Negotiation Programs

Javier Esparza[1] and Jörg Desel[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
esparza@tum.de
[2] Fakultät für Mathematik und Informatik, FernUniversität in Hagen, Germany
joerg.desel@fernuni-hagen.de

Abstract. We introduce a global specification language for distributed negotiations, a recently introduced concurrent computation model with atomic negotiations combining synchronization of participants and choice as primitive. A token game on distributed negotiations determines reachable markings which enable possible next atomic negotiations. In a *deterministic* distributed negotiation, each participant can always be engaged in at most one next atomic negotiation. In a *sound* distributed negotiation, every atomic negotiation is enabled at some reachable marking, and from every reachable marking the final marking of the distributed negotiation can be reached. We prove that our specification language has the same expressive power as sound and deterministic negotiations, i.e., every program can be implemented by an equivalent sound and deterministic negotiation and every sound and deterministic negotiation can be specified by an equivalent program, where a program and a negotiation are equivalent if they have the same Mazurkiewicz traces and thus the same concurrent runs. The translations between negotiations and programs require only linear time.

1 Introduction

Multi-party negotiation as a concurrent computation model has been recently introduced in [1, 2] as a formalization of the negotiation paradigm given e.g. in [3, 4]. In this model, distributed negotiations are described by combining atomic negotiations, called *atoms*. Each atom has a number of *parties* (the set of agents involved in it), and a set of possible *outcomes*. The parties of an atom agree on an outcome, which transforms the internal state of the parties, and determines the atoms each party is ready to engage in next. If each agent is always willing to engage in at most one atom, the negotiation is called *deterministic*.

For an example, consider the left part of Figure 1, which shows a deterministic negotiation with agents 1 to 4. Atoms are represented by black bars with white circles (*ports*) for the respective participating agents. Initially all agents are ready to engage in the *initial atom* n_0, where they decide whether to start discussing a proposal (outcome `y`(es)) or not (`n`(o)). If the agents agree on `n`, then the negotiation terminates with the *final atom* n_f. If they agree on `y`, then the agents build two teams to study and modify the proposal in parallel: agents 1 and 2 "move" to atom n_1, and agents 3 and 4 to n_2. After n_1 and n_2, the

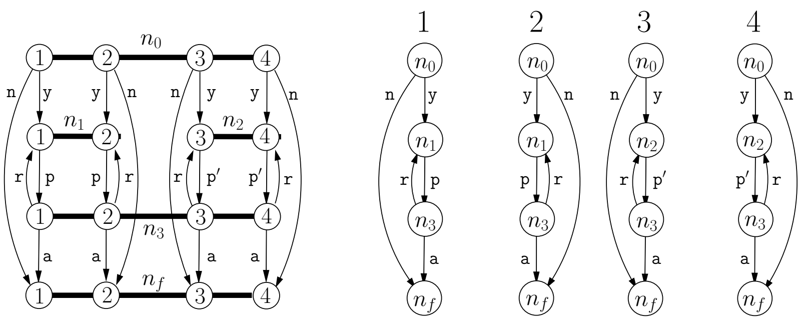


Fig. 1. Two negotiations between four agents.

four agents decide in n_3 whether to accept (outcome a) or reject (r) the revised proposal; in case of rejection, the two teams work again on revisions.

Negotiations can deadlock. For instance, if, in our example, the r-arc from port 2 of atom n_3 would lead to n_f rather than to n_1, then the negotiation reaches a deadlock after the execution of $\mathtt{y\,p\,p'\,r\,p'}$. Loosely speaking, a negotiation is *sound* if each atom can be executed in some reachable state and, whatever its current state, it can always finish, i.e., execute the final atom. In particular, soundness implies deadlock-freedom.

In this paper we investigate negotiations from a programming language point of view. Negotiations can be seen as concurrent compositions of flowcharts, one for each agent. For example, the negotiation on the left of Figure 1 is the composition of the four flowcharts shown on the right. So, just as flowcharts (or if-goto programs) model unstructured sequential programs, negotiations model unstructured concurrent programs. The Böhm-Jacopini theorem, often called the Structure Theorem [5][3], states that every flowchart has an equivalent structured program [5–7]. This raises the question we investigate in the paper: Is there a "Structure Theorem" for negotiations similar to the Böhm-Jacopini theorem for sequential computation? We give a positive answer for deterministic negotiations with a surprising twist: We exhibit a programming language with the same expressive power as *sound* deterministic negotiations. In other words, every syntactically correct program is guaranteed by construction to be sound, *and* for every sound deterministic negotiation there is an equivalent program exhibiting the same degree of concurrency. A similar question has frequently been studied for process models given as Petri nets or BPMN-diagrams, relating these models to programs in some execution language such as BPEL. In this setting, by now only partial solutions have been obtained. For example, [8] shows how to find so-called blocks in diagrams, each corresponding to a XOR-split/XOR-join-couple or to an AND-split/AND-join-couple. Process models with nested blocks are always sound and can easily be translated in a programming language, but not all sound process models have nested blocks.

An example program of our language is given in Figure 2. This program is equivalent to the negotiation of Figure 1. The first two lines of the program

[3] See [6], which convincingly argues that it should be considered a folk theorem.

```
agent  $a_1, a_2, a_3, a_4$
outcome  y, n, a, r : $\{a_1, \ldots, a_4\}$; p : $\{a_1, a_2\}$; p$'$ : $\{a_3, a_4\}$
do [] y : (p $\|$ p$'$) $\circ$
          do [] a : end [] r : (p $\|$ p$'$) loop od
          end
   [] n : end
od
```

Fig. 2. Program equivalent to the negotiation of Figure 1

specify the agents of the system, and, for each outcome, the set of agents that have to agree to choose the outcome. The outer **do** $\cdots$ **od** block corresponds to the atom n_0. The block offers a choice between outcomes `y` and `n`; in the language, outcomes are prefixed by the [] operator. After outcome `y`, the two outcomes `p` and `p`$'$ can be taken concurrently (actually, `p` is here an abbreviation of **do** p: **end od**, a block with only one possible outcome). The operator $\circ$ is the *layer composition* operator of Zwiers [9]. In every execution of $P_1 \circ P_2$, all actions of P_1 in which an agent a participates take place before all actions of P_2 in which a participates. If the sets of agents involved in P_1 and P_2 are disjoint, then P_1 and P_2 can be executed concurrently, and in this case we write $P_1 \parallel P_2$ (our language has only layer composition as primitive, and concurrent composition is just a special case). Finally, the block **do** [] `a`: **end** [] `r`: (`p` $\|$ `p`$'$) **loop od** offers a choice between two alternatives, corresponding to the outcomes `a` and `r`. The alternatives are labeled with the keywords **end** and **loop** respectively, which indicate what happens after the chosen alternative has been executed: in the case of a **loop**, the block restarts, and for an **end** it terminates.

While we have presented both negotiations and negotiation programs as data-less computational models, data can easily be added to both. In fact, in [1, 2] each agent is assumed to have an internal state (which can be given by the valuation of a set of local variables), and an outcome of an atom with a set X of agents is assigned a state transformer relation which only applies to the internal states of the involved agents. For programs, we can assign to each agent a set of local variables, and to each outcome of an atom a guarded command over (a subset of) the local variables of the participating agents of the atom. For instance, assume that the purpose of the negotiation of Figure 1 is to fix a price. Agent a_i stores his current proposal for the price in a local variable x_i $(1 \leq i \leq 4)$. The outcome `n` (no need to negotiate) is assigned the guard $x_1 = x_2 = x_3 = x_4$, while `y` is assigned its negation. The outcome `p` is assigned a command $x_1, x_2 := f(x_1, x_2)$, where f represents a (possibly nondeterministic) function that returns an agreed price between agents a_1 ad a_2. We proceed similarly with `p`$'$ and a function g. If the two proposed prices agree, the program terminates. Otherwise, a new price is negotiated by means of a third function h, and sent to the four agents.
Figure 3 shows a concrete negotiation program with data which corresponds to the abstract program of Figure 2. The i-th agent stores its current price in a variable x_i. If the prices are initially different, then agents 1 and 2 and agents 3 and 4 build two teams and come up with new suggestions for the price, a

```
agent $a_1$ var $x_1$ :int
...
agent $a_4$ var $x_4$ :int

1   do [] $\neg(x_1 = x_2 = x_3 = x_4)$ :
2          $\{x_1, x_2 := f(x_1, x_2) \parallel x_3, x_4 := g(x_3, x_4)\} \circ$
3          do [] $(x_2 = x_3)$: end
4             [] $(x_2 \neq x_3)$:
5                $x_2, x_3 = h(x_2, x_3) \circ$
6                $\{x_1, x_2 := f(x_1, x_2) \parallel x_3, x_4 := g(x_3, x_4)\}$
7                end
8           od  loop
9        [] $(x_1 = x_2 = x_3 = x_4)$ : end
10  od
```

Fig. 3. A concrete program corresponding to the abstract program of Figure 2

process encapsulated in the functions f and g. The new suggestions are stored in x_2 and x_3. If x_2 and x_3 are not equal, then agents 2 and 3 come up with a new suggestion (function h), which is then sent again to the two teams.

Notice that, according to the above recursive procedure, the set of agents executing the guards $(x_2 = x_3)$ and $(x_2 \neq x_3)$ must be equal, and this set must be a superset of the set of agents executing lines **5** and **6**. Since all variables appear in these lines, all agents must participate in the execution of the guards. If only agents a_2 and a_3 execute the guards, then the program may deadlock, because after line 2, process 1 does not know whether it has to execute line 6 or finish.

The paper is structured as follows. In the following section, we recall definitions and notations for negotiations. Section 3 introduces negotiation programs formally. In Section 4, we show how to derive a negotiation from a program. Section 5 is devoted to the converse direction, which is based on a technical result given in Section 6. Our result can be viewed as a solution to the *realizability problem*, as posed for other models, which will be discussed in Section 7.

2 Negotiations: Syntax and Semantics

We recall the main definitions of [1] for syntax and semantics of negotiations. However, here we do not consider states of agents and their transformations. Throughout the paper, we fix a finite set A of *agents* representing potential parties of negotiations.

Definition 1. *A* negotiation atom*, or just an* atom*, is a pair $n = (P_n, R_n)$, where P_n is a nonempty set of* parties *(participants) and R_n is a finite, nonempty set of* results*. For each result r, the pair (n, r), also denoted by r_n, is the* outcome *of n.*

A negotiation is a composition of atoms. We add a *transition function* $\mathcal{X}$ that assigns to each triple (n, a, r) consisting of an atom n, a party a of n, and a result r of n a set $\mathcal{X}(n, a, r)$ of atoms, the set of atomic negotiations agent a is ready to engage in after the atom n, if the result of n is r.

Definition 2. *Given a finite set of atoms N, let $T(N)$ denote the set of triples (n, a, r) such that $n \in N$, $a \in P_n$, and $r \in R_n$.*

A negotiation *is a tuple $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$, where $n_0, n_f \in N$ are the* initial *and* final *atoms, and $\mathcal{X}\colon T(N) \to 2^N$ is the* transition function*, such that*

(1) every agent of A participates in both n_0 and n_f;

(2) for every $(n, a, r) \in T(N)$: $\mathcal{X}(n, a, r) = \emptyset$ iff $n = n_f$.

The negotiation $\mathcal{N}$ is deterministic *if $|\mathcal{X}(n, a, r)| = 1$ for each $(n, a, r) \in T(N)$ satisfying $n \neq n_f$. We write $\mathcal{X}(n, a, r) = n'$ instead of $\mathcal{X}(n, a, r) = \{n'\}$.*

In this paper we consider only deterministic negotiations. In the graphical representation of a deterministic negotiation, an arc from the port of agent a in atom n, labeled by r, leads to the port of a in the unique atom of $\mathcal{X}(n, a, r)$. In the negotiation of Figure 1, the atom n_0 has possible results y and n while n_1 only has the result p. By definition, the final atom n_f has results, too. Since after each outcome (n_f, e) no agent is ready to engage in any atom, these results are not represented in the figure. Whenever we choose disjoint names for results, as we did in this example, we do not have to distinguish results and outcomes.

A *marking* of a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ is a mapping $x\colon A \to 2^N$. Intuitively, $x(a)$ is the set of atoms that agent a is currently ready to engage in next. The *initial* and *final* markings, denoted by x_0 and x_f respectively, are given by $x_0(a) = \{n_0\}$ and $x_f(a) = \emptyset$ for every $a \in A$.

A marking x *enables* an atom n if $n \in x(a)$ for every $a \in P_n$. If x enables n, then n can take place and its parties agree on a result r; we say that the outcome (n, r) *occurs*. The occurrence of (n, r) produces a next marking x' given by $x'(a) = \mathcal{X}(n, a, r)$ for every $a \in P_n$, and $x'(a) = x(a)$ for every $a \in A \setminus P_n$. We write $x \xrightarrow{(n,r)} x'$ to denote this, and call it a *small step*. By this definition, always either $x(a) = \{n_0\}$ or $x(a) = \mathcal{X}(n, a, r)$ for some atom n and outcome r. Therefore, for deterministic negotiations, $x(a)$ always contains at most one atom. We write $x_1 \xrightarrow{\sigma}$ to denote that there is a sequence $\sigma = (n_1, r_1) \ldots (n_k, r_k) \ldots$ of small steps such that $x_1 \xrightarrow{(n_1,r_1)} x_2 \xrightarrow{(n_2,r_2)} \cdots \xrightarrow{(n_k,r_k)} x_{k+1} \cdots$ We call σ *occurrence sequence* from the marking x_1, or enabled by x_1. If σ is finite then we write $x_1 \xrightarrow{\sigma} x_{k+1}$ and call x_{k+1} *reachable* from x_1. If x_1 is the initial marking, then we call σ *initial occurrence sequence*. If moreover x_{k+1} is the final marking, then σ is a *large step*.

The marking x_f can only be reached by the occurrence of (n_f, e) (e being a possible result of n_f), and it does not enable any atom. Any other marking that does not enable any atom is considered a *deadlock*.

We represent a marking x of the negotiation of Figure 1 by the vector $(x(1), x(2), x(3), x(4))$. With this notation, one of the occurrence sequences is:

$$\begin{array}{llllll} (n_0, n_0, n_0, n_0) & \xrightarrow{\mathtt{y}} & (n_1, n_1, n_2, n_2) & \xrightarrow{\mathtt{p}} & (n_3, n_3, n_2, n_2) & \xrightarrow{\mathtt{p}'} \\ (n_3, n_3, n_3, n_3) & \xrightarrow{\mathtt{a}} & (n_f, n_f, n_f, n_f) & \xrightarrow{\mathtt{e}} & (\emptyset, \emptyset, \emptyset, \emptyset) & \end{array}$$

Following [10, 11], we introduce a notion of well-behavedness of negotiations:

Definition 3. *A negotiation is* sound *if* (a) *every atom is enabled at some reachable marking, and* (b) *every initial occurrence sequence is either a large step or can be extended to a large step.*

Sound negotiations are necessarily deadlock-free. A sound negotiation also has no livelocks, i.e., it cannot reach a behaviour from which it is impossible to reach the the final marking. However, sound negotiations may not terminate. In the rest of this paper, we often consider the set of all sound and deterministic negotiations. We introduce the abbreviation *SDN* for the elements of this set.

Two distinct atoms which are both enabled at a reachable marking are *concurrently enabled*. Hence two possible next outcomes (n_1, r_1) and (n_2, r_2) are concurrent if $n_1 \neq n_2$, and they are *alternative* if $n_1 = n_2$ and $r_1 \neq r_2$. In an occurrence sequence, concurrently occurring outcomes are ordered arbitrarily. Conversely, two subsequent outcomes in an occurrence sequence occur concurrently if and only if the sets of agents participating in the respective atoms are disjoint. This fact is utilized by the concurrent semantics of negotiations, the *Mazurkiewicz trace semantics.*

A Mazurkiewicz trace language [12] is based on a finite alphabet Σ (of events) and a *dependence relation* $D \subseteq \Sigma \times \Sigma$ which is reflexive and symmetric. The *independence relation* $I = (\Sigma \times \Sigma) \setminus D$ is symmetric and irreflexive. Two subsequent independent events of a sequential observation of a concurrent run can be interchanged, and the resulting sequence is an observation of the same run, whereas the order of two subsequent dependent events matters.

Given any finite sequence σ of events over Σ, $[\sigma]$ denotes the least set of sequences which contains σ and is closed under permutation of subsequent independent events (i.e., if $\sigma_1\, a\, b\, \sigma_2 \in [\sigma]$ and $(a, b) \in I$ then $\sigma_1\, b\, a\, \sigma_2 \in [\sigma]$). Each such $[\sigma]$ is called a *trace*, and each set of traces is a *trace language*. Formally, a trace language is defined on a *distributed alphabet* (Σ, I), where Σ is an alphabet and $I \subseteq \Sigma \times \Sigma$ is an independence relation.

Traces can be composed in a natural way: for $\sigma_1, \sigma_2 \in \Sigma^*$, $[\sigma_1] \cdot [\sigma_2] := [\sigma_1\, \sigma_2]$ (it is easy to see that this is well-defined, i.e., for $[\sigma_1'] = [\sigma_1]$ and $[\sigma_2'] = [\sigma_2]$ we have $[\sigma_1\, \sigma_2] = [\sigma_1'\, \sigma_2']$). Similarly, we define composition of trace languages: if A and B are sets of traces, then $A \cdot B := \{a \cdot b \mid a \in A, b \in B\}$.

The Kleene star applied to a trace, $[\sigma]^*$, denotes the languages of all $[\sigma]^i$, for $i = 0, 1, 2, \ldots$. Similarly, for a trace language A, A^* is the union of all A^i.

Definition 4. *Let $\mathcal{N}$ be a negotiation and let Σ be the set of all outcomes of $\mathcal{N}$. Define the independence relation I by $((n_1, r_1), (n_2, r_2)) \in I$ if $P_{n_1} \cap P_{n_2} = \emptyset$ (i.e., n_1 and n_2 are* independent *if they have disjoint sets of agents). The set of* traces *of $\mathcal{N}$, denoted by $T(\mathcal{N})$, is the set of traces over (Σ, I) given by $T(\mathcal{N}) = \{[\sigma] \mid \sigma$ is a large step of $\mathcal{N}\}$.*

The outcomes $(n_1, \mathtt{p})$ and $(n_2, \mathtt{p}')$ of the negotiations of Figure 1 are independent. The set of traces is the set $\{[\sigma] \mid \sigma \in (\mathtt{n} + \mathtt{y\,p\,p'}\,(\mathtt{r\,p\,p'})^*\,\mathtt{a})\}$ (we abbreviate an outcome (n, r) to the result $\mathtt{r}$). For instance, we have $[\mathtt{y\,p\,p'\,r\,p\,p'\,a}] = \{\mathtt{y\,p\,p'\,r\,p\,p'\,a}, \mathtt{y\,p'\,p\,r\,p\,p'\,a}, \mathtt{y\,p\,p'\,r\,p'\,p\,a}, \mathtt{y\,p'\,p\,r\,p'\,p\,a}\}$.

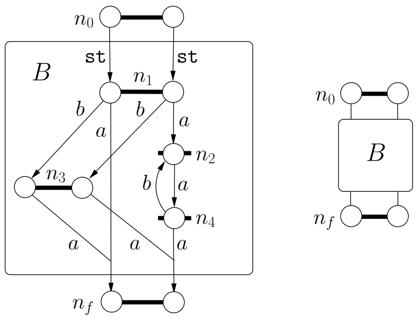


Fig. 4. Boxes.

It is convenient to assume that the initial and final atoms of a negotiation are distinct and have one single result each, for which we use the symbols `st` and `end`, respectively. We will moreover require that no port of the initial atom has an ingoing arc. If the initial atom n_0 does not satisfy this, then we add a new initial atom n_0' with a single result `st` and set $\mathcal{X}(n_0', a, \texttt{st}) = n_0$ for each agent a. For the final atom, we can easily replace all the results by a single result `end`.

Definition 5. *A negotiation* $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ *is* normed *if* n_0 *and* n_f *are distinct and have one single result, called* `st` *for* n_0 *and* `end` *for* n_f*, and satisfies* $n_0 \notin \mathcal{X}(n, a, r)$ *for each atom* n*,* $a \in P_n$ *and* $r \in R_n$*. The* normed trace semantics *of* $\mathcal{N}$ *is the set of traces* $[\![\mathcal{N}]\!] = \{\sigma \mid \texttt{st}\,\sigma\,\texttt{end} \in T(\mathcal{N})\}$.

We use the abstract graphical representation of a normed negotiation shown in Figure 4; we draw a box around its *body* and give it a name, in this case B. Due to the convention above, for each agent there is exactly one arc connecting its port in the initial atom to the body. However, there may be several arcs from the body to the port of an agent in the final atom, although we represent them as one arc. Observe that a negotiation is completely determined by its body,, the initial and final atoms just play the rôle of a wrapper.

3 Negotiation Programs

In this section, we provide a language for the specification of negotiations. As we have abstracted from states and state transformations of negotiations, we also abstract from data but concentrate on the communication between agents.

Agents can agree on negotiation outcomes. For the language, we therefore define a set of *outcome names* or *names* $\mathcal{R}$ (without stating anything about atomic negotiations yet). We fix a function $\ell \colon \mathcal{R} \to 2^A$ that assigns to each name a nonempty set of agents, intuitively the set of agents that have to agree on the outcome to be taken. For every set $X \subseteq A$, we denote by $\mathcal{R}_X$ the set of names $\texttt{r} \in \mathcal{R}$ such that $\ell(\texttt{r}) = X$.

Definition 6. *Let $\mathcal{NP}$ be the grammar consisting of the following productions for every $X \subseteq A$, every $X' \subseteq X$, and every $Y, Z \subseteq X$ such that $Y \cup Z = X$:*

$$\begin{aligned}
\mathit{prog}[X] &::= \epsilon \\
&\quad\ \mathbf{do}\ \{[]\ \mathit{endalt}[X]\}^+\ \{[]\ \mathit{loopalt}[X]\}^*\ \mathbf{od} \\
&\quad\ \mathit{prog}[Y]\ \circ\ \mathit{prog}[Z] \\
\mathit{endalt}[X] &::= \mathit{name}[X] : \mathit{prog}[X']\ \mathbf{end} \\
\mathit{loopalt}[X] &::= \mathit{name}[X] : \mathit{prog}[X']\ \mathbf{loop} \\
\mathit{name}[X] &::= \mathit{element\ of\ } \mathcal{R}_X
\end{aligned}$$

where, as usual, ϵ is the empty expression, $\{\}^+$ stands for "one or more instances of", and $\{\}^*$ for "zero or more instances of".

For every $X \subseteq A$, the negotiation programs over X *are the expressions derivable in $\mathcal{NP}$ from the nonterminal $\mathit{prog}[X]$.*

In the rest of the paper we use P_X to denote a program over the set X of agents. With this syntax, if $P_{X'}$ is a subprogram of P_X, then necessarily $X' \subseteq X$.

Intuitively, the semantics of negotiation programs is as follows:

- ϵ stands for a terminated negotiation
- **do** `body` **od** describes a negotiation starting with an atomic negotiation among the agents of X, in which they agree on one of the alternatives in the body. If they agree on an end-alternative $\mathtt{a}\colon P_{X'}$ **end**, then the program continues with $P_{X'}$ and terminates when (and if) $P_{X'}$ terminates. If they agree on a loop-alternative $\mathtt{a}\colon P_{X'}$ **loop**, then, after $P_{X'}$ terminates (if it does), the program restarts.
- $P_Y \circ P'_Z$ combines sequential and concurrent composition.If $Y \cap Z = \emptyset$, then P_Y and P'_Z are executed concurrently, and we may write $P_Y \parallel P'_Z$ instead of $P_Y \circ P'_Z$.

Formally, the semantics of a negotiation program is a set of traces over a distributed alphabet. We define the alphabet first.

Definition 7. *Given a set of agents A, outcome names $\mathcal{R}$ and a labeling function ℓ as above, the distributed alphabet over A is the pair (Σ, I), where $\Sigma = \mathcal{R}$ and $(\mathtt{a}, \mathtt{b}) \in I$ iff $\ell(a) \cap \ell(b) = \emptyset$. That is, two outcome names are independent if their corresponding sets of agents are disjoint.*

The semantics of a negotiation program P_X over a set of agents $X \subseteq A$ is the set of traces $[\![P_X]\!]$ over the distributed alphabet (Σ, I) inductively defined as follows, where E^i_X and L^j_X denote end- and loop-alternatives, respectively:

$$\begin{aligned}
[\![\epsilon]\!] &= \{[\epsilon]\} \\
[\![\mathbf{do} \mathop{[]}_{i=0}^{k} E^i_X \mathop{[]}_{j=1}^{m} L^j_X\ \mathbf{od}]\!] &= \left(\bigcup_{j=1}^{m} [\![L^j_X]\!]\right)^* \cdot \left(\bigcup_{i=0}^{k} [\![E^i_X]\!]\right) \\
[\![\mathtt{a}\colon P_{X'}]\!] &= \{[\mathtt{a}]\} \cdot [\![P_{X'}]\!] \\
[\![P_Y \circ P'_Z]\!] &= [\![P_Y]\!] \cdot [\![P'_Z]\!]
\end{aligned}$$

We use an abbreviation for **do** $\cdots$ **od** constructs with only one alternative (which must be an **end**-alternative): we shorten **do** $[]\, a\ :\ \epsilon$ **end od** to just a.

In our example, the body of the program shown in Figure 2 has the same semantics as the negotiation of Figure 1. Observe that we need to duplicate the subprogram ($\mathtt{p} \parallel \mathtt{p}'$). This is, however, already necessary in sequential computations. Consider the degenerate negotiation with only one agent obtained by "projecting" the negotiation of Figure 1 onto the first agent (shown on the right of the figure). The language of the program is given by the regular expression $\mathtt{yp}(\mathtt{rp})^*\mathtt{a}$, which also contains two occurrences of $\mathtt{p}$. No regular expression for this language contains only one occurrence of $\mathtt{p}$.

The main result of this paper, proved in the next sections, shows the equivalence between negotiation programs and sound deterministic negotiations, where a negotiation program and a SDN are equivalent if they have the same set of Mazurkiewicz traces. This equivalence not only preserves the occurrence sequences, but also concurrency. In particular, in the SDN for a program $P_1 \parallel P_2$, the negotiations for P_1 and P_2 are indeed executed concurrently. So the theorem shows that every specification is deadlock-free and can be implemented, and every sound implementation can be specified.

Theorem 1. *(a) For every negotiation program P there is a normed SDN $\mathcal{N}$ with the same set of agents such that $[\![P]\!] = [\![\mathcal{N}]\!]$. Moreover, the number of atoms and outcomes of $\mathcal{N}$ is equal to the number of* **do***-blocks of P plus 2, and the total number of outcomes of $\mathcal{N}$ is equal to the total number of alternatives of P plus 1.*

(b) For every normed SDN $\mathcal{N}$ there is a negotiation program P with the same set of agents such that $[\![P]\!] = [\![\mathcal{N}]\!]$.

In (b), the size of P can be exponential in the size of $\mathcal{N}$. This is already the case for negotiations with one single agent, in which $\mathcal{N}$ is essentially a deterministic finite automaton, and P corresponds to a regular expression for this automaton, which can be exponentially larger than the automaton itself.

4 From Programs to Normed SDNs

We show that for every negotiation program P there is a normed SDN $\mathcal{N}$ such that $[\![P]\!] = [\![\mathcal{N}]\!]$, by induction over the structure of P. First we give a SDN for the empty program, and then we give deterministic negotiations for $P_1 \circ P_2$ and **do** $[]_{i=1}^{k} \mathtt{a}_i : P_i$ **end** $[]_{j=k+1}^{k+\ell} \mathtt{a}_j : P_j$ **loop**, assuming we have produced negotiations for all P_i. In all cases, the proof that the negotiation is sound and has the same traces as the program follows easily from the definitions, and is omitted.

Definition 8. *The empty normed negotiation over a set X of agents is $\mathcal{N}^{\epsilon}_X = (\{n_0, n_f\}, n_0, n_f, \mathcal{X})$ with $\mathcal{X}(n_0, a, \mathtt{st}) = n_f$, $\mathcal{X}(n_f, a, \mathtt{end}) = \emptyset$ for each $a \in X$.*

Lemma 1. $[\![\epsilon]\!] = \{[\epsilon]\} = [\![\mathcal{N}^{\epsilon}_X]\!]$ *for every $\emptyset \neq X \subseteq A$.*

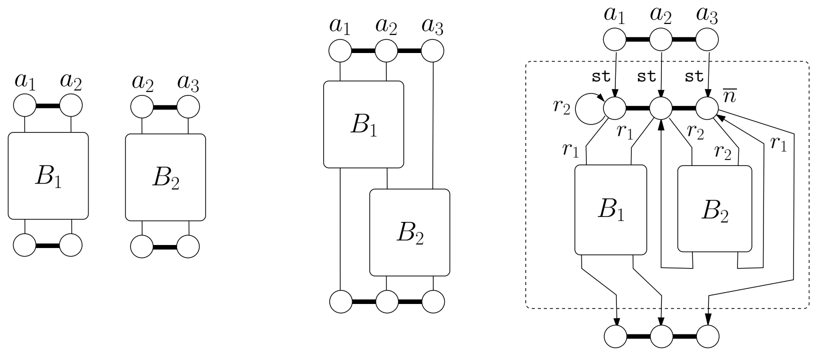


Fig. 5. The concatenation and the prefix operation.

Figure 5 illustrates the concatenation (middle) of two negotiations (left) with bodies B_1, B_2 over two not disjoint sets of agents.

Definition 9. *Let $\mathcal{N}_1 = (N_1, n_{01}, n_{f1}, \mathcal{X}_1)$, $\mathcal{N}_2 = (N_2, n_{02}, n_{f2}, \mathcal{X}_2)$ be negotiations over (not necessarily disjoint) sets of agents A_1, A_2 satisfying $N_1 \cap N_2 = \emptyset$. The negotiation $\mathcal{N}_1 \circ \mathcal{N}_2 = (N, n_0, n_f, \mathcal{X})$ over agents $A_1 \cup A_2$ is defined by:*

– $N = (N_1 \setminus \{n_{01}, n_{f1}\}) \cup (N_2 \setminus \{n_{02}, n_{f2}\}) \cup \{n_0, n_f\}$

– $\mathcal{X}(n_0, a, \mathtt{st}) = \begin{cases} \mathcal{X}_1(n_{01}, a, \mathtt{st}) \text{ if } a \in A_1 \\ \mathcal{X}_2(n_{02}, a, \mathtt{st}) \text{ if } a \in A_2 \setminus A_1 \end{cases}$

– *For every $n \in N_1$, for every $a \in P_n, r \in R_n$:*

$$\mathcal{X}(n,a,r) = \begin{cases} \mathcal{X}_1(n,a,r) \text{ if } \mathcal{X}_1(n,a,r) \neq n_{f1} \\ n_f \text{ if } \mathcal{X}_1(n,a,r) = n_{f1}, a \in A_1 \setminus A_2, \\ \mathcal{X}_1(n_{01},a,r) \text{ if } \mathcal{X}_1(n,a,r) = n_{f1}, a \in A_1 \cap A_2, \end{cases}$$

– *For every $n \in N_2$, for every $a \in P_n, r \in R_n$:*

$$\mathcal{X}(n,a,r) = \begin{cases} \mathcal{X}_2(n,a,r) \text{ if } \mathcal{X}_2(n,a,r) \neq n_{f2} \\ n_f \text{ if } \mathcal{X}_1(n,a,r) = n_{f2} \end{cases}$$

Lemma 2. *If $[\![P_1]\!] = [\![\mathcal{N}_1]\!]$ and $[\![P_2]\!] = [\![\mathcal{N}_2]\!]$, then $[\![P_1 \circ P_2]\!] = [\![\mathcal{N}_1 \circ \mathcal{N}_2]\!]$.*

Prefixing negotiations by an atom that chooses which negotiation to execute next is illustrated in Figure 5 (right) for the special case of **do** [] $r_1 : P_1$ **end** [] $r_2 : P_2$ **loop od**, where P_1, P_2 are programs over agents $\{a_1, a_2\}$ and $\{a_2, a_3\}$, respectively. As for concatenation, the textual definition is a bit laborious.

Definition 10. *Let $\mathcal{N}_1, \ldots \mathcal{N}_{k+\ell}$ be negotiations over (not necessarily disjoint) sets of agents $A_1, \ldots, A_{k+\ell}$. Let $\mathcal{N}_i = (N_i, n_{0i}, n_{fi}, \mathcal{X}_i)$ for every $1 \leq i \leq k + \ell$, where the N_i are pairwise disjoint. The negotiation*

$$\mathit{choice}[\mathcal{N}_1, \ldots, \mathcal{N}_k; \mathcal{N}_{k+1}, \ldots, \mathcal{N}_{k+\ell}] = (N, n_0, n_f, \mathcal{X})$$

over agents $A = \bigcup_{i=1}^{k+\ell} A_i$ is defined as follows:

– $N = \{\overline{n}, n_0, n_f\} \cup \bigcup_{i=1}^{k+\ell} N_i \setminus \{n_{0i}, n_{fi}\}$
– $\mathcal{X}(n_0, a, \mathbf{st}) = \overline{n}$ *for every* $a \in A$
– *For every* $1 \leq i \leq k$*:* $\mathcal{X}(\overline{n}, a, r_i) = \begin{cases} \mathcal{X}_i(n_{0i}, a, r_i) & \text{if } a \in A_i \\ n_f & \text{if } a \notin A_i \end{cases}$
– *For every* $k+1 \leq i \leq k+\ell$*:* $\mathcal{X}(\overline{n}, a, r_i) = \begin{cases} \mathcal{X}_i(n_{0i}, a, r_i) & \text{if } a \in A_i \\ \overline{n} & \text{if } a \notin A_i \end{cases}$
– *For every* $1 \leq i \leq k$*,* $n \in N_i$*,* $a \in P_n, r \in R_n$*:*

$$\mathcal{X}(n, a, r) = \begin{cases} \mathcal{X}_i(n, a, r) \text{ if } \mathcal{X}_i(n, a, r) \neq n_{fi} \\ n_f \text{ if } \mathcal{X}_i(n, a, r) = n_{fi} \end{cases}$$

– *For every* $k+1 \leq i \leq k+\ell$*,* $n \in N_i$*,* $a \in P_n, r \in R_n$*:*

$$\mathcal{X}(n, a, r) = \begin{cases} \mathcal{X}_i(n, a, r) \text{ if } \mathcal{X}_i(n, a, r) \neq n_{fi} \\ \overline{n} \text{ if } \mathcal{X}_i(n, a, r) = n_{fi} \end{cases}$$

Lemma 3. *Let* $P = \mathbf{do} \, []_{i=1}^{k} \, a_i : P_i \, \mathbf{end} \, []_{j=k+1}^{k+\ell} \, a_j : P_j \, \mathbf{loop}$*. If* $[\![P_i]\!] = [\![\mathcal{N}_i]\!]$ *for every* $1 \leq i \leq k+\ell$*, then* $[\![P]\!] = [\![\mathit{choice}[\mathcal{N}_1, \ldots \mathcal{N}_k; \mathcal{N}_{k+1}, \ldots, \mathcal{N}_{k+\ell}]]\!]$*.*

5 From Normed SDNs to Programs

We show that for every normed SDN $\mathcal{N}$ there is a negotiation program P with the same agents such that $[\![P]\!] = [\![\mathcal{N}]\!]$. For this we use the results of [1, 2] on *reduction rules*. Although we generally abstract from data aspects in this paper, states and state transformations are helpful to understand the reduction rules.

Each agent $a \in A$ has a (possibly infinite) nonempty set Q_a of *internal states*. We denote by Q_A the cartesian product $\prod_{a \in A} Q_a$. For each atom n and result $r \in R_n$, there is a *state transformer* $\delta_n(r)$ representing a non-deterministic state transforming function (this non-determinism is not related to the previously defined determinism of negotiations). Formally, $\delta_n(r)$ is a left-total relation $\delta_n(r) \subseteq Q_A \times Q_A$ satisfying: if $((q_{a_1}, \ldots, q_{a_{|A|}}), (q'_{a_1}, \ldots, q'_{a_{|A|}})) \in \delta_n(r)$ then $q_{a_i} = q'_{a_i}$ for all $a_i \notin P_n$ (only the internal states of parties of n can be transformed). We assign to each large step $\sigma = (n_0, r_0) \ldots (n_f, r_f)$ a transformer $\delta_\sigma = \delta(n_0, r_0) \cdots \delta(n_f, r_f)$ (concatenation is the usual concatenation of relations). The *summary transformer* of negotiation $\mathcal{N}$ and result r_f of the final atom n_f, $\delta_{\mathcal{N}}(r_f)$, is the union of all δ_σ for large steps σ ending with (n_f, r_f).

Two negotiations $\mathcal{N}_1$ and $\mathcal{N}_2$ over A are *semantically equivalent*, denoted $\mathcal{N}_1 \equiv \mathcal{N}_2$, if either both are not sound or if both are sound, their final atoms have the same results and $\delta_{\mathcal{N}_1}(r_f) = \delta_{\mathcal{N}_2}(r_f)$ for every final result r_f.

A *reduction rule*, or just a rule, is a binary relation on the set of negotiations. Given a rule R, we write $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ for $(\mathcal{N}_1, \mathcal{N}_2) \in R$. A rule R is *correct* if $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ implies that $\mathcal{N}_1 \equiv \mathcal{N}_2$) and therefore in particular that $\mathcal{N}_1$ is sound iff $\mathcal{N}_2$ is sound.

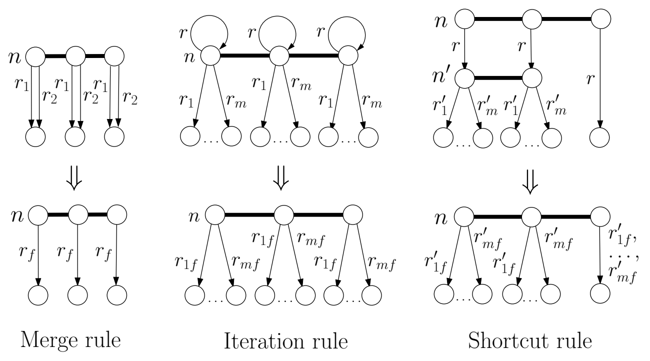


Fig. 6. The reduction rules

Given a set of rules $\mathcal{R} = \{R_1, \ldots, R_k\}$, we denote by $\mathcal{R}^*$ the reflexive and transitive closure of $R_1 \cup \ldots \cup R_k$. We say that $\mathcal{R}$ is *complete with respect to a class of negotiations* if $\mathcal{N} \xrightarrow{\mathcal{R}^*} \mathcal{N}_{min}$ holds for every negotiation $\mathcal{N}$ in the class, where $\mathcal{N}_{min}$ is a minimal negotiation of that class. In the class of sound negotiations, each minimal negotiation has a single atom, which is both initial and final. In the class of normed sound negotiations, each minimal negotiation has two atoms, an initial and a final one, and the initial one has only one result, `st`, which sends all agents to the final atom.

Given a reduction rule R, we say that R^{-1} is its associated *synthesis rule.* By the definition of completeness, for every normed SDN $\mathcal{N}$ over the set X of agents there is a chain $\mathcal{N}^{\epsilon}_X = \mathcal{N}_1 \equiv \mathcal{N}_2 \equiv \ldots \equiv \mathcal{N}_m = \mathcal{N}$ where each negotiation is obtained from the previous one through the application of a synthesis rule.

We will prove the existence of a sequence $\epsilon = P_1 \equiv P_2 \equiv \ldots \equiv P_m = P$ of programs such that $[\![P_i]\!] = [\![\mathcal{N}_i]\!]$ for every $1 \leq i \leq n$. We do so by proving the following statement for each synthesis rule R^{-1} in the following complete set of reduction rules: if $(\mathcal{N}, \mathcal{N}') \in R^{-1}$ and there is P such that $[\![P]\!] = [\![\mathcal{N}]\!]$, then there exists P' such that $[\![P']\!] = [\![\mathcal{N}']\!]$.

We repeat the correct and complete set of rules for normed SDNs from [2]. Rules are described by a *guard* and an *action*; $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ holds if $\mathcal{N}_1$ satisfies the guard and $\mathcal{N}_2$ is a possible result of applying the action to $\mathcal{N}_1$. The rules introduced in [1, 2] are summarized in Figure 6. The transformations of state transformers (δ_n) are actually not important in the present context but are provided for the sake of completeness.

Merge rule. Intuitively, this rule (Figure 6, left) merges two outcomes with identical next enabled atoms into one single outcome with a fresh label.

Guard: N contains an atom n with distinct outcomes $r_1, r_2 \in R_n$ such that $\mathcal{X}(n, a, r_1) = \mathcal{X}(n, a, r_2)$ for every $a \in A_n$.

Action: (1) $R_n \leftarrow (R_n \setminus \{r_1, r_2\}) \cup \{r_f\}$, with r_f being a fresh label.
(2) For all $a \in P_n$: $\mathcal{X}(n, a, r_f) \leftarrow \mathcal{X}(n, a, r_1)$.
(3) $\delta(n, r_f) \leftarrow \delta(n, r_1) \cup \delta(n, r_2)$.

Iteration rule. The rule replaces the iteration of an outcome r followed by some other outcome by one outcome r_f with the same effect (Figure 6, middle).

Guard: N contains an atom n with an outcome r
such that $\mathcal{X}(n, a, r) = n$ for every party a of n.
Action: (1) $R_n \leftarrow \{r'_f \mid r' \in R_n \setminus \{r\}\}$, with r'_f being a fresh label.
(2) For all $a \in P_n$: $\mathcal{X}(n, a, r'_f) \leftarrow \mathcal{X}(n, a, r') \setminus \{n\}$.
(3) For every $r'_f \in R_n$: $\delta_n(r'_f) \leftarrow \delta_n(r)^* \, \delta_n(r')$.

Shortcut rule. The shortcut rule merges the outcomes of two atoms that can occur subsequently into one single outcome with the same effect (Figure 6, right).

Given atoms n, n', we say that (n, r) *unconditionally enables* n' if $P_n \supseteq P_{n'}$ and $\mathcal{X}(n, a, r) = n'$ for every $a \in P_{n'}$. If (n, r) unconditionally enables n' then, for *every* marking x that enables n, the marking x' given by $x \xrightarrow{(n,r)} x'$ enables n'. Moreover, n' can only be disabled by its own occurrence.

Guard: N contains two distinct atoms $n, n' \neq n_0$
such that (n, r) unconditionally enables n'.
Action:
(1) $R_n \leftarrow (R_n \setminus \{r\}) \cup \{r'_f \mid r' \in R_{n'}\}$, with r'_f being fresh labels.
(2) For all $a \in P_{n'}$, $r' \in R_{n'}$: $\mathcal{X}(n, a, r'_f) \leftarrow \mathcal{X}(n', a, r')$.
For all $a \in P \setminus P_{n'}$, $r' \in R_{n'}$: $\mathcal{X}(n, a, r'_f) \leftarrow \mathcal{X}(n, a, r)$.
(3) For all $r' \in R_{n'}$: $\delta_n(r'_f) \leftarrow \delta_n(r)\delta_{n'}(r')$.
(4) If $\mathcal{X}^{-1}(n') = \emptyset$ after (1)-(3), then remove n' from N, where
$\mathcal{X}^{-1}(n') = \{(\tilde{n}, \tilde{a}, \tilde{r}) \in T(N) \mid n' \in \mathcal{X}(\tilde{n}, \tilde{a}, \tilde{r})\}$.

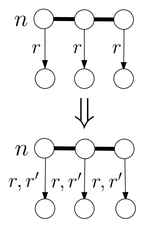
Theorem 2. *[1, 2] The merge, shortcut, and iteration rules are complete and correct for the class of deterministic negotiations (and thus preserve soundness as well as unsoundness). Moreover, every SDN with k atoms can be completely reduced by means of a polynomial number (in k) of applications of the rules.*

For defining according program rules, it is convenient to introduce *labeled programs*, in which each **do**. . .**od**-block carries a label. Two blocks carry the same label if and only if they are syntactically identical.

A labeled program P over a set of agents A *matches* a normed negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$, denoted by $P \sim_A \mathcal{N}$, if each block P' of P is labeled with an atom $n' \in N \setminus \{n_0, n_f\}$ having the same agents and outcomes as P', and for each atom $n' \in N \setminus \{n_0, n_f\}$ some block of P is labeled by n.

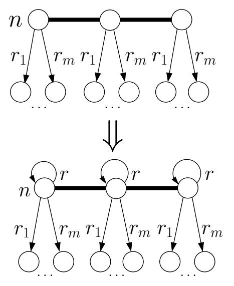
For each of the rules above we prove the following statement: if $(\mathcal{N}, \mathcal{N}') \in R^{-1}$ and there is a negotiation program P such that $[\![P]\!] = [\![\mathcal{N}]\!]$ and $P \sim_A \mathcal{N}$, then there exists a negotiation program P' such that $[\![P']\!] = [\![\mathcal{N}']\!]$, and $P' \sim_A \mathcal{N}'$. For the merge and iteration rules this is very simple, but the shortcut rule is nontrivial.

In the rest of the section, **kwd** (for keyword) stands for either **end** or **loop**.



$$\mathbf{do} \cdots [] \; r : P \; \mathbf{kwd} \; [] \cdots \; \mathbf{od}$$
$$\Downarrow$$
$$\mathbf{do} \cdots [] \; r : P \; \mathbf{kwd} \; [] \; r' : P \; \mathbf{kwd} \; [] \cdots \; \mathbf{od}$$

Fig. 7. Program rule for the (inverse of the) merge rule



$$\mathbf{do} \overset{m}{\underset{i=1}{[]}} \; r_i : P_i \; \mathbf{kwd} \; \mathbf{od}$$
$$\Downarrow$$
$$\mathbf{do} \overset{m}{\underset{i=1}{[]}} \; r_i : P_i \; \mathbf{kwd} \; [] \; r : \mathbf{loop} \; \mathbf{od}$$

Fig. 8. Program rule for the (inverse of the) iteration rule

Merge rule.

Lemma 4. *Let* $(\mathcal{N}, \mathcal{N}') \in M^{-1}$, *where* M *is the binary relation of the merge rule. If there is* P *such that* $[\![P]\!] = [\![\mathcal{N}]\!]$ *and* $P \sim_A \mathcal{N}$, *then there exists* P' *such that* $[\![P']\!] = [\![\mathcal{N}']\!]$, *and* $P' \sim_A \mathcal{N}'$.

Proof. Let (n, r) be the outcome of $\mathcal{N}$ to which the synthesis rule is applied. Since $P \sim_A \mathcal{N}$, all blocks of P labeled by n are identical and have the form

$$n :: \mathbf{do} \cdots [] \; r : P_r \; \mathbf{kwd} \; [] \cdots \mathbf{od} \tag{1}$$

for some program P_r. If P' is the result of replacing all blocks labeled by n by

$$n :: \mathbf{do} \cdots [] \; r : P_r \; \mathbf{kwd} \; [] \; r' : P_r \; \mathbf{kwd} \; [] \cdots \; \mathbf{od}$$

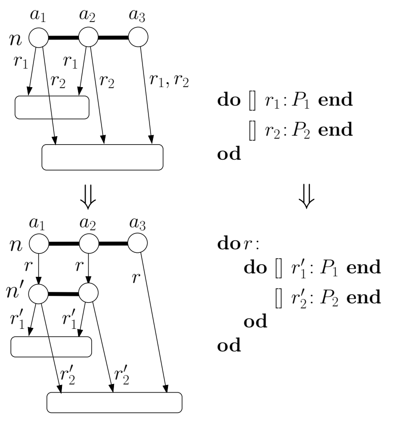
then we clearly have $[\![P']\!] = [\![\mathcal{N}']\!]$, and $P' \sim_A \mathcal{N}'$. □

Observe that, due to the duplication of P_r, the size of P' can be essentially twice the size of P.

Iteration rule.

Lemma 5. *Let* $(\mathcal{N}, \mathcal{N}') \in I^{-1}$, *where* I *is the binary relation of the iteration rule. If there is* P *such that* $[\![P]\!] = [\![\mathcal{N}]\!]$ *and* $P \sim_A \mathcal{N}$, *then there exists* P' *such that* $[\![P']\!] = [\![\mathcal{N}']\!]$, *and* $P' \sim_A \mathcal{N}'$.

Proof. Let n be the atom of $\mathcal{N}$ to which the synthesis rule adds one more outcome, and let X be the set of agents of n. Since $P \sim_A \mathcal{N}$, all blocks of P labeled



```
do [] r1: P1 end
   [] r2: P2 end
od
```

```
do r:
   do [] r1': P1 end
      [] r2': P2 end
   od
od
```

Fig. 9. The naïve program rule for the shortcut rule fails

by n are identical and have the form

$$n :: \textbf{do} \mathop{[]}_{i=1}^{m} r_i : P_i \ \textbf{kwd}_i \ \textbf{od} \tag{2}$$

Let P' be the result of replacing all blocks labeled by n by

$$n :: \textbf{do} \mathop{[]}_{i=1}^{m} r_i : P_i \ \textbf{kwd}_i \ [] \ r : \textbf{loop od}$$

Then we clearly have $[\![P']\!] = [\![\mathcal{N}']\!]$, and $P' \sim_A \mathcal{N}'$. □

Shortcut rule. The shortcut rule presents a problem, illustrated in Figure 9. The left part of the figure represents an application of the synthesis rule. Let $(\mathcal{N}, \mathcal{N}') \in S^{-1}$ be this application, where S is the binary relation of the shortcut rule. The program for $\mathcal{N}$ must contain a block labeled by n with set of agents $\{a_1, a_2, a_3\}$ and two outcomes r_1, r_2, as shown in the upper-right part of the figure. Assume, as shown in the figure, that P_1 and P_2 have $\{a_1, a_2\}$ and $\{a_1, a_2, a_3\}$ as sets of parties, respectively. Then the program for $\mathcal{N}'$ must still contain a **do**-block P for the atom n, but now with a single outcome r leading to a second **do**-block P' with two outcomes r'_1 and r'_2, leading to the programs P_1 and P_2. Since the outcome r only has a_1 and a_2 as parties, P' has to be a program derived from the nonterminal $\langle \text{prog} \rangle_{\{a_1, a_2\}}$. But then, since P_2 has $\{a_1, a_2, a_3\}$ as parties, it cannot be a subprogram of P'.

Fortunately, we can *sidestep* the problem by having a close look at the completeness proofs of [1, 2]. Those proofs imply the following result: completeness is retained if the shortcut rule is restricted to two special cases.

Definition 11. *The* one-outcome shortcut rule *is like the shortcut rule, but with the additional condition in its guard that the atom n' has only one outcome. The* same-parties shortcut rule *is like the shortcut rule, but with the additional condition in its guard that atoms n and n' have identical sets of parties.*

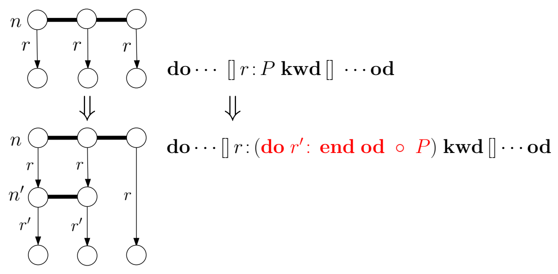

n
r
r
r
do ··· [] r : P kwd [] ··· od
⇓
⇓
n
r
r
n′
r
r′
r′
do ··· [] r : (do r′ : end od ∘ P) kwd [] ··· od

Fig. 10. Program rule mimicking the (inverse of the) one-outcome shortcut rule

The proof of this completeness result is non-trivial, and we delay it to Section 6. Assuming the result holds, we show next that we find program transformations matching the inverses of the one-outcome and same-parties shortcut rules.

Lemma 6. *Let* $(\mathcal{N}, \mathcal{N}') \in O^{-1}$*, where* O *is the binary relation of the one-outcome shortcut rule. If there is* P *such that* $[\![P]\!] = [\![\mathcal{N}]\!]$ *and* $P \sim_A \mathcal{N}$*, then there exists* P' *such that* $[\![P']\!] = [\![\mathcal{N}']\!]$*, and* $P' \sim_A \mathcal{N}'$*.*

Proof. Let n be the atom of $\mathcal{N}$ with an outcome r to which the inverse of the one-outcome rule is applied. Given a set T of traces, let $T[n, r, n', r']$ be the result of replacing in T each trace of the form $[\sigma_1\,(n, r)\,\sigma_2]$ by the trace $[\sigma_1\,(n, r)\,(n', r')\,\sigma_2]$. It follows easily from the definition of $\mathcal{N}$ and $\mathcal{N}'$ that $[\![\mathcal{N}']\!] = [\![\mathcal{N}]\!][r, r']$. The construction is illustrated in Figure 10.

Since $P \sim_A \mathcal{N}$, all blocks of P labeled by n are identical and have the form

$$B = n :: \mathbf{do} \cdots [] \;\; r : P_r \;\mathbf{kwd}\; [] \;\ldots \mathbf{od}\;.$$

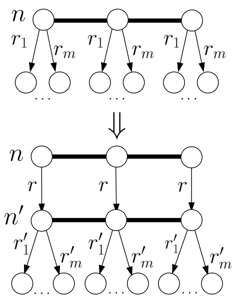
Let $P[B/B']$ be the result of replacing all blocks labeled by n by

$$B' = n :: \mathbf{do} \cdots [] \;\; r : (\mathbf{do}\; r' : \mathbf{end\;od} \;\circ\; P_r) \;\mathbf{kwd}\; [] \;\ldots \mathbf{od}\;.$$

By the definition of the program semantics we have $[\![B']\!] = [\![B]\!][n, r, n', r']$. We prove $[\![P[B/B']]\!] = [\![P]\!][n, r, n', r']$ by induction on the structure of P, which, taking $P' = P[B/B']$, concludes the proof.

– If $P = B$, then apply $P[B/B'] = B'$ and $[\![B']\!] = [\![B]\!][n, r, n', r']$.
– If $P = \mathbf{do}\, []_{i=1}^{m}\;\; r_i : P_i \;\mathbf{kwd}_i\; \mathbf{od}$, where $\mathbf{kwd}_i = \mathbf{end}$ for $1 \leq i \leq m'$ and $\mathbf{kwd}_i = \mathbf{loop}$ for $m' < i \leq m$, then $P[B/B'] = \mathbf{do}\, []_{i=1}^{m}\;\; r_i : P_i[B/B']\;\mathbf{kwd}_i\; \mathbf{od}$. By induction hypothesis $[\![P_i[B/B']]\!] = [\![P_i]\!][n, r, n', r']$, and so we get

$$\begin{aligned}
&[\![P[B/B']]\!]\\
&= \left(\textstyle\bigcup_{i=m'+1}^{m}[\![P_i[B/B']]\!]\right)^* \textstyle\bigcup_{j=1}^{m'}[\![P_j[B/B']]\!]\\
&= \left(\textstyle\bigcup_{i=m'+1}^{m}[\![P_i]\!][n, r, n', r']\right)^* \textstyle\bigcup_{j=1}^{m'}[\![P_j]\!][n, r, n', r'] \quad \text{(induction hypothesis)}\\
&= \textstyle\bigcup_{j=1}^{m'}\bigcup_{i=m'+1}^{m}\left([\![P_i]\!][n, r, n', r']\right)^* [\![P_j]\!][n, r, n', r']\\
&= \textstyle\bigcup_{j=1}^{m'}\bigcup_{i=m'+1}^{m}\left([\![P_i]\!]^*[\![P_j]\!]\right)[n, r, n', r']\\
&= \left(\left(\textstyle\bigcup_{i=m'+1}^{m}[\![P_i]\!]\right)^* \textstyle\bigcup_{j=1}^{m'}[\![P_j]\!]\right)[n, r, n', r']\\
&= [\![P]\!][n, r, n', r']
\end{aligned}$$



do $[]_{i=1}^{m}\ r_i : P_i$ **kwd**$_i$ **od**

$\Downarrow$

do $[]\ r :$ **do** $[]_{i=1}^{m}\ r'_i : P_i$ **kwd**$_i$ **od end**
od

Fig. 11. Program rule for the (inverse of the) same-parties shortcut rule

– If $P = P_1 \circ P_2$, then

$$\begin{aligned}
&[\![P[B/B']]\!]\\
&= [\![P_1[B/B']]\!] \cdot [\![P_2[B/B']]\!]\\
&= [\![P_1]\!][n,r,n',r'] \cdot [\![P_2]\!][n,r,n',r'] \quad \text{(induction hypothesis)}\\
&= ([\![P_1]\!] \cdot [\![P_2]\!])[n,r,n',r']\\
&= [\![P_1 \circ P_2]\!][n,r,n',r']
\end{aligned}$$

□

Lemma 7. *Let $(\mathcal{N}, \mathcal{N}') \in O^{-1}$, where O is the binary relation of the same parties shortcut rule. If there is P such that $[\![P]\!] = [\![\mathcal{N}]\!]$ and $P \sim_A \mathcal{N}$, then there exists P' such that $[\![P']\!] = [\![\mathcal{N}']\!]$, and $P' \sim_A \mathcal{N}'$.*

Proof. Let n be the atom of $\mathcal{N}$ with outcome r to which the inverse of the same-parties rule is applied. Given a set T of traces, let $T[n, r, n', r'_1, \ldots, r'_m]$ be the result of replacing in T each trace of the form $[\sigma_1(n, r_i)\sigma_2]$ by the trace $[\sigma_1(n, r)(n', r'_i)\sigma_2]$. It follows easily from the definition of $\mathcal{N}$ and $\mathcal{N}'$ that $[\![\mathcal{N}']\!] = [\![\mathcal{N}]\!][n, r, n', r'_1, \ldots, r'_m]$. Figure 11 illustrates this construction.

Since $P \sim_A \mathcal{N}$, all blocks of P labeled by n are identical. Let B be the syntactic expression of the block. Let $B' =$ **do** $r : B$ **end od**. Then $[\![B']\!] = [\![B]\!][n, r, n', r'_1, \ldots, r'_m]$. Let $P[B/B']$ be the result of replacing all occurrences of B in P by B'. An induction proof analogous to that of Lemma 6 shows that $[\![P[B/B']]\!] = [\![B]\!][n, r, n', r'_1, \ldots, r'_m]$. Taking $P' = P[B/B']$ we are done. □

This concludes the proof of Theorem 1 (modulo the remaining proof obligation discharged to Section 6). It was shown in [2] that every SDN $\mathcal{N}$ can be completely reduced by means of $O(a^4 \cdot r)$ applications of the rules, where a and r are the number of atoms and the total number of results of $\mathcal{N}$. Since the program rule for the inverse of the merge rule can at most duplicate the size of the program, and the other program rules only increase its size by a constant, we obtain an upper bound of $O(2^{a^4 \cdot n})$ for the size of the program P equivalent to $\mathcal{N}$. A program of linear size can be obtained by enriching the programming language with procedures. Instead of duplicating program P_r in the proof of Lemma 4, we call twice a procedure with body P_r.

6 Completeness of Rules for normed SDNs

It remains to show that the merge, iteration, one-outcome shortcut and same-parties shortcut rules are complete for normed SDNs, i.e., that they reduce every normed SDN to a negotiation with just two atoms.

Definition 12. *A* cycle *of a negotiation $\mathcal{N}$ is a sequence of outcomes (n_1, r_1), $\dots, (n_k, r_k)$ such that there are agents $a_1, \dots, a_k$ and $n_2 \in \mathcal{X}(n_1, a_1, r_1)$, $n_3 \in \mathcal{X}(n_2, a_2, r_2)$, $\dots, n_1 \in \mathcal{X}(n_k, a_k, r_k)$. The negotiation $\mathcal{N}$ is called* cyclic *if it contains a cycle, and* acyclic *otherwise.*

We consider the acyclic and cyclic cases separately.

The completeness of the rules (merge, iteration, one-outcome shortcut and same-parties shortcut) in the acyclic case was proven in [1]:

Lemma 8. *The merge rule, iteration rule, one-outcome shortcut rule and same-parties shortcut rule are complete for sound deterministic acyclic SDNs.*

Proof. This claim is an immediate consequence of Lemma 1 in [1] (our one-outcome shortcut rule is called d-shortcut rule there). Actually, Lemma 1 in [1] states that whenever the merge rule and the same-parties shortcut rule are not applicable to a sound deterministic acyclic negotiation then every agent participates in all atoms with more than one output. If the negotiation under consideration is not minimal yet, we can apply the shortcut rule to atoms n and n'. Since the same-parties shortcut rule is not applicable, n' has less parties than n, and hence not all agents participate in n'. Therefore n' can have only one outcome, and the conditions of the one-outcome shortcut rule are satisfied. $\square$

For the cyclic case, we have a closer look to the results of [2]:

Definition 13. *A* loop *is an occurrence sequence σ such that $x \xrightarrow{\sigma} x$ for some marking x reachable from the initial marking x_0. A* minimal loop *is a loop σ satisfying the property that there is no other loop σ' such that the set of atoms in σ' is a proper subset of the set of atoms in σ.*

Lemma 9 (Lemma 1 of [2]).

(1) Every cyclic SDN has a loop.
(2) The set of atoms of a minimal loop generates a strongly connected subgraph of the graph of the considered negotiation.

Usually, more than one atom is involved in a loop, and these atoms have different sets of parties. For sound deterministic negotiations, it was proven in [2] that at least one of these atoms involve all parties that participate in any of these atoms. These atoms are called *synchronizers* of the loop. In turn, a synchronizer of one loop can synchronize other loops as well. For a single atom n we consider the *fragment* of the negotiation which is constituted by all atoms and outcomes appearing in any loop synchronized by the atom n (which is nonempty only if n

is a synchronizer of at least one loop). Each fragment is cyclic by construction. Now we are looking for a fragment with the property that all its cycles pass through its generating synchronizer n. It is not difficult to see that this property is satisfied by minimal fragments, which do not properly include any smaller ones: if a cycle of a fragment does not pass through the generating synchronizer n, then there is an according loop for this cycle, which again has a synchronizer n', and the fragment generated by n' is smaller than the one generated by n.

The procedure introduced in [2] shows that a minimal fragment generated by a synchronizer n can be viewed as an acyclic sound negotiation starting with n and ending with (a copy of) n, and can thus be reduced by the same rules as for the acyclic case. This procedure ends with a minimal cycle, which enables the iteration rule. After applying this rule, the cycle vanishes. The complete procedure deletes this way cycle by cycle, until the negotiation is acyclic and can be reduced to a minimal one as above.

Another important point made in [2] is that the atoms of a minimal fragment enjoy the following property: Each atom is either a synchronizer (and has hence the same parties as the generating atom) or has no *exits*, which means that all outcomes of the atom are also outcomes of the fragment. This implies that it suffices to apply the restricted same-parties and one-outcome shortcut rules instead of the general shortcut rule also for the acyclic case, as we will argue next. We have recalled above that the restricted rules suffice for sound and deterministic acyclic negotiations, and we reduce the fragment exactly like a corresponding acyclic negotiation. If a same-parties shortcut rule is applied in the fragment, then the same rule applies to the entire negotiation. The one-outcome shortcut rule, however, requires that the reduced negotiation (called n' in the definition) has only one output. Even if this is the case within the fragment, additional outputs might exist in the entire negotiation. However, in this case this atom must be a synchronizer, and thus all parties of the fragment participate in this atom. In particular, it cannot have less parties than the other atom of the rule (called n in the definition), which implies that the additional guard of the same-parties rule is also fulfilled. In other words: For each application of the one-outcome shortcut rule in the fragment, which is not at the same time an application of the same-parties shortcut rule, the reduced atom (n') has only one outcome in the negotiation, too, and hence, the same application of the shortcut rule in the negotiation is also a one-outcome shortcut reduction.

These considerations, all from [2], prove the following lemma:

Lemma 10. *The merge rule, iteration rule, one-outcome shortcut rule and same-parties shortcut rule are complete for sound deterministic cyclic SDNs.*

Finally, recall that completeness of a set of rules means that each negotiation can be reduced to a minimal one. Minimal negotiations have a single atom, whereas minimal normed negotiations have two. Since we apply the reduction rules to normed negotiations, we still have to show that we are always able to end the reduction procedure with a minimal normed negotiation.

Theorem 3. *The merge rule, iteration rule, one-outcome shortcut rule and same-parties shortcut rule are complete for normed SDNs.*

Proof. This proof is heavily based on Lemma 8 and Lemma 10. We only have to show that for every normed SDN $\mathcal{N}$ at least one rule can be applied that does not spoil the normedness property.

By definition of the rules, application of the merge rule or of the iteration rule transforms a normed SDN into a normed SDN. For the shortcut rule, the derived negotiation might be not normed, if the rule is applied to the initial atom n_0 and its unique successor. However, it suffices to consider the restricted variants of the one-outcome shortcut rule and the same-parties shortcut rule. We moreover rule out the case that the negotiation before transformation is already a minimal normed one, i.e., we assume that it has more than two atoms. For the one-outcome shortcut rule, in the resulting negotiation, the initial atom still has one outcome only, by definition of the shortcut rule. For the same-parties shortcut rule, however, this is not necessarily the case. So we consider this case in the sequel and assume that the same-parties shortcut rule can be applied to the initial atom n_0 and its successor n_1 of a normed negotiation.

By definition of a normed negotiation, none of the ports of the initial atom has an ingoing arc. Since the same-parties shortcut rule is applicable, n_1 contains the same parties as n_0, and since n_0 is the initial atom, all agents participate in both atoms. So it is obvious that the negotiation obtained after deletion of n_0, taking n_1 as initial atom, is also sound (but not normed in general). This smaller negotiation $\mathcal{N}'$ can be reduced to a minimal negotiation by the merge rule, the iteration rule and the two restricted variants of the shortcut rule. We consider two cases: If $\mathcal{N}'$ is already minimal, it consists of a single atom. Then the considered negotiation with n_0 is already a minimal normed SDN. If $\mathcal{N}'$ is not minimal, then one of the rules can be applied to $\mathcal{N}'$. The same rule can be applied to $\mathcal{N}$, referring to the same involved atoms. $\square$

7 Conclusions

We have introduced a specification language for deterministic negotiations. The language has a very special feature: every program of the language is sound (the program can terminate from every reachable state, meaning in particular that the program is deadlock-free) *and* every sound negotiation can be specified in the language. So the language provides a syntactic characterization of soundness.

Design requirements for distributed systems are often captured with the help of scenarios, specifying the interactions that take place between sequential processes. There exist different formal notations for scenarios, depending on the underlying communication mechanism between processes. Formal notations also permit to specify multiple scenarios by means of operations like choice, concatenation, and repetition. A set of scenarios specified using such operations can be viewed as an early model of the system analyzable using formal techniques.

A key feature of scenario-based notations is that they present a global view of the system as a set of concurrent executions representing use cases. While this view is usually more intuitive for developers, implementations require a concurrent composition of sequential models, i.e., of state machines. A specification is realizable if there exists a set of state machines, one for each sequential component, whose set of concurrent behaviours coincides with the set globally specified. The *realizablity problem* consists of deciding if a given specification is realizable and, if so, computing a realization, i.e., a set of state machines. The problem has been studied for various formalisms.

For negotiations, the realizability problem reads as follows: given a syntactically correct negotiation program, is there a sound deterministic negotiation with the same behaviour? The results of this paper show that, for deterministic negotiations, the realizability problem is far more tractable than in other languages, because the answer to the above question is always positive. In turn, negotiation programs are expressively complete: every sound deterministic negotiation diagram has an equivalent negotiation program. Finally, negotiation programs can be distributed in linear time. We provided an algorithm to derive a deterministic negotiation from a program that generalizes classical constructions to derive an automaton from a regular expression. The negotiation is then projected onto its components.

Negotiations are closely related to workflow Petri nets representing business processes, and deterministic negotiations to free-choice workflow nets. Our future work transfers the concepts of this paper to the area of business processes.

References

1. Esparza, J., Desel, J.: On negotiation as concurrency primitive. In: CONCUR. (2013) 440–454
2. Esparza, J., Desel, J.: On negotiation as concurrency primitive II: Deterministic cyclic negotiations. In: FoSSaCS. (2014) 258–273
3. Davis, R., Smith, R.G.: Negotiation as a metaphor for distributed problem solving. Artificial intelligence **20**(1) (1983) 63–109
4. Jennings, N.R., Faratin, P., Lomuscio, A.R., Parsons, S., Wooldridge, M.J., Sierra, C.: Automated negotiation: prospects, methods and challenges. Group Decision and Negotiation **10**(2) (2001) 199–215
5. Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Commun. ACM **9**(5) (May 1966) 366–371
6. Harel, D.: On folk theorems. Commun. ACM **23**(7) (1980) 379–389
7. Kozen, D., Tseng, W.L.D.: The Böhm-Jacopini theorem is false, propositionally. In: MPC. (2008) 177–192
8. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data Knowl. Eng. **68**(9) (2009) 793–818
9. Zwiers, J.: Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship. Volume 321 of Lecture Notes in Computer Science. Springer (1989)
10. van der Aalst, W.M.P.: The application of Petri nets to workflow management. J. Circuits, Syst. and Comput. **08**(01) (1998) 21–66
11. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Asp. Comput. **23**(3) (2011) 333–363
12. Diekert, V., Rozenberg, G., Rozenburg, G.: The book of traces. Volume 15. World Scientific (1995)