Multiway Synchronization Verified with Coupled Simulation

Joachim Parrow* and Peter Sjödin

Swedish Institute of Computer Science, Box 1263, S-16428 Kista, Sweden
and
Department of Computer Systems, Uppsala University, Sweden

Abstract. We consider the problem of implementing multiway synchronization in a distributed environment providing only binary asynchronous communication. Our implementation strategy is formulated as a transformation on transition systems and we give a distributed algorithm for multiway synchronization. Correctness assertions and proofs are based on a new method: *coupled simulations.* The coupled simulation equivalence is weaker than observation equivalence and stronger than testing equivalence and combines some of their advantages. Like observation equivalence (and unlike testing) it is established through case analysis over single transitions. Like testing equivalence (and unlike observation) it allows an internal choice to be distributed onto several internal choices. The latter is particularly important when relating our distributed implementations to their specifications.

1 Introduction

In this paper we will formally verify a strategy for implementing multiway synchronizations between processes in an environment supporting only binary asynchronous communication. This type of environment is common in physically distributed systems, and multiway synchronization is a communication primitive that has attained much interest [1, 3, 4, 7, 8, 9, 10, 16]. We will here concentrate on the verification methodology, i.e. the formulation and proof of the relevant correctness properties.

Our point of departure will be a *specification* of a system of n "client" processes, written $P_1 \mid\mid \cdots \mid\mid P_n$. A process P_i can perform two kinds of actions: *synchronization actions* involve multiway synchronizations over ports with other processes, and *external actions* are interactions between a single process and the environment of the system. An example specification with four client processes is depicted in Fig. 1. The lines connecting the processes represent multiway synchronization capabilities; thus a is a synchronization between all four processes, b is between P_1, P_2 and P_3, and c is between P_3 and P_4. A process can only participate in one synchronization at a time, so there may be conflicts between different enabled synchronizations. Suppose that the specification in Fig. 1 is in a state where all three multiway synchronizations are enabled. Since there is one process that is involved in all synchronizations, namely P_3, only one of the synchronizations will take place. The choice of what synchronization takes place is nondeterministic.

* Partly funded by NUTEK as part of Esprit Basic Research Action 3006: Concur.

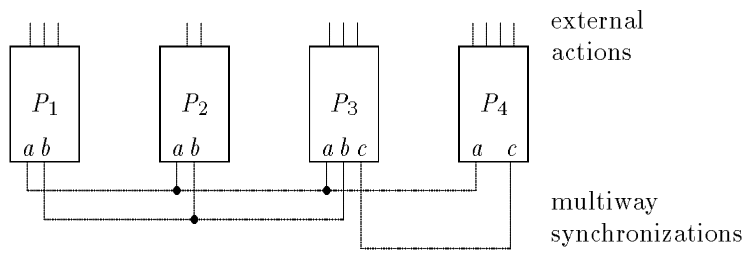


Fig. 1. A specification with four processes.

The first step in formulating a general strategy for implementing multiway synchronization is to replace multiway synchronizations by binary interactions with a monolithic *central synchronizer*. We call the resulting system a *central implementation*, and the central implementation of the specification $P_1 \mid\mid \cdots \mid\mid P_n$ is written $\mathcal{C}(P_1, \ldots, P_n)$. Each client process P_i here communicates, on actions req_i, its intentions to participate in synchronizations and receives confirmations about scheduled interactions on actions $conf_i$. A central implementation of a specification is achieved by transforming each client process so that it uses req_i and $conf_i$ actions rather than multiway synchronization actions, and by adding the central synchronizer as an extra process with no external actions. The central implementation of the specification in Fig. 1 is depicted in Fig. 2.

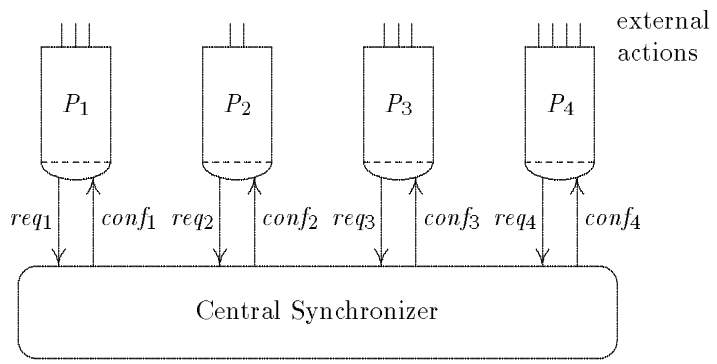


Fig. 2. A central implementation.

Traditionally, correctness proofs for multiway synchronization algorithms [2, 6, 10, 14, 15] amount to proving particular safety properties such as deadlock freedom and particular liveness properties such as "an enabled synchronization must eventually be scheduled for execution". But our implementation strategy admits a more general correctness assertion, namely that the implementation behaves as prescribed by the specification (i.e., as prescribed by $P_1 \parallel \cdots \parallel P_n$). We will compare the specification and the implementation with respect to how they communicate with their environments. In essence, we will require the implementation and specification to have exactly the same computational behaviour on external actions.

Both our specification and implementation may be nondeterministic, and various implementation preorders and behaviour equivalences have been suggested for comparing such behaviours. We will throughout this study use equivalences, which means that we do not allow the implementation to resolve any nondeterminism present in the specification; all of the specified behaviours must be implemented. When comparing $P_1 \parallel \cdots \parallel P_n$ with $\mathcal{C}(P_1, \ldots, P_n)$ we will use observation equivalence [12]. This is stronger than e.g. testing equivalence [5] (at least for the class of convergent systems which is of interest for this paper), so equivalent processes cannot be separated by any amount of testing. Furthermore observation equivalence can be shown by establishing weak bisimulations, a convenient proof technique applicable directly on transition systems.

A central implementation is not realistic if the client processes execute on physically distributed computers and communicate over an asynchronous network. Therefore we replace the central synchronizer with a *distributed synchronizer* which in turn consists of mediators and ports. There is one *mediator* for each client process; the mediator can be thought of as representing the client process in negotiations to establish multiway synchronizations. There is one *port* for each multiway synchronization action in the specification; the port establishes that all required clients participate in the synchronization. Mediators and ports interact asynchronously, each such interaction is always between one mediator and one port. The distributed implementation of n processes is written $\mathcal{D}(P_1, \ldots, P_n)$. Figure 3 contains the distributed implementation corresponding to Fig. 1 and 2.

Our goal is to determine to what extent $P_1 \parallel \cdots \parallel P_n$ and $\mathcal{D}(P_1, \ldots, P_n)$ are equivalent. A first step, as mentioned above, is to establish that $P_1 \parallel \cdots \parallel P_n$ is observation equivalent to $\mathcal{C}(P_1, \ldots, P_n)$. We then compare the distributed and central synchronizers. It turns out that these are *not* observation equivalent; the reason is that in the central synchronizer the choice between competing multiway synchronizations is resolved in one single step, whereas in the distributed synchronizer the corresponding choice can be resolved gradually over several steps. Observation equivalence is here a too strong criterion since it requires a direct correspondence between choice points in computations.

We therefore have to look for weaker behaviour equivalences. An interesting candidate is testing equivalence, which implies that both synchronizers satisfy the same tests on the external actions. Unfortunately, in order to establish testing equivalence one must do a case analysis on all possible computations (this is in contrast to observation equivalence which can be established through a case analysis over all single computation steps, a more manageable task). We will therefore introduce a new equivalence which we call *coupled simulation* equivalence, $=_{\mathrm{cs}}$. For convergent systems this lies between testing and observation equivalence. The main point is that coupled simulation equivalence can be established by case analysis over single computation steps and yet does not require an exact correspondence between choice points in computations. We will establish that the distributed and central synchronizers are coupled simulation equivalent, and that the context of the synchronizers

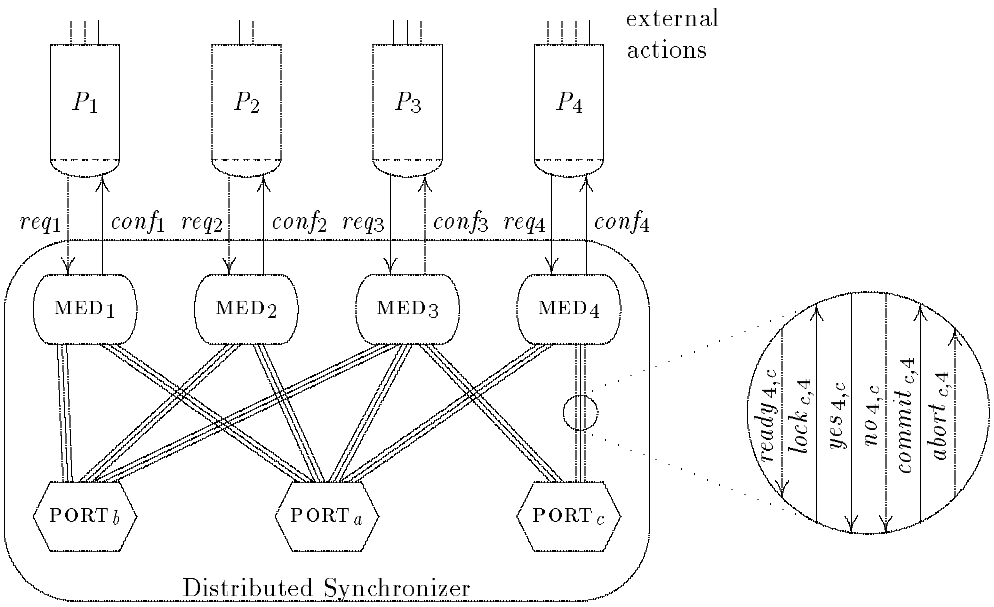


Fig. 3. A distributed implementation. The mediator for client i appears as MED$_i$ and the port for action l as PORT$_l$. A port and a mediator interact over six actions (*ready*, ..., *abort*).

preserves this equivalence. Thus we can conclude $P_1 \mid\mid \cdots \mid\mid P_n =_{\mathrm{cs}} \mathcal{D}(P_1, \ldots, P_n)$. In order to use coupled simulations we must first establish convergence of the involved processes. This result is also interesting in itself since it implies absence of livelocks in the implementation.

In summary, this paper addresses the problem of verifying systems where decisions on a high level of abstraction are regarded as atomic but in the implementation are made distributedly and incrementally.

The rest of the paper is structured as follows. In Sect. 2 we define transition systems, equivalences and parallel composition formally. Section 3 contains the definition and verification of central implementations. In Sect. 4 we define distributed synchronizers and prove them equivalent to the corresponding central synchronizers. Section 5 contains a brief summary, ideas for further work, and related work on verifications of multiway synchronization.

2 Preliminaries

2.1 Transition Systems

We assume a set of *actions* Act ranged over by k, l. The *silent step* τ is not in Act, and we use α to range over $Act \cup \{\tau\}$. A silent step represents an internal computation step of an entity.

A (labelled) transition system represents an entity that can move between different states, and in doing so performs actions or silent steps:

Definition 1 (transition system). A *transition system* is a pair $(\mathbf{Q}, \longrightarrow)$ where $\mathbf{Q}$ is a nonempty set, called the *states*, and $\longrightarrow \subseteq \mathbf{Q} \times (Act \cup \{\tau\}) \times \mathbf{Q}$ is called the *transition relation*. A *rooted transition system* is a transition system with a distinguished initial state. We use Q and Q' to range over rooted transition systems.

We use $q \xrightarrow{\alpha} q'$ to mean $(q, \alpha, q') \in \longrightarrow$. We also use $Q \xrightarrow{\alpha} Q'$ to mean that $q \xrightarrow{\alpha} q'$ where q and q' are initial states of Q and Q' respectively, and that Q and Q' have the same set of states and transition relation. We define $Q \xrightarrow{\alpha}$ to mean $\exists Q' : Q \xrightarrow{\alpha} Q'$; further, Q is *unstable* if $Q \xrightarrow{\tau}$, otherwise Q is *stable*. If Q' is reachable by zero or more transitions from Q we call Q' a *derivative* of Q, or sometimes that Q' is a *state* in Q.

Given a transition relation $\longrightarrow$ we construct, in the standard way, an *experiment* relation $\Longrightarrow$ which does not record the silent steps. Thus, $Q \Longrightarrow Q'$ and $Q \stackrel{\tau}{\Longrightarrow} Q'$ mean $Q(\xrightarrow{\tau})^* Q'$, and $Q \stackrel{l}{\Longrightarrow} Q'$ for $l \in Act$ means $Q \Longrightarrow\xrightarrow{l}\Longrightarrow Q'$.

A state q with an infinite sequence of τ-transitions $q \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \cdots$ is called *divergent*, and a transition system is divergent if it contains a reachable divergent state, otherwise it is *convergent*.

2.2 Equivalence Relations

Definition 2 (observation equivalence). A binary relation $\mathcal{S}$ on transition systems is called a *(weak) simulation* if

$$\text{whenever } Q\mathcal{S}R \text{ and } Q \xrightarrow{\alpha} Q', \text{ then } \exists R' : R \stackrel{\alpha}{\Longrightarrow} R' \text{ and } Q'\mathcal{S}R' \ .$$

The relation $\mathcal{S}$ is a (weak) *bisimulation* if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are weak simulations. Two transition systems are *observation equivalent*, written $\approx$, if their initial states are related by a weak bisimulation.

Definition 3 (coupled simulation equivalence). A pair $(\mathcal{S}_1, \mathcal{S}_2)$ of binary relations on transition systems is called a *coupled simulation* if

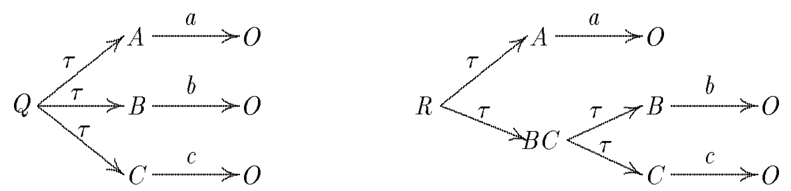
1. If $Q\mathcal{S}_1R$ then both of:
   (a) $Q \xrightarrow{\alpha} Q'$ implies $\exists R' : R \stackrel{\alpha}{\Longrightarrow} R'$ and $Q'\mathcal{S}_1R'$;
   (b) If Q is stable then $Q\mathcal{S}_2R$.
2. If $Q\mathcal{S}_2R$ then both of:
   (a) $R \xrightarrow{\alpha} R'$ implies $\exists Q' : Q \stackrel{\alpha}{\Longrightarrow} Q'$ and $Q'\mathcal{S}_2R'$;
   (b) If R is stable then $Q\mathcal{S}_1R$.

Two transition systems are *cs-equivalent*, written $=_{\mathrm{cs}}$, if their initial states are related by both elements of a coupled simulation.

Thus, $(\mathcal{S}_1, \mathcal{S}_2)$ is a coupled simulation if both $\mathcal{S}_1$ and $\mathcal{S}_2^{-1}$ are simulations which are "coupled" at the stable states.

Proposition 4. $\approx \subset =_{\mathrm{cs}}$.

The inclusion is strict as demonstrated by the following example, where $Q \not\approx R$ and $Q =_{\mathrm{cs}} R$.



This example demonstrates the significant difference between $=_{\mathrm{cs}}$ and $\approx$; the choice to preempt a must in the left-hand transition system imply a choice between b and c, while in the right-hand system there is an extra state, namely BC, where a is preempted but the choice between b and c has not been resolved. We have established the general result that $=_{\mathrm{cs}}$ on convergent systems implies testing equivalence (the simple proof of this is beyond the scope of the present paper). $=_{\mathrm{cs}}$ shares with $\approx$ the nice property that it can be established by simulations which involve case analysis over individual transitions; this is in contrast with testing equivalence which involves case analysis over sequences of transitions. We will thus use both $\approx$ and $=_{\mathrm{cs}}$ as practical proof methods for convergent systems.

Proposition 5. $=_{\mathrm{cs}}$ is an equivalence relation on convergent transition systems.

We omit the straightforward proof. $=_{\mathrm{cs}}$ is not transitive on divergent transition systems; we will however only use it on convergent systems.

2.3 Processes and Parallel Composition

Definition 6 (process). A *process* P is a rooted transition system with an associated *synchronization sort*, $L(P)$, which is a set of actions such that each state Q of P is either

1. *waiting*, meaning that $\forall \alpha : Q \xrightarrow{\alpha}$ implies $\alpha \in L(P)$, or
2. *external*, meaning that $\neg\exists\alpha \in L(P) : Q \xrightarrow{\alpha}$.

The synchronization sort of a process will be significant only when processes are composed in parallel. Intuitively, when P occurs in a parallel composition then the actions in $L(P)$ represent synchronizations with other entities in the composition, while actions not in $L(P)$, which we call *external* for P, are interactions with the environment outside the parallel composition. We restrict attention to processes where external and internal interactions never compete, thus a state must either be waiting or external. (This is not a vital restriction, our protocols and proofs can be extended to allow competition between internal and external synchronizations, see [17].)

It will sometimes be convenient to identify a process with its underlying transition system. Thus, if P and P' have transition systems Q and Q' respectively we use $P \xrightarrow{\alpha} P'$ to mean that $Q \xrightarrow{\alpha} Q'$ and that P and P' have the same synchronization sort. Similarly we will also use $P \stackrel{\alpha}{\Longrightarrow} P'$ and $P \xrightarrow{\alpha}$ on processes, and say that a process is divergent if its transition system is divergent. $P =_{\mathrm{cs}} P'$ will mean that $L(P) = L(P')$ and $Q =_{\mathrm{cs}} Q'$.

A process or transition system P is called *synchronization deterministic* (or *s-deterministic* for short) if for all derivatives P' and all actions $l \in L(P)$ it holds that whenever $P' \xrightarrow{l} P_1$ and $P' \xrightarrow{l} P_2$ then $P_1 = P_2$. We will here confine interest to s-deterministic client processes. Again, this is a technical convenience which simplifies our reasoning but it is not crucial for the results.

Definition 7 (parallel composition). The parallel composition $P_1 \mid\mid \cdots \mid\mid P_n$, sometimes written $\prod_{i=1}^{n} P_i$, of processes $P_1, \ldots, P_n$ is the rooted transition system whose states and transitions can be generated from the following rules:

1. For all $i, 1 \leq i \leq n$ and all $\alpha \notin L(P_i)$: if $P_i \xrightarrow{\alpha} P_i'$ then
$$P_1 \mid\mid \cdots \mid\mid P_i \mid\mid \cdots \mid\mid P_n \xrightarrow{\alpha} P_1 \mid\mid \cdots \mid\mid P_i' \mid\mid \cdots \mid\mid P_n \ .$$
2. For all l: if $\forall i : (l \in L(P_i)$ implies $P_i \xrightarrow{l} P_i')$, then
$$P_1 \mid\mid \cdots \mid\mid P_n \xrightarrow{\tau} P_1' \mid\mid \cdots \mid\mid P_n'$$
(where P_j' is defined to be P_j whenever $l \notin L(P_j)$).

Clause 2 here corresponds to an internal synchronization at l; this involves all processes with l in their sort and results in a silent step. Note that our parallel composition operator is not associative—formally, the result of a parallel composition is a rooted transition system and not a process. We will need the following simple results:

Lemma 8. If $P =_{\mathrm{cs}} P'$ then $P_1 \mid\mid \cdots \mid\mid P_n \mid\mid P \ =_{\mathrm{cs}} \ P_1 \mid\mid \cdots \mid\mid P_n \mid\mid P'$.

Proof: Omitted.

Lemma 9. If $P =_{\mathrm{cs}} P'$ and both P and $P_1 \mid\mid \cdots \mid\mid P_n \mid\mid P'$ are convergent, then also $P_1 \mid\mid \cdots \mid\mid P_n \mid\mid P$ is convergent.

Proof: Omitted.

3 Central Implementation

3.1 Implementation Transformation

Let P be a client process and i its index, i.e. a natural number, in a specification. Then $\mathcal{R}_i(P)$ is a process which, instead of participating in multiway synchronizations on actions in $L(P)$, interacts with a synchronizer on the following actions:

$req_i(L)$ where L is a set of actions; this is a request from the process to the synchronizer to participate in any action in L.

$conf_i(l)$ where l is an action; this is a confirmation from the synchronizer to the process that a multiway synchronization on l has been scheduled.

Note that the index of the client (the number i) is part of the actions; this ensures that all interactions will be binary, namely between one client process and the synchronizer.

Definition 10 (implementation transformation). Let P be any process and i a natural number. The processes $\mathcal{R}_i(P)$ and $\mathcal{W}_i(P)$ are defined to have the following transitions:

1. If $P \xrightarrow{\alpha} Q$ and $\alpha \notin L(P)$ then $\mathcal{R}_i(P) \xrightarrow{\alpha} \mathcal{R}_i(Q)$.
2. If P is waiting and $L = \{l : P \xrightarrow{l}\}$ then $\mathcal{R}_i(P) \overset{req_i(L)}{\longrightarrow} \mathcal{W}_i(P)$.
3. If $P \xrightarrow{l} P'$ then $\mathcal{W}_i(P) \overset{conf_i(l)}{\longrightarrow} \mathcal{R}_i(P')$.

Further, $L(\mathcal{R}_i(P))$ and $L(\mathcal{W}_i(P))$ are both defined to be

$$\{conf_i(l) : l \in L(P)\} \cup \{req_i(L) : L \subseteq L(P)\} \ .$$

The purpose of $\mathcal{R}_i$ is to implement a small protocol for interaction with the synchronizer. If P is in a waiting state where a set of actions L are enabled (i.e., $P \xrightarrow{l}$ for all l in L), then L represents the multiway synchronizations in which P wants to participate. So the transformed processes $\mathcal{R}_i(P)$ can do the action $req_i(L)$, signifying "Process i requests to do a synchronization in L", and thereby reach state $\mathcal{W}_i(P)$. This state can continue with a $conf_i(l)$ action, signifying that "the multiway synchronization on l has been achieved". The transformation leaves external action unaffected.

The notation "$req_i(L)$" and "$conf_i(l)$" suggests that l and L are "data" transferred in the interaction. There is no need to introduce data transfer in a formal way; the actions are atomic and there is one such action for each appropriate l and L.

3.2 Central Synchronizer

The central synchronizer is a monolithic entity which interacts with the transformed client processes. A state of the central synchronizer is written $\textsc{csynch}(RQ, CF)$ where both RQ and CF are sets of offers. An *offer* is a pair (i, l) where $l \in L(P_i)$. The parameters mean:

RQ**:** If $(i, l) \in RQ$ then client process i has requested to participate in a synchronization on l.

CF**:** If $(i, l) \in CF$ then a synchronization on l has been scheduled and the synchronizer shall confirm this to process i.

Definition 11 (central synchronizer). The *central synchronizer* for a sequence $P_1, \ldots, P_n$ of processes is a process with synchronization sort

$$\bigcup_i \{conf_i(l) : l \in L(P_i)\} \cup \{req_i(L) : L \subseteq L(P_i)\} \ .$$

The states of the central synchronizer are of the form $\textsc{csynch}(RQ, CF)$, and the transitions from $\textsc{csynch}(RQ, CF)$ to $\textsc{csynch}(RQ', CF')$ are given by the following table:

precondition	action	RQ'	CF'
$\neg\exists l : (i, l) \in RQ \cup CF$	$req_i(L)$	$RQ \cup \{(i, l) : l \in L\}$	–
$S \subseteq RQ \wedge Cand(S)$	τ	$RQ - \{(i, l) : \exists k : (i, k) \in S\}$	$CF \cup S$
$(i, l) \in CF$	$conf_i(l)$	–	$CF - \{(i, l)\}$

In this and subsequent similar tables each line corresponds to a set of transitions, namely the transitions which satisfy the precondition. The columns "RQ'" and "CF'" give the resulting state $\text{CSYNCH}(RQ', CF')$; the symbol "–" means that the value does not change. In the second line, $Cand(S)$ means that the set of offers S is a *candidate synchronization*, i.e. that all offers in S have the same action l and $S = \{(i, l) : l \in L(P_i)\}$.

The initial state of the central synchronizer is $\text{CSYNCH}(\{\}, \{\})$ and we will abbreviate it as CSYNCH_0.

The first transition type in the table corresponds to process i requesting synchronizations on actions in L. The second type corresponds to the selection of a multiway synchronization at some particular action l. This is an internal step within the synchronizer, which selects (among the offers in RQ) a "candidate synchronization", i.e. a set of offers where all required processes—processes with l in the synchronization sort—are present. The third type corresponds to the synchronizer confirming to process i that the synchronization on l has taken place.

Definition 12 (central implementation). The *central implementation* of processes $P_1, \ldots, P_n$ is written $\mathcal{C}(P_1, \ldots, P_n)$ and is defined to be

$$\mathcal{R}_1(P_1) \mid\mid \cdots \mid\mid \mathcal{R}_n(P_n) \mid\mid \text{CSYNCH}_0$$

(where CSYNCH_0 is the initial state of the central synchronizer for $P_1, \ldots, P_n$).

We have established two correctness properties of the central implementation (the proofs are omitted for lack of space). The first is:

Theorem 13. For s-deterministic processes $P_1, \ldots, P_n$:

$$P_1 \mid\mid \cdots \mid\mid P_n \;\approx\; \mathcal{C}(P_1, \ldots, P_n) \ .$$

This guarantees that both sides can mimick each other indefinitely as far as external actions are concerned. However, it does not guarantee anything about silent steps; in particular it does not imply divergence freedom. So our second correctness property is:

Theorem 14. For s-deterministic processes $P_1, \ldots, P_n$: if $P_1 \mid\mid \cdots \mid\mid P_n$ is convergent, then $\mathcal{C}(P_1, \ldots, P_n)$ is convergent.

4 Distributed Implementation

4.1 Overview

The distributed synchronizer for processes $P_1, \ldots, P_n$ consists of *mediators*, one for each process, and *ports*, one for each action $l \in \bigcup_i L(P_i)$. Mediator i synchronizes with process P_i via req_i and $conf_i$ actions. Ports and mediators communicate via *messages*. Messages are transferred asynchronously from a sender process to a receiver process. We model this as (two-way) synchronizations where the sender and receiver are represented as indices of actions and where receivers are always prepared to accept a message (i.e. receptions are enabled in all states). For example, the action $ready_{i,l}$ represents the transmission of a *ready* message from mediator i to port l. Thus for mediator i this action corresponds to a transmission, and for port l it corresponds to a reception.

Port l is responsible for establishing multiway synchronizations at l. The algorithm for establishing synchronizations is a *two-phase locking* algorithm, and the following messages are used in communication between mediators and ports.

$ready_{i,l}$ is an indication from mediator i to port l that process P_i has requested, by a $req_i(L)$ synchronization where $l \in L$, to synchronize at l.

$lock_{l,i}$ is a request from port l to lock mediator i.

$yes_{i,l}$ is a positive response, from mediator i to port l, to a lock request.

$no_{i,l}$ is a negative response, from mediator i to port l, to a lock request.

$commit_{l,i}$ is an indication, from port l to mediator i, that P_i has been scheduled for a multiway synchronization at l.

$abort_{l,i}$ is an indication, from port l to mediator i, that a multiway synchronization at l could not be established.

Informally, the algorithm works as follows. After a $req_i(L)$ synchronization mediator i sends to each port $l \in L$ a $ready_{i,l}$ message. When a port l has received sufficient *ready* messages to determine that a multiway synchronization is possible between a set of processes I, it can initiate the locking procedure.

In the first phase, port l attempts to lock each mediator in I. To lock a mediator i, the port sends a *lock* message and then awaits a reply. If the mediator replies by a *yes* message, the mediator is thereby locked and the port continues to lock the next mediator. When all mediators have been locked, the second phase of the locking procedure is commenced: to each mediator $i \in I$, a $commit_{l,i}$ message is sent. When mediator i has received a $commit_{l,i}$ message, it signals to process P_i that it participated in a multiway synchronization at l, by a $conf_i(l)$ synchronization.

It can happen that a mediator i cannot be locked for a synchronization at a port l. This happens, for instance, when P_i has already participated in another multiway synchronization. Then mediator i rejects the lock request from l by responding with a *no* message. If port l at that point has managed to lock some mediators, it must unlock them by sending an *abort* message to each of them.

A mediator can only be locked by one port at a time. Should a mediator receive lock requests while being locked, the mediator does not respond until it receives a *commit* or *abort*. If the mediator receives a *commit* (an indication that a synchronization was established), the mediator rejects all other lock requests (by sending *no* messages). Should the mediator receive an *abort* (an indication that a synchronization could not be established), the mediator may respond with a *yes* to any delayed lock request. To prevent deadlocks, mediators are locked in the order defined by their indices, so whenever a port should lock mediators i and j, and $i < j$, it will lock i before j.

4.2 Mediators

A state of a mediator for process i is written $\textsc{med}_i(R, Q, V, c)$ where R, Q and V are sets of actions and c is an action or the special symbol $\perp$. The parameters have the following meanings:

R is the indices of ports to which *ready* messages should be sent. If the last request from the mediator's client was a $req_i(L)$-synchronization, then $R \subseteq L$.

Q is the indices of ports that have sent *lock* requests to the mediator and to which the mediator has not yet responded. If $l \in Q$, then a $lock_{l,i}$-synchronization has taken place and either a $yes_{i,l}$- or a $no_{i,l}$-synchronization will take place.

V is the valid actions, i.e. the actions the client currently is prepared to do. If $V \neq \{\}$ then the last request from the client process was $req_i(V)$.

c is the action to which the mediator has committed itself, that is, to which the mediator has sent a *yes* message and from which the mediator expects to receive a *commit* or an *abort* message. If the mediator has not committed itself, then c is $\perp$.

If c is different from $\perp$ and V is empty, then the mediator has received a *commit* message and should synchronize with its process by a *conf* action.

Definition 15 (mediator). Let K be a synchronization sort and i a natural number. Then the *mediator* for i and K is a process with synchronization sort

$$\bigcup_{l \in K} \{ready_{i,l}, lock_{l,i}, yes_{i,l}, no_{i,l}, commit_{l,i}, abort_{l,i}\} \ .$$

The states of a mediator are of the form $\text{MED}_i(R, Q, V, c)$, and the transitions from $\text{MED}_i(R, Q, V, c)$ to $\text{MED}_i(R', Q', V', c')$ are given by the following table:

precondition	action	R'	Q'	V'	c'
$c = \perp \wedge V = \{\}$	$req_i(L)$	L	–	L	–
$l \in R$	$ready_{i,l}$	$R - \{l\}$	–	–	–
–	$lock_{l,i}$	–	$Q \cup \{l\}$	–	–
$l \in Q \wedge l \in V \wedge c = \perp$	$yes_{i,l}$	–	$Q - \{l\}$	–	l
$l \in Q \wedge l \notin V$	$no_{i,l}$	–	$Q - \{l\}$	–	–
–	$commit_{l,i}$	$\{\}$	–	$\{\}$	–
–	$abort_{l,i}$	–	–	–	$\perp$
$V = \{\} \wedge c \neq \perp$	$conf_i(c)$	–	–	–	$\perp$

The initial state is $\text{MED}_i(\{\}, \{\}, \{\}, \perp)$, i.e. all action sets are empty and c is $\perp$.

Note that actions corresponding to asynchronous receptions have the trivial precondition "–" which always holds.

4.3 Ports

A state of a port for multiway synchronization action l is written $\text{PORT}_l(T, N, w, Y, A)$ where T, N, Y and A are sets of process indices, i.e. natural numbers, and w is a process index or the special symbol $\perp$. The parameters have the following meanings:

T is the indices of mediators that have sent *ready* messages to the port, but are not currently involved in a locking procedure of the port.

N is the indices of mediators that have sent *ready* messages to the port, and that the port has decided to lock for a synchronization, but has not yet locked.

w is the index of the mediator that the port is waiting for. If $w = i$ then the port has sent a *lock* message to mediator i, and is waiting for a a *yes* or a *no* message from i. In states where the port is not waiting for any mediator, $w = \perp$.

Y is the indices of mediators that have responded with *yes* messages to lock requests, and thereby have committed themselves to a synchronization at l. The mediator will eventually send a *commit* or an *abort* message to each mediator in Y.

A is the indices of mediators that have responded with *yes* messages to lock requests, and to which the mediator should send *abort* messages.

If N is non-empty, or w is different from $\perp$, then the port is in the first phase of the locking procedure. If Y is non-empty, N is empty and $w = \perp$, then the port is in the second phase of the locking procedure and should send *commit* messages to all mediators in Y.

Definition 16 (port). Let l be an action and I a set of natural numbers. Then the *port* for l and I is a process with synchronization sort

$$\bigcup_{i \in I} \{ready_{i,l}, lock_{l,i}, yes_{i,l}, no_{i,l}, commit_{l,i}, abort_{l,i}\} \ .$$

The states of a port are of the form $\text{PORT}_l(T, N, w, Y, A)$, and the transitions from $\text{PORT}_l(T, N, w, Y, A)$ to $\text{PORT}_l(T', N', w', Y', A')$ are collected in the following table:

precondition	action	T'	N'	w'	Y'	A'
–	$ready_{i,l}$	$T \cup \{i\}$	–	–	–	–
$Cand(\{(i,l) : i \in T\})$ $\wedge N = \{\} \wedge w = \perp$	τ	$\{\}$	T	–	–	–
$N \neq \{\} \wedge w = \perp \wedge i = least(N)$	$lock_{l,i}$	–	$N - \{i\}$	i	–	–
–	$yes_{i,l}$	–	–	$\perp$	$Y \cup \{i\}$	–
–	$no_{i,l}$	$T \cup N$	$\{\}$	$\perp$	$\{\}$	Y
$N = \{\} \wedge w = \perp \wedge i \in Y$	$commit_{l,i}$	–	–	–	$Y - \{i\}$	–
$i \in A$	$abort_{l,i}$	$T \cup \{i\}$	–	–	–	$A - \{i\}$

Here $least(N)$ gives the least element of the set of natural numbers N. The initial state is $\text{PORT}_l(\{\}, \{\}, \perp, \{\}, \{\})$, i.e. all index sets are empty and w is $\perp$.

4.4 Distributed Synchronizer

Definition 17 (distributed synchronizer). The *distributed synchronizer* for a sequence $P_1, \ldots, P_n$ of processes is a process with synchronization sort

$$\bigcup_i \{conf_i(l) : l \in L(P_i)\} \ \cup \ \{req_i(L) : L \subseteq L(P_i)\} \ .$$

Let $\widetilde{L} = \bigcup_i L(P_i)$. The states of the distributed synchronizer are of the form

$$\prod_{i=1}^{n} \text{MED}_i(R_i, Q_i, V_i, c_i) \ || \ \prod_{l \in \widetilde{L}} \text{PORT}_l(T_l, N_l, w_l, Y_l, A_l)$$

where $\textsc{med}_i(R_i, Q_i, V_i, c_i)$ is the mediator for i and $\widetilde{L}$ and $\textsc{port}_l(T_l, N_l, w_l, Y_l, A_l)$ is the port for l and $\{1, \ldots, n\}$. The initial state of the distributed synchronizer, written $\textsc{dsynch}_0$, is the parallel composition of the initial states of its mediators and ports.

Definition 18 (distributed implementation). The *distributed implementation* of processes $P_1, \ldots, P_n$ is written $\mathcal{D}(P_1, \ldots, P_n)$ and is defined to be

$$\mathcal{R}_1(P_1) \mid\mid \cdots \mid\mid \mathcal{R}_n(P_n) \mid\mid \textsc{dsynch}_0$$

(where $\textsc{dsynch}_0$ is the initial state of the distributed synchronizer for $P_1, \ldots, P_n$).

So $\mathcal{D}(P_1, \ldots, P_n)$ differs from $\mathcal{C}(P_1, \ldots, P_n)$ only in that $\textsc{dsynch}_0$ replaces $\textsc{csynch}_0$.

4.5 Correctness Properties

Theorem 19. For s-deterministic processes $P_1, \ldots, P_n$:

$$P_1 \mid\mid \cdots \mid\mid P_n \quad =_{\mathrm{cs}} \quad \mathcal{D}(P_1, \ldots, P_n) \ .$$

To prove this theorem it suffices, in view of Proposition 4, Lemma 8 and Theorem 13, to establish:

Lemma 20. $\textsc{csynch}_0 =_{\mathrm{cs}} \textsc{dsynch}_0$.

To prove this lemma we need a coupled simulation between the central and the distributed synchronizer:

Definition 21. Let $CS = \textsc{csynch}(RQ, CF)$ range over derivatives of $\textsc{csynch}_0$ for a sequence of processes $P_1, \ldots, P_n$ and

$$DS \quad = \quad \prod_{i=1}^{n} \textsc{med}_i(R_i, Q_i, V_i, c_i) \mid\mid \prod_{l \in \widetilde{L}} \textsc{port}_l(T_l, N_l, w_l, Y_l)$$

range over derivatives of $\textsc{dsynch}_0$ for the same process sequence. Let *Reqs* and *Confs* be the following functions from derivatives of $\textsc{dsynch}_0$ to offer sets:

$$\begin{aligned} \mathit{Reqs}(DS) &\stackrel{\text{def}}{=} \{(i, l) : l \in V_i \wedge \neg(\exists k : i \in Y_k \wedge N_k = \{\} \wedge w_k = \bot)\}, \\ \mathit{Confs}(DS) &\stackrel{\text{def}}{=} \{(i, l) : (c_i = l \wedge V_i = \{\}) \vee (i \in Y_l \wedge N_l = \{\} \wedge w_l = \bot)\} \ . \end{aligned}$$

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be relations between derivatives of $\textsc{csynch}_0$ and $\textsc{dsynch}_0$ as follows:

$$\begin{aligned} \mathcal{S}_1 &= \left\{ (CS, DS) : \begin{pmatrix} RQ = \mathit{Reqs}(DS) \wedge CF = \mathit{Confs}(DS) \wedge \\ \forall i, l : R_i = N_l = Y_l = A_l = \{\} \wedge w_l = \bot \end{pmatrix} \right\}, \\ \mathcal{S}_2 &= \{(CS, DS) : RQ = \mathit{Reqs}(DS) \wedge CF = \mathit{Confs}(DS)\} \ . \end{aligned}$$

Obviously $(\textsc{csynch}_0, \textsc{dsynch}_0) \in \mathcal{S}_1 \cap \mathcal{S}_2$. So in order to prove Lemma 20 we have to prove:

Lemma 22. $(\mathcal{S}_1, \mathcal{S}_2)$ is a coupled simulation.

The second correctness criterion guarantees that the distributed implementation is convergent:

Theorem 23. For s-deterministic processes $P_1, \ldots, P_n$: if $P_1 \mid\mid \cdots \mid\mid P_n$ is convergent, then $\mathcal{D}(P_1, \ldots, P_n)$ is convergent.

To prove this theorem it is enough, by Lemma 9 and Theorem 14, to establish the following lemma:

Lemma 24. DSYNCH_0 is convergent.

We finally give an example that demonstrates why the distributed synchronizer and the central synchronizer are not observation equivalent. Assume a specification with three client processes P_1, P_2, and P_3 and three possible (two-way) synchronizations: P_1 and P_2 can synchronize with each other at both a and b, and P_2 and P_3 can synchronize with each other at c. Suppose all three clients have requested to synchronize, then all three synchronizations are enabled but since P_2 must participate in every synchronization only one can occur.

The central synchronizer selects, by an internal transition, one of the three possible synchronizations and then confirms the selected synchronization to the clients. Hence any state in the central synchronizer is either before the choice has been made, in which case all three synchronizations are still possible, or after the choice, in which case only the chosen synchronization is possible.

In the distributed synchronizer, on the other hand, the choice may be resolved incrementally by first excluding the possibility of an a synchronization (cf. the example in Section 2.2). Consider the state where ports a and b have requested to lock mediator 1, and port c has requested to lock mediator 2. If mediator 1 sends a $yes_{1,b}$ to port b then synchronizations on b and c are still possible. But a synchronization on port a can only happen if b relinquishes its claim on mediator 1 (through $abort_{b,1}$); this only happens if mediator 2 refuses b ($no_{2,b}$), and the only possible reason for such a refusal is that c has been selected ($yes_{2,c}$), whence a c synchronization (and not a) will occur.

5 Conclusions

We have presented a new way to formulate and prove correctness properties of distributed algorithms through a new equivalence relation $=_{\text{cs}}$, coupled simulation equivalence (cs-equivalence), on transitions systems. It is weaker than observation equivalence but stronger than e.g. testing equivalence, and it is motivated by its practical applicability; it can, in contrast to testing equivalence, be established by case analysis over individual transitions. Although it is not suitable for divergent transition systems (systems with infinite sequences of internal transitions), we believe that the cs-equivalence is useful, for instance for implementation-oriented verifications where divergence often is regarded as harmful (a divergent system has potential "livelocks").

We have demonstrated the use of the cs-equivalence by verifying a strategy for implementing multiway synchronization. We started from a specification of a set of processes, composed in parallel, that communicate by multiway synchronizations.

The specification was transformed into an implementation consisting of another set of processes and a synchronizer which are composed in parallel. We first defined a *central* synchronizer, such that the implementation with the central synchronizer is convergent and observation equivalent to its specification. We then defined a *distributed* synchronizer, which is convergent and cs-equivalent to the central synchronizer. Cs-equivalence is preserved by parallel composition, wherefore we conclude that the implementation with the distributed synchronizer is cs-equivalent to the specification.

The fact that we have used an equivalence (and not an asymmetric implementation preorder such as the testing preorder) means that our implementation strategy does not exclude any synchronization possibilities in the specification. Thus the implementation will inherit all global properties such as fairness or deadlock freedom from the specification.

Other proposals for distributed algorithms for multiway synchronization can be found in [2, 6, 10, 11, 14, 15]. We have given priority to verification and thus chosen a simple algorithm, so the algorithm is not optimal, in terms of the number of messages required to establish a synchronization, and there are several possible improvements of the algorithm. For a discussion on this topic we refer to [17], where one of the authors presents and verifies a strategy for generating distributed implementations from LOTOS specifications, including process creation and communication with environment.

Mitchell [13] gives a transformation from CCS expressions to distributed implementations with a distributed algorithm to implement the CCS (two-way) synchronization. A result similar to ours was obtained; the implementation is not observation equivalent to the transformed CCS expression. The distributed algorithm is a busy-wait algorithm and, consequently, it causes divergence in the implementation so testing equivalence does not hold either. Mitchell therefore introduces *weak-must* testing equivalence, which is insensitive to certain kinds of divergence, but concludes that its lack of "bisimulation style" proof technique makes it difficult to use.

We have left the modal logic characteristics and algebraic theory of the cs-equivalence as future work. The distributed algorithm we have proposed can be improved in several ways; we believe that the results of such improvements would be (at least) cs-equivalent to the algorithm given in this paper, and therefore also correct according to our criteria. Our investigation could also consider distributed algorithms for multiway synchronization proposed by others.

References

1. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.
2. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, 1989.

4. A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, July 1987.
5. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1,2):83–133, 1984.
6. P. Eklund. Synchronizing multiple processes in common handshakes. Reports on Computer Science and Mathematics 39, Åbo Akademi, Finland, 1984.
7. M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, November 1989.
8. N. Francez and I. R. Forman. Superimposition for interacting processes. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 1990.
9. N. Francez, B. Hailpern, and G. Taubenfeld. Script: A communication abstraction mechanism. *Science of Computer Programming*, 6(1):35–88, January 1986.
10. Y.-J. Joung and S. A. Smolka. Coordinating first-order multiparty interactions. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 209–220, Orlando, Florida, January 1991.
11. D. Kumar. An implementation of N-party synchronization using tokens. In *Proc. 10th International Conference on Distributed Computing Systems*, pages 320–327. IEEE, 1990.
12. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
13. K. Mitchell. *Implementations of Process Synchronization and their Analysis*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1986.
14. M. H. Park and M. Kim. A distributed synchronization scheme for fair multi-process handshakes. *Information Processing Letters*, 34(3):131–138, April 1990.
15. S. Ramesh. A new and efficient implementation of multiprocess synchronization. In *Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 387–401. Springer-Verlag, 1987.
16. S. Ramesh and S. L. Mehndiratta. A methodology for developing distributed programs. *IEEE Transactions on Software Engineering*, SE-13(8):967–976, August 1987.
17. P. Sjödin. *From LOTOS Specifications to Distributed Implementations*. PhD thesis, Department of Computer Systems, Uppsala University, Uppsala, Sweden, 1991.

This article was processed using the LaTeX macro package with LLNCS style