coupled* simulation relation (c-simulation) on states of transition systems. A coupled simulation relation is a pair $(\mathcal{R}_1, \mathcal{R}_2)$ of relations where:

- Whenever $S_1 \mathcal{R}_1 S_2$ and there is a transition $S_2 \xrightarrow{a} S_2'$ then there is a state S_1' and a sequence of transitions $S_1 \overset{a}{\Rightarrow} S_1'$ such that $S_1' \mathcal{R}_1 S_2'$. Furthermore, if S_2 is stable, then also $S_1 \mathcal{R}_2 S_2$.
- Whenever $S_1 \mathcal{R}_2 S_2$ and there is a transition $S_1 \xrightarrow{a} S_1'$ then there is a state S_2' and a sequence of transitions $S_2 \overset{a}{\Rightarrow} S_2'$ such that $S_1' \mathcal{R}_2 S_2'$. Furthermore, if S_1 is stable, then also $S_1 \mathcal{R}_1 S_2$.

Finally, we say that two transition systems are coupled simulation equivalent, or *cs-equivalent*, if their initial states are related by a coupled simulation relation. The cs-equivalence has the property that two systems that are cs-equivalent cannot be separated by any amount of testing [17]. For example, they have the same deadlock properties, and the same set of possible sequences of communication events.

The correctness criterion that the two MSPs are indistinguishable by client processes then becomes: *The ideal MSP and the distributed MSP are cs-equivalent.*

In order to establish cs-equivalence between the two MSPs, we formulate a coupled simulation in terms of two functions over states of the distributed MSP: $RQ(C_1, \ldots, c_n, T_1, \ldots, n_m)$ and $CF(C_1, \ldots, c_n, T_1, \ldots, n_m)$. There is one function for each variable in the ideal MSP; the idea is that the two functions should mimic the two state variables of the ideal MSP. Roughly speaking, for states of the distributed and ideal MSP that are related through the coupled simulation, $CF(C_1, \ldots, c_n, T_1, \ldots, n_m)$ equals CF and $RQ(C_1, \ldots, c_n, T_1, \ldots, n_m)$ equals RQ.

The key to defining these functions is to identify a transition of the distributed MSP that corresponds to the internal, second transition class of the ideal MSP – the transition by which the ideal MSP definitely decides to schedule an action. In the general case, we cannot pinpoint one single transition in the distributed MSP that does this, since the decision can be resolved gradually through a sequence of internal transitions. Fortunately, the cs-equivalence gives us some degree of freedom here, and we can pick any transition after which the decision is irrevocable. For this we can choose, for instance, the internal transition corresponding to a communication along a *yes* channel. That is, we say that the deciding transition is when the last mediator in the chain moves the action from its L variable to its C variable (transition class 6 in Table 5).

We formulate the two functions in the following way:

$$RQ(C_1, \ldots, c_n, T_1, \ldots, n_m)$$
$$= \{(i,a) : i \in P(a) \wedge a \in E_i \wedge \neg \exists j : (j > i \wedge a \in C_j)\}$$
$$CF(C)1, \ldots, c_n, T_1, \ldots, n_m)$$
$$= \{(i,a) : i \in P(a) \wedge (a = c_i \vee \exists j : (j > i \wedge a \in C_j))\}.$$

The first part of the coupled simulation relation (relation $\mathcal{R}_1$), which says how the ideal MSP simulates the distributed MSP, is then straightforward:

Definition of $\mathcal{R}_1$

$$(RQ, CF)\mathcal{R}_1(C_1, \ldots, c_n, T_1, \ldots, n_m)$$
$$\text{if} \begin{pmatrix} RQ = RQ(C_1, \ldots, c_n, T_1, \ldots, n_m) \\ \wedge\ CF = CF(C_1, \ldots, c_n, T_1, \ldots, n_m) \end{pmatrix}.$$

Note that, since the functions only depend on a subset of the variables of the distributed MSP, a state of the ideal MSP can simulate more than one state of the distributed MSP. For instance, internal transitions in the distributed MSP corresponding to the forwarding of lock requests (communications along *lock* channels) are simulated by the ideal MSP remaining in the same state. For the second simulation relation, we can use this fact to simplify the forthcoming proof by reducing the number of states of the distributed MSP that are covered by the relation. In other words, for each state of the ideal MSP, it is sufficient to find one simulating state in the distributed MSP. By reducing the number of simulating states, we will reduce the number of cases we need to examine in the proofs. In addition, in this way we can again circumvent the problem of pinpointing a decision in the distributed MSP, by letting the simulation relation only cover states where we know that the distributed MSP has not eliminated any possible actions. For this, we pick only those states where there are no ongoing negotiations (i.e. no port or mediator has outstanding lock requests), and all mediators have advertised all their requests to ports (i.e. all mediators have an empty R variable). The second simulation relation then becomes:

Definition of $\mathcal{R}_2$

$$(RQ, CF)\mathcal{R}_2(C_1, \ldots, c_n, T_1, \ldots, n_m)$$
$$\text{if} \begin{pmatrix} RQ = RQ(C_1, \ldots, c_n, T_1, \ldots, n_m) \\ \wedge\ CF = CF(C_1, \ldots, c_n, T_1, \ldots, n_m) \\ \wedge\ \forall i, a : (R_i = L_i = A_i = C_i = \emptyset \wedge w_i = \perp \wedge \neg n_a) \end{pmatrix}.$$

It is easy to see that the initial states of the ideal and distributed MSP are related by $\mathcal{R}_1$ and $\mathcal{R}_2$. Thus, to show cs-equivalence, we need to show that $\mathcal{R}_1$ and $\mathcal{R}_2$ together form a coupled simulation relation. This proof is straightforward, but somewhat lengthy: for each of the transition classes of the ideal and the distributed MSP, show that the coupled simulation relation is preserved by the transition. The details of the proof are omitted.

Finally, to complete a correctness proof we should show that the distributed MSP does not have any non-terminating internal loops, i.e. that it is free from live-locks. Absence of live-locks is a desirable property in itself, since it guarantees that the MSP will always make progress. In addition, this requirement makes sure that the second rule of the c-simulation has effect: it guarantees that there are stable states, and therefore some states in the two MSPs have to be coupled.

To demonstrate that there are no potential live-locks, we should show that the distributed MSP can only do a finite number of internal transitions before it has to communicate along *req* or *conf*. To do this, we formulate a *well-founded order* on states of the distributed MSP. The idea here is to assign a certain 'weight' to each state, and show that each internal transition brings the distributed MSP to a state with strictly less weight. We let the weight be a natural number, so the decrease is always a (positive) integer number. Furthermore, we define the weight in such a way that it has a lower limit. These facts taken together guarantee that there are no infinitely long sequences of internal transitions – whatever state the distributed MSP is in, it an only do so many internal transitions as it takes to bring its weight down to the minimum. The proof that the weight strictly decreases is by case analysis over the individual transition classes. The weight function is a polynomial in number of elements in mediator and port state variables, and in indices of mediators. The function is somewhat involved, and we omit its presentation and the proof details.

6. Conclusions

We have presented a specification of a protocol for multiway synchronization of processes. The algorithm is designed for large-scale environments with many processes, where it is not suitable to implement process synchronization in a centralized manner. Many different algorithms have been proposed for implementing multiway synchronization [2, 11–15, 18]; the protocol presented here is an improved version, in terms of the number of messages needed to establish synchronization, of our previously published algorithm [10].

We have outlined a way of demonstrating the correctness of the protocol, based on the cs-equivalence relation between transition systems. As a starting point, we defined the transition system for a simple algorithm, the ideal MSP, and the intention is that this specification should be so simple that it is obviously correct. We then specified our algorithm, the distributed MSP, consisting of a set of processes that cooperate to establish synchronizations. Thereafter, we defined a relation between the states of the two transition systems, and outlined that this relation is a cs-equivalence relation. Thus, by showing that the behaviour of a complex, distributed algorithm is equivalent to that of a simple, ideal algorithm, we can conclude that the distributed algorithm is correct. Furthermore, to complete the analysis, we show that the distributed MSP does not have any live-locks, that is, that there are no nonterminating internal loops. This property is desirable in itself, but we also need to establish this since or definition of cs-equivalence is only relevant for transition systems without live-locks. A discussion of alternative characterizations of cs-equivalence that apply also to transition systems with infinite internal loops can be found in Ref. [19].

The main difference between the behaviours of the ideal and distributed MSP is that when there are alternative synchronizations possible, the ideal MSP selects a synchronization to schedule with a single, internal step, whereas the distributed MSP resolves this choice gradually, through sequences of internal transitions. This makes it impossible to use, for instance, the popular *observation equivalence* [16] as correctness criterion for our algorithm, since this relation would require that internal decisions are resolved in the same way in both MSPs.

There are equivalence relations that are less discriminating than weak observation equivalence with respect to internal decisions, for example *testing equivalence* [20] (for systems without live-locks, cs-equivalence implies testing equivalence). But testing equivalence has one drawback: unlike observation equivalence and cs-equivalence, it cannot be established through case analysis of individual transitions, which is a proof method that is feasible even for large transition systems. So, in summary, the advantages of cs-equivalence are that it has a convenient proof method, and that it is less discriminating than for example observation equivalence with respect to how internal decisions are made.

References

[1] C.A.R. Hoare, Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[2] R.J.R. Back and R. Kurki-Suonio, Distributed cooperation with action systems. ACM Trans. Programming Languages and Systems, 10(4) (1988) 513–554.

[3] A. Charlesworth, The multiway rendezvous. ACM Trans. Programming Languages and Systems, 9(2) (July 1987) 350–366.

[4] N. Francez, B. Hailpern and G. Taubenfeld, Script: A communication abstraction mechanism and its verification. Science of Computer Programming, 6(1) (January 1986) 35–88.

[5] D. Kumar, An implementation of N-party synchronization using tokens. Proc. 10th Int. Conf. Distributed Computing Systems, pp. 320–327, 1990.

[6] S. Ramesh and S.L. Mehndiratta, A methodology for developing distributed programs. IEEE Trans. Software Engineering, SE-13(8) (August 1987) 967–976.

[7] T. Bolognesi, E. Najm and P. Tilanus, G-LOTOS: A graphical language for concurrent systems. Computer Networks & ISDN Systems, 26(9) (May 1994) 1101–1127.

[8] CCITT 1988, Recommendation Z.100: Specification and Description Language SDL. Blue Book, Volume X.1.

[9] D. Harel, Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3) (1987) 231–274.

[10] J. Parrow and P. Sjödin, Multiway synchronization verified with coupled simulation. In W.R. Cleaveland, ed., Proc. CONCUR '92, vol 630 of Lecture Notes in Computer Science, pp. 518–533, Springer-Verlag, 1992.

[11] R. Bagrodia, Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Trans. Software Engineering, 15(9) (September 1989) 1053–1065.

[12] G.N. Buckley and A. Silberschatz, An effective implementation for the generalized input-output construct of CSP. ACM Trans. Programming Languages and Systems, 5(2) (April 1983) 223–235.

[13] K.M. Chandy and J. Misra, Parallel Program Design: a Foundation. Addison-Wesley, Reading, MA, 1988.

[14] M. Choy and A.K. Singh, Efficient implementation of synchronous communication over asynchronous networks. J. Parallel and Distributed Computing, 26(2) (April 1995) 166–180.

[15] S. Ramesh, A new and efficient implementation of multiprocess synchronization. In Parallel Architectures and Languages Europe, vol 259 of Lecture Notes in Computer Science, pp. 387–401, Springer-Verlag, 1987.

[16] R. Milner, Communication and Concurrency, Prentice Hall, Englewood Cliffs, NJ, 1989.

[17] R.J. van Glabbeek, The linear times – branching time spectrum II. In E. Best, ed., Proc. CONCUR '93, vol 715 of Lecture Notes in Computer Science, pp. 60–80, Springer-Verlag, 1993.

[18] Y.-J. Joung and S.A. Smolka, Coordinating first-order multiparty

interactions. ACM Trans. Programming Languages and Systems, 16(3) (May 1994) 954–985.

[19] J. Parrow and P. Sjödin, The complete axiomatization of cs-congruence. In P. Enjalbert, E.W. Mayr and K.W. Wagner, eds., Proc. STACS 94, vol 775 of Lecture Notes in Computer Science, pp. 557–568, Springer-Verlag, 1994.

[20] M. Hennessy, Algebraic Theory of Processes, The MIT Press, Cambridge, MA, 1988.