Abstract—Negotiation diagrams are a model of concurrent computation akin to workflow Petri nets. Deterministic negotiation diagrams, equivalent to the much studied and used free-choice workflow Petri nets, are surprisingly amenable to verification. Soundness (a property close to deadlock-freedom) can be decided in \( P \text{TIME} \). Further, other fundamental questions like computing summaries or the expected cost, can also be solved in \( P \text{TIME} \) for sound deterministic negotiation diagrams, while they are \( \text{PSPACE} \)-complete in the general case.

In this paper we generalize and explain these results. We extend the classical “meet-over-all-paths” (MOP) formulation of static analysis problems to our concurrent setting, and introduce Mazurkiewicz-invariant analysis problems, which encompass the questions above and new ones. We show that any Mazurkiewicz-invariant analysis problem can be solved in \( P \text{TIME} \) for sound deterministic negotiations whenever it is in \( P \text{TIME} \) for sequential flow-graphs—even though the flow-graph of a deterministic negotiation diagram can be exponentially larger than the diagram itself. This gives a common explanation to the low-complexity of all the analysis questions studied so far. Finally, we show that classical gen/kill analyses are also an instance of our framework, and obtain a \( P \text{TIME} \) algorithm for detecting anti-patterns in free-choice workflow Petri nets.

Our result is based on a novel decomposition theorem, of independent interest, showing that sound deterministic negotiation diagrams can be hierarchically decomposed into (possibly overlapping) smaller sound diagrams.

1. Introduction

Concurrent systems are difficult to analyze due to the state explosion problem. Unlike for sequential systems, the flow graph of a concurrent system is often exponential in the size of the system, so that analysis techniques for sequential systems cannot be directly applied. One approach to analyze concurrent systems is to take a general model and design heuristics that work for relevant examples. Another, that we pursue in this paper, is to find a restricted class of concurrent systems and design provably efficient algorithms for particular analysis problems for this class.

In [6] Esparza and Desel introduced negotiation diagrams, a model of concurrency closely related to workflow Petri nets. Workflow nets are a very successful formalism for the description of business processes, and a backend for graphical notations like BPMN (Business Process Modeling Notation), EPC (Event-driven Process Chain), or UML Activity Diagrams (see e.g. [28], [30]). In a nutshell, negotiation diagrams are workflow Petri nets that can be decomposed into communicating sequential Petri nets, a feature that makes them more amenable to theoretical study, while the translation into workflow nets (described in [3]) allows to transfer results and algorithms to business process applications.

A negotiation diagram describes a distributed system with a fixed set of sequential processes. The diagram is composed of “atomic negotiations”, each one involving a (possibly different) subset of processes. An atomic negotiation starts when all its participants are ready to engage in it, and concludes with the selection of one out of a fixed set of possible outcomes; for each participant process, the outcome determines which atomic negotiations the process is willing to engage in at the next step. As workflow Petri nets, negotiations can simulate linearly bounded automata, and so all interesting analysis problems are \( \text{PSPACE} \)-hard for them.

A negotiation is deterministic if for every process the outcome of an atomic negotiation completely determines the next atomic negotiation the process should participate in. As shown in [3], the connection between negotiations diagrams and workflow Petri nets is particularly tight in the deterministic case: Deterministic negotiation diagrams are essentially isomorphic to the class of free-choice workflow nets, a class important in practice\(^1\) and extensively studied, see e.g. [8], [9], [10], [13], [16], [17], [29]). The state space of deterministic negotiations/free-choice workflow nets can grow exponentially in their size, and so they are subject to the state explosion problem. However, theoretical research has shown that, remarkably, several fundamental problems can be solved in polynomial time by means of algorithms that avoid direct exploration of the state space (contrary to other techniques, like partial-order reduction, that only reduce the number of states to be explored, and still have exponential worst-case complexity). First, it can be checked in \( P \text{TIME} \) if a deterministic negotiation diagram is sound [7], a variant of deadlock-freedom property [17]\(^2\). Then, for

---

\(^1\) For example, 70% of the almost 2000 workflow nets from the suite of industrial models studied in [8], [12], [31] are free-choice.

\(^2\) About 50% of the free-choice workflow nets from the suite mentioned above are sound.
sound deterministic negotiation diagrams PTIME algorithms have been proposed for: the summarization problem [8], the problem of computing the expected cost of a probabilistic free-choice workflow net [9], and the identification of some anti-patterns [10].

In this paper we develop a generic approach to the static analysis of sound deterministic negotiation diagrams. It covers all the problems above as particular instances, and new ones, like the computation of the best-case/worst-case execution time. The approach is a generalization to the concurrent setting of the classical lattice-based approach to static analysis of sequential flow-graphs [22]. A flow-graph consists of a set of nodes, modeling program points, and a set of edges, modeling program instructions, like assignments or guards3. In the lattice-based approach one (i) defines a lattice $D$ of dataflow informations capturing the analysis at hand, (ii) assigns semantic transformers $[a]: D \rightarrow D$ to each action $a$ of the flow-graph, (iii) assigns to a path $a_1 \cdots a_n$ of the flow graph the functional composition $[a_n] \circ \cdots \circ [a_1]$ of the transformers, and (iv) defines the result of the analysis as the “Merge Over all Paths”, i.e. the joint of the transformers of all execution paths, usually called the MOP-solution or just the MOP of the dataflow problem. So performing an analysis amounts to computing the MOP of the flow-graph for the corresponding lattice and transformers.

Katoen et al. have recently shown in [20], [5] that in order to adequately deal with quantitative analyses of concurrent systems, like expected costs, one needs a semantics that distinguishes between the inherent nondeterminism of each sequential process, and the nondeterminism introduced by concurrency (the choice of the process that should perform the next step). Following these ideas, we introduce a semantics in which the latter is resolved by an external scheduler, and define the MOP for a given scheduler. The result of a dataflow analysis is then given by the infimum or supremum, depending on the application, of the MOPs for all possible schedulers.

The contributions of the paper are the following:

(1) We present an extension of a static analysis framework to deterministic negotiation diagrams. In particular, we identify the class of Mazurkiewicz-invariant frameworks that respect the concurrency relation in negotiations. We prove a theorem showing a first important property of sound deterministic negotiations, namely that the MOP is independent of the scheduler for Mazurkiewicz invariant frameworks. This allows to compute the result of the analysis by fixing a scheduler, and computing the MOP for it. As another motivation for Mazurkiewicz-invariant frameworks we observe that there are static analysis frameworks for which analysis is NP-hard, even for sound deterministic negotiation diagrams.

(2) The main contribution of the paper is a method to compute MOP problems for sound deterministic negotiation diagrams. The method does not require the computation of the reachable configurations. We prove a novel decomposition theorem showing that a deterministic negotiation diagram is composed of smaller subnegotiations involving only a subset of the processes, and that these subnegotiations are themselves sound. This allows us to define a generic PTIME algorithm for computing the MOP for Mazurkiewicz-invariant static analysis frameworks.

(3) Finally, we show that the problems studied in [8], [9], and others, are Mazurkiewicz-invariant. Further, we show that the MOP of an important class of analyses – all four flavors of gen/kill problems, well known in the static analysis community – can be reformulated as invariant frameworks, and computed in PTIME.

Organization of the paper: Section 2 introduces the negotiation model and static analysis frameworks. Section 3 proves the decomposition theorem. Section 4 presents the algorithm to compute the MOP of an arbitrary Mazurkiewicz-invariant analysis framework. Section 5 deals with gen/kill analyses. Omitted proofs can be found in the extended version [?] .

Related work. As we have mentioned, deterministic negotiations are very close to free-choice workflow Petri nets, also called workflow graphs. Algorithms for the analysis of specific properties of these nets have been studied in [8], [9], [10], [13], [16], [17], [27], [29]. We have already described above the relation to these works.

We discuss the connection to work on static analysis for (abstract models of) programming languages. The synchronization-sensitive analysis of concurrent programs has been intensively studied (see e.g [1], [2], [11], [14], [15], [18], [21], [23], [25], [26]). A fundamental fact is that interprocedural synchronization-sensitive analysis is undecidable [23], and intraprocedural synchronization-sensitive analysis has high complexity (ranging from PSPACE-completeness to EXPSPACE-completeness, depending on the communication primitive), see e.g. [?]). This is in sharp contrast to the linear complexity of static analysis in the size of the flow graph for sequential programs, and causes work on the subject to roughly split into two research directions. The first one aims at obtaining decidability or low complexity of analyses by restricting the possible synchronization patterns. Many different restrictions have been considered: parbegin-parend constructs [11], [26], generalizations thereof (see e.g. [21]), synchronization by nested locks (see e.g. [14]), and asynchronous programming (see e.g. [18]). The other direction does not restrict the synchronization patterns, at the price of worst-case exponential analysis algorithms (see e.g. [2], [15], where control-flow of parallel programs is modelled by Petri nets, and a notion similar to Mazurkiewicz invariance is also used).

Compared with these papers, the original feature of our work is that we obtain polynomial analysis algorithms without restricting the possible synchronization patterns; instead, deterministic negotiation diagrams restrict the interplay between synchronization and choice. This distinction can be best appreciated when we compare these formalisms, but...
excluding choice. In the programming languages of [11], [14], [18], [21], [26], excluding choice means excluding if-then-else or alternative constructs, while for deterministic negotiations it means considering the special case in which every node has exactly one outcome. Sound deterministic negotiation diagrams can model all synchronization patterns given in terms of Mazurkiewicz traces, which is not the case for the formalisms of [11], [14], [18], [21], [26]. For example, the languages of [11], [26] cannot model a derivation pattern with three processes, A, B, C in which first A synchronizes with B, then A synchronizes with C, and finally B synchronizes with C. Observe that on the other hand, negotiations are finite state, whereas the other formalisms we have mentioned have non-determinism, recursion, and possibly, thread creation.

2. Negotiations

A negotiation diagram \( \mathcal{N} \) is a tuple \( \langle \text{Proc}, N, \text{dom}, R, \delta \rangle \), where Proc is a finite set of processes (or agents) and \( N \) is a finite set of nodes where the processes can synchronize to choose an outcome. The function \( \text{dom} : \mathcal{N} \to \mathcal{P} \text{(Proc)} \) associates to every node \( n \in N \) the (non-empty) set \( \text{dom}(n) \) of processes participating in it. Nodes are denoted as \( m \) or \( n \), and processes as \( p \) or \( q \); possibly with indices. The set of possible outcomes of nodes is denoted \( R \), and we use \( a, b, \ldots \) to range over its elements. Every node \( n \in N \) has its set of possible outcomes \( \text{out}(n) \subseteq R \).

The control flow in a negotiation diagram is determined by a partial transition function \( \delta : N \times R \times P \to \mathcal{P}(N) \), telling that after the outcome \( a \) of node \( n \), process \( p \in \text{dom}(n) \) is ready to participate in any of the nodes in the set \( \delta(n, a, p) \). So for every \( n' \in \delta(n, a, p) \) we have \( p \in \text{dom}(n') \cap \text{dom}(n) \), and for every \( n, a \in \text{out}(n) \) and \( p \in \text{dom}(n) \) the result \( \delta(n, a, p) \) is defined. Observe that nodes may have one single participant process, and/or have one single outcome. A location is a pair \( (n, a) \) such that \( a \in \text{out}(n) \), and we define its domain as \( \text{dom}(n) \).

![Negotiation Diagram](image)

**Figure 1.** A negotiation diagram with two processes.

**Example:** Figure 1 shows a negotiation diagram for (a slight modification of) the insurance claim example of [28] (see also Fig. 2 of [8] for a workflow Petri net model). The diagram describes a workflow for handling insurance claims by an insurance company with two departments \( D_1 \) and \( D_2 \). The processes of the negotiation are \( D_1 \) and \( D_2 \). The nodes \( n_1, n_4, n_7 \) have domain \( \{D_1, D_2\} \), and \( n_1 \) and \( n_3 \) have domain \( D_1 \), and \( n_2, n_5, n_6 \) have domain \( D_2 \). After the claim is registered, outcome \( \text{reg} \) involves both processes, \( D_1 \) sends a questionnaire to the client, and concurrently \( D_2 \) makes a first evaluation of the claim. After the client’s answer is received or a time-out occurs (outcome \( \text{tou} \)), both departments decide together at node \( n_4 \) whether to process the claim or not. In both cases \( D_1 \) has nothing further to do, and moves to the final node \( n_7 \). If the decision is to process the claim, then \( D_2 \) moves to \( n_5 \), and the claim is processed, possibly several times, until a satisfactory result is achieved (outcome \( \text{OK} \)), after which \( D_2 \) also moves to \( n_7 \).

A configuration of a negotiation diagram is a function \( C : \text{Proc} \to \mathcal{P}(N) \) mapping each process \( p \) to the set of nodes in which \( p \) is ready to engage. A node \( n \) is enabled in a configuration \( C \) if \( n \in C(p) \) for every \( p \in \text{dom}(n) \), that is, if all processes that participate in \( n \) are ready to proceed with it. A configuration is a deadlock if it has no enabled node. If node \( n \) is enabled in \( C \), and \( a \) is an outcome of \( n \), then we say that location \( (n, a) \) can be executed, and its execution produces a new configuration \( C' \) given by \( C'(p) = \delta(n, a, p) \) for \( p \in \text{dom}(n) \) and \( C'(p) = C(p) \) for \( p \notin \text{dom}(n) \). We denote this by \( C \xrightarrow{(n,a)} C' \). For example, in Figure 1 we have \( C \xrightarrow{(n_4,\text{reg})} C' \) for \( C(D_1) = \{n_0\} = C(D_2) \) and \( C'(D_1) = \{n_1\}, C'(D_2) = \{n_2\} \).

A run of a negotiation diagram \( \mathcal{N} \) from a configuration \( C_1 \) is a finite or infinite sequence of locations \( w = (n_1, a_1)(n_2, a_2) \ldots \) such that there are configurations \( C_2, C_3, \ldots \) with

\[
C_1 \xrightarrow{n_1,a_1} C_2 \xrightarrow{(n_2,a_2)} \ldots
\]

We denote this by \( C_1 \xrightarrow{w} C_k \) or \( C_1 \xrightarrow{w} C_k \) if the sequence is finite and finishes with \( C_k \). In the latter case we say that \( C_k \) is reachable from \( C_1 \) on \( w \). We simply call it reachable if \( w \) is irrelevant, and write \( C_1 \xrightarrow{w} C_k \).

Negotiation diagrams come equipped with two distinguished initial and final nodes \( n_{\text{init}} \) and \( n_{\text{fin}} \), in which all processes in Proc participate. The initial and final configurations \( C_{\text{init}}, C_{\text{fin}} \) are given by \( C_{\text{init}}(p) = \{n_{\text{init}}\} \) and \( C_{\text{fin}}(p) = \{n_{\text{fin}}\} \) for all \( p \in \text{Proc} \). A run is successful if it starts in \( C_{\text{init}} \) and ends in \( C_{\text{fin}} \). We assume that every node (except for \( n_{\text{fin}} \)) has at least one outcome. In Figure 1, \( n_{\text{init}} = n_0 \), and \( n_{\text{fin}} = n_7 \).

A negotiation diagram \( \mathcal{N} \) is sound if every partial run starting at \( C_{\text{init}} \) can be completed to a successful run. If a negotiation diagram has no infinite runs, then it is sound iff it has no reachable deadlock configuration. The negotiation diagram of Figure 1 is sound.

Process \( p \) is deterministic in a negotiation diagram \( \mathcal{N} \) if for every \( n \in N \), and \( a \in R \), the set \( \delta(n, a, p) \) of possible successor nodes is either a singleton or the empty set. A negotiation diagram is deterministic if every process \( p \in \text{Proc} \) is deterministic. The negotiation diagram of Figure 1 is deterministic.

4. \( R \) stands for result; we prefer to avoid the confusing symbol \( O \).
The graph of a negotiation diagram has \( N \) as set of vertices, and there is an edge \( n \xrightarrow{p,a} n' \) iff \( n' \in \delta(n,a,p) \). Observe that \( p \in \text{dom}(n) \cap \text{dom}(n') \).

A negotiation diagram is acyclic if its graph is acyclic. Acyclic negotiation diagrams cannot have infinite runs, so as mentioned above, soundness is equivalent to deadlock-freedom.

### 2.1. Static analysis frameworks

Let \( (D, \sqcup, \sqcap, \sqsubseteq, \sqsupseteq, \top, \bot) \) be a complete lattice. A function \( f : D \to D \) is monotonic if \( d \sqsubseteq d' \) implies \( f(d) \sqsubseteq f(d') \), and distributive if \( f(\sqcap D') = \sqcap \{ f(d) \mid d \in D' \} \).

An analysis framework\(^5\) of a negotiation diagram is a lattice together with a mapping \( [\quad] \) that assigns to each outcome \( \ell \) a monotonic and distributive function \( [\ell] \) in the lattice. Abusing language, we use \([\quad] \) to denote a framework.

A negotiation diagram has two kinds of nondeterminism, one that picks a node among the ones enabled at a configuration, and a second kind which picks an outcome. We distinguish the two by letting a scheduler to decide the first kind. This is an important design choice, motivated by modeling issues: In distributed systems, one often has information about how the outcomes are picked, but not about the way nondeterminism due to concurrency is resolved. In particular, one may have probabilistic information about the former, but not about the latter. This point has been discussed in detail by Katoen et al.\([20],[5]\), who also advocate the separation of the two kinds of nondeterminism.

A scheduler of \( N \) is a partial function \( S \) that assigns to every run \( C_{init} \xrightarrow{w} C \) a node \( S(w) \) enabled at \( C \), if it exists. A finite initial run \( w = \ell_1 \cdots \ell_k \), where \( \ell_i = (n_i, a_i) \), is compatible with \( S \) if \( S(\ell_1 \cdots \ell_i) = n_{i+1} \) for every \( 1 \leq i \leq k - 1 \).

For example, a scheduler for the negotiation diagram in Figure 1 can give preference to \( n_1 \) and \( n_3 \) over \( n_2 \). The successful runs compatible with this scheduler are given by the regular expression (omitting the nodes of the locations) \( reg \text{send}(\text{tout}|\text{rec}) \text{eval}(\text{npr}|\text{pr}(\text{done nOK})*\text{doneOK}) \).

The abstract semantics of a finite run \( w = \ell_1 \cdots \ell_k \) is the function \( [w] := [\ell_k] \circ [\ell_{k-1}] \circ \cdots \circ [\ell_1] \). The abstract semantics of \( N \) with respect to a scheduler \( S \) is the function \([N, S] \) defined by \([N, S] = \bigsqcup \{ [w] \mid w \text{ is a successful run of } N \text{ compatible with } S \}\). The extension of \( \bigsqcup \) to functions is defined pointwise.

The abstract semantics \([N] \) of \( N \) is defined as either \( \bigsqcup \{ [N, S] \mid S \text{ is a scheduler of } N \} \), or as \( \bigsqcup \{ [N, S] \mid S \text{ is a scheduler of } N \} \), depending on the application.

In classical static analysis, analysis frameworks are over flow-graphs, instead of negotiation diagrams\([22]\). Flow-graphs describe sequential programs. Loosely speaking, a flow-graph is a graph whose nodes are labeled with program points, and whose edges are labeled with program instructions (assignments or guards). The mapping \([\ ]\) assigns to an edge the relation describing the effect of the assignment or guard on the program variables. We can see a flow-graph as a degenerate negotiation diagram in which all nodes have one single process. In this case every reachable configuration enables at most one node, and so there is a unique scheduler. So, in this case, the abstract semantics of a flow-graph is the standard “Meet Over all Paths” (MOP), defined by \([N] = \bigsqcup \{ [w] \mid w \text{ is a successful run} \} \).

Several interesting analyses are instances of our framework.

### 2.1.1. Input/output semantics

Let \( V \) be a set of variables and \( Z \) the set of values. A valuation is a function \( V \to Z \), and \( Val \) denotes the set of all valuations. An element of \( D \) is a set \( T \subseteq Val \). The join and meet lattice operations are set union and intersection. For each location \( \ell = (n, a) \), the function \([\ell] \) describes for each input valuation \( v \in Val \) the set of output valuations \( [\ell](\{v\}) \) that are possible if \( n \) ends with result \( a \). For any set \( T \) of valuations the function is defined by \([\ell](T) = \bigcup_{v \in T} [\ell](\{v\}) \). The semantics \([N] \) is the relation that assigns to every initial valuation the possible final valuations after a successful run.

### 2.1.2. Detection of anti-patterns

Actions in business processes generate, use, modify, and delete resources (for example, a document can be created by a first department, read and used by a second, and classified as confidential by a third). Anti-patterns are used to describe runs that do not correctly access resources; for example, a resource is used before it is created, or a resource is created and then never used. Examples of anti-patterns can be found in\([27]\). They can be easily formalized as analyses frameworks. Consider for example two locations \( \ell_1 \) and \( \ell_2 \) that generate a resource, and a set \( K \) of locations that delete it. We wish to know if a given deterministic sound negotiation diagram has a successful run that belongs to \( L = \mathcal{L}^* \ell_1(\overline{K})^* \ell_2 \mathcal{L}^* \)

where \( \overline{K} \) denotes the set of locations not in \( K \). In other words, is there a scenario where a resource is generated twice without deleting it in between.

To encode this problem in our static analysis framework, we take \( D = \{0, 1, 2\} \) with the natural order together with \( \min, \max \) as \( \sqcap \) and \( \sqcup \), respectively. Intuitively, \( 0 \) says that the sequence does not have a suffix of the form \( \ell_1(\overline{K})^* \), \( 1 \) says that it has such a suffix, and \( 2 \) that it has a subword \( \ell_1(\overline{K})^* \ell_2 \). The semantics of a location is a monotone and distributive function from \( D \) to \( D \) reflecting this intuition:

\[
[\ell_1](x) = \begin{cases} 2 & \text{if } x = 2 \\ 1 & \text{otherwise} \end{cases} \quad [\ell_2](x) = \begin{cases} 0 & \text{if } x = 0 \\ 2 & \text{otherwise} \end{cases}
\]

\(^6\) Some classical literature uses \( \bigsqcup \) instead of \( \bigcup \) and speaks of the “Meet Over all Paths”, but other standard texts, e.g. [22], use \( \bigcup \).
$[\ell](x) = \begin{cases} 
 x & \text{if } \ell \in K \\
 2 & \text{if } \ell \in K \text{ and } x = 2 \\
 0 & \text{if } \ell \in K \text{ and } x = 0,1
\end{cases}$

2.1.3. Minimal/maximal expected cost. We let $D = \{(p,c) \mid p \in \mathbb{R}_+^N, c \in \mathbb{R}\}$, where we interpret $p$ as a probability and $c$ as a cost. We take $(p_1,c_1) \sqcup (p_2,c_2) = (p_1 + p_2,c_1p_1 + c_2p_2)$ and $(p_1,c_1) \sqsubseteq (p_2,c_2)$ if $p_1 \leq p_2$ and $c_1 \leq c_2$.

We define a function $\text{Prob}: \mathbb{N} \times R \rightarrow [0,1]$ such that $\text{Prob}(n,a) = 0$ if $a \notin \text{out}(n)$, and $\sum_{a \in R} \text{Prob}(n,a) = 1$ for every $n \in \mathbb{N}$. Intuitively, $\text{Prob}(n,a)$ is the probability that node $n$ yields the outcome $a$. We also define a cost function $\text{Cost}: \mathbb{N} \times R \rightarrow \mathbb{R}$ that assigns to each process $p$ the cost of a flow-graph does not satisfy the ascending chain condition, but yields a set of linear fixed point equations over the rational numbers, which can be solved using standard techniques.

In the concurrent case, the correspondence between MOP and MFP is more delicate. Given a scheduler $S$, we can construct the reachability graph of the negotiation diagram, corresponding to the runs compatible with $S$. If the graph is finite (for instance, this is always the case if the scheduler is memoryless, i.e., the node selected by the scheduler to extend a run depends only on the configuration reached by the run), then $[\mathcal{N},S]$ can be computed as the MFP of this graph, seen as a sequential flow-graph. The corresponding set of linear fixed point equations has one equation for each configuration of the graph. However, this approach has two problems:

(a) The number of schedulers is infinite, and non-memoryless schedulers may generate an infinite reachability graph; so we do not obtain an algorithm for computing $[\mathcal{N}]$.

(b) Even for memoryless schedulers, the size of the reachability graph may grow exponentially in the size of the negotiation diagram. So the algorithm for computing $[\mathcal{N},S]$ needs exponential time, also for lattices with only two elements.

In the remaining of this section we introduce a Mazurkiewicz-invariant analysis framework, and show that for this framework and for the class of sound deterministic negotiation diagrams we can overcome these two obstacles. In Section 2.3 we solve problem (a): We show that $[\mathcal{N}] = [\mathcal{N},S]$ for every scheduler $S$ (Theorem 1 below), and so that it suffices to compute $[\mathcal{N},S]$ for a scheduler $S$ of our choice. In the rest of the paper we solve problem (b): We give a procedure that computes $[\mathcal{N}]$ without ever constructing the reachability graph of the negotiation diagram. The procedure reduces the problem of computing the MOP to computing the MFP of a polynomial number of (sequential) flow-graphs, each of them of size at most linear in the size of the negotiation diagram. This shows that the MOP can be computed in polynomial time for a sound deterministic negotiation diagram iff it can be computed in polynomial time for a sequential flow-graph.

If we remove any of the three conditions of our setting (Mazurkiewicz-invariance, soundness, determinism), then there exist frameworks with the following property: deciding if the MOP has a given value is polynomial in the sequential case (i.e., for flow graphs of sequential programs), but at least NP-hard for negotiations.

We sketch the NP-hardness proof for deterministic, sound negotiations where the framework is not Mazurkiewicz-invariant. Consider the NP-hard problem 1-in-3-SAT, where it is asked if for a CNF formula with $k$ variables and $m$ clauses there is an assignment that sets exactly one literal true in each clause. We have $k$ processes $p_1, \ldots, p_k$, one for each variable $x_i$. We describe the (acyclic) deterministic, sound negotiation $\mathcal{N}$. The initial node of $\mathcal{N}$ has a single outcome, that leads process $p_i$ to
a node with domain \( \{ p_i \} \). From there \( p_i \) branches for the two possible values for \( x_i \). The “true” branch is a line with outcomes corresponding to clauses that become “true” when \( x_i \) is true, and analogously for the “false” branch - in both cases respecting the order of clauses. Let us denote by \( C_j \) an outcome corresponding to the \( j \)-th clause. The lattice \( D \) has elements \( \bot < 1, \ldots, m+1 < \top \); so there are \( m+1 \) pairwise incomparable elements together with \( \bot \) and \( \top \). For every node \( n \) and clause \( C_j \) we set \( \{ (n, C_j) \} (j) = j+1 \). \( \{ (n, C_j) \} (\top) = \top \) and \( \{ (n, C_j) \} (\bot) = \bot \) otherwise. Moreover over \( [\ell] \) is the identity function for all other locations \( \ell \). This framework is monotonic and distributive. For a run \( w \) of \( N \) we have \( [w](1) = m+1 \) if the subsequence of clauses appearing in \( w \) is exactly \( C_1 \ldots C_m \); otherwise \( [w](1) = \bot \). Since \( [N] \) is the \( \sqcup \) over all runs, we get that the 1-in-3-SAT instance is positive iff \( [N](1) = m+1 \). In the sequential case, the analysis can be done in polynomial time, since the lattice \( D \rightarrow D \) has the height \( O(m) \).

The proofs of the other two cases (where determinism or soundness are removed) follow easily from a simple construction shown in Theorem 1 of [6]: Given a deterministic linearly bounded automaton \( A \) and a word \( w \), one can construct in polynomial time a negotiation diagram \( N_A \) having one single run that simulates the execution of \( A \) on the input \( w \). This gives PSPACE-hardness for essentially all non-trivial frameworks, Mazurkiewicz invariant or not.

2.3. Mazurkiewicz-invariant analysis frameworks

We introduce the notion of Mazurkiewicz equivalence between runs (also called trace equivalence in the literature [4]). Two equivalent runs started in the same configuration will end up in the same configuration. We call an analysis framework Mazurkiewicz-invariant if the values of equivalent runs are the same. We then show that the MOP of a Mazurkiewicz-invariant analysis is independent of the scheduler.

**Definition 1.** Two nodes \( n, m \) of a negotiation diagram are independent if \( \text{dom}(n) \cap \text{dom}(m) = \emptyset \). Two locations are independent if their nodes are independent. Given two finite sequences of locations \( w_1, w_2 \), we write \( w_1 \sim w_2 \) if \( w_1 = w_1' \ell_1 \ell_2 w_2' \) and \( w_2 = w_2' \ell_1 \ell_2 w_1' \) for independent locations \( \ell_1, \ell_2 \). Mazurkiewicz equivalence, denoted by \( \equiv \), is the reflexive-transitive closure of \( \sim \).

The next lemma says that Mazurkiewicz equivalent runs have the same behaviors.

**Lemma 1.** If \( C_1 \xrightarrow{w} C_2 \) and \( v \equiv w \), then \( C_1 \xrightarrow{v} C_2 \). In particular, if \( w \) is a (successful) run, then \( v \) is.

Interestingly Mazurkiewicz equivalence behaves very well with respect to schedulers.

**Lemma 2.** Let \( N \) be a deterministic negotiation diagram and let \( S \) be a scheduler of \( N \). For every successful run \( w \) there is exactly one successful run \( v \equiv w \) that is compatible with \( S \).

We observe that Lemma 2 may not hold for runs that are not successful nor for non-deterministic negotiation diagrams.

We can now define Mazurkiewicz-invariant analysis frameworks, and prove that they are independent of schedulers.

**Definition 2.** An analysis framework is Mazurkiewicz-invariant if \([\ell_1] \circ [\ell_2] = [\ell_2] \circ [\ell_1] \) for every two independent outcomes \( \ell_1, \ell_2 \).

**Theorem 1.** Let \( N \) be a negotiation diagram, and let \([\_]\) be an analysis framework for \( N \). If \( N \) is deterministic and \([\_]\) is Mazurkiewicz-invariant, then \([N, S] = [N, S']\) for every two schedulers \( S, S' \), and so \([N] = [N, S]\) for every scheduler \( S \).

It turns out that many interesting analysis frameworks are Mazurkiewicz-invariant, or Mazurkiewicz-invariant under natural conditions. Let us look at the examples from Section 2.1.

The **input/output framework** is Mazurkiewicz-invariant if \([\ell_1][\ell_2] ([\{ v \}]) = [\ell_2][\ell_1] ([\{ v \}]) \) holds whenever \( \ell_1 \) and \( \ell_2 \) are independent. This is not always the case, but holds e.g. when all variables of \( V \) are local variables. Formally, the set \( V \) of variables is partitioned into sets \( V_p \) of local variables for each process \( p \). Further, \([\ell]\) involves only the local variables of the processes involved in \( \ell \). Letting \( (v_\ell, v) \) denote a valuation of \( V \), split into a valuation \( v_\ell \) of the variables of the processes of \( \ell \) and a valuation \( v \) of the rest, we have \([\ell](v_\ell, v) = (v_\ell', v') \) and \([\ell'](v_\ell, v) = [\ell](v_\ell, v') \) for every \( v_\ell, v, v' \).

The **anti-pattern framework** is not Mazurkiewicz-invariant. For example if we take some \( \ell \in K \) independent of \( \ell_1 \) then \( [\ell_1 \ell_2] \neq [\ell \ell_2] \). However, in Section 5 we will show that there is a Mazurkiewicz-invariant framework for anti-patterns.

The **minimal/maximal expected cost framework** is Mazurkiewicz-invariant. Indeed, it satisfies \([\ell_1] \circ [\ell_2] = [\ell_2] \circ [\ell_1] \) for all outcomes \( \ell_1, \ell_2 \), independent or not. Further, by Theorem 1 the expected cost is the same for every scheduler, and so the result of the analysis is the expected cost of the negotiation diagram.

The **best/worse case execution framework** is Mazurkiewicz-invariant. Intuitively, the scheduler introduces an artificial linearization of the nodes, which are however being executed in parallel. As in the previous case, the result of the analysis is the best-case/worst-case execution time of the negotiation diagram (if the negotiation diagram is cyclic and the cycle non-zero time, then the worst-case execution time is infinite).

Our next goal is a generic algorithm for computing the MOP of Mazurkiewicz-invariant frameworks for sound deterministic negotiation diagrams. This will be done in Section 4, but before we will need some results on decomposing negotiation diagrams.
3. Decomposing Sound Negotiation Diagrams

We associate with every node \( n \) and every location \( \ell \) of a sound deterministic negotiation diagram \( \mathcal{N} \) a “subnegotiation” \( \mathcal{N}_{|n} \) and \( \mathcal{N}_{|\ell} \), and prove that it is also sound. In Section 4 we use these subnegotiations to define an analysis algorithm for sound deterministic negotiation diagrams. We illustrate the results of this section on the example of Figure 2.

![Figure 2. A negotiation diagram with three processes.](image)

Intuitively, \( \mathcal{N}_{|n} \) contains the nodes \( n' \) such that \( \text{dom}(n') \subseteq \text{dom}(n) \), with transitions inherited from \( \mathcal{N} \), and \( n \) as initial node. The non-trivial part is to define the final node and show that \( \mathcal{N}_{|n} \) is sound. Given a location \( \ell = (n,a) \), the negotiation \( \mathcal{N}_{|\ell} \) contains the part of \( \mathcal{N}_{|n} \) reachable by executions that start with \( \ell \) and afterwards only use nodes with domains strictly included in \( \text{dom}(n) \). Figure 3 shows some of the subnegotiations we will obtain for some nodes and locations of Figure 2.

![Figure 3. Subnegotiations \( \mathcal{N}_{|n_3} \) (top left), \( \mathcal{N}_{|(n_1,a)} \) (bottom left), and \( \mathcal{N}_{|(n_2,a)} \) (right) of the negotiation diagram of Figure 2. Nodes unreachable from the initial node are not shown.](image)

The rest of the section first presents a theorem showing the existence and uniqueness of some special configurations (Section 3.1), and then uses it to define \( \mathcal{N}_{|n} \) and \( \mathcal{N}_{|\ell} \), and prove their soundness (Section 3.2).

3.1. Unique maximal configurations

Given a node \( n \) of a sound and deterministic negotiation, we prove the existence of a unique reachable configuration \( I(n) \) enabling \( n \) and only \( n \). Then we show the following: if we start from \( I(n) \) as initial configuration, “freeze” the processes of \( \text{Proc} \setminus \text{dom}(n) \), and let the processes of \( \text{dom}(n) \) execute maximally (i.e., until they cannot execute any node without the help of processes of \( \text{Proc} \setminus \text{dom}(n) \)), then we always reach the same “final” configuration \( F(n) \).

Additionally, given a location \( \ell = (n,a) \) we show: If from \( I(n) \) we execute \( \ell \) and let the processes of \( \text{dom}(n) \) proceed until no enabled node \( m \) satisfies \( \text{dom}(m) \subseteq \text{dom}(n) \), then again we always reach the same “final” configuration \( F(\ell) \).

Let \( X \subseteq \text{Proc} \) be a set of processes. A sequence of locations \( \ell_1, \ldots, \ell_k \) is an \( X \)-sequence if the domains of all \( \ell_i \) are included in \( X \); it is a strict \( X \)-sequence if moreover the domains of all \( \ell_i \), but possibly \( \ell_1 \), are strictly included in \( X \). We write (strict) \( n \)-sequence for (strict) \( \text{dom}(n) \)-sequence.

We write \( n' \preceq n \) if \( \text{dom}(n') \subseteq \text{dom}(n) \), and \( n' \prec n \) if \( \text{dom}(n') \subset \text{dom}(n) \) (and similarly for \( \ell' \preceq \ell \) etc). Our goal is to prove:

**Theorem 2.** Let \( m \) be a reachable node of a sound deterministic negotiation diagram \( \mathcal{N} \).

(i) There is a unique reachable configuration \( I(m) \) of \( \mathcal{N} \) that enables \( m \), and no other node.

(ii) There is a unique configuration \( F(m) \) such that

- \( F(m) \) is reachable from \( I(m) \) by means of an \( m \)-sequence, and
- for every node \( n \) enabled at \( F(m) \), \( \text{dom}(n) \) is not included in \( \text{dom}(m) \).

(iii) For every location \( \ell \) of \( m \) there is a unique configuration \( F(\ell) \) such that

- \( F(\ell) \) is reachable from \( I(m) \) by means of a strict \( m \)-sequence starting with \( \ell \), and
- for every node \( n \) enabled at \( F(\ell) \), \( \text{dom}(n) \) is not strictly included in \( \text{dom}(m) \).

E.g., in Figure 2 we have \( I(n_1) = (n_1,n_8,n_8) \) (an abbreviation for \( I(n_1)(p_1) = \{n_1\}, I(n_1)(p_2) = \{n_8\}, I(n_1)(p_3) = \{n_8\} \); \( I(n_2) = (n_8,n_2,n_2) \); and \( I(n_3) = (n_8,n_3,n_7) \). Moreover, \( F(n_1) = F(n_2) = (n_8,n_8,n_8) \); and \( F(n_3) = (n_8,n_7,n_7) \). Further, we get \( F(n_7,a) = (n_8,n_7,n_7) \) and \( F(n_7,b) = (n_8,n_2,n_2) \).

The proof of Theorem 2 is quite involved. The theorem is a consequence of the Unique Configuration lemma (Lemma 4 below), which relies on the Domination lemma (Lemma 3 below), which in turn is based on results of [7], [10].

A local path of a negotiation diagram \( \mathcal{N} \) is a path \( n_0 \xrightarrow{p_{0},a_0} n_1 \xrightarrow{p_{1},a_1} \ldots \xrightarrow{p_{k-1},a_{k-1}} n_k \) in the graph of \( \mathcal{N} \). A local path is a local circuit if \( k > 0 \) and \( n_0 = n_k \). A local path is reachable if some node in the path is reachable. The domination lemma says that every local circuit has some node containing all processes of the circuit.

**Lemma 3.** *(Domination Lemma)* Let \( \mathcal{N} \) be a deterministic sound negotiation diagram. Every reachable local circuit of \( \mathcal{N} \) contains a dominant node, i.e. a node \( n \) such that \( m \preceq n \), for every node \( m \) of the circuit.
The unique configuration lemma says that if two enabled configurations agree on a set of processes \( X \) and every enabled action in one of the two configurations needs a process from \( X \), then the two configurations are actually the same.

**Lemma 4. (Unique Configuration Lemma)** Let \( X \subseteq \text{Proc} \) be a set of processes. Let \( C_1, C_2 \) be reachable configurations such that (1) \( C_1(p) = C_2(p) \) for every \( p \in X \), and (2) every node \( v \) enabled at \( C_1 \) or \( C_2 \) satisfies \( \text{dom}(v) \cap X \neq \emptyset \). Then \( C_1 = C_2 \).

The above lemma gives Theorem 2 rather directly. For (i), take \( X = \text{dom}(m) \). Suppose that there are two configurations \( I_1 \) and \( I_2 \) as in (i). The hypotheses of Lemma 4 are satisfied, and so \( I_1 = I_2 \). The case (ii) is equally easy, while (iii) is only a bit more involved.

### 3.2. Subnegotiations for nodes and locations

We use Theorem 2 to define the subnegotiations \( N|_n^\ell \) and \( N|_\ell \) for each node \( n \) and location \( \ell \) of a sound deterministic negotiation diagram \( N \), and prove that they are sound.

**Definition 3.** Let \( N \) be a sound deterministic negotiation diagram and let \( n \) be a reachable node of \( N \). The negotiation diagram \( N|_n \) contains all the nodes and locations that appear in the \( n \)-sequences \( u \) such that \( I(n) \xrightarrow{u} F(n) \), plus a new final node \( n|_n^{\text{fin}} \) with \( \text{dom}(n|_n^{\text{fin}}) = \text{dom}(n) \). The initial node is \( n \), and the transition function \( \delta|_n \) is defined as follows. For given \( m, a, p \), we set:

\[
\delta|_n(m, a, p) = \begin{cases} 
\delta(m, a, p) & \text{if } \delta(m, a, p) \neq F(n)(p) \\
\delta|_n(m, a, p) & \text{if } \delta(m, a, p) = F(n)(p)
\end{cases}
\]

An example of \( N|_n \) is given on the left of Figure 3.

**Lemma 5.** If \( N \) is a sound deterministic negotiation diagram then so is \( N|_n \).

The subnegotiation \( N|_\ell \) induced by a location \( \ell = (n, a) \) is defined analogously to \( N|_n \), with two differences. First, in \( N|_\ell \) the node \( n \) has as the unique outcome. Second, the domain of every node of \( N|_\ell \), except the node itself, is strictly included in \( \text{dom}(n) \).

**Definition 4.** Let \( N \) be a deterministic sound negotiation diagram, let \( n \) be a reachable node of \( N \), and let \( \ell = (n, a) \) for some outcome \( a \) of \( n \). The negotiation diagram \( N|_\ell \) contains all the nodes and locations that appear in the strict \( n \)-sequences \( u \) such that \( I(n) \xrightarrow{u} F(\ell) \), plus a new final node \( n|_\ell^{\text{fin}} \). The initial node is \( n \), it has the unique outcome \( a \), and the transition function \( \delta|_\ell \) is defined as follows. For given \( m, b, p \) we set:

\[
\delta|_\ell(m, b, p) = \begin{cases} 
\delta(m, b, p) & \text{if } \delta(m, b, p) \neq F(\ell)(p) \\
\delta|_\ell(m, b, p) & \text{if } \delta(m, b, p) = F(\ell)(p)
\end{cases}
\]

Figure 3 shows (on the right) \( N|_\ell \) for the location \( \ell = (n_2, a) \) of the negotiation diagram of Figure 2.

**Lemma 6.** If \( N \) is a sound deterministic negotiation diagram, then so is \( N|_\ell \).

### 4. Computing the MOP

We use the decomposition of Section 3 to define an algorithm computing the MOP for Mazurkiewicz-invariant frameworks and arbitrary sound deterministic negotiation diagram. The idea is to repeatedly reduce parts of the negotiation diagram without changing the meaning of the whole negotiation. When reduction will be no longer possible, the negotiation diagram will have only one location, whose value will be the value of the negotiation. The goal of this section is to present Algorithm 1 and Theorem 3 that is our main result.

As we have seen in the previous section, for every node \( n \), the negotiation diagram \( N|_n \) is sound and the final configuration \( F(n) \) is unique. Thus we can safely replace all the transitions from \( n \) by one transition going directly to \( F(n) \) and assign to this transition the value of \( N|_n \). This requires to be able to compute \( F(n) \) as well as the value of \( N|_n \). For this we proceed by induction on the domain of \( n \) starting from nodes with the smallest domain. As we will see, this will require us to compute MOP only for negotiations of two (very) special forms.

**One-trace negotiations.** These are acyclic negotiation diagrams in which every node has one single outcome. In this degenerate case, all the executions of the negotiation diagram are Mazurkiewicz equivalent; moreover, by acyclicity, the trace contains every location at most once. Since the analysis framework is Mazurkiewicz-invariant, we have \([N] = [w]\) for any successful run \( w \). A successful run can be computed by just executing the negotiation diagram with some arbitrary scheduler. Once a successful run \( w \) is computed, we extract from it a flow-graph with \( |w| \) nodes, that is actually a sequence, and compute MFP.

**Replications.** Intuitively, a replication is a negotiation diagram in which all processes are involved in every node, and all processes move uniformly, that is, after they agree on an outcome they all move to the same node. Formally, a negotiation diagram is a replication if for every reachable node \( n \) and every outcome \( (n, a) \) there is a node \( m \) such that \( \delta(n, a, p) = m \) for every process \( p \). Observe that, in particular, all nodes of a replication have the same domain.

It follows that (the reachable part of) a replication is a flow-graph “in disguise”. More precisely, we can assign to it a flow-graph having one node for every reachable node, and an edge for every location \( (n, a) \), leading from \( n \) to \( \delta(n, a, p) \), where \( p \) can be chosen arbitrarily out of \( \text{dom}(n) \). It follows immediately that the MOP for the negotiation diagram is equal to the MFP of this flow-graph.

**Definition 5.** A node \( n \) is reduced if it has one single outcome, and for this single outcome \( a \) we have \( \delta(n, a, p) = F(n)(p) \) for every \( p \in \text{dom}(n) \).

A location \( \ell = (n, a) \) is reduced if \( \delta(n, a, p) = F(\ell)(p) \) for every \( p \in \text{dom}(n) \).

Now we will define an operation of reducing nodes and locations in a negotiation diagram; this is the core operation
Algorithm 1 Algorithm computing MOP for a sound deterministic negotiation diagram \( \mathcal{N} \).

1: while \( \mathcal{N} \) has non-reduced nodes do
2: \( m := \prec \)-minimal, non-reduced node
3: \( X = \text{dom}(m) \)
4: for every \( \ell = (n,a) \) with \( \text{dom}(n) = X \) do
5: \# \( \mathcal{N}_{|\ell} \) is a one-trace negotiation \#
6: \[ n, a \ell := \text{MOP}(\mathcal{N}_{|\ell}) \]
7: \( \mathcal{N} := \text{Red}_\ell(\mathcal{N}) \)
8: end for
9: for every node \( n \) such that \( \text{dom}(n) = X \) do
10: \# \( \mathcal{N}_{|n} \) is a replication \#
11: \[ n, a_n := \text{MOP}(\mathcal{N}_{|n}) \]
12: end for
13: end for
14: end while
15: return \[ \ell \]

Algorithm 1. For a location \( \ell = (n,a) \), the operation \( \text{Red}_\ell(\mathcal{N}) \) removes the transition of \( n \) on \( \ell \) and adds a new transition on \( (n,a_\ell) \). Similarly \( \text{Red}_n(\mathcal{N}) \), but this time it removes all transitions from \( n \) and adds a single new transition on \( (n,a_n) \).

**Definition 6.** Let \( \mathcal{N} = \langle \text{Proc}, N, \text{dom}, R, \delta \rangle \) be a sound deterministic negotiation diagram and let \( \ell = (n,a) \) be a non-reduced outcome of \( \mathcal{N} \). The negotiation diagram \( \text{Red}_\ell(\mathcal{N}) \) has the same components as \( \mathcal{N} \) but for out and \( \delta \) that are subject to the following changes:

- \( \text{out}(n) := \{ \text{out}(n) \setminus \{a\} \} \cup \{a_\ell\} \);
- \( \delta(n,a_\ell,p) := F(\ell)(p) \) for every process \( p \in \text{dom}(n) \).

The negotiation diagram \( \text{Red}_n(\mathcal{N}) \) is defined similarly but now:

- \( \text{out}(n) := \{a_n\} \);
- \( \delta(n,a_n,p) := F(n)(p) \) for every process \( p \in \text{dom}(n) \).

The next lemma states that these reduction operations preserve the meaning of a negotiation diagram.

**Lemma 7.** Let \( \mathcal{N} \) and \( \ell = (n,a) \) be as in Definition 6. Assign to the new location \( \ell' = (n,a_\ell) \) the mapping \( [\ell'] := [\mathcal{N}_{|\ell}] \). Then \([\mathcal{N}] = [\text{Red}_\ell(\mathcal{N})] \). Analogously, \( [\mathcal{N}] = [\text{Red}_n(\mathcal{N})] \) when we assign \([n,a_n] = [\mathcal{N}_{|n}] \).

At this point we can examine Algorithm 1. The algorithm repeatedly applies reduction operations to a given negotiation diagram. Thanks to Lemma 7 these reductions preserve the meaning of the negotiation diagram. At every reduction, the number of reachable locations in the negotiation diagram decreases. So the algorithm stops, and when it stops the negotiation diagram has only one reachable location. The abstract semantics of this location is equal to the abstract semantics of the original negotiation diagram.

This argument works if indeed we can compute \( \text{MOP}(\mathcal{N}_{|\ell}) \) and \( \text{MOP}(\mathcal{N}_{|n}) \) in lines 6 and 11 of the algorithm, respectively. For this it is enough to show that the invariants immediately preceding these lines hold, as this would mean that we deal with special cases we have discussed at the beginning of this section. The following lemma implies that the invariants indeed hold.

**Lemma 8.** Let \( \mathcal{N} \) be a sound deterministic negotiation diagram, and \( n \) a node such that all nodes \( m \prec n \) are reduced in \( \mathcal{N} \).

1. if \( a \) is an outcome of \( n \), then for \( \ell = (n,a) \) the negotiation diagram \( \mathcal{N}_{|\ell} \) is a one-trace negotiation.
2. if all locations \( \ell \preceq n \) are reduced, then \( \mathcal{N}_{|n} \) is a replication.

**Example:** Consider the negotiation diagram of Figure 2. Assume that all locations have cost 1, and that the probability of a location \( \ell = (n,a) \) is \( 1/\text{out}(n) \) (so, for example, the locations \((n_3,a) \) and \((n_3,b) \) have probability 1/2, while \((n_0,a) \) has probability 1). We compute the expected cost of the diagram using Algorithm 1.

The minimal non-reduced nodes w.r.t. \( \preceq \) are \( n_3, n_4, n_5, n_6 \). All their locations satisfy \([\mathcal{N}_{|n}] = [\ell] \). The algorithm computes \( \text{MOP}(\mathcal{N}_{|n_3}) \) for \( i = 3, 4, 5, 6 \). The subnegotiations \( \mathcal{N}_{|n_3} \) and \( \mathcal{N}_{|n_5} \) are shown in Figure 3; the other two are similar. \( \text{MOP}(\mathcal{N}_{|n_3}) \) is the expected cost of reaching \( n_{fin} \) from \( n_3 \) in \( \mathcal{N}_{|n_3} \). Since \( \mathcal{N}_{|n_3} \) is a replication (in fact, it is even a flow-graph), we can compute it as the least solution of the following fixed point equation, where we abbreviate \( \text{Prob}(\ell) \) to \( P(\ell) \) and \( \text{Cost}(\ell) \) to \( C(\ell) \):

\[
(p,c) = \left( p \cdot P(n_3,b) \cdot P(n_5,a) + P(n_3,a) \right) \\
\times \left( P(n_3,b) \cdot P(n_5,a) \cdot (C(n_3,b) + C(n_5,a) + c) + P(n_3,a) \cdot C(n_3,a) \right)
\]

which gives

\[
(p,c) = \left( \frac{1}{2}p + \frac{1}{2} \cdot \left( 2 + c + \frac{1}{2} \right) \right)
\]

with least fixed point \((1,3)\), which of course can be computed by just solving the linear equation. So \( \text{MOP}(\mathcal{N}_{|n_3}) = (1,3) \). We obtain \([n_3,a_3] = [n_4,a_4] = 3, [n_5,a_5] = [n_6,a_6] = 4, \) and the reduced negotiation diagram at the top of Figure 4. Observe that nodes \( n_5 \) and \( n_6 \) are no longer reachable.

The minimal non-reduced nodes are now \( n_2 \) and \( n_7 \). The locations of \( n_7 \) satisfy \([\mathcal{N}_{|n}] = [\ell] \). For the location \((n_2,a) \) we obtain \( \text{MOP}(\mathcal{N}_{|(n_2,a)}) = (1,1 + 3 + 3) = (1,7) \); this we can easily do because \( \mathcal{N}_{|(n_2,a)} \) is a one-trace negotiation; we just pick any successful run, for example \((n_2,a)(n_3,a_n)(n_4,a_n) \), and add the costs. After reducing \( \mathcal{N}_{|(n_2,a)} \) we obtain the negotiation diagram in the middle of Figure 4; the non-reachable nodes \( n_3, \ldots, n_6 \) are no longer displayed, and all locations have cost 1 but \((n_2,a_{(n_2,a)}) \), which has cost 7. Further, all locations \( \ell \) of \( n_2 \) and \( n_7 \).
satisfy \([N]\|_{\ell} = [\ell]\). We compute \(N|_{n_2}\) and \(N|_{n_2}\) by a fixed point calculation analogous to the one above (observe that both of them are replications), and after reduction obtain the negotiation diagram at the bottom of the figure, with \([n_7, a_{n_2}] = (1,9)\) and \([n_2, a_{n_2}] = (1,7+9) = (1,16)\). Now, the minimal non-reduced node is \(n_0\). We compute \(MOP(N|_{(n_0,a)}) \) (\(N|_{(n_0,a)}\) is a one-trace negotiation), and obtain \([n_0, a] = (1, 1+1+16) = (1,18)\), which is the final result.

\[\text{Figure 4. Three reduced negotiation diagrams constructed by Algorithm 1 started on the negotiation diagram of Figure 2.}\]

Finally, to estimate the complexity of the algorithm, we should also examine how to calculate \(\text{Red}_x(N)\) and \(\text{Red}_x(N)\) in lines 7 and 14 of the algorithm. In order to calculate \(\text{Red}_x(N)\) we need to know \(F(\ell)\). The invariant says that \(N|_{\ell}\) at that point has one trace. So it is enough to execute this trace in \(N|_{\ell}\) to reach \(F(\ell)\). Similarly, for \(\text{Red}_x(N)\) we need to know \(F(n)\). The invariant says that \(N|_{n}\) is a replication, so it is just a flow graph with one final node \(n_0\). We can execute in \(N\) any path leading to the exit to calculate \(F(n)\). To sum up, the calculations of \(\text{Red}_x(N)\) and \(\text{Red}_x(N)\) in lines 7 and 14 can be done in linear time with respect to the size of \(N\).

We summarize the results of this section:

**Theorem 3.** Let \(N\) be a sound deterministic negotiation diagram, and \([\_]\) a Mazurkiewicz invariant analysis framework. Algorithm 1 stops and outputs \([N]\). The complexity of the algorithm is \(O(|N|(C + |\_|))\) where \(|N|\) is the size of \(N\) and \(C\) is the cost of \([\_]\) analysis for flow-graphs of size \(|N|\).

5. Anti-Patterns and Gen/Kill Analyses

We saw in Section 2.1 that the problem of detecting an anti-pattern can be naturally captured as an analysis framework, which however is not Mazurkiewicz-invariant. We now show that, while the natural framework is not Mazurkiewicz invariant, an equivalent framework that returns the same result is. Then we sketch how this result generalizes to arbitrary Gen/Kill analysis frameworks, a much studied class [19].

We need some basic notions of Mazurkiewicz trace theory [4]. Given a sequence \(w = w_1 \cdots w_n \in \mathcal{L}^*\), define \(i \sqsubseteq j\) for two positions \(i, j\) of \(w\) if \(i < j\) and \(w_i\) is not independent from \(w_j\) (see Definition 1). Further, define \(\sqsubseteq\) is the transitive closure of the relation \(\sqsubseteq\). It is well-known that if \(w = v\), i.e., if \(w, v\) are Mazurkiewicz equivalent, then \(\sqsubseteq_w\) and \(\sqsubseteq_v\) are isomorphic as labeled partial orders (the labels being the locations). We write \(btw(i, j)\) for the set of positions between \(i\) and \(j\), i.e., \(\{k : i \sqsubseteq k \sqsubseteq j\}\).

Recall that anti-pattern analysis asked if there is an execution with the property “\(w \in \mathcal{L}\)” for the language \(L = \mathcal{L}^*\ell_1(K)^*\ell_2\mathcal{L}^*\). Instead of this property consider the following property of \(w\):

\((*)\) there are two positions \(i, j\) such that \(w_i = \ell_1\), \(w_j = \ell_2\), \(btw(i, j) \cap K = \emptyset\), and \(j \sqsubseteq i\).

The special case of this condition is when \(btw(i, j) = \emptyset\); then, since not \(j \sqsubseteq i\), we actually know that the two positions \(i, j\) are concurrent, so \(w\) is Mazurkiewicz equivalent to \(\ell_1\ell_2\nu\), for some \(\nu\).

**Lemma 9.** A negotiation diagram \(N\) has a successful run \(w \in \mathcal{L}\) iff it has a successful run \(v\) with property \((*)\).

It remains to set a static analysis framework for tracking property \((*)\). Since this is a property of Mazurkiewicz traces, the framework is Mazurkiewicz-invariant.

We need one more piece of notation. For a word \(w\) and a process \(p\) we write \(btw(\ell_1, p)\) for the set \(btw(i, j)\) where \(i\) is the last occurrence of \(\ell_1\), and \(j\) is the last occurrence of a location using process \(p\).

We define now an auxiliary function \(\alpha\) from sequences to \(\mathcal{P}(\text{Proc})^2 \cup \{\top\}\). We set \(\alpha(w) = \top\) if \(w\) has property \((*)\), otherwise \(\alpha(w) = (P_A, P_B)\) where

- \(p \in P_A\) if \(btw(\ell_1, p) \neq \emptyset\) and \(btw(\ell_1, p) \cap K = \emptyset\),
- \(p \in P_B\) if \(btw(\ell_1, p) \neq \emptyset\) and \(btw(\ell_1, p) \cap K \neq \emptyset\).

We describe a \(\text{PTIME}\) computable function \(F\) such that for every sequence \(w\) and location \(\ell\):

\[\alpha(w, \ell) = F(\alpha(w), \ell)\]

For defining \(F\) we first describe the update of \(btw(\ell_1, p)\) when extending \(w\) by \(\ell\). Let us detail the more interesting
case where \( \ell \neq \ell_1 \). Observe that the set of positions \( \text{btw}(\ell_1, p) \) does not change if \( p \notin \text{dom}(\ell) \). If \( p \in \text{dom}(\ell) \) then the update of \( \text{btw}(\ell_1, p) \) is the union of \( \text{btw}(\ell_1, q) \) over all \( q \in \text{dom}(\ell) \), plus \( \ell \). According to these observations, we define the update of \( F \) in the case \( \ell \neq \ell_1 \) as follows. If \( p \notin \text{dom}(\ell) \) then process \( p \) remains in its set, \( P_A \) or \( P_B \). If \( p \in \text{dom}(\ell) \), then \( p \) goes into the set \( P_B \) if either there was some \( q \in \text{dom}(\ell) \) in \( P_B \), or \( \ell \in K \) and there is some \( q \in \text{dom}(\ell) \) in \( P_A \); \( p \) goes into the set \( P_A \) if \( \ell \notin K \), no \( q \in \text{dom}(\ell) \) is in \( P_B \) and there is at least one \( q \in \text{dom}(\ell) \) in \( P_A \).

The function \( F \) can be extended to a monotonic and distributive function \( \hat{F} \) on \( \mathcal{P}(\text{Proc})^3 \cup \{\top\} \) ordered componentwise, by turning \( \alpha(w) \) into a partition of \( \text{Proc} \) (adding a component \( P_C = \text{Proc} \setminus (P_A \cup P_B) \)) and embedding it into a suitable function over \( \mathcal{P}(\text{Proc})^3 \cup \{\top\} \).

With the help of the function \( \hat{F} \) we define now the value of each location:

\[
[\ell](P_A, P_B, P_C) = \hat{F}((P_A, P_B, P_C), \ell).
\]

Observe that this gives us \([w] = \alpha(w)\) for every sequence \( w \). The above discussion yields two lemmas showing that \([\_]\) is a Mazurkiewicz invariant analysis framework that can be computed in \textsc{PTime}, since \( \hat{F} \) can be computed in \textsc{PTime}.

Lemma 10. \([\_]\) is Mazurkiewicz-invariant.

Lemma 11. Consider a negotiation diagram \( N \) over set of locations \( L \). For every sequence \( w \in L^* ; \, [w](\emptyset, \emptyset, \text{Proc}) = \top \) iff \( w \in L \). Moreover, \([N](\emptyset, \emptyset, \text{Proc}) = \top \) iff \( N \) has a successful execution in \( L \).

5.1. Generalization to Gen/Kill analyses

We consider general Gen/Kill analyses\(^8\). We are given a set of locations \( G \subseteq L \) that generate something, and a set of locations \( K \subseteq L \) (not necessarily disjoint with \( G \)) that kill this something. The lattice \( D \) has just two elements \( \{0, 1\} \), with \( \wedge \) and \( \lor \) as lattice operations, and the transformer of a program instruction \( \ell \) is of the form \([\ell](u) = (u \wedge (\ell \notin K)) \lor (\ell \in G) \). Classical examples from the static analysis of programs are the computation of reaching definitions, available expressions, live variables, very busy expressions, where the “something” are values assigned to a variable or an expression [22]. The four main classes of Gen/Kill analyses differ only on whether control-flow is interpreted forward or backwards, and on whether we do “merge over all paths” or “meet over all paths”.

- **may/forward.** For some configuration \( C \) there is an execution \( C_{init} \xrightarrow{w} C \) with \( w \in L^* G(\overline{K})^* \ell \).
- **must/forward.** For every configuration \( C \) and every execution \( C_{init} \xrightarrow{w} C \), if \( w \) ends with \( \ell \) then \( w \in L^* G(\overline{K})^* \ell \).
- **may/backward.** For some reachable configuration \( C \) there is an execution \( C \xrightarrow{w} C_{fin} \) with \( w \in \ell (\overline{K})^* G L^* \).
- **must/backward.** For every reachable configuration \( C \) and every execution \( C \xrightarrow{w} C_{fin} \), if \( w \) starts with \( \ell \) then \( w \in \ell (\overline{K})^* G L^* \).

Observe that in backward properties we require that a configuration \( C \) is reachable from the initial configuration. For forward properties we do not need to assume that a configuration is co-reachable from the final configuration, as this will be immediately implied by soundness.

The two existential properties above can be expressed in terms of the existence of successful executions of a particular form: executions \( C_{init} \xrightarrow{w} C_{fin} \) with \( w \) belonging to some language. The same is true for the negation of the universal properties. Consider the languages given by regular expressions:

1. \( E_1 = L^* G(\overline{K})^* \ell L^* \),
2. \( E_2 = (\overline{K} \cup \overline{G})^* \ell L^* \cup L^* (K \cap G)(\overline{K} \cup \overline{G})^* \ell L^* \),
3. \( E_3 = L^* \ell \overline{K} G(\overline{K})^* \),
4. \( E_4 = L^* \ell (\overline{K} \cup \overline{G})^* \ell L^* (\overline{K} \cup \overline{G})^* (K \cap G) L^* \).

**Lemma 12.** For a sound negotiation diagram we have the following:

- **may/forward** is equivalent to \( \exists C_{init} \xrightarrow{w} C_{fin} \) with \( w \in E_1 \),
- **negation of must/forward** is equivalent to \( \exists C_{init} \xrightarrow{w} C_{fin} \) with \( w \in E_2 \),
- **may/backward** is equivalent to \( \exists C_{init} \xrightarrow{w} C_{fin} \) with \( w \in E_3 \),
- **negation of must/backward** is equivalent to \( \exists C_{init} \xrightarrow{w} C_{fin} \) with \( w \in E_4 \).

The resource analysis at the beginning of this section corresponds to \( E_3 \). For each one of \( E_1, E_2, E_4 \) it is easy to produce an analogue of Lemma 9 reformulating the property in trace terms. This allows us to check all properties in polynomial time using our algorithm for Mazurkiewicz invariant analysis frameworks.

6. Conclusions

Previous work had identified deterministic negotiations – a model of concurrency essentially isomorphic to free-choice workflow Petri nets – as a class that has both practical relevance for business process modeling, and admits \textsc{PTime} analysis for several important properties once negotiations are assumed to be sound. Moreover soundness is a natural prerequisite that can be checked in \textsc{PTime}.

We have proposed a general notion of Mazurkiewicz-invariant analysis frameworks. We have shown that computing the MOP in such frameworks for sound deterministic negotiations is as easy as computing it for sequential flow graphs (while computing the MOP of general frameworks takes exponentially longer, unless \textsc{PTime}=\textsc{NP}). This result not only subsumes all previous \textsc{PTime} results on analysis of sound deterministic negotiations, but also yields \textsc{PTime} algorithms for new problems, like the computation of the best-case/worst-case execution time, the detection of anti-patterns, and general gen/kill analysis problems. The result is particularly interesting for gen/kill problems: While

8. Although not in bitvector form, which we leave for future work (see the conclusions).
their natural formulation is not in terms of Mazurkiewicz-invariant frameworks, we have shown that they can be reformulated as such.

In future work we plan to improve the degree of the polynomial bounding the runtime of our algorithm. Since our decomposition does not partition a negotiation into disjoint parts, when computing MOPs of subnegotiations we are redoing computations. Bounding the size of overlaps looks like a promising way of bringing the complexity on a par with the sequential case. Section 6 on gen/kill analyses raises a further question. In the sequential case, a gen/kill analysis can be simultaneously computed for all program points and all program variables (for example, one can compute for each program point the set of live variables at that point). This is not yet the case in our algorithm. In fact, it is not clear what is a program point in a negotiation: If one takes a configuration as a program point, then, since the number of reachable configurations can grow exponentially in the size of the negotiation diagram, any algorithm that explicitly computes the MOP for each reachable configuration has exponential worst-case complexity.

Acknowledgments. We thank the anonymous reviewers for useful remarks, and Jörg Desel, Philipp Hoffmann, and Denis Kuperberg for helpful discussions.

References