On Implementation of Global Concurrent Systems with Local Asynchronous Controllers

Blaise Genest

Department of Computer Science, Warwick, Coventry, CV4 7AL, UK

Abstract. The classical modelization of concurrent system behaviors is based on observing execution sequences of global states. This model is intuitively simple and enjoys a variety of mathematical tools, e.g. finite automata, helping verifying concurrent systems. On the other hand, parallel composition of local controllers are needed when dealing with the actual implementation of concurrent models. A well known tool for turning global observation into local controllers is Zielonka's theorem, and its derivatives. We give here another algorithm, simpler and cheaper than Zielonka's theorem, in the case where the events observed do not include communication but only local asynchronous actions. In a developer point of view, it means that she does not have to explicitly specify the messages needed, which will be added (if allowed) automatically by the implementation algorithm.

1 Introduction

Specifying the behavior of software systems in such a way that formal methods can be applied and validation tasks can be automated, is a challenging goal. While research has brought strong results and tools for simple systems, complex systems still lack powerful techniques. For instance, concurrent systems such as message passing systems are still hard to cope with.

One of the challenging problem concerning concurrent systems is to design parallel algorithms. The problem is that it is easy to think in a sequential way, and to observe and model the global behavior of such a system. On the other hand, it is much harder to build local controllers, that is controllers that have only a local view of the behaviors. They have to deal with partial information, consisting of local behaviors plus information brought by messages from other processes. One method used for the implementation is to automatically turn, when possible, a global specification into a set of local controllers. Zielonka's theorem [22] states that the regular models that can be implemented by local controllers in term of asynchronous automata are exactly those closed by commutation. This theorem was further used and extended to give local controllers in term of communicating automata, with channels either bounded [18, 11, 12], or existentially-bounded [9] (that is a set of bounded executions suffices to represent up to commutation every execution of the system, even the unbounded ones [13, 14, 9]). This method has several major drawbacks. The first one is that the complexity of these Zielonka-based theorems is of several exponents [11]. Moreover, the implementation obtained in terms of communicating automata has unavoidable deadlocks, which is unacceptable for an implementation. Last, but not least, it forces the designer to provide every single message that will be needed: forgetting a message can compromise the implementability of a specification. These messages are used as support for passing information with control data (or gossiping in the terminology of [18, 11]).

Our starting point is to implement a FIFO, **deadlock-free communicating automata**, with possibly more messages than specified by a regular set of executions. Of course, using all channels for messages is neither interesting (every system could be sequentialized, thus implemented), nor wanted (the less messages, channels and synchronizations used, the better), nor possible (some channel might not be available, as a direct communication from a PDA to a satellite). Thus, an architecture is set once for all, by means of a set of couples of processes, standing for the directed available channels. Since the implementation algorithm can add messages, we assume that no message is provided by the specification, even if we could technically express them. The specification is thus given by a finite automaton, whose transitions are labeled by local actions. Until this point, the background is general enough to use either synchronous or asynchronous messages, and to use open or closed systems. For instance, an action of a synchronous system would be a tuple of local actions (possibly the null action), one for each process. Here, we consider asynchronous systems that are closed, modeling communicating protocols, where every action is associated with one process. We want to obtain a deadlock-free communicating automaton such that the projection on local actions of its set of (partial) executions is exactly the language given as specification. Thus, every state is considered as final, for both the specification and the implementation. This non trivial restriction is natural when deadlocks are forbidden. Like Zielonka's theorem, the closure under a (semi) commutation related to the architecture is necessary. But unlike it, it is not always sufficient, due to the deadlock-free requirement.

Results: We first show that in the subcase where one process (the boss) can send messages to any other process (possibly via other processes), then an easy implementation exists as soon as the specification is closed by commutation.

We then give two different notions of deadlock-freeness (strong and weak), together with algorithms to test their implementability in the general setting. *Strong deadlock-freeness* allows only choices that are local, which we translate for the specification into the forward diamond property [6]. A linear size implementation can then be obtained. *Weak deadlock-freeness* allows global-choices (that is, choices involving the decision from several processes), which can be really tricky to implement. Nonetheless, we show the decidability of the weak deadlock-free implementability too, which yields an implementation exponential in the number of global choices. More precisely, we show that there are finitely many decisions to make, enabling us to test them all to find a potential unpredictable choice. The implementation is obtained by ruling out the global choices at the beginning of the implementation, and then by using the small implementation obtained in case there are only local-choices. It is worth noting that usually, the number of global choices is restricted, which limits the use of the expansive step of ruling out the global choices. We also define *very weak deadlock-freeness*, but argue against it, even though the algorithm for implementability is easier than for the weak deadlock freeness. We give examples that separate these three deadlock-free notions in section 3.3.

Related Work: Other papers deal with deadlock-free implementation. [1, 3] give an algorithm for the implementation only when no information can be exchanged by processes, avoiding many protocols from being implementable. Moreover, [8, 2] consider only local specifications (local-choice CMSC-graph [10] and a local flavor of EMSO logic), unlike the global specification in our paper. In the non FIFO case (two messages with different contents can overtake each other), Petri nets implementation can be used [5], allowing much more systems to be implemented. However unwanted scenarios can be implied.

Several papers were also published considering open systems. Synchronous open systems become quickly undecidable due to undecidability for partial information games [20], both if the specification is global [19, 17] or local [15]. This basic undecidability result cannot be translated to the asynchronous case, so decidability is more likely: some promising results in restricted cases were published, as [7, 16].

2 Preliminaries

Until the end of the paper, we consider that the set $\mathcal{P}$ of processes is fixed. We want to implement a global specification $\mathcal{S}$ by a set $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ of local controllers for process p, using some communicating channels Ch.

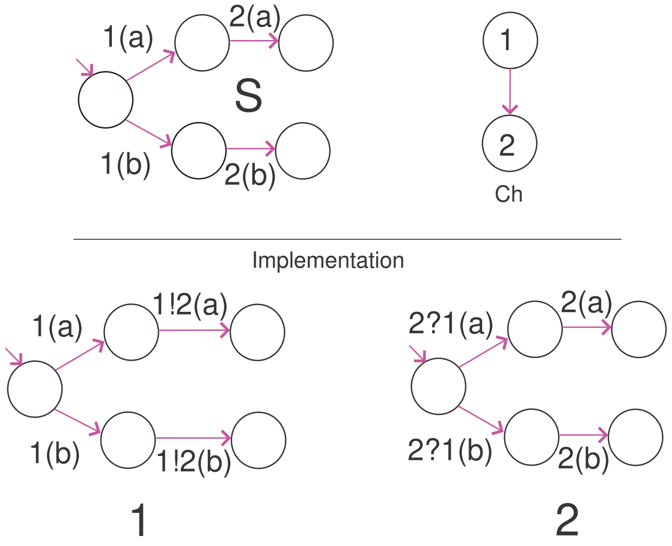

Fig. 1. An example of a specification S, an architecture Ch, and a communicating finite-state machine (CFM) on processes $\{1, 2\}$.

A specification is given by an automaton $\mathcal{S}$, whose transitions are labeled by local events $p(a)$, meaning that process $p \in \mathcal{P}$ does the action a. Here, we think that the specification to implement does not provide the messages yet. Indeed, it is the task of the implementation to find which messages are to be sent. Actually, the specification stands for the possibilities of executions, which is an arena the system should never exit. That is, $u \in \mathcal{L}(\mathcal{S})$ means that there exists a path labeled by u in $\mathcal{S}$ (we consider that every state of $\mathcal{S}$ is final).

We now define our target model, communicating finite state machines, which are restricted since every state is considered as final.

A communicating finite-state machine (CFM) $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ [4] consists of finite automata $\mathcal{A}_p$ associated with processes $p \in \mathcal{P}$, that communicate over unbounded, error-free, FIFO channels. With each pair $(p, q) \in \mathcal{P}^2$ of distinct processes we associate a channel $C_{p,q}$. The content of a channel is a word over a finite alphabet $\mathcal{C}$. Each $\mathcal{A}_p$ is described by a tuple $\mathcal{A}_p = (S_p, A_p, \rightarrow_p)$ consisting of a set of local states S_p, a set of actions name A_p, and a transition relation $\rightarrow_p \subseteq S_p \times A_p \times S_p$. An action of $\mathcal{A}_p$ is either a local action $p(a)$, a send $p!q(a)$ or a receive $p?q(a)$, with $p, q \in \mathcal{P}$. Sending message $p!q(a)$ means that a is appended to the channel $C_{p,q}$. Receiving message $p?q(a)$ means that a must be the first message in $C_{q,p}$, which will be then removed from $C_{q,p}$. The function P associates an event with its process: $P(p(a)) = P(p!q(a)) = P(p?q(a)) = p$ for all $p, q \in \mathcal{P}$. Every computation begins in the initial state s^0, where $s^0 \in \prod_{p\in\mathcal{P}} S_p$.

A run of a CFM is a word x such that for all $p \in \mathcal{P}$, the projection of x on events of process p is a run of A_p, and such that any receive $p?q(a)$ is enabled, that is the first letter of $C_{q,p}$ before $p?q(a)$ is a. We say that a run is successful if every channel is empty at the end of the run. Moreover, we say that a CFM is *very weakly deadlock-free* if every run can be extended to a successful run. The set of successful runs generated by $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$, the language of $\mathcal{A}$. With this definition of run, a CFM satisfies the FIFO restriction, that is, the n-th send event s from p to q is received by the n-th receive r from p to q. Thus, we call (s, r) a message if s and r are the n-th send and receive on the same channel, and define a mapping $m(s) = r$ and an order $s <_m r$ iff (s, r) is a message. Let x be a run of a CFM. The process order of x is $a <_p b$ if $P(a) = P(b) = p \in \mathcal{P}$ and a appears before b in x. The visual order $<$ of x is the transitive closure of $\bigcup_p <_p \cup <_m$. For every set of event X of x, we denote $\text{Future}(X) = \{z \mid y < z$ for some $y \in X\}$.

An architecture is a graph $(\mathcal{P}, Ch)$ where $\mathcal{P}$ is the set of processes and $Ch \subseteq \mathcal{P} \times \mathcal{P}$ is the set of directed channels. The channels which can be used have to be provided as an architecture since some channel might not be available (there is no way to talk to a sensor designed only to send some information, nor to send a message directly to a satellite with a small device). We call q a child of p for $(p, q) \in Ch$. We assume that for all $p \in \mathcal{P}$, $(p, p) \in Ch$. We will prove later that we can assume without loss of generality that $(\mathcal{P}, Ch)$ is an acyclic graph closed by transitivity. We thus call a process p minimal for Ch if there is no process $q \neq p$ with $(q, p) \in Ch$.

3 Implementability

3.1 Properties for Implementability

Given a specification $\mathcal{S}$, we want to obtain, when possible, a deadlock-free communicating automaton $\mathcal{A}$ on processes $\mathcal{P}$, which uses only the channels Ch, such that the projection $\pi(\mathcal{L}(\mathcal{A}))$ of $\mathcal{L}(\mathcal{A})$ on local actions $\{p(a) \mid p \in \mathcal{P}, a \in \Sigma\}$ equals $\mathcal{L}(\mathcal{S})$. We will consider several deadlock-free restrictions, but all will respect the very weak condition at least. The size $|\mathcal{P}|$ of $\mathcal{P}$ is the number of processes. The size $|\mathcal{A}|$ of an automaton $\mathcal{A}$ is its number of nodes. The size $|\mathcal{A}|$ of a CFM $\mathcal{A} = (\mathcal{A}_p)$ is $|\mathcal{A}| = \sum_{p \in \mathcal{P}} |\mathcal{A}_p|$.

We show now that we can assume without loss of generality that Ch is closed by transitivity, that is $Ch = Ch^*$. If two processes p, q are such that $(p,q) \in Ch^*$, then any implementation $\mathcal{A}$ for $Ch \cup (p,q)$ can be translated into an implementation $\mathcal{A}'$ for Ch. Let $p_0 \cdots p_n$ be some sequence of processes with $(p_i, p_{i+1}) \in Ch$ and $p_0 = p, p_n = q$. It suffices to change every action $p!q(a)$ into $p!p_1(fw, a)$ (for every a, (fw, a) is a new symbol meaning forward a), and $q?p(a)$ into $q?p_{n-1}(fw, a)$. Moreover, simple loops are added on every state of processes $p_i, i \in \{1, \cdots, n-1\}$, labeled by $p_i?p_{i-1}(fw, a)$ then $p_i!p_{i+1}(fw, a)$. Obtaining an implementation for Ch from an implementation $\mathcal{A}$ for Ch^* is of size at most $|\mathcal{P}||\mathcal{A}|$. Hence, from now on, we will assume that Ch is closed by transitivity.

We show now that we can assume without loss of generality that Ch is loop-free (remember that Ch is closed by transitivity). If two processes $p \neq q$ are such that $(p,q) \in Ch$ and $(q,p) \in Ch$, then any implementation $\mathcal{A}$ for $\mathcal{P} \setminus q$, where every action $q(a)$ is replaced by an action $p(r, a)$, can be translated into an implementation for $\mathcal{P}$. It suffices to change every actions $p(r, a)$ in $\mathcal{A}$ into two actions $p!q(a), p?q(ack)$. Moreover, the process q is easily made of loops $q?p(a), q(a), q!p(ack)$. The aim of the acknowledgements ack is to ensure the good timing of actions with respect to p and q. Obtaining an implementation for $\mathcal{P}$ from an implementation $\mathcal{A}$ for $\mathcal{P} \setminus q$ is of size at most $|\mathcal{P}||\mathcal{A}|$.

Moreover, the specification $\mathcal{S}$ can be assumed to be closed by the semi-commutation: $ab \rightarrowtail ba$ if $(P(a), P(b)) \notin Ch$ (see [6] for a formal definition of semi commutation). First, notice that for any CFM $\mathcal{A}$, if $uxyv \in \mathcal{L}(\mathcal{A})$ and $y \notin \text{Future}(x)$, then $uyxv \in \mathcal{L}(\mathcal{A})$. Let $\mathcal{S}$ be a specification not closed by semi $\rightarrowtail$ commutation: there is $uabv \in \mathcal{S}$ with $ab \rightarrowtail ba$ and $ubav \notin \mathcal{S}$. Assume by contradiction that there exists an implementation $\mathcal{A}$ of $\mathcal{S}$. Then $u'awbv' \in \mathcal{L}(\mathcal{A})$ with $\pi(u'awbv') = uabv$. Let $w = w_b w'$ with w_b in the past of b. Since b is not in the future of w', $u'aw_b bw'v' \in \mathcal{L}(\mathcal{A})$. We have that $w_b \cap \text{Future}(a) = \emptyset$, else there would exist in w a sequence of messages from a to b, which is not possible since $(P(a), P(b)) \notin Ch$ and Ch is closed by transitivity. So $u'w_b abw'v' \in \mathcal{L}(\mathcal{A})$, hence $u'w_b baw'v' \in \mathcal{L}(\mathcal{A})$, so $\pi(u'w_b baw'v') = uabv \in \mathcal{S}$, which is false. Hence, being closed by semi commutation is a necessary condition for being implementable.

We thus consider for the rest of the paper that $\mathcal{S}$**, given as minimal and deterministic**, is diamond. That is if there exist 3 states r, s, t with $r \xrightarrow{a} s \xrightarrow{b} t$ and $ab \rightarrowtail ba$, then there exists also a fourth state s' with $r \xrightarrow{b} s' \xrightarrow{a} t$, which translates the semi-commutation [6].

3.2 An Easy Subcase

Here, we give a subcase where the diamond property is enough to ensure the implementability. We do not need it for solving the general problem, but it is an easy implementation which stands for a good first example.

Namely, in case where there is a unique minimal process for Ch called $boss$, then any specification is implementable. The idea is that $boss$ makes the choices and dictates their behaviors to other processes.

Proposition 1 *If $\mathcal{S}$ is an automaton whose language is closed by semi $\rightarrowtail$-commutation, and there is a unique minimal process boss for Ch, then $\mathcal{S}$ is implementable by a CFM of size $O(|\mathcal{S}| \cdot |\mathcal{P}|!)$.*

Proof. We define now $\mathcal{A}_p$ for $p \neq boss$ as a program, which can be easily turned into an automaton.

while(true)
{ wait until $p?boss(a)$;
If $a = top(q)$ then $p!q(top)$;
If $a = wait(Q)$ then for all $q \in Q, p?q(top)$;
If $a = do(b)$ then b; }

We define now the automaton for boss. It is built upon $\mathcal{S}$, with two differences. A memory is kept recording the ordering of the last event on each process. More formally, let $\prec$ be an order on $\mathcal{P}$. After each transition labeled by an event on process p, the order $\prec$ is updated such that the restriction of $\prec$ on $\mathcal{P} \setminus \{p\}$ is kept, and for all $q \neq p$, $q \prec p$. In the initial state, the order $1 \prec \cdots \prec n$ is chosen (but any other is fine).

Consider a transition labeled by a with $P(a) = p$, from a state with an order $\prec$ between the processes. Let Q be the set of processes q with $p \prec q$ and $(q, p) \in Ch$. Now, we replace this transition labeled by a by a sequence of transitions $boss!q(top(p))$ for all $q \in Q$, then $boss!p(wait(Q))$, then $boss!p(do(a))$.

We have easily that $\pi_{boss!p(do(a))}(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{S})$, so it is closed by semi-commutation. Now, we can prove easily that $\pi_{boss!p(do(a))}(\mathcal{L}(\mathcal{A})) = \pi(\mathcal{L}(\mathcal{A}))$. □

3.3 Several Deadlock-free Restrictions

If the implementation seems really easy in the case where there is a boss, many problems can arise in the case where there are several minimal processes in Ch.

Consider first the specification $\mathcal{S}_0$ made of three states $\{s_1, s_2, s_3\}$ with $s_1 \xrightarrow{p(a)} s_2$ and $s_1 \xrightarrow{q(b)} s_3$, with p, q two distinct minimal processes of Ch. While $\mathcal{S}_0$ satisfies the diamond property, it is not implementable in this architecture. The reason is that p and q can act separately, and since they have no common information (they are minimal in Ch), they can also act in the same time in any implementation, yielding the execution $p(a)q(b)$ which is not in the specification.

When there is a process r with $(r,p) \in Ch$ and $(r,q) \in Ch$, then r can choose between p and q and inform either p or q (but not both) that it has to act, which is exactly what happens when there is a unique minimal process in Ch.

When a choice between 3 processes is involved, then a solution can be much difficult to obtain. Consider an architecture with 6 processes $\{1,2,3,4,5,6\}$ with $Ch = \{(1,4),(1,5),(2,4),(2,6),(3,5),(3,6)\}$, that is every pair of processes from $\{4,5,6\}$ has a common ancestor, but there is no common ancestor to $\{4,5,6\}$. Assume that the specification $\mathcal{S}_1$ allows $4(a)5(a)$, $5(a)6(a)$ or $4(a)6(a)$ up to commutation but not $4(a)5(a)6(a)$. Since every state of $\mathcal{S}_1$ is final, we have $\mathcal{L}(\mathcal{S}_1) = \{\epsilon, 4(a), 5(a), 6(a), 4(a)5(a), 5(a)6(a), 4(a)6(a)\}$ (up to commutation). There is a way to implement $\mathcal{S}_1$ with a very weakly deadlock free CFM $\mathcal{A}$. Processes $1,2,3$ guess two processes among $4,5,6$ and send this guess to their children (two processes among $4,5,6$). Process $p \in \{4,5,6\}$ does a if and only if the two guesses it received are the same (we could deal with some difference too) and contain p. We indeed have $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{S}_1)$. Moreover, $\mathcal{A}$ is very weakly deadlock-free because every message is received (the channels are empty) and the state reached is final. The problem is that if the guesses $1,2,3$ are not coherent (which is usually the case), then the CFM sends its messages and no local action happens, which is a kind of deadlock as well, even though it is allowed since $\epsilon \in \mathcal{L}(\mathcal{S}_1)$. We think this is due to a weakness of the (very weakly) deadlock-freeness as it is defined, and we propose two alternatives in our context where $\pi(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{S})$.

Definition 1 *A very weakly deadlock-free CFM is strongly deadlock-free if for every run x, if there exists an accepting run y with $\pi(y) = \pi(x)a$, then x can be extended into an accepting run xz with $\pi(z) = a$.*

A very weakly deadlock-free CFM is weakly deadlock-free if for every run x, if there exists an accepting run y with $\pi(y) = \pi(x)a$, then x can be extended into an accepting run.

The difference between the two definitions is that if a word x can be extended in k different ways in $\mathcal{S}$, then any run of a strongly deadlock-free CFM implementing x must be extendable in k different ways, while a weakly deadlock-free CFM must be extendable in at least one way. Strong deadlock-freeness is more restrictive than weak deadlock-freeness, but also easier to implement.

First, it is easy to show that the specification $\mathcal{S}_1$ defined above cannot be implemented by a weakly deadlock-free CFM, and hence neither by a strongly deadlock-free CFM. Then, consider the same architecture as for $\mathcal{S}_1$, that is $\mathcal{P} = \{1,2,3,4,5,6\}$ with $Ch = \{(1,4),(1,5),(2,4),(2,6),(3,5),(3,6)\}$. This time, let $\mathcal{S}_2 = \{4(a)5(a)6(b), 4(a)5(b)6(a), 4(b)5(a)6(a), 4(b)5(b)6(b)\}$ (up to commutation). Consider the following CFM $\mathcal{A}$: Process i guesses a bit b_i, where $i \in \{1,2,3\}$. It sends b_i to its two children. Each process $j \in \{4,5,6\}$ checks if the two bits it received from its two parents are equal. If yes, then it does b, else it does a. Either the three bits are equal, thus the CFM yields the execution $4(b)5(b)6(b)$ (up to commutation), or two bits are equal, let say $b_1 = b_2$, and then 4 does b, while $5,6$ does a. The CFM $\mathcal{A}$ is not strongly deadlock-free since the run x where $1, 2, 3$ have sent their bit can be extended in only one way, while $\pi(x) = \epsilon$ can be extended in many other ways. But $\mathcal{A}$ is weakly deadlock-free. While strong deadlock-freeness usually suffices, it sometimes forbids implementing legitimate specifications. For instance, the specification $\mathcal{S}_2$ corresponds to an election of two or none processes among three (or distributed mutual exclusion).

The proposition 2 shows that no strongly deadlock-free CFM can implement $\mathcal{S}_2$. First, let us define the forward diamond property:

Definition 2 *A deterministic and diamond automaton $\mathcal{S}$ fulfills the forward diamond property if for every states r, s, s' with $r \xrightarrow{a} s$, $r \xrightarrow{b} s'$ and $ab \rightarrowtail ba$ or $ba \rightarrowtail ab$ (that is $P(a) \neq P(b)$ since Ch is acyclic), there exists a fourth state t with $r \xrightarrow{a} s \xrightarrow{b} t$ and $r \xrightarrow{b} s' \xrightarrow{a} t$.*

Notice that $\mathcal{S}_2$ does not fulfill the forward diamond property.

Proposition 2 *Let $\mathcal{S}$ be a deterministic and diamond specification. If $\mathcal{S}$ is implemented by a strongly deadlock-free CFM $\mathcal{A}$, then $\mathcal{S}$ fulfills the forward diamond property.*

Proof. By contradiction, assume that $\mathcal{S}$ is implemented by a strongly deadlock-free CFM $\mathcal{A}$ and that there exists $r \xrightarrow{a} s$, $r \xrightarrow{b} s'$ and $(P(a), P(b)) \notin Ch$ but there is no transition labeled by b coming from s (else, by the diamond (non forward) property, there would be a transition labeled by q coming from s' as well). Let x be a run of $\mathcal{S}$ stopping in r. Since $xa \in \mathcal{L}(\mathcal{S})$ and $\mathcal{A}$ is an implementation of $\mathcal{S}$, there exists a run va of $\mathcal{A}$ with $\pi(v) = x$. Since $\mathcal{A}$ is strongly deadlock-free, there exists w with $\pi(w) = \epsilon$ and vwb is a run of $\mathcal{A}$. Let $w = w_b w'$ with w_b the past of b. The word $vw_b b$ is also an execution of $\mathcal{A}$. Since $(P(a), P(b)) \notin Ch$, $w_b b$ contains no events on p with $(p, P(b)) \notin Ch$. The projection of $vaw_b b$ on p with $(p, P(b)) \notin Ch$ is the same as the projection of va, which is an execution on p. The projection of $vaw_b b$ on q with $(q, P(b)) \in Ch$ is the same as the projection of $vw_b b$, which is an execution on q. So $vaw_b b$ is an execution of $\mathcal{A}$ which means that $\pi(vaw_b b) = xab \in \mathcal{L}(\mathcal{S})$, a contradiction.

□

Last, notice that the proof of proposition 1 builds a weakly deadlock-free CFM. There are cases where it cannot be made strongly deadlock-free because of the proposition 2.

4 Strongly Deadlock-free Implementation

We show in this section that proposition 2 is an equivalence, namely that $\mathcal{S}$ fulfills the forward diamond properties if and only if it is implementable by a strongly deadlock-free CFM.

We first build the communicating automaton $\mathcal{A}_{\mathcal{S}} = (\mathcal{A}_p)_{p \in \mathcal{P}}$ we use for the implementation, based on the automaton $\mathcal{S}$. For each process p, we let $\mathcal{S}_p$ be the accessible nodes of $\mathcal{S}$ from the initial node by any action of processes $\{q \mid (q, p) \in Ch\}$ (recall that $(p, p) \in Ch$). We then define $\mathcal{A}_p$ from the automaton

$\mathcal{S}_p$, where the labels $a \in \Sigma_q, q \neq p$ of the transitions are replaced by $p?q(a)$. Furthermore, every transition labeled by $a \in \Sigma_p$ is replaced by a sequence of transitions labeled by a then by $p!q(a)$ for all $(p,q) \in Ch$. That is, when an action is done on p, then p informs each of its children. Conversely, the projection of $u \in \mathcal{A}_\mathcal{S}$ on p defines a run in S_p.

Proposition 3 *Let $\mathcal{S}$ be a deterministic specification. If $\mathcal{S}$ fulfills the diamond and forward diamond properties, then $\mathcal{S}$ is implementable, and one of its strongly deadlock-free implementation is $\mathcal{A}_\mathcal{S}$, of size $O(|\mathcal{P}| \cdot |\mathcal{S}|)$.*

Proof. We let $\mathcal{A} = \mathcal{A}_\mathcal{S}$. We show that $\pi(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{S})$, and that $\mathcal{A}$ is strongly deadlock-free. Notice that both diamond properties hold for $\mathcal{S}_p, p \in \mathcal{P}$.

$\mathcal{L}(\mathcal{S})$ **is included into** $\pi(\mathcal{L}(\mathcal{A}))$**.** Let $u \in \mathcal{L}(\mathcal{S})$. We define an execution v by replacing every action $p(a)$ by a sequence of actions $p(a), p!q(a), q?p(a)$ for all $(p,q) \in Ch$. We thus have $\pi(v) = u$.

We show now that $v \in \mathcal{L}(\mathcal{A})$, which ends the proof of $\mathcal{L}(\mathcal{S}) \subseteq \pi(\mathcal{L}(\mathcal{A}))$. Let $p \in \mathcal{P}$. Let $u \rightarrowtail u_p u'$ with u_p containing every event of u on p or on $(q,p) \in Ch$, and none on other processes. Since $\mathcal{S}$ is closed by commutation, $u_p u' \in \mathcal{L}(\mathcal{S})$, and thus $u_p \in \mathcal{L}(\mathcal{S}_p)$. The execution of $\mathcal{A}_p$ which corresponds to u_p is exactly the projection v_p of v on p. Thus for all p, $v_p \in \mathcal{L}(\mathcal{A}_p)$, that is $v \in \mathcal{L}(\mathcal{A})$.

$\pi(\mathcal{L}(\mathcal{A}))$ **is included into** $\mathcal{L}(\mathcal{S})$**.** Let $u \in \mathcal{L}(\mathcal{A})$. Assume by contradiction that $v = \pi(u)$ is not an execution of $\mathcal{S}$. Let $v = v'p(a)w$ with v' the longest prefix with $v' \in \mathcal{L}(\mathcal{S})$. We let t be the state of $\mathcal{S}$ where the path labeled by v' ends. We define $v' \rightarrowtail v'_p v''$, with v'_p containing every event of v' on p or on $(q,p) \in Ch$, and none on other processes. Since $\mathcal{S}$ is diamond, $v'_p v'' \in \mathcal{L}(\mathcal{S})$ labels a path in $\mathcal{S}$ which ends in t. Moreover, $v'_p \in \mathcal{L}(\mathcal{S}_p)$, and let s be the state reached by the path labeled by v'_p. That is, $s \xrightarrow{v''} t$.

Let u_p be the projection of u on p, which belongs to $\mathcal{L}(\mathcal{A}_p)$, and w_p be the associated execution of $\mathcal{L}(\mathcal{S}_p)$. Notice that v'_p may not be a prefix of w_p since p may hear (receive the messages) about actions of an ancestor process much later than when it did happen. Anyway, it is a prefix of some equivalent run of w_p because of the FIFO condition. Hence, since $\mathcal{S}_p$ is closed by semi commutation, $w_p \in \mathcal{L}(\mathcal{S})$, and $w_p \rightarrowtail v'_p p(a) x$, there is a transition from s labeled by $p(a)$ in $\mathcal{S}_p$, so in $\mathcal{S}$.

Applying inductively the forward diamond property of $\mathcal{S}$ to s, to the transition labeled by $p(a)$, and to those leading to t and labeled by an event on a process not in p or in $(q,p) \in Ch$, we get a transition from t labeled by $p(a)$ in $\mathcal{S}$. This means $v'p(a) \in \mathcal{L}(\mathcal{S})$, contradicting the maximality of v'. So $\pi(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{S})$.

$\mathcal{A}$ **is very weakly deadlock-free**. Considering $\mathcal{L}(\mathcal{A})$ is closed by prefix, the only reason for $\mathcal{A}$ to have a very weak deadlock would be not to be able to receive some message $p!q(a)$ sent. We show that this is not possible.

Assume that there is a partial execution u of $\mathcal{A}$ with an unreceived send event $c = p!q(a)$. We choose c to be the minimal unreceived send in u sent by the smallest process possible. We show that $u\,d$ is also a partial execution, with d the receive $q?p(a)$.

Let u_q be the projection of u on q, which belongs to $\mathcal{A}_q$, and w_q be the associated execution in $\mathcal{S}_q$. Let $w_q \rightarrowtail w_p w'$ with w_p containing all events on process p or on $(k,p) \in Ch$. Because every send sent by any $k \neq p, (k,p) \in Ch$ is already received (by minimality of c), w_p contains exactly all the events of u on $k \neq p, (k,p) \in Ch$, and by FIFO, all the events until c on p.

Since u contains the send c, it means that there exists a prefix $u'_p c$ of the projection of u on p. Let $w'_p p(a)$ be the execution of $\mathcal{S}_p$ associated with $u'_p c$. We have $w_p \rightarrowtail w'_p x$, that is $w_q \rightarrowtail w'_p y$. By the diamond property of $\mathcal{S}_q$, we have states s, t with $s_0 \xrightarrow{w'_p} s \rightarrow^* t$ and $s_0 \xrightarrow{w_q} t$. Consider now that c is enabled after u'_p, so $p(a)$ is enabled after w'_p, so after s.

Applying the forward diamond property inductively in $\mathcal{S}_q$, to the transition labeled by $p(a)$, and the sequence of transitions leading to t (labeled by an event on a process different that p), $p(a)$ is enabled after t. The meaning in $\mathcal{A}_q$ is that d can be received by q after the projection of u on q, that is $u\,d$ is a partial execution of $\mathcal{A}$.

$\mathcal{A}$ **is strongly deadlock-free**. By the very construction of $\mathcal{A}$, if an execution x is such that $\pi(x)a \in \mathcal{S}$, then a transition labeled by a on $P(a)$ is possible after x. $\square$

Remark 1.
1. Figure 1 depicts a specification and its corresponding implementation given by proposition 3.
2. One definition of a deterministic CFM is that if $\pi(x) = \pi(y)$ for two runs x, y of a CFM, then x and y are equal up to commutation. It makes sense in our case since only the projection on local events matters. Then any deterministic implementation needs to be strongly-deadlock free.
3. The CFM obtained in proposition 3 is deterministic for the determinacy definition of [11]. That is, each automaton $\mathcal{A}_p$ is deterministic, and there is no node r with two transitions $r \xrightarrow{p!q(a)} s$ and $r \xrightarrow{p!q(b)} s$ labeled by different messages data $a \neq b$.
4. The CFM obtained is not universally bounded. That is, there are specifications for which for every b, the implementation obtained has a run which uses more than b messages in some channel at one time.
5. The existential boundedness is defined in [13] (see also [14, 9]). The CFM obtained in proposition 3 is not existentially bounded. Anyway, it is easy to obtain a strongly deadlock-free implementation which is existentially 1 bounded (but it is not of linear size). It suffices for a process p to simulate every process q with $(q,p) \in Ch$. For every state where two messages from different processes can be received (that is the two simulated states are not stuck), the transition receiving the message from the lower process (for Ch) can be deleted. It makes the CFM even more deterministic. The diamond property on $\mathcal{S}$ will allow process p to receive this message later, when the higher process will be stuck waiting for the same message, preserving the deadlock-freeness.

5 Weakly Deadlock-free Implementation

5.1 Global Choices

The challenge with weakly deadlock-free implementation is that global choices (choices involving the decision from several processes) are possible. In contrast, being forward diamond means that every choice in the specification can be made locally by one process. The strategy we use to solve the weakly deadlock-free implementability is to make a precomputation on the specification (if possible) to have a forward diamond specification. Hence, we can use the strongly deadlock-free implementation, whose algorithm is fast and produce small implementations.

The important step in our algorithm is proposition 5 which states that the number of global choices is finite. Then proposition 6 states that there exists a finite number of different ways to take a decision, implying they can all be tested. A crucial hypothesis for proposition 5 is to choose $\mathcal{S}$ deterministic, hence we will assume that it is the case from now on (we can anyway determinize and minimize the specification, preserving the diamond property).

First, we give a syntactical characterization of global choices. We denote by $L(s)$ the language of $\mathcal{S}$ with initial node s. We denote by L_Q the projection of L on set of processes Q, where Q is such that if $q \in Q$ and $(p, q) \in Ch$, then $p \in Q$. In particular, if L satisfies the diamond property, L_Q is the accessible part of L by events in Q. This is the case for $L(s)$ for every node s of $\mathcal{S}$.

Definition 3 *A transition $s \xrightarrow{\tau} t$ is a global choice if $L_Q(s) \neq L_Q(t)$ for $Q = \{k \mid (P(\tau), k) \notin Ch\}$. We denote by $\mathcal{G}$ the set of global choices.*

Notice that $\mathcal{S}$ satisfies the forward diamond property iff $\mathcal{G} = \emptyset$.

First, we can test for global choice involving two process p, q without common ancestor. If such global choices exist, no implementation is possible since p and q cannot take a coherent decision.

Proposition 4 *Assume that a transition $s \xrightarrow{\tau} t$ is such that $L_K(s) \neq L_K(t)$ with $K = \{q \mid \nexists k, (k, q) \in Ch \wedge (k, P(\tau)) \in Ch\}$. Then $\mathcal{S}$ is not (very) weakly deadlock-free implementable.*

Proof. By contradiction, assume that $\mathcal{A}$ is a very weak deadlock-free implementation of $\mathcal{S}$. Let $u \in \mathcal{L}(\mathcal{S})$ such that this execution ends in the state s. We decompose u in u_1, u_2, with $u_1 = \pi_K(u)$. Let $v \in L_K(s) \setminus L_K(t)$. So $u_1 u_2 v, u_1 u_2 \tau \in \mathcal{L}(\mathcal{S})$, but $u_1 u_2 \tau v \notin \mathcal{L}(\mathcal{S})$, neither $u_2 \tau u_1 v \notin \mathcal{L}(\mathcal{S})$ (by semi commutation of $\mathcal{S}$). So there exists $wxy, w'x'\tau \in \mathcal{L}(\mathcal{A})$, with $\pi(w) = \pi(w') = u_1$, $\pi(x) = \pi(x') = u_2$, $\pi(y) = v$, and $wy = \pi_K(xwy)$. We have wy is a partial execution of the processes in K since they cannot receive any message from other processes. In the same line, $x'\tau$ is a partial execution of $\mathcal{A}$. Hence $x'\tau wy$ is a partial execution of $\mathcal{A}$. Since $\mathcal{A}$ is a very weakly deadlock-free implementation of $\mathcal{S}$, we have $\pi(x'\tau wy) = u_2 \tau u_1 v \in \mathcal{L}(\mathcal{S})$, a contradiction. $\square$

5.2 A Finite Number of Decisions

We prove here that there is no loop around a global choice. That is, there is only a finite number of "questions" whether to take a choice or not.

Proposition 5 *Let $\mathcal{S}$ be a deterministic automaton that fulfills the diamond property. There is no loop $s \xrightarrow{\tau} t \xrightarrow{y} s$ with $(s \xrightarrow{\tau} t) \in \mathcal{G}$ a global choice.*

Proof. Since $s \xrightarrow{\tau} t$ is a global choice, there exists $x \in L_Q(s), x \notin L_Q(t)$ with $Q = \{k \mid (P(\tau), k) \notin Ch\}$. Now, for z the projection of y on Q, we have $y \rightarrowtail zz'$. We can merge τ with z', and renaming s, we obtain $s \xrightarrow{z'} t \xrightarrow{z} s$. We thus have the semi commutation $z'x \rightarrowtail xz'$ and $z'z \rightarrowtail zz'$. Because of the diamond property, $x \in L_Q(s)$ for the new s, and still $x \notin L_Q(t)$. Hence $s \neq t$.

Since $z'z \rightarrowtail zz'$, applying the diamond property, we obtain a state s' with $s \xrightarrow{z} s' \xrightarrow{z'} s$. Because $z'x \in L(s')$, $z'x \rightarrowtail xz'$ and the diamond property, $x \in L_Q(s')$. This means $s' \neq t$. The fact that $s' \neq s$ comes from the determinacy of $\mathcal{S}$, since $s' \xrightarrow{z'} s$ and $s \xrightarrow{z'} t$ with $s \neq t$. Assume by induction that we have constructed n different states s_n in this way. Then since $z'z \rightarrowtail zz'$, applying the diamond property, we obtain a state s_{n+1} with $s_n \xrightarrow{z} s_{n+1} \xrightarrow{z'} s_n$. Since $x \in L_Q(s_n)$ by induction hypothesis, and $z'x \rightarrowtail xz'$, we have $x \in L_Q(s_{n+1})$. So $s_{n+1} \neq t$. Assume that $s_{n+1} = s_i$. Then $s_i \xrightarrow{z'} s_{n+1}$. But we have already $s_i \xrightarrow{z'} s_{i-1}$. So by determinacy, $s_{n+1} = s_{i-1}$ as well. We come to a contradiction when $s_{n+1} = s$ means that $s_{n+1} = t$ by determinacy, which is known to be false.

We can iterate this to obtain an infinite number of nodes, using the diamond property and the determinacy, which is a contradiction with the finiteness of the automaton. □

Since the number of decisions to make is finite, they can be made once at the beginning, as a subset $G \subseteq \mathcal{G}$ of the global choices which can be used. However, not every subset can be chosen. Let $\mathcal{S}^G$ be the specification obtained from $\mathcal{S}$ by deleting every global choice not in $\mathcal{G}$. We say that G is compatible if:

- There exists $G \subseteq H$ with $\mathcal{S}^H$ diamond and forward diamond (a transition of $\mathcal{S}^H$ may be a global choice of $\mathcal{S}$ but is not in $\mathcal{S}^H$).
- Every $x \in \mathcal{L}(\mathcal{S}^G)$ that cannot be extended in $\mathcal{S}^G$ cannot be extended in $\mathcal{S}$ either.

We denote by $\mathcal{F}$ the set of compatible sets $G \subseteq \mathcal{G}$. Let $\mathcal{F}_p$ denote the set of projections of subsets of $\mathcal{F}$ on process p. We can easily compute $(\mathcal{F}_p)_{p \in \mathcal{P}}$.

5.3 Implementing Locally the Global Choices

Process p shall choose one set of global choices it can use among $\mathcal{F}_p$. If it could choose a set outside, then it would lead to an implementation which is not weakly deadlock-free (because a run cannot be extended), or which breaks the requirement $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{S})$.

First, we test whether $\bigcup_{G\in\mathcal{F}} \mathcal{L}(\mathcal{S}^G) = \mathcal{L}(\mathcal{S})$, else every implementation $\mathcal{A}$ will satisfy $\pi(\mathcal{L}(\mathcal{A})) \subseteq \bigcup_{G\in\mathcal{F}} \mathcal{L}(\mathcal{S}^G) \subsetneq \mathcal{L}(\mathcal{S})$: $\mathcal{S}$ cannot be implemented.

To help processes choosing locally a global choice among $\mathcal{F}$ (and not a combination of choices which is not in $\mathcal{F}$), they are helped by information sent by their ancestors. We can model the choices made by a function *choice* which associates each process p and each tuple of information sent to a set of $\mathcal{F}_p$ which the process p will execute if it receives the information. Of course, this function depends for p only on the information sent by ancestors of p. Let I_p be the set of different information that can be sent by process p. Let $I = I_1 \times \cdots \times I_n$ be the set of tuples of information.

We then say that $\mathcal{F}$ is implementable with I information if there exists a function *choice* as described above, such that:
- for every tuple $i \in I$, let $f(i) = \bigcup_{p\in\mathcal{P}} choice(p)(i)$,
- for every tuple $i \in I$ of information sent, $f(i) \in \mathcal{F}$, that is every global choice made by *choice* is compatible.
- $\bigcup_i \mathcal{L}(\mathcal{S}^{f(i)}) = \mathcal{L}(\mathcal{S})$, that is we get the whole language of $\mathcal{S}$ by implementing the function *choice*.

Notice that *choice* is defined by $f : I \to \mathcal{F}$ with $f(i) = \bigcup_{p\in\mathcal{P}} choice(p)(i)$. We show here that the information I can be limited. We can assume without loss of generality that minimal processes only send some information.

Proposition 6 *If $\mathcal{F}$ is implementable, then it is implementable with* $\log(|\mathcal{F}|)$ *bits of information sent by every minimal process p.*

Proof. Since $\mathcal{F}$ is implementable, there are a set of information I and $f : I \to \mathcal{F}$ which implements $\mathcal{G}$. Assume that $|I_1| > |\mathcal{F}| = n_0$ different information on process 1. We want to show that some information x is useless, that is the restriction $f|_{I\setminus\{x\}}$ still implements $\mathcal{G}$. We partition the space into n_0 pigeon holes: A tuple of information $i \in I$ is in the pigeon hole $f(i)$.

Obviously, if $\bigcup_{i\in I\setminus x} \mathcal{L}(\mathcal{S}^{f(i)}) = \mathcal{L}(\mathcal{S})$, then we can forget the information $x \in I_1$. In particular, it is the case if every pigeon hole which contains a tuple using x contains also a tuple not using x. Assume by contradiction that this is not the case. So for all information $x \in I_1$, there exists a pigeon hole which contains only information whose first component is x. Applying Dirichlet's pigeon hole lemma, since there are $n_0 < |I_1|$ pigeon holes, there is a pigeon hole associated with two different information, a contradiction.

□

We can thus test if $\mathcal{F}$ is implementable or not by testing every function of choice f using information $|I_p| = |\mathcal{F}| \leq 2^{|\mathcal{G}|}$ for each minimal process p (there are at most $|\mathcal{F}|^{|\mathcal{F}|^{|\mathcal{P}|}}$ such functions), telling us whether $\mathcal{S}$ is weakly deadlock-free implementable or not. It gives us the following theorem:

Theorem 1 *Let $\mathcal{S}$ be a deterministic and diamond specification. It is decidable to test whether a specification $\mathcal{S}$ is weakly deadlock-free implementable. The complexity is EXPSPACE($m|\mathcal{G}|$) and the implementation obtained is of size at most $O(|\mathcal{P}| \cdot |\mathcal{S}| \cdot 2^{m|\mathcal{G}|})$, where $|\mathcal{G}| \leq |\mathcal{S}|$ is the number of global choices and $m \leq |\mathcal{P}|$ the number of minimal processes.*

Proof. If $\mathcal{F}$ is implementable by f using information $|I_p| = |\mathcal{F}|$ for each minimal process p, then a weakly deadlock-free implementation is obtained in two steps. The first step is that every minimal process guesses $\log(|\mathcal{F}|) \leq |\mathcal{G}|$ bits of information and sends it to each of its children. Then every process q computes $G = f(q)(i)$. The second step is for process q to run the CFM obtained as in section 4 from the specification $\mathcal{S}_q^G$. Every CFM $\mathcal{A}^i$ obtained from the information $i \in I$ is weakly deadlock-free and satisfies $\pi(\mathcal{L}(\mathcal{A}^i)) \subseteq \mathcal{L}(\mathcal{S})$ by the definition of $\mathcal{G}$. Moreover, $\mathcal{G}$ is implemented by f, which means that $\bigcup_{i \in I} \pi(\mathcal{L}(\mathcal{A}^i)) = \mathcal{L}(\mathcal{S})$. $\square$

Remark 2. We can show that transitions $s \xrightarrow{\tau} t$ with $L_Q(s) \neq L_Q(t)$ (cf proposition 4) are the only reasons a specification may not be *very weakly deadlock-free implementable*, which gives a polynomial time algorithm to test implementability. If there is no such transition, each minimal process guesses a subset $G \in \mathcal{G}$. A process does nothing if the guesses of its parents are not consistent, else it runs the implementation of section 4 on S^G. The implementation obtained is also of size at most $O(|\mathcal{P}| \cdot |\mathcal{S}| \cdot 2^{m|\mathcal{G}|})$.

6 Conclusion

Unlike the deadlock-free implementability of global specifications when no message can be added (but data can) for which no result but heuristics are known [8], the implementability of a global specification $\mathcal{S}$ is decidable when messages can be added.

- The *very weak deadlock-free* implementability can be tested in PTIME and the implementation obtained is of exponential size. However, this deadlock-free restriction is too weak since the implementation is allowed to stop, while actions can still happen according to the specification.
- The *strong deadlock-free* implementability can be tested in PTIME and yields an implementation of size $O(|\mathcal{P}||\mathcal{S}|)$. The implementation enjoys many good properties, while it doesn't compromise any deterministic implementation. For instance, the *usb* protocol [21] is strongly deadlock-free implementable since it does not involve global choices.
- The *weak deadlock-freeness* allows global choices, contrary to the strong deadlock-freeness. Its much more involved implementation yields a single exponential size implementation. However, our algorithm becomes tractable for really small number of global choices, which we think is the case in real-life examples. In the unlikely case where there are a lot of global choices, the help from the user can still be asked, by highlighting the global choices, asking if they are really needed, and if yes, asking for a redesign of the specification to explain how these choices can be locally taken.

Acknowledgements : I would like to thank Anca Muscholl, Doron Peled, Igor Walukiewicz for fruitful discussions and an annonymous referee for his comments.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *ICALP'01*, LNCS 2076, pp.797-808, 2001.
2. B. Bollig and M. Leucker. Message-Passing Automata are expressively equivalent to EMSO Logic. In *CONCUR'04*, LNCS 3170, pp. 146-160, 2004.
3. N. Baudru and R. Morin. Safe implementability of regular message sequence chart specifications. In *SNPD'03*, pp 210–217. ACIS, 2003.
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):pp.323-342, 1983.
5. B. Caillaud, P. Darondeau, L. Hélouët, and G. Lesventes. HMSCs as partial specifications... with PNs as completions. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, Nantes, France 2000.
6. V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.
7. P. Gastin, B. Lerman, M. Zeitoun. Distributed Games with Causal Memory Are Decidable for Series-Parallel Systems. In *FSTTCS'04*, LNCS 3328, pp. 275-286, 2004.
8. B. Genest. Compositional Message Sequence Charts (CMSCs) are better to Implement than MSCs. In *TACAS'05*, LNCS , pp. 429-444, 2005.
9. B. Genest, D. Kuske and A. Muscholl. A Kleene Theorem and Model Checking for a Class of Communicating Automata. In *DLT'04*, LNCS 3340, pp. 30-48, 2004.
10. E. Gunter, A. Muscholl, and D. Peled. Compositional Message Sequence Charts. In *TACAS'01*, LNCS 2031, pp. 496–511, 2001. Journal version *International Journal on Software Tools for Technology Transfer (STTT)* 5(1): 78-89 (2003).
11. J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni and P. Thiagarajan. A Theory of Regular MSC Languages. To appear in *Information and Computation*, available at http://www.comp.nus.edu.sg/~thiagu/public_papers/icregmsc.pdf.
12. D. Kuske. Regular sets of infinite message sequence charts. In *Information and Computation 187*, Academic Press, pp.80-109, 2003.
13. M. Lohrey and A. Muscholl. Bounded MSC communication. *Information and Computation*, (189):135–263, 2004.
14. P. Madhusudan and B. Meenakshi. Beyond Message Sequence Graphs. In *FSTTCS'01*, LNCS 2245, pp. 256-267, 2001.
15. P. Madhusudan and P.S. Thiagarajan. Distributed Controller Synthesis for Local Specifications. In *ICALP'01*, LNCS 2076, pp. 396-407,2001.
16. P. Madhusudan and P.S. Thiagarajan. A Decidable Class of Asynchronous Distributed Controllers. In *CONCUR'02*, LNCS 2421, pp. 145-160, 2002.
17. S. Mohalik and I. Walukiewicz. Distributed Games. In *FSTTCS'03*, LNCS 2914, pp. 338-351, 2003.
18. M. Mukund, K. Narayan Kumar and M. Sohoni. Synthesizing Distributed Finite-State Systems from MSCs. In *CONCUR 2000*, LNCS 1877 , pp. 521-535, 2000.
19. A. Pnueli and R. Rosner. Distributed Reactive Systems Are Hard to Synthesize. In *FOCS 1990*, pp. 746-757, 1990.
20. J.H. Reif. The Complexity of Two-player Games of Incomplete Information. *J. Comp. Sys. Sci. 29*, pp 274-301, 1984.
21. USB 1.1 specification, available at http://www.usb.org/developers/docs/usbspec.zip
22. W. Zielonka. Note on finite asynchronous automata, R.A.I.R.O. *Informatique Théorique et Applications*, 21:pp.99-135, 1987.