Symmetric and Synchronous Communication in Peer-to-Peer Networks

Andreas Witzel[1,2]

[1] University of Amsterdam, Plantage Muidergracht 24, 1018TV Amsterdam
[2] CWI, Kruislaan 413, 1098SJ Amsterdam, The Netherlands

Abstract. Motivated by distributed implementations of game-theoretical algorithms, we study symmetric process systems and the problem of attaining common knowledge between processes. We formalize our setting by defining a notion of peer-to-peer networks[3] and appropriate symmetry concepts in the context of Communicating Sequential Processes (*CSP*) [1]. We then prove that *CSP* with input and output guards makes common knowledge in symmetric peer-to-peer networks possible, but not the restricted version which disallows output statements in guards and is commonly implemented. Our results extend [2].
An extended version is available at `http://arxiv.org/abs/0710.2284` .

1 Introduction

1.1 Motivation

Our original motivation comes from the distributed implementation of game-theoretical algorithms (see e.g. [3] for a discussion of the interface between game theory and distributed computing). Two important issues in the domain of game theory have always been knowledge, especially common knowledge, and symmetry between the players, also called anonymity. We will describe these issues and the connections to distributed computing in the following two paragraphs, before we motivate our choice of process calculus and the overall goal of the paper.

Common Knowledge and Synchronization. The concept of common knowledge has been a topic of much research in distributed computing as well as in game theory. When do processes or players "know" some fact, mutually know that they know it, mutually know that they mutually know that they know it, and so on ad infinitum? And how crucial is the difference between arbitrarily, but finitely deep mutual knowledge and the limit case of real common knowledge?

In distributed computing, the classical example showing that the difference is indeed essential is the scenario of Coordinated Attack [4]. The game-theoretical incarnation of the underlying issue is the Electronic Mail Game [5,6].

[3] Please note that we are *not* dealing with fashionable incarnations such as file-sharing networks, but merely use this name for a mathematical notion of a network consisting of directly connected peers "treated on an equal footing", i.e. not having a client-server structure or otherwise pre-determined roles.

The basic insight of these examples is that two agents that communicate through an unreliable channel can never achieve common knowledge, and that their behavior under finite mutual knowledge can be strikingly different.

These issues are analyzed in detail in [7], in particular in a separately published part [8], including a variant where communication is reliable, but message delivery takes an unknown amount of time. Even in that variant, it is shown that only finite mutual knowledge can be attained.

However, in a synchronous communication act, sending and receiving of a message is, by definition, performed simultaneously. In that way, the agents obtain not only the pure factual information content of a message, but the sender also knows that the receiver has received the message, the receiver knows that the sender knows that, and so on ad infinitum. The communicated information immediately becomes common knowledge.

Attaining common knowledge and achieving synchronization between processes are thus closely related. Furthermore, synchronization is in itself an important subject, see e.g. [9].

Symmetry and Peer-to-peer Networks. In game theory, players are assumed to be anonymous and treated on an equal footing, in the sense that their names do not play a role and no single player is a priori distinguished from the others [10,11].

In distributed computing, too, this kind of symmetry between processes is desirable to avoid a predetermined assignment of roles to processes and improve fault tolerance, modularity, and load balancing [12].

We will consider symmetry on two levels. Firstly, the communication network used by the processes should be symmetric to some extent in order not to discriminate single processes a priori on a topological level; we will formalize this requirement by defining peer-to-peer networks. Secondly, processes in symmetric positions of the network should have equal possibilities of behavior; this we will formalize in a semantic symmetry requirement on the possible computations.

Communicating Sequential Processes (CSP). Since we are interested in synchronization and common knowledge, a process calculus which supports synchronous communication through primitive statements clearly has some appeal. We will focus on one of the prime examples of such calculi, namely *CSP*, introduced in [1] and revised in [13,14], since it supports synchronous communication through primitive statements. Furthermore, it has been implemented in various programming languages, among the best-known of which is Occam [15]. We thus have at our disposal a theoretical framework and programming tools which in principle could give us synchronization and common knowledge “for free”.

However, symmetric situations are a reliable source of impossibility results [16]. In particular, the restricted dialect CSP_{in} which was, for implementation issues [17], chosen to be the theoretical foundation of Occam is provably [2] less expressive than the general form, called $CSP_{i/o}$. CSP_{in} has been used throughout the history of Occam, up to and including its latest variant Occam-π [18].

This generally tends to be the case for implementations of *CSP*, one notable exception being a very recent extension [19] of JCSP[4] to $CSP_{i/o}$.

Some of the resulting restrictions of CSP_{in} can in practice be overcome by using helper processes such as buffers [20]. Our goal therefore is to formalize the concepts mentioned above, extend the notion of peer-to-peer networks by allowing helper processes, and examine whether synchronization is feasible in either of these two dialects of *CSP*. We will come to the result that, while the problem can (straightforwardly) be solved in $CSP_{i/o}$, it is impossible to do so in CSP_{in}. Our setting thus provides an argument in favor of the former's rare and admittedly more complicated implementations, such as JCSP.

1.2 Related Work

This paper builds upon [2], where a semantic characterization of symmetry for *CSP* is given and fundamental possibility and impossibility results for the problem of electing a leader in networks of symmetric processes are proved for various dialects of *CSP*. More recently, this has inspired a similar work on the more expressive π-calculus [21], but the possibility of adding helper processes is explicitly excluded.

There has been research on how to circumvent problems resulting from the restrictions of CSP_{in}. However, solutions are typically concerned only with the factual content of messages and do not preserve synchronicity and the common knowledge creating effect of communication, for example by introducing buffer processes [20].

The same focus on factual information holds for general research on synchronizing processes with asynchronous communication. For example, in [9] one goal is to ensure that a writing process knows that no other process is currently writing; whether this is common knowledge, is not an issue.

The problem of Coordinated Attack has also been studied for models in which processes run synchronously [16]; however, the interesting property of *CSP* is that processes run asynchronously, which is more realistic in physically distributed systems, and synchronize only at communication statements.

Since we focus on the communication mechanisms, the results will likely carry over to other formalisms with synchronous communication facilities comparable to those of *CSP*.

1.3 Overview of the Paper

In Sect. 2 we give a short description of *CSP* and the dialects that we are interested in, define some basic concepts from graph theory, and recall the required notions and results for symmetric electoral systems from [2].

In Sect. 3 we formally define the problem of pairwise synchronization that we will examine, give a formalization of peer-to-peer networks which ensures a certain kind of symmetry on the topological level, and describe in what ways

[4] A Java$^{\text{TM}}$ implementation and extension of *CSP*.

we want to allow them to be extended by helper processes. We adapt a concept from [2] to capture symmetry on the semantic level.

Section 4 contains two positive results and the main negative result saying that pairwise synchronization of peer-to-peer networks of symmetric processes is not obtainable in CSP_{in}, even if we allow extensions through buffers or similar helper processes. Section 5 concludes.

2 Preliminaries

2.1 CSP

A *CSP process* consists of a sequential program which can use, besides the usual *local* statements, two *communication* statements:

- $P\,!\,message$ to send (output) the given message to process P;
- $P\,?\,variable$ to receive (input) a message from P into the given local variable.

Communication is *synchronous*, i.e., send and receive instructions block until their counterpart is available, at which point the message is transferred and both participating processes continue execution. Note that the communication partner P is statically defined in the program code.

There are two *control structures* (see Fig. 1). Each guard is a Boolean expression over local variables (which, if omitted, is taken to be true), optionally followed by a communication statement. A guard is *open* if its Boolean expression evaluates to true and its communication statement, if any, can currently be performed. A guard is *closed* if its Boolean expression evaluates to false. Note that a guard can thus be neither open nor closed.

```
[  $guard_1 \rightarrow command_1$
□ $guard_2 \rightarrow command_2$
...
□ $guard_k \rightarrow command_k$  ]
```

(a) Non-deterministic selection

```
*[  $guard_1 \rightarrow command_1$
 □ $guard_2 \rightarrow command_2$
 ...
 □ $guard_k \rightarrow command_k$  ]
```

(b) Non-deterministic repetition

Fig. 1: Control structures in *CSP*.

The selection statement *fails* and execution is aborted if all guards are closed. Otherwise execution is suspended until there is at least one open guard. Then one of the open guards is selected non-deterministically, the required communication (if any) performed, and the associated command executed.

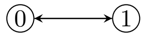
The repetition statement keeps waiting for, selecting, and executing open guards and their associated commands until all guards are closed, and then exits normally; i.e., program execution continues at the next statement.

We will sometimes use the following abbreviation to denote multiple branches of a control structure (for some finite set X): $\Box_{x \in X}\ guard_x \rightarrow command_x$

Various dialects of *CSP* can be distinguished according to what kind of communication statements are allowed to appear in guards. Specifically, in CSP_{in} only input statements are allowed, and in $CSP_{i/o}$ both input and output statements are allowed (within the same control structure). For technical reasons, CSP_{in} has been suggested from the beginning [1] and is indeed commonly used for implementations, as mentioned in Sect. 1.1.

Definition 1. *A* communication graph *(or* network*) is a directed graph without self-loops. A* process system *(or simply* system*) $\mathcal{P}$ with communication graph $G = (V, E)$ is a set of component processes $\{P_v\}_{v \in V}$ such that for all $v, w \in V$, if the program run by P_v (resp. P_w) contains an output command to P_w (resp. input command from P_v) then $(v, w) \in E$. In that case we say that G* admits *$\mathcal{P}$. We identify vertices v and associated processes P_v and use them interchangeably.*

Example 1. Figure 2 shows a simple network G with the vertex names written inside the vertices, and a $CSP_{i/o}$ program run by two processes which make up a system $\mathcal{P} := \{P_0, P_1\}$. Obviously, G admits $\mathcal{P}$. The intended behavior is that the processes send each other, in non-deterministic order, a message containing their respective process name.



(a) Network G

```
  $recd$ := false
  $sent$ := false
*[  $\neg recd \wedge P_{i+1}\,?\,x \rightarrow recd$ := true
  $\square\ \neg sent \wedge P_{i+1}\,!\,i \rightarrow sent$ := true  ]
```

(b) Program of process P_i

Fig. 2: Network and program run by P_0 and P_1 in Example 1. Addition of process names here and in all further example programs is modulo 2.

Definition 2. *A* state *of a system $\mathcal{P}$ is the collection of all component processes' (local) variables together with their current execution positions. A* computation step *is a transition from one state to another, involving either one component process executing a local statement, or two component processes jointly executing a pair of matching (send and receive) communication statements. The valid computation steps are determined by the state of the system.*

A computation *is a maximal sequence of valid computation steps, i.e. a sequence which is not a prefix of any other sequence of valid computation steps. A computation*

- *is* properly terminated *if all component processes have completed their last instruction,*
- diverges *if it is infinite, and*
- *is in* deadlock *if it is finite but not properly terminated.*

2.2 Graph Theory

We state some fundamental notions concerning directed finite graphs, from here on simply referred to as graphs.

Definition 3. *Two vertices $a, b \in V$ of a graph $G = (V, E)$ are* strongly connected *if there are paths from a to b and from b to a; G is strongly connected if all pairs of vertices are. Two vertices $a, b \in V$ are* directly connected *if $(a, b) \in E$ or $(b, a) \in E$; G is directly connected if all pairs of vertices are.*

Definition 4. *An* automorphism *of a graph $G = (V, E)$ is a permutation σ of V such that for all $v, w \in V$, $(v, w) \in E$ implies $(\sigma(v), \sigma(w)) \in E$. The* automorphism group *$\varSigma_G$ of a graph G is the set of all automorphisms of G. The least $p > 0$ with $\sigma^p = \mathrm{id}$ is called the* period *of σ, where by* id *we denote the identity function defined on the domain of whatever function it is compared to.*

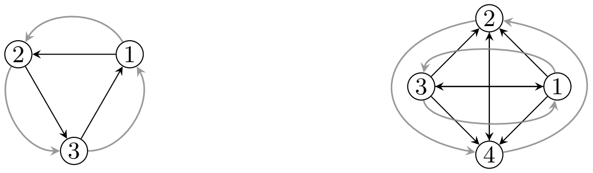
The orbit *of $v \in V$ under $\sigma \in \varSigma_G$ is $O_v^\sigma := \{\sigma^p(v) \mid p \geq 0\}$. An automorphism σ is* well-balanced *if the orbits of all vertices have the same cardinality, or alternatively, if for all $p \geq 0$,*

$$\sigma^p(v) = v \text{ for some } v \in V \text{ implies } \sigma^p = \mathrm{id} \quad .$$

We will usually consider the (possibly empty) set $\varSigma_G^{wb} \setminus \{\mathrm{id}\}$ of non-trivial *well-balanced automorphisms of a graph G, that is those with period greater than* 1.

A subset $W \subseteq V$ is called invariant *under $\sigma \in \varSigma_G$ if $\sigma(W) = W$; it is called invariant under $\varSigma_G$ if it is invariant under all $\sigma \in \varSigma_G$.*

Example 2. Figure 3 shows two graphs G and H and well-balanced automorphisms $\sigma \in \varSigma_G$ with period 3 and $\tau \in \varSigma_H$ with period 2. We have $\varSigma_H = \{\mathrm{id}, \tau\}$, so $\{1, 3\}$ and $\{2, 4\}$ are invariant under $\varSigma_H$.



(a) Graph G, $\sigma \in \varSigma_G$

(b) Graph H, $\tau \in \varSigma_H$

Fig. 3: Two graphs with non-trivial well-balanced automorphisms, indicated by gray, bent arrows.

2.3 Symmetric Electoral Systems

We take over the semantic definition of symmetry from [2]. As discussed there, syntactic notions of symmetry are difficult to formalize properly; requiring that "all processes run the same program" does not do the job. We will skip the formal details since we are not going to use them. The interested reader is referred to [2].

Definition 5 (adapted from [2, Definition 2.2.2]). *A system $\mathcal{P}$ with communication graph $G = (V, E)$ is* symmetric *if for each automorphism $\sigma \in \varSigma_G$*

and each computation C of $\mathcal{P}$, there is a computation C' of $\mathcal{P}$ in which, for each $v \in V$, process $P_{\sigma(v)}$ performs the same steps as P_v in C, modulo changing via σ the process names occurring in the computation (e.g. as communication partners).

The intuitive interpretation of this symmetry notion is as follows. Any two processes which are not already distinguished by the communication graph G itself, i.e. which are related by some automorphism, must have equal possibilities of behavior. That is, whatever behavior one process exhibits in some particular possible execution of the system (i.e., in some computation), the other process must exhibit in some other possible execution of the system, localized to its position in the graph by appropriate process renaming. Taken back to the syntactic level, this can be achieved by running the same program in both processes, which must not make use of any externally given distinctive features like, for example, an ordering of the process names.

Example 3. The system from Fig. 2 is symmetric. It is easy to see that, if we swap all names 0 and 1 in any computation of $\mathcal{P}$, we still have a computation of $\mathcal{P}$. Note that programs are allowed to access the process names, and indeed they do; however, they do not, for example, use their natural order to determine which process sends first.

Example 4. On the other hand, the system $\mathcal{Q} = \{Q_0, Q_1\}$ where each Q_i runs the following program is not symmetric:

```
[  $i = 0 \rightarrow Q_{i+1} \,!\, i$
□ $i = 1 \rightarrow Q_{i+1} \,?\, x$  ]
```

We now recall a classical problem for networks of processes, and then restate the impossibility result which our paper builds on.

Definition 6 (from [2, Definition 1.2.1]). *A system $\mathcal{P}$ is an* electoral system *if*

(i) all computations of $\mathcal{P}$ are properly terminating and
(ii) each process of $\mathcal{P}$ has a local variable `leader`, and at the time of termination all these variables contain the same value, namely the name of some process $P \in \mathcal{P}$.

Theorem 1 (from [2, Theorem 3.3.2]). *Suppose a network G admits some well-balanced automorphism σ different from* id. *Then G admits no symmetric electoral system in CSP_{in}.*

3 Setting the Stage

3.1 Pairwise Synchronization

Intuitively, if we look at synchronization as part of a larger system, a process is able to synchronize with another process if it can execute an algorithm such that a direct communication (of any message) between the two processes takes place. This may be the starting point of some communication protocol to exchange more information, or simply be taken as an event creating common knowledge about the processes' current progress of execution.

Communication in *CSP* always involves exactly two processes and facilities for synchronous broadcast do not exist, thus synchronization is inherently pairwise only. This special case is still interesting and has been subject to research, see e.g. [22].

Focusing on the synchronization algorithm, we want to guarantee that it allows all pairs of processes to synchronize. To this end, we require existence of a system where in all computations, all pairs of processes synchronize. Most probably, in a real system not all pairs of processes need to synchronize in all executions. However, if one has an algorithm which in principle allows that, then one could certainly design a system where they actually do; and, vice versa, if one has a system which is guaranteed to synchronize all pairs of processes, then one can obviously use its algorithms to synchronize any given pair. Therefore we use the following formal notion.

Definition 7. *A system $\mathcal{P}$ of processes* (pairwise) synchronizes $\mathcal{Q} \subseteq \mathcal{P}$ *if all computations of $\mathcal{P}$ are finite and properly terminating and contain, for each pair $P_a, P_b \in \mathcal{Q}$, at least one direct communication from P_a to P_b or from P_b to P_a.*

Example 5. The system from Fig. 2 synchronizes $\{P_0, P_1\}$.

Note that the program considered so far is not a valid CSP_{in} program, since there an output statement appears within a guard. If we want to restrict ourselves to CSP_{in} (for example, to implement the program in Occam), we have to get rid of that statement. Attempts to simply move it out of the guard fail since the symmetric situation inevitably leads to a system which may deadlock.

To see this, consider the system $\mathcal{P}' = \{P'_0, P'_1\}$ running the following program:

```
  $recd$ := false
  $sent$ := false
*[  $\neg recd \wedge P'_{i+1}\,?\,x \rightarrow recd$ := true
  $\square\ \neg sent \rightarrow P'_{i+1}\,!\,i;\ sent$ := true  ]
```

There is no guarantee that not both processes enter the second clause of the repetition at the same time and then block forever at the output statement, waiting for each other to become ready for input. A standard workaround [20] for such cases is to introduce buffer processes mediating between the main processes, in our case resulting in the extended system $\mathcal{R} = \{R_0, R'_0, R_1, R'_1\}$:

```
  $recd$ := false
  $sent$ := false
*[  $\neg recd \wedge R'_{i+1}\,?\,x \rightarrow recd$ := true
  $\square\ \neg sent \rightarrow R'_i\,!\,i;\ sent$ := true  ]
```

(program of main process R_i)

```
$R_i\,?\,y$
$R_{i+1}\,!\,y$
```

(program of buffer process R'_i)

While the actual data transmitted between the main processes remains the same, this system obviously cannot synchronize $\{R_0, R_1\}$, since there is no direct communication between them. This removes the synchronizing and common knowledge creating effects of communication. Mutual knowledge can only be achieved to a finite (if arbitrarily high) level, as discussed in Sect. 1.1.

The obvious question now is: Is it possible to change the program or use buffer or other helper processes in more complicated and smarter ways to negotiate between the main processes and aid them in establishing direct communications?

To attack this question, in the following Sect. 3.2 we will formalize the kind of communication networks we are interested in and define how they may be extended in order to allow for helper processes without affecting the symmetry inherent in the original network.

3.2 Peer-to-peer networks

The idea of peer-to-peer networks is to have nodes which can communicate with each other directly and on an equal footing, i.e. there is no predetermined client/server architecture or central authority coordinating the communication. We first formalize the topological prerequisites for this, and then adapt the semantic symmetry requirement to our setting.

Definition 8. *A* peer-to-peer network *is a communication graph $G = (V, E)$ with at least two vertices (also called nodes) such that*

(i) G is strongly connected,
(ii) G is directly connected, and
(iii) we have $\Sigma_G^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$.

In this definition, (i) says that each node has the possibility to contact (at least indirectly) any other node, reflecting the fact that there are no predetermined roles; (ii) ensures that all pairs of nodes have a direct connection at least in one direction, without which pairwise synchronization by definition would be impossible; and (iii) requires a kind of symmetry in the network. This last item is implied by the more intuitive requirement that there be some $\sigma \in \Sigma_G$ with only one orbit, i.e. an automorphism relating all nodes to each other and thus making sure that they are topologically on an equal footing. The requirement we use is less restrictive and suffices for our purposes.

Example 6. See Fig. 3 for two examples of peer-to-peer networks.

We will consider extensions allowing for helper processes while preserving the symmetry inherent in the network. We view the peers, i.e. the nodes of the original network, as processors each running a main process, while the added nodes can be thought of as helper processes running on the same processor as their respective main process.

Definition 9. *Let $G = (V, E)$ be a peer-to-peer network, then $G' = (V', E')$ is a* symmetry-preserving extension *of G iff there is a collection $\{S_v\}_{v \in V}$ partitioning V' such that*

(i) for all $v \in V$, we have $v \in S_v$;

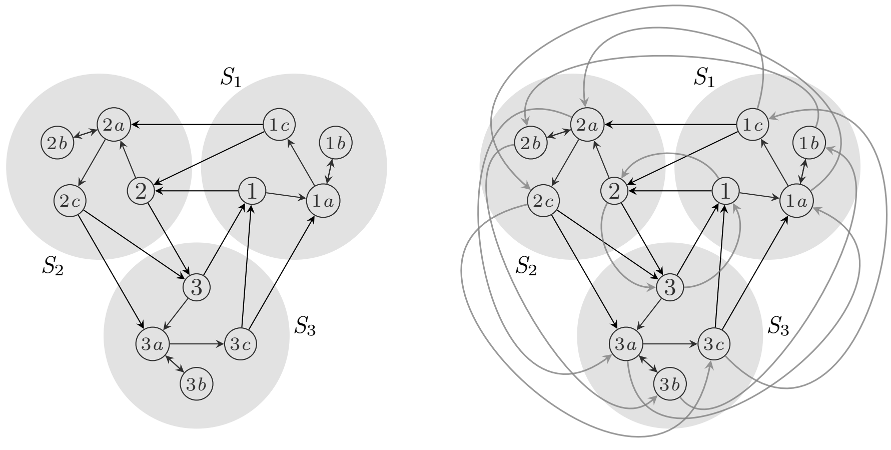
(ii) all $v \in V$, $v' \in S_v \setminus \{v\}$ are strongly connected (possibly via nodes $\notin S_v$);
(iii) for all $v, w \in V$, $E' \cap (S_v \times S_w) \neq \emptyset$ iff $(v, w) \in E$;
(iv) there is, for each $\sigma \in \Sigma_G$, an automorphism $\iota_\sigma \in \Sigma_{G'}$ extending σ such that $\iota_\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$.

Remark 1. In general, the collection $\{S_v\}_{v \in V}$ may not be unique. When we refer to it, we implicitly fix an arbitrary one.

Intuitively, these requirements are justified as follows:

(i) Each S_v can be seen as the collection of processes running on the processor at vertex v, including its main process P_v.
(ii) The main process should be able to communicate (at least indirectly) in both ways with each helper process.
(iii) While communication links within one processor can be created freely, links between processes on different processors are only possible if there is a physical connection, that is a connection in the original peer-to-peer network; also, if there was a connection in the original network, then there should be one in the extension in order to preserve the network structure.
(iv) Lastly, to preserve symmetry, each automorphism of the original network must have an extension which maps all helper processes to the same processor as their corresponding main process.

Example 7. See Fig. 4 for an example of a symmetry-preserving extension. Note that condition (iii) of Definition 9 is liberal enough to allow helper processes to communicate directly with processes running on other processors, and indeed, e.g. $2c$ has a link to 3. It also allows several communication links on one physical connection, reflected by the fact that there are three links connecting S_2 to S_3.



(a) Symmetry-preserving extension of the network from Fig. 3(a).

(b) Extended automorphism ι_σ as required by Definition 9.

Fig. 4: A symmetry-preserving extension (illustrating Definition 9).

We will need the following immediate fact later on.

Fact 1 *As a direct consequence of Definitions 8 and 9, any symmetry-preserving extension of a peer-to-peer network is strongly connected.*

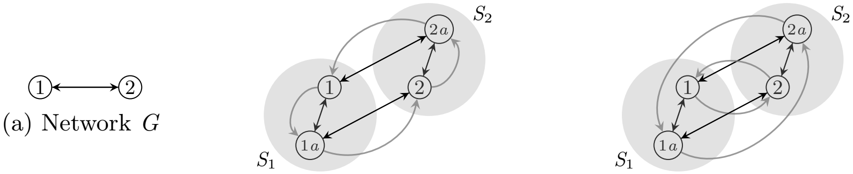
3.3 G-symmetry

Corresponding to the intuition of processors with main and helper processes, we weaken Definition 5 such that only automorphisms are considered which keep the set of main processes invariant and map helper processes to the same processor as their main process. There are cases (as in Fig. 8 later in this paper) where the main processor otherwise would be required to run the same program as some helper process.

Definition 10 (G-symmetry). *A system $\mathcal{P}$ whose communication graph G' is a symmetry-preserving extension of some peer-to-peer network $G = (V, E)$ is called* G-symmetric *if Definition 5 holds with respect to those automorphisms $\sigma \in \Sigma_{G'}$ satisfying, for all $v \in V$, (i) $\sigma(V) = V$ and (ii) $\sigma(S_v) = S_{\sigma(v)}$.*

This is weaker than Definition 5, since there we require the condition to hold for all automorphisms.

Example 8. To illustrate the impact of G-symmetry, Fig. 5 shows a network G and an extension where symmetry relates all processes which each other. G-symmetry disregards the automorphism which causes this and considers only those which keep the set of main processes invariant, i.e. the nodes of the original network G, thus allowing them to behave differently from the helper processes.



(a) Network G

(b) Extension of G and an automorphism mixing main and helper processes

(c) Extension of G and the only automorphism taken into account by G-symmetry

Fig. 5: A network G and an extension which has an automorphism mixing main and helper processes, disregarded by G-symmetry.

4 Results

4.1 Positive Results

Theorem 2. *Let $G = (V, E)$ be a peer-to-peer network. Then G admits a symmetric system pairwise synchronizing V in $CSP_{i/o}$.*

Proof. A system which at each vertex $v \in V$ runs the program shown below is symmetric and pairwise synchronizes V. Each process simply waits for each other process in parallel to become ready to send or receive a dummy message, and exits once a message has been exchanged with each other process.

```
  for each $w \in V$ do $sync_w :=$ false
  $W_{in} := \{w \in V \mid (w, v) \in E\}$
  $W_{out} := \{w \in V \mid (v, w) \in E\}$
*[
  $\Box_{w \in W_{in}}\ \neg sync_w \wedge P_w\,?\,x \rightarrow sync_w :=$ true
  $\Box_{w \in W_{out}}\ \neg sync_w \wedge P_w\,!\,0 \rightarrow sync_w :=$ true
  ]
```

□

By dropping the topological symmetry requirement for peer-to-peer networks, under certain conditions we get a positive result even for CSP_{in}.

Theorem 3. *Let $G = (V, E)$ be a network satisfying only the first two conditions of Definition 8, i.e. G is strongly connected and directly connected. If G admits a symmetric electoral system and there is some vertex $v \in V$ such that $(v, a) \in E$ and $(a, v) \in E$ for all $a \in V$, then G admits a symmetric system pairwise synchronizing V in CSP_{in}.*

Proof (sketch). First, the electoral system is run to determine a temporary leader v'. When the election has terminated, v' chooses a coordinator v that is directly and in both directions connected to all other vertices, and broadcasts its name. Broadcasting can be done by choosing a spanning tree and transmitting the broadcast information together with the definition of the tree along the tree, as in the proof of [2, Theorem 2.3.1, Phase 2] (the strong connectivity which is required there holds for G by assumption). After termination of this phase, the other processes each send one message to v and then wait to receive commands from v according to which they perform direct communications with each other, while v receives one message from each other process and uses the obtained order to send out the commands. □

Example 9. See Fig. 6 for an example of a network which admits a symmetric system pairwise synchronizing all its vertices in CSP_{in}. The fact that the network admits a symmetric electoral system can be established as for [2, Fig. 4] (note that the edges between the lower nodes are only in one direction).

This result could be generalized, e.g. by weakening the conditions on v and taking care that the commands will reach the nodes at least indirectly. Since our main focus is the negative result, we will not pursue this further.

4.2 Negative Result

In the following we will establish the main result saying that, even if we extend a peer-to-peer network G by helper processes (in a symmetry-preserving way), it

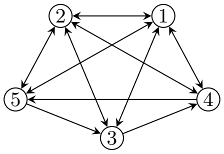


Fig. 6: A network which by Theorem 3 admits a symmetric system pairwise synchronizing all its vertices in CSP_{in}.

is not possible to obtain a network which admits a G-symmetric system pairwise synchronizing the nodes of G in CSP_{in}.

To this end, we derive a contradiction with Theorem 1 by proving the following intermediate steps (let G denote a peer-to-peer network and G' a symmetry-preserving extension):

- Lemma 1: If G' admits a G-symmetric system pairwise synchronizing the nodes of G in CSP_{in}, it admits a G-symmetric electoral system in CSP_{in}.
- Lemma 2: G' has a non-trivial well-balanced automorphism taken into account by G-symmetry (i.e. satisfying the two conditions of Definition 10).
- Lemma 3: We can extend G' in such a way that there exists a non-trivial well-balanced automorphism (derived from the previous result), G-symmetry is reduced to symmetry, and admittance of an electoral system is preserved.

Lemma 1. *If some symmetry-preserving extension of a peer-to-peer network $G = (V, E)$ admits a G-symmetric system pairwise synchronizing V in CSP_{in}, then it admits a G-symmetric electoral system in CSP_{in}.*

Proof. The following steps describe the desired electoral system (using the fact that under G-symmetry processes of nodes $\in V$ may behave differently from those of nodes $\notin V$):

- All processes run the assumed G-symmetric pairwise synchronization program, with the following modification for the processes in $\mathcal{P} := \{P_v \mid v \in V\}$ (intuitively this can be seen as a kind of knockout tournament, similar to the proof of [2, Theorem 4.1.2, Phase 1]):
  - Each of these processes has a local variable `winning` initialized to `true`.
  - After each communication statement with some other $P \in \mathcal{P}$, insert a second communication statement with P in the same direction:
    - If it was a “send” statement, send the value of `winning`.
    - If it was a “receive” statement, receive a Boolean value, and if the received value is `true`, set `winning` to `false`.

  Note that, since the program pairwise synchronizes V, each pair of processes associated to vertices in V has had a direct communication at the end of execution, and thus there is exactly one process in the whole system which has a local variable `winning` containing `true`.

- After the synchronization program terminates the processes check their local variable `winning`. The unique process that still has value `true` declares itself the leader and broadcasts its name; all processes set their variable `leader` accordingly. As in the proof of Theorem 3, broadcasting can be done using a spanning tree. The required strong connectivity is guaranteed by Fact 1. □

Lemma 2. *For any symmetry-preserving extension $G' = (V', E')$ of a peer-to-peer network $G = (V, E)$, there is $\sigma' \in \Sigma^{wb}_{G'} \setminus \{\mathrm{id}\}$ such that $\sigma'(V) = V$ and $\sigma'(S_u) = S_{\sigma'(u)}$ for all $u \in V$.*

Proof. Take an arbitrary $\sigma \in \Sigma^{wb}_{G} \setminus \{\mathrm{id}\}$ (exists by Definition 8) and let ι, to save indices, denote the ι_σ required by Definition 9. If $\iota \in \Sigma^{wb}_{G'} \setminus \{\mathrm{id}\}$ we are done; otherwise we construct a suitable σ' (Example 10 illustrates this proof).

Let p denote the period of σ and pick an arbitrary $v \in V$. For simplicity, we assume that σ has only one orbit; if it has several, the proof extends straightforwardly by picking one v from each orbit in parallel.

For all $u \in S_v$ let $p_u := |O^\iota_u|$ and note that for all $t \in O^\iota_u$ we have $p_t = p_u$, and $p_u \geq p$ since ι maps each S_v to $S_{\sigma(v)}$ and these sets are pairwise disjoint. We define $\sigma' : V' \to V'$ and then prove the claims.

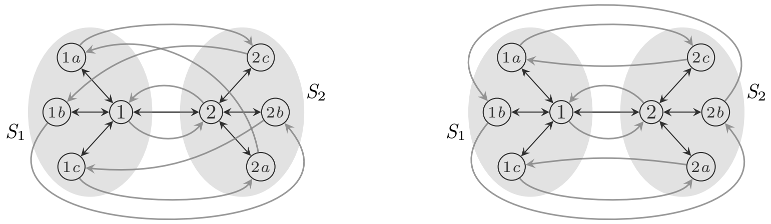
$$\sigma'(u) := \begin{cases} \iota^{p_u - p + 1}(u) & \text{if } u \in S_v \\ \iota(u) & \text{otherwise.} \end{cases}$$

- $\sigma'(V) = V$, $\sigma' \neq \mathrm{id}$: Follows from $\iota \restriction_V = \sigma$ and $p_v = p$ and thus $\sigma' \restriction_V = \sigma$ (where $f \restriction_X$ denotes the restriction of a function f to the domain X)
- $\sigma' \in \Sigma_{G'}$: With (iv) from Definition 9 we obtain that, for $u \in S_v$, p_u must be a multiple of p, and $\sigma'(O^\iota_u \cap S_v) = \iota(O^\iota_u \cap S_v)$, thus σ' is a permutation of V' since ι is one. Furthermore, for $t, u \in S_v$, we have $\iota^{p_t(p_u-1)}(t) = t$ and $\iota^{p_u(p_t-1)}(u) = u$ and therefore σ' also inherits edge-preservation from ι by
$$(\sigma'(t), \sigma'(u)) = (\iota^{p_t - p + 1}(t), \iota^{p_u - p + 1}(u)) = (\iota^{p_t p_u - p + 1}(t), \iota^{p_t p_u - p + 1}(u)) \ .$$
- $\sigma'(S_u) = S_{\sigma'(u)}$, σ' well-balanced: The above-mentioned fact that for all $u \in S_v$ we have $\sigma'(O^\iota_u \cap S_v) = \iota(O^\iota_u \cap S_v)$, together with (iv) from Definition 9 implies that also $\sigma'(S_u) = S_{\sigma(u)}$ for all $u \in V$. For all $v' \in V'$, well-balancedness of σ and disjointness of the S_u imply that $\sigma'^q(v') \neq v'$ for $0 < q < p$. On the other hand, since each orbit of σ has size p and contains exactly one element from S_v (namely v), we have that
$$\begin{aligned} \sigma'^p(v') &= \iota^{(p_u - p + 1) + (p - 1)}(v') && \text{for some } u \in O^\iota_{v'} \\ &= \iota^{p_u}(v') = \iota^{p_{v'}}(v') = v' \ . \end{aligned}$$
□

Example 10. In Fig. 7(a), we have $p = 2$ (the period of $\sigma = \iota_\sigma \restriction_{\{1,2\}}$), and we pick vertex $v = 2$. For the elements of S_2, we obtain $p_2 = p = 2$ and $p_{2a} = p_{2b} = p_{2c} = 6$. Thus σ' is defined as follows:

$$\sigma'(u) = \begin{cases} \iota(u) & \text{if } u = 2 \\ \iota^5(u) & \text{if } u \in S_2 \setminus \{2\} \\ \iota(u) & \text{if } u \in S_1 \ . \end{cases}$$

This σ', depicted in Fig. 7(b), satisfies the claims of Lemma 2.



(a) ι_σ as required by Definition 9 (b) σ' constructed from ι_σ as in Lemma 2

Fig. 7: An extended peer-to-peer network G' illustrating Lemma 2.

Lemma 3. *Any symmetry-preserving extension $G' = (V', E')$ of a peer-to-peer network $G = (V, E)$ can be extended to a network H such that*

(i) $\Sigma_H^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$, and
(ii) if G' admits a G-symmetric electoral system in CSP_{in}, then H admits a symmetric electoral system in CSP_{in}.

Proof. The idea is to add an “identifying structure” to all elements of V, which forces all automorphisms to keep V invariant and map the S_v to each other correspondingly (see Fig. 8). Formally, let $K = |V'|$ and, denoting the inserted vertices by $i_{\cdot,\cdot}$, for each $v \in V$ let $I_v := \bigcup_{k=1}^{K} \{i_{v,k}\}$ and

$$E_v := \{(v, i_{v,1})\} \cup \bigcup_{k=1}^{K-1} \{(i_{v,k}, i_{v,k+1}), (i_{v,k+1}, v)\} \cup \bigcup_{w \in S_v} \{(i_{v,K}, w)\} \ ,$$

and let $H := \left(V' \cup \bigcup_{v\in V} I_v, E' \cup \bigcup_{v \in V} E_v\right)$. Now we can prove the two claims.

(i) Let $\sigma \in \Sigma_{G'}^{wb} \setminus \{\mathrm{id}\}$ with $\sigma(V) = V$ and $\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$ (such a σ exists by Lemma 2), then $\sigma \cup \bigcup_{v\in V} \bigcup_{k=1}^{K} \{i_{v,k} \mapsto i_{\sigma(v),k}\} \in \Sigma_H^{wb} \setminus \{\mathrm{id}\}$.

(ii) H is still a symmetry-preserving extension of G via (straightforward) extensions of the S_v. The discriminating construction has the effect that Σ_H consists only of extensions, as above, of those $\sigma \in \Sigma_{G'}$ for which $\sigma(V) = V$ and $\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$. Thus, any G-symmetric system with communication graph H is a symmetric system with communication graph H. Additionally, the set of all $i_{v,k}$ is invariant under Σ_H due to the distinctive structure of the I_v, thus the associated processes are allowed to differ from those of the remaining vertices. A symmetric electoral system in CSP_{in} can thus be obtained by running the original G-symmetric electoral system on all members of G' and having each $v \in V$ inform $i_{v,1}$ about the leader, while all $i_{v,k}$ simply wait for and transmit the leader information. □

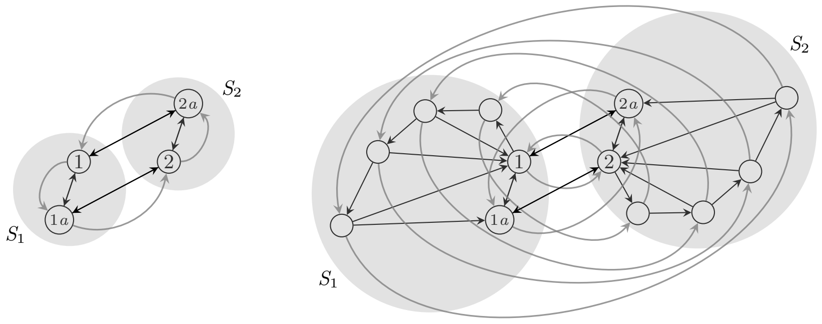


Fig. 8: A network with an automorphism disregarded by G-symmetry, and the extension given in Lemma 3 invalidating automorphisms of this kind shown with the only remaining automorphism.

Theorem 4. *There is no symmetry-preserving extension of any peer-to-peer network $G = (V, E)$ that admits a G-symmetric system pairwise synchronizing V in CSP_{in}.*

Proof. Assume there is such a symmetry-preserving extension G'. Then by Lemma 1 it also admits a G-symmetric electoral system in CSP_{in}. According to Lemma 3, there is then a network H with $\Sigma_H^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$ that admits a symmetric electoral system in CSP_{in}. This is a contradiction to Theorem 1. □

5 Conclusions

We have provided a formal definition of peer-to-peer networks and adapted a semantic notion of symmetry for process systems communicating via such networks. In this context, we have defined and investigated the existence of pairwise synchronizing systems, which are directly useful because they achieve synchronization, but also because they create common knowledge between processes. Focusing on two dialects of the *CSP* calculus, we have proved the existence of such systems in $CSP_{i/o}$, as well as the impossibility of implementing them in CSP_{in}, even allowing additional helper processes like buffers. We have also mentioned a recent extension to JCSP to show that, while CSP_{in} is less complex and most commonly implemented, implementations of $CSP_{i/o}$ are feasible and do exist.

A way to circumvent our impossibility result is to remove some requirements. For example, we have sketched a construction for non-symmetric systems in CSP_{in}. In general, if we give up the symmetry requirement, $CSP_{i/o}$ can be implemented in CSP_{in} [2, p. 197].

Another way is to weaken the notion of common knowledge or approximate it [8], which may suffice in settings where the impact decreases significantly as the depth of mutual knowledge increases, see e.g. [23].

However, if one is interested in symmetric systems and exact common knowledge, as in the game-theoretical settings described in Sect. 1.1, then our results show that $CSP_{i/o}$ is a suitable formalism, while CSP_{in} is insufficient. Already in the introducing paper [1], the exclusion of output guards from *CSP* was recognized as reducing expressivity and being programmatically inconvenient, and soon it was deemed technically not justified [24,17] and removed in later versions of *CSP* [13, p. 227].

Some existing proposals for implementations of input and output guards and synchronous communication could be criticized for simply shifting the problems to a lower level, notably for not being symmetric themselves or for not even being strictly synchronous in real systems due to temporal imprecision [8].

However, it is often useful to abstract away from implementation issues on the high level of a process calculus or a programming language (see e.g. [25, Section 10]). For these reasons, we view our setting as an argument for implementing $CSP_{i/o}$ rather than CSP_{in}.

Acknowledgments

I would like to thank my supervisor Krzysztof Apt for his support, helpful comments and suggestions, as well as four anonymous referees for their feedback which helped to improve the paper.

This research was supported by a GLoRiClass fellowship funded by the European Commission (Early Stage Research Training Mono-Host Fellowship MEST-CT-2005-020841).

References

1. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21** (1978) 666–677
2. Bougé, L.: On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. Acta Informatica **25** (February 1988) 179–201
3. Halpern, J.Y.: A computer scientist looks at game theory. Games and Economic Behavior **45** (October 2003) 114–131
4. Gray, J. In: Notes on Data Base Operating Systems. Volume 60 of Lecture Notes in Computer Science. Springer-Verlag (1978) 393–481
5. Rubinstein, A.: The electronic mail game: Strategic behavior under "almost common knowledge". The American Economic Review **79** (June 1989) 385–391
6. Morris, S.: Coordination, communication, and common knowledge: A retrospective on the electronic-mail game. Oxf Rev Econ Policy **18** (December 2002) 433–445
7. Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press (1995)
8. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM **37** (1990) 549–587
9. Schneider, F.B.: Synchronization in distributed programs. ACM Trans. Program. Lang. Syst **4** (1982) 125–148
10. Osborne, M.J.: An Introduction to Game Theory. Oxford University Press, New York (August 2003)

11. Moulin, H.: Axioms of Cooperative Decision Making. Cambridge University Press (1988)
12. Andrews, G.R.: Concurrent Programming: Principles and Practice. Addison Wesley (July 1991)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc (1985)
14. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. John Wiley and Sons, Inc (1999)
15. INMOS Ltd.: occam 2 Reference Manual. Prentice-Hall (1988)
16. Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing **16** (2003) 121–163
17. Buckley, G.N., Silberschatz, A.: An effective implementation for the generalized input-output construct of csp. ACM Trans. Program. Lang. Syst **5** (1983) 223–235
18. Welch, P.: An occam-pi Quick Reference (1996–2007) `https://www.cs.kent.ac.uk/research/groups/sys/wiki/OccamPiReference`.
19. Welch, P., Brown, N., Moores, J., Chalmers, K., Sputh, B.: Integrating and extending JCSP. In McEwan, A.A., Schneider, S., Ifill, W., Welch, P., eds.: Communicating Process Architectures, IOS Press (2007)
20. Jones, G.: On guards. In Muntean, T., ed.: Parallel Programming of Transputer Based Machines (OUG-7), Amsterdam, IOS Press (1988) 15–24
21. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Mathematical Structures in Computer Science **13** (2003) 685–719
22. Parikh, R., Krasucki, P.: Communication, consensus, and knowledge. Journal of Economic Theory **52** (October 1990) 178–189
23. Weinstein, J., Yildiz, M.: Impact of higher-order uncertainty. Games and Economic Behavior **60** (July 2007) 200–212
24. Bernstein, A.: Output guards and nondeterminism in "Communicating Sequential Processes". ACM Trans. Program. Lang. Syst **2** (1980) 234–238
25. Kurki-Suonio, R.: Towards programming with knowledge expressions. In: 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL), St. Petersburg Beach, Florida, ACM Press (1986) 140–149