Journal of Artificial Intelligence Research 52 (2015) 543-600 Submitted 09/14; published 04/15

Distributed Evaluation of Nonmonotonic Multi-context Systems

Minh Dao-Tran DAO@KR.TUWIEN.AC.AT
Thomas Eiter EITER@KR.TUWIEN.AC.AT
Michael Fink FINK@KR.TUWIEN.AC.AT
Thomas Krennwallner TKREN@KR.TUWIEN.AC.AT
Institute für Informationssysteme, TU Wien
Favoritenstrasse 9-11, A-1040 Vienna, Austria

Abstract

Multi-context Systems (MCSs) are a formalism for systems consisting of knowledge bases (possibly heterogeneous and non-monotonic) that are interlinked via bridge rules, where the global system semantics emerges from the local semantics of the knowledge bases (also called "contexts") in an equilibrium. While MCSs and related formalisms are inherently targeted for distributed settings, no truly distributed algorithms for their evaluation were available. We address this shortcoming and present a suite of such algorithms which includes a basic algorithm DMCS, an advanced version DMCSOPT that exploits topology-based optimizations, and a streaming algorithm DMCS-STREAMING that computes equilibria in packages of bounded size. The algorithms behave quite differently in several respects, as experienced in thorough experimental evaluation of a system prototype. From the experimental results, we derive a guideline for choosing the appropriate algorithm and running mode in particular situations, determined by the parameter settings.

1. Introduction

In the last decade, there has been an increasing interest in systems that comprise information from multiple knowledge bases. This includes a wide range of application fields such as data integration, multi-agent systems, argumentation and many others. To picture a more concrete real-world application, we may consider METIS (Velikova et al., 2014), an industrial prototype system for facilitating timely human decision making in maritime control. In this application, human operators need support to determine whether a ship entering a port might hide its identity for illegal activities or might be a high risk for environmental hazard. To access such risks, METIS relies on a number of heterogeneous external information sources such as the commercial ship database IHS *Fairplay*,[1] ship tracking websites,[2] and news items for history of pollution events the ship may have been involved in.

The rise of the Word Wide Web and distributed systems has propelled this development, and to date several AI-based formalisms are available to host multiple, possibly distributed knowledge bases in a compound system. Well-known such formalisms are distributed SAT solving (Hirayama & Yokoo, 2005), distributed constraint satisfaction (Faltings & Yokoo, 2005; Yokoo & Hirayama, 2000), distributed ontologies in different flavors (Homola, 2010), MWeb (Analyti, Antoniou, & Damásio, 2011), and different approaches to multi-context systems (Giunchiglia & Serafini, 1994; Ghidini & Giunchiglia, 2001; Brewka, Roelofsen, & Serafini, 2007; Brewka & Eiter, 2007; Bikakis

1. www.ihs.com/products/maritime-information/
2. marinetraffic.com, myship.com

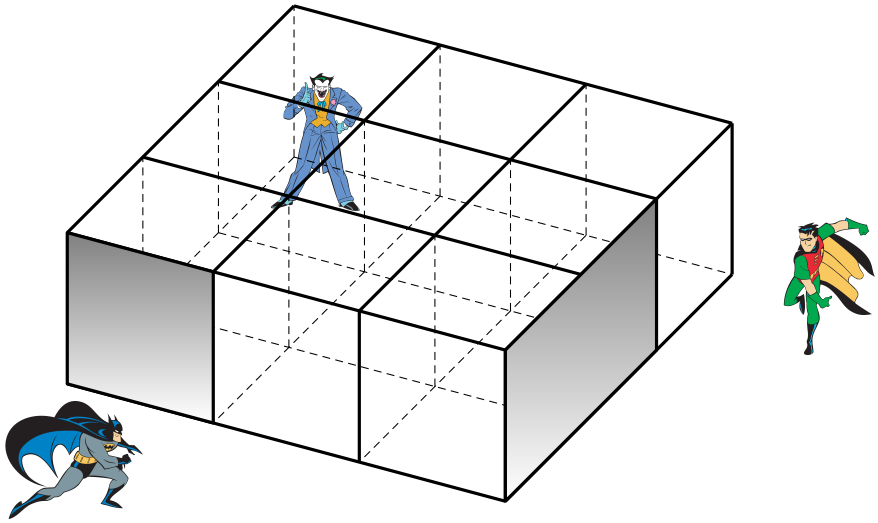


Figure 1: Pinpointing Joker

& Antoniou, 2010) rooted in McCarthy's (1993) work; among them, we focus here on *Heterogeneous Nonmonotonic Multi-context Systems* (MCSs) (Brewka & Eiter, 2007).

As a generalization of previous proposals, MCSs are a powerful formalism to specify systems of knowledge bases that may have different formats and reasoning powers, ranging from simple query answering over a relational database to reasoning over description logic knowledge bases (see Baader et al., 2003), as well as to nonmonotonic formalisms such as default logic (Reiter, 1980) or answer set programs (Gelfond & Lifschitz, 1991). To allow for heterogeneous knowledge bases and to deal with the impedance mismatch between them, MCSs abstract knowledge bases to plain mathematical structures; on top, special *bridge rules* interlink the knowledge bases, where a *bridge rule* adds a formula to a knowledge base, depending on certain beliefs at other knowledge bases. Hence the semantics of a knowledge base with associated bridge rules, which forms a *context*, depends on the other contexts, possibly in a cyclic manner. Based on this, MCSs have an equilibrium semantics in terms of global states in which every context adopts an abstract "local model," called belief set, that is conformant with the local models adopted by the other contexts and in addition obeys the bridge rules. The following simple example, which is a paraphrase of Ghidini and Giunchiglia's (2001) Magic Box, illustrates the power of this idea,

Example 1 Suppose that in a computer game, players Batman and Robin chased player Joker to a partially occluded area, as shown in Figure 1; Robin is wounded and cannot read his distance to objects. Neither Batman nor Robin can tell Joker's exact position on the 3×3 box: Batman can only assure that he is not in columns 2 and 3, while Robin can only tell that he is on row 1. However, if they exchange their partial knowledge, they can pinpoint Joker to row 1 and column 1.

We can model Batman and Robin as contexts whose local knowledge bases include their information about Joker's position, which is exchanged using bridge rules, such as "$at\_row(X) \leftarrow (2 : at\_row(X))$." for Batman, which informally "imports" Robin's knowledge (context 2) about row positions; a full encoding is given in Example 2. The equilibrium of the emerging MCS discloses then Joker's position to both Batman and Robin.

Although MCSs and related formalisms inherently target distributed systems, no truly distributed algorithms for computing equilibria of MCSs were available. Brewka and Eiter (2007) encoded equilibria into HEX-programs (Eiter, Ianni, Schindlauer, & Tompits, 2005), which can be evaluated using the dlvhex solver. However, while this approach elegantly offers full heterogeneity, it is fully centralized and needs technical assumptions. Roelofsen, Serafini, and Cimatti (2004) had proposed earlier an algorithm to check satisfiability of homogeneous, monotonic MCS with a centralized control that accesses contexts in parallel (hence is not truly distributed). Bikakis and Antoniou (2010) instead gave a distributed algorithm for their defeasible multi-context systems; however, the latter have homogeneous (possibly nonmonotonic) contexts with a particular type of semantics, and the algorithm serves query answering but not model building.

The lack of distributed algorithms for evaluating MCSs based on local context handlers is due to several obstacles:

- The abstract view of local semantics as belief sets limits for an algorithm at the global level interference with the knowledge bases and the evaluation process at each context.

- Towards real life applications, certain levels of information hiding and security are required (e.g. for information exchange between knowledge bases of companies) such that only selected information is transferred between contexts via well-defined interfaces. This prevents a context from getting more insight about its neighbors for optimization, for instance to learn conflicts (i.e., joint beliefs leading to contradiction) across contexts.

- The MCS system topology, i.e., the structure of context linkage, might be unknown at a context; this disables decomposing the system for more efficient, modular evaluation.

- The bridge rules might fuel a cyclic information flow through a group of contexts. Even if each context is easy to evaluate (e.g., all knowledge bases are acyclic logic programs), such global cycles require nontrivial care.

In this article, we address these obstacles and present results towards efficient distributed evaluation of MCSs. Our main contributions are a suite of generic algorithms DMCS, DMCSOPT, and DMCS-STREAMING which work truly distributed, and their implementation in a system prototype. In more detail, the contributions are as follows.

1.1 Algorithms and Optimization Techniques

(1) Our first, basic algorithm DMCS aims at a fully distributed setting and we deal with the obstacles above in a generic way: contexts just exchange belief sets and the call history (i.e., the access path traversing bridge rules), but no further information. At the global level, belief states are formed as tuples of belief sets; each context with bridge rules must respect the belief sets of its neighbors when computing its own belief sets using a local solver for its knowledge base. Cycles are detected from the call history, if a context gets a request and finds itself in the call history; to break a cycle, a guessing technique is used with checks on the return path.

(2) By localizing a context's knowledge about the system and information exchange, DMCS can fairly easily adapt to context changes (additions or deletions), but at the same time faces some scalability issues. To enhance the performance in an optimized version DMCSOPT, we disclose meta-level information to contexts, viz. (i) the topology of context dependencies, which is exploited for decomposing the MCS into sub-MCSs (*blocks*) that are linked in a *block-tree*, and (ii) the interface between contexts, for optimizing the data transfer between blocks. Here (i) breaks cycles in

advance and (ii) significantly reduces duplicate local evaluation; each yields a remarkable performance gain.

(3) Still as DMCS and DMCSOPT compute all equilibria of an MCS, they can not escape from scalability and memory issues, as multiple local belief sets can lead to combinatorial explosion at the global level. We thus consider computing equilibria in a streaming mode; to this end, contexts pass their belief sets not in one shot to their parents but gradually in small packages. Memory blowup can be thus avoided and moreover contexts can continue earlier than when they wait for all answers from all neighbors. This approach seems more user-friendly as equilibria gradually appear rather than all at once, possibly after long time; and one may quit the computation after seeing sufficiently many results (i.e., equilibria).

1.2 Implementation and Experiments

We have implemented the algorithms in a system prototype. To assess the effects of the optimization techniques, we have set up a benchmarking system and conducted comprehensive experiments with MCSs of various topologies and interlinking. The results confirm our expectation of the optimization techniques in general; in a nutshell, (i) the decomposition technique clearly improves the performance in the non-streaming mode; (ii) streaming is worthwhile as it may still find answers while non-streaming times out; (iii) for streaming, choosing the package size is very important; (iv) the system topology is important as some optimization techniques show drastic improvements for specific topologies; (v) sometimes, the techniques yield no gain but incur overhead.

The results of this work provide not only truly distributed algorithms for evaluating MCSs, but through this also for distributed versions of non-monotonic knowledge base formalisms as such (e.g., for distributed answer set programs), and the underlying principles and techniques might be exploited in related contexts. Furthermore, they may provide a basis for the evaluation of extensions and generalizations of MCSs, such as non-ground MCSs (Fink, Ghionna, & Weinzierl, 2011), managed MCSs (Brewka, Eiter, Fink, & Weinzierl, 2011), supported MCS (Tasharrofi & Ternovska, 2014), or reactive MCSs (Goncalves, Knorr, & Leite, 2014; Brewka, Ellmauthaler, & Pührer, 2014).

1.3 Organization

The remainder of this article is organized as follows. The next section provides preliminaries on Multi-context Systems. Section 3 introduces the basic distributed algorithm DMCS, while Section 4 develops the optimized algorithm DMCSOPT; Section 5 presents then the streaming algorithm DMCS-STREAMING. Experimental results of the prototype implementation are reported in Section 6. In Section 7, we consider related works, and in Section 8 we summarize and address further and open issues. To increase readability, proofs have been moved to the Appendix.

2. Preliminaries

This sections briefly introduces the preliminaries needed for the rest of the article.

2.1 Multi-context Systems

First, we present the formalization of Heterogeneous Nonmonotonic Multi-context Systems (MCSs) proposed by Brewka and Eiter (2007) and further described by Brewka, Eiter, and Fink (2011),

which serves as the base of this work. The idea behind MCSs is to allow different logics to be used in different contexts, and to model information flow among contexts via bridge rules. The notion of *logic* is defined as follows.

Definition 1 (cf. Brewka & Eiter, 2007) *A* logic $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ *is composed of the following components:*

1. $\mathbf{KB}_L$ *is the set of well-formed knowledge bases of* L*, each of which consists of a set of elements called formulas;*
2. $\mathbf{BS}_L$ *is the set of possible belief sets, where each* $S \in \mathbf{BS}_L$ *is a set of elements called beliefs; and*
3. $\mathbf{ACC}_L \colon \mathbf{KB}_L \to 2^{\mathbf{BS}_L}$ *is a function describing the "semantics" of the logic, by assigning to each element of* $\mathbf{KB}_L$ *a set of acceptable sets of beliefs.*

This notion of logic is very generic, and it abstracts the formation of an agent's beliefs to a bare minimum. Structure of formulas (both in the knowledge base and the belief sets) is dismissed, and they are viewed as "naked elements." Likewise no particular inference mechanism is associated with a knowledge base, nor are any logical properties imposed on belief sets; the term "belief" reflects that statements held by the agent might be on an epistemic basis, without going into further detail. The assignment of acceptable beliefs sets to a knowledge base, each of which is intuitively a set of beliefs that an agent is willing to adopt given the knowledge base, captures that in some logics (e.g., in nonmonotonic logics) multiple or even no acceptable belief sets are possible.

This abstract model allows us to capture a range of different logics for knowledge representation and reasoning, including classical logic, modal logics, epistemic logics, spatial logics, description logics etc, but also nonmonotonic logics such as default logic (Reiter, 1980) or answer set programs (Gelfond & Lifschitz, 1991), in different varieties and settings. A comparison to other formalisms is given by Brewka et al. (2011). For example, classical (propositional or predicate logic) may be modeled as follows:

- $\mathbf{KB}$: the set of (well-formed) sentences over a signature Σ,
- $\mathbf{BS}$: the set of deductively closed sets S of Σ-sentences, (i.e., $Cn(S) = S$, where $Cn(\cdot)$ denotes deductive closure),
- $\mathbf{ACC}(kb)$: the singleton containing the deductive closure of kb, i.e., $\mathbf{ACC}(kb) = \{Cn(kb)\}$.

For an example of nonmonotonic logics, (disjunctive) logic programs under answer set semantics (Gelfond & Lifschitz, 1991) can be modeled by

- $\mathbf{KB}$: the set of logic programs over a signature Σ,
- $\mathbf{BS}$: the set of consistent sets of literals over Σ,
- $\mathbf{ACC}(kb)$: the set $AS(kb)$ of answer sets of kb according to Gelfond and Lifschitz (1991).[3]

We refer to this setting, which will be used repeatedly in the sequel, as *Answer Set Programming (ASP)*. Note that the answer sets of a knowledge base kb amount to particular 3-valued models of kb; intuitively, if a positive literal p is in an answer set S, then p is known to be true, and if a negative

3. As common, we exclude inconsistent answer sets admitted by Gelfond and Lifschitz (1991).

literal $\neg p$ is in S, then p is known to be false, where "known" means that the literal is present as a fact or derivable from rules; if neither p nor $\neg p$ is in S, then the truth value of p is unknown. The above MCS modeling is a possible worlds (scenarios) view via answer sets, which can be generated by an answer set solver. However, ASP and its implementations also capture inference (truth of a query in some respectively all answer sets) and further forms of belief set formation.

Bridge rules. Based on logics, bridge rules are introduced to provide a uniform way of interlinking heterogeneous information sources as follows.

Definition 2 (cf. Brewka & Eiter, 2007) *Let $L = \{L_1, \ldots, L_n\}$ be a (multi-)set of logics. An L_k-bridge rule over L, $1 \leq k \leq n$, is of the form*

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \text{not}\,(c_{j+1} : p_{j+1}), \ldots, \text{not}\,(c_m : p_m) \tag{1}$$

where (i) for each $1 \leq i \leq m$, $c_i \in \{1, \ldots, n\}$ and p_i is an element of some belief set of L_{c_i}, and (ii) for each $kb \in \mathbf{KB}_k$, it holds that $kb \cup \{s\} \in \mathbf{KB}_k$.

Informally, bridge rules refer in their bodies to other contexts (identified by c_i) and can thus add information to a context's knowledge base depending on what is believed or disbelieved in other contexts. In contrast to Giunchiglia's (1992) multi-context systems, there is no single, global set of bridge rules; each context knows only its own bridge rules.

Now that the means for connecting contexts is available, MCSs can be formally defined.

Definition 3 (Brewka & Eiter, 2007) *A* multi-context system (MCS) *$M = (C_1, \ldots, C_n)$ consists of a collection of contexts $C_i = (L_i, kb_i, br_i)$ where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ is a knowledge base, and br_i is a set of L_i-bridge rules over $\{L_1, \ldots, L_n\}$.*

Example 2 (cont'd) The scenario from Example 1 can be formalized by an MCS $M = (C_1, C_2)$, where in both contexts L_1, L_2 are instances of Answer Set Programming, and:

- $kb_1 = F \cup F_1 \cup \left\{ \begin{array}{c} at\_col(X) \leftarrow see\_col(X). \\ \neg at\_col(X) \leftarrow \neg see\_col(X). \end{array} \right\} \cup R$,
- $br_1 = \left\{ \begin{array}{c} at\_row(X) \leftarrow (2 : at\_row(X)). \\ \neg at\_row(X) \vee covered\_row(X) \leftarrow \text{not}\,(2 : see\_row(X)), (1 : row(X)). \end{array} \right\}$
- $kb_2 = F \cup F_2 \cup \left\{ \begin{array}{c} at\_row(X) \leftarrow see\_row(X). \\ \neg at\_row(X) \leftarrow \neg see\_row(X). \end{array} \right\} \cup R$,
- $br_2 = \left\{ \begin{array}{c} at\_col(X) \leftarrow (1 : at\_col(X)). \\ \neg at\_col(X) \vee covered\_col(X) \leftarrow \text{not}\,(1 : see\_col(X)), (2 : col(X)). \end{array} \right\}$,

where

- $F = \{row(1).\ row(2).\ row(3).\ col(1).\ col(2).\ col(3).\}$,
- $F_1 = \{\neg see\_col(2).\ \neg see\_col(3).\}$,
- $F_2 = \{see\_row(1).\}$, and

- $R = \left\{ \begin{array}{l} joker\_in \leftarrow at\_row(X). \\ joker\_in \leftarrow at\_col(X). \end{array} \right\} \cup$

$$\left\{ \begin{array}{c} at\_row(X) \leftarrow joker\_in, row(X), \text{not } \neg at\_row(X). \\ \neg at\_row(X) \leftarrow joker\_in, row(X), at\_row(Y), X \neq Y. \\ at\_col(X) \leftarrow joker\_in, col(X), \text{not } \neg at\_col(X). \\ \neg at\_col(X) \leftarrow joker\_in, col(X), at\_col(Y), X \neq Y. \end{array} \right\}.$$

Here, X and Y are variables used in schematic rules, and they range over rows resp. columns (i.e., 1,2,3). Intuitively, C_1 formalizes Batman's knowledge about the scene and C_2 that of Robin. In the knowledge bases kb_1 and kb_2, the facts F represent the box of size 3×3, while F_1 and F_2 state what Batman and Robin see, viz. that Joker is not in columns 2 and 3 respectively that he is on row 1. The next two rules simply map sensed locations to respective facts. Informally, the rules in R make a guess on the row and the column where Joker is, if he is concluded to be in the box (first two rules); this may lead to multiple belief sets. Importantly, Batman adjusts his knowledge base depending on beliefs communicated by Robin (bridge rules br_1) and vice versa (bridge rules br_2).

For convenience, we introduce the following notation and conventions. For an MCS $M = (C_1, \ldots, C_n)$, we denote by $\mathbf{B}_i$ the set of all beliefs that can occur in belief sets of context C_i, i.e., $\mathbf{B}_i = \bigcup_{S \in \mathbf{BS}_i} S$, and we let $\mathbf{B}_M = \bigcup_{i=1}^{n} \mathbf{B}_i$ (simply $\mathbf{B}$, if M is understood). Without loss of generality, we assume that for distinct contexts C_i and C_j, $\mathbf{B}_i \cap \mathbf{B}_j = \emptyset$, and that for any bridge atom of the form $(i : b_i)$ appearing in any bridge rule in M, it holds that $b_i \in \mathbf{B}_i$.

2.2 Semantics of Multi-context Systems

The semantics of an MCS is defined in terms of special belief states, which are sequences $S = (S_1, \ldots, S_n)$ such that each S_i is an element of $\mathbf{BS}_i$. Intuitively, S_i should be a belief set of the knowledge base kb_i; however, also the bridge rules must be respected. To this end, kb_i is augmented with the conclusions of its bridge rules that are applicable. More precisely, a bridge rule r of form (1) is applicable in S, if $p_i \in S_{c_i}$, for $1 \leq i \leq j$, and $p_k \notin S_{c_k}$, for $j + 1 \leq k \leq m$. We denote by $head(r)$ the head of r, and by $app(R, S)$ the set of bridge rules $r \in R$ that are applicable in S. Then,

Definition 4 (Brewka & Eiter, 2007) *A belief state $S = (S_1, \ldots, S_n)$ of an MCS $M = (C_1, \ldots, C_n)$ is an* equilibrium*, if $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, for all $1 \leq i \leq n$.*

An equilibrium thus is a belief state which contains for each context an acceptable belief set, given the belief sets of the other contexts.

Example 3 (cont'd) The MCS M in Example 2 has the single equilibrium $S = (S_1, S_2)$ where $S_1 = F \cup F_1 \cup F_3$ and $S_2 = F \cup F_2 \cup F_3$ where $F_3 = \{joker\_in, at\_row(1), \neg at\_row(2), \neg at\_row(3), at\_col(1), \neg at\_col(2), \neg at\_col(3)\}$. This equilibrium indeed reflects the intuition in the scenario in Example 1, where Batman and Robin together can infer the location of Joker, while any single one of them cannot accomplish this task without communication.

Example 4 Let $M = (C_1, C_2, C_3, C_4)$ be an MCS such that all L_i are ASP logics, with signatures $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$, $\Sigma_3 = \{c, d, e\}$, and $\Sigma_4 = \{f, g\}$. Suppose

- $kb_1 = \emptyset$, $br_1 = \{a \leftarrow (2 : b), (3 : c)\}$;

- $kb_2 = \emptyset$, $br_2 = \{b \leftarrow (4 : g)\}$;
- $kb_3 = \{c \leftarrow d;\ d \leftarrow c\}$, $br_3 = \{c \vee e \leftarrow \text{not}\,(4 : f)\}$;
- $kb_4 = \{f \vee g \leftarrow\}$, $br_4 = \emptyset$.

One can check that $S = (\{a\}, \{b\}, \{c, d\}, \{g\})$ is an equilibrium of M.

The computation of equilibria for a given MCS has been realized by a declarative implementation using HEX-programs (Eiter et al., 2005) which can be evaluated using the dlvhex system.[4] The idea is to translate an MCS into a HEX-program with (i) disjunctive facts for guessing the truth values of beliefs, (ii) HEX-rules for capturing bridge rules, and (iii) constraints with external atoms for capturing the acceptability functions. For further details on a concrete implementation of this approach, we refer the reader to the MCS-IE system (Bögl, Eiter, Fink, & Schüller, 2010). In this article, we pursue a more sophisticated approach, i.e., we design and implement *distributed* algorithms, to compute equilibria of MCSs. During evaluation, there is no centralized component that controls the communication between contexts. Each context independently runs an instance of the algorithm and communicates with each other to exchange beliefs as well as to detect and break cycles. These novel contributions are described in the next sections.

3. Basic Algorithm (DMCS)

This section introduces a very first, basic, truly distributed algorithm for evaluating equilibria of an MCS. The algorithm takes a general setting as input, that is, each context has only minimal knowledge about the whole system; or in other words, it just knows the interface with direct neighbors (parents and child contexts) but not the topological information or any further metadata of the system. Under this setting, we concentrate on *distributeness*. Section 4 shifts the focus towards optimization techniques when more metadata is provided.

Taking a local stance, we consider a context C_k and compute those parts of (potential) equilibria of the system which contain coherent information from all contexts that are "reachable" from C_k.

3.1 Basic Notions

We start with some basic concepts. The import closure formally captures reachability.

Definition 5 (Import Closure) *Let $M = (C_1, \ldots, C_n)$ be an MCS. The* import neighborhood of context *C_k, $k \in \{1, \ldots, n\}$, is the set*

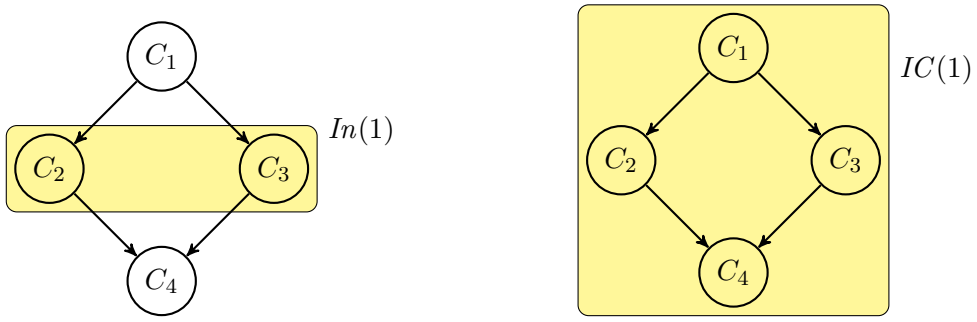
$$In(k) = \{c_i \mid (c_i : p_i) \in B(r), r \in br_k\}.$$

Furthermore, the import closure $IC(k)$ of C_k *is the smallest set S such that (i) $k \in S$ and (ii) for all $i \in S$, $In(i) \subseteq S$.*

Equivalently, we can define the import closure constructively by $IC(k) = \{k\} \cup \bigcup_{j \geq 0} IC^j(k)$, where $IC^0(k) = In(k)$, and $IC^{j+1}(k) = \bigcup_{i \in IC^j(k)} In(i)$.

Example 5 Consider M in Example 4. Then $In(1) = \{2, 3\}$, $In(2) = In(3) = \{4\}$, and $In(4) = \emptyset$; the import closure of C_1 is $IC(1) = \{1, 2, 3, 4\}$ (see Figure 2).

4. www.kr.tuwien.ac.at/research/systems/dlvhex/



(a) Import neighborhood of C_1 (b) Import closure of C_1

Figure 2: Import neighborhood and Import closure

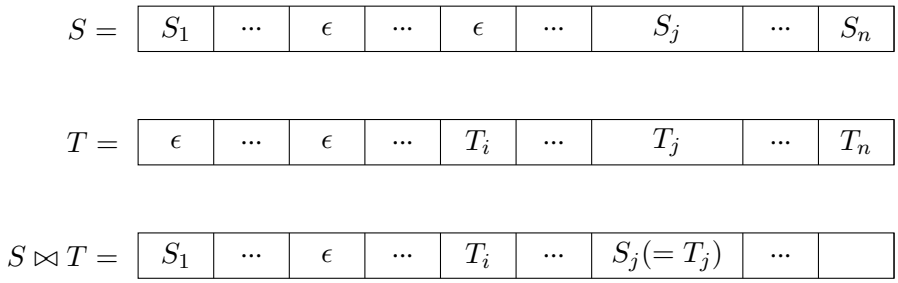


Figure 3: Joining partial belief states

Based on the import closure we define partial equilibria.

Definition 6 (Partial Belief States and Equilibria) *Let $M = (C_1, \ldots, C_n)$ be an MCS, and let $\epsilon \notin \bigcup_{i=1}^{n} \mathbf{BS}_i$. Then a sequence $S = (S_1, \ldots, S_n)$ such that $S_i \in \mathbf{BS}_i \cup \{\epsilon\}$, for all $1 \leq i \leq n$, is a* partial belief state (PBS) *of M, and S is a* partial equilibrium (PE) of M w.r.t. C_k, $k \in \{1, \ldots, n\}$, *if $i \in IC(k)$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and $i \notin IC(k)$ implies $S_i = \epsilon$, for all $1 \leq i \leq n$.*

Note that $IC(k)$ essentially defines a subsystem M' of M that is connected by bridge rules. We use PEs of M instead of equilibria of M' to keep the original MCS M intact.

For combining partial belief states $S = (S_1, \ldots, S_n)$ and $T = (T_1, \ldots, T_n)$, we define their *join* $S \bowtie T$ as the partial belief state $(U_1, \ldots, U_n)$ such that

$$U_i = \begin{cases} S_i, & \text{if } T_i = \epsilon \text{ or } S_i = T_i, \\ T_i, & \text{if } T_i \neq \epsilon \text{ and } S_i = \epsilon, \end{cases} \text{, for all } 1 \leq i \leq n$$

(see Figure 3). Note that $S \bowtie T$ is void, if some couples S_i, T_i are from $\mathbf{BS}_i$ but different. Naturally, the *join* of two sets $\mathcal{S}$ and $\mathcal{T}$ of partial belief states is then $\mathcal{S} \bowtie \mathcal{T} = \{S \bowtie T \mid S \in \mathcal{S}, T \in \mathcal{T}\}$.

Example 6 Consider two sets of partial belief states:

$$\mathcal{S} = \{\, (\epsilon, \{b\}, \epsilon, \{\neg f, g\}),\ (\epsilon, \{\neg b\}, \epsilon, \{f, \neg g\}) \,\} \text{ and}$$
$$\mathcal{T} = \{(\epsilon, \epsilon, \{\neg c, \neg d, e\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{c, d, \neg e\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{\neg c, \neg d, \neg e\}, \{f, \neg g\})\}\,.$$

Their join is given by

$$\mathcal{S} \bowtie \mathcal{T} = \left\{ \begin{array}{l} (\epsilon, \{b\}, \{\neg c, \neg d, e\}, \{\neg f, g\}), \quad (\epsilon, \{b\}, \{c, d, \neg e\}, \{\neg f, g\}), \\ (\epsilon, \{\neg b\}, \{\neg c, \neg d, \neg e\}, \{f, \neg g\}) \end{array} \right\}.$$

3.2 The Basic Algorithm

Given an MCS M and a starting context C_k, we aim at finding all PEs of M w.r.t. C_k in a distributed way. To this end, we design an algorithm DMCS whose instances run independently at a node for each context and communicate with each other for exchanging sets of partial belief states. This provides a method for distributed model building, and the DMCS algorithm can be applied to any MCS provided that appropriate solvers for the respective context logics are available. As a main feature of DMCS, it can also compute *projected partial equilibria*, i.e., PEs projected to a relevant part of the beliefs showing up in C_k's import closure. This can be exploited for specific tasks like, e.g., local query answering or consistency checking. When computing projected PEs, the information communicated between contexts is minimized, keeping communication cost low.

In the sequel, we present a basic version of the algorithm, abstracting from low-level implementation issues; the overall MCS structure is assumed to be unknown at context nodes. The idea is as follows: starting from context C_k, we visit its import closure by expanding the import neighborhood at each context C_i like in a depth-first search (DFS), until a leaf context is reached or a cycle is detected, by finding the current context in the set $hist$ of already visited contexts. A leaf context simply computes its local belief sets, transforms them into partial belief states, and returns this result to its parent (invoking context, Figure 4a). In case of a cycle (Figure 4c), the context C_i which detects the cycle must also break it, by (i) guessing belief sets for its "export" interface, (ii) transforming the guesses into partial belief states, and (iii) returning them to the invoking context.

An intermediate context C_i produces partial belief states that will be joined, i.e., consistently combined, with partial belief states of its neighbors; to enable this, C_i returns its local belief sets, joined with the results of its own neighbors (Figure 4b).

For computing projected PEs, the algorithm offers a parameter $\mathcal{V}$ called the *relevant interface* which must fulfill some conditions w.r.t. import closure that we next discuss.

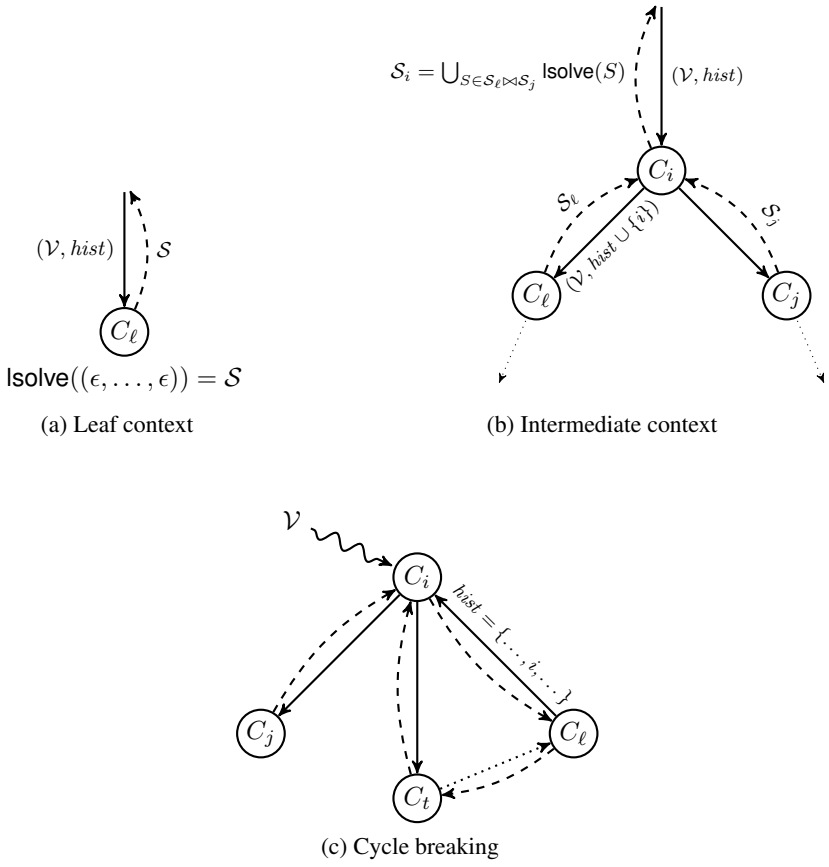
Notation. Given a (partial) belief state S and set $\mathcal{V} \subseteq \mathbf{B}$ of beliefs, we denote by $S|_\mathcal{V}$ the *restriction of S to $\mathcal{V}$*, i.e., the (partial) belief state $S' = (S_1|_\mathcal{V}, \ldots, S_n|_\mathcal{V})$, where $S_i|_\mathcal{V} = S_i \cap \mathcal{V}$ if $S_i \neq \epsilon$, and $\epsilon|_\mathcal{V} = \epsilon$; for a set $\mathcal{S}$ of (partial) belief states, we let $\mathcal{S}|_\mathcal{V} = \{S|_\mathcal{V} \mid S \in \mathcal{S}\}$. Next,

Definition 7 (Recursive Import Interface) *For an MCS $M = (C_1, \ldots, C_n)$ and $k \in \{1, \ldots, n\}$, we call $\mathcal{V}(k) = \{p_i \mid (c_i : p_i) \in B(r), r \in br_k\}$ the* import interface *of context C_k and $\mathcal{V}^*(k) = \bigcup_{i \in IC(k)} \mathcal{V}(i)$ the* recursive import interface *of context C_k.*

For a correct relevant interface $\mathcal{V}$, we have two extremal cases: (1) $\mathcal{V} = \mathcal{V}^*(k)$ and (2) $\mathcal{V} = \mathcal{V}_\mathbf{B} = \mathbf{B}$. In (1), DMCS basically checks for consistency on the import closure of C_k by computing PEs projected to interface beliefs. In (2), it computes PEs w.r.t. C_k. In between, by providing a fixed interface $\mathcal{V}$, problem-specific knowledge (such as query variables) and/or infrastructure information can be exploited to keep computations focused on relevant projections of partial belief states.

The projections of partial belief states are cached in every context such that recomputation and recombination of belief states with local belief sets are kept at a minimum.

We assume that each context C_k has a background process (or daemon in Unix terminology) that waits for incoming requests of the form $(\mathcal{V}, hist)$, upon which it starts the computation outlined



(a) Leaf context

(b) Intermediate context

(c) Cycle breaking

Figure 4: Basic distributed algorithm - casewise

in Algorithm 1. This process also serves the purpose of keeping the cache $c(k)$ persistent. We write $C_i.\mathsf{DMCS}(\mathcal{V}, hist)$ to specify that we send $(\mathcal{V}, hist)$ to the process at context C_i and wait for its return message.

Algorithm 1 uses the following primitives:

- function $\mathsf{lsolve}(S)$ (Algorithm 2): augments the knowledge base kb of the current context with the heads of bridge rules in br that are applicable w.r.t. partial belief state S, computes local belief sets using function $\mathbf{ACC}$, combines each local belief set with S, and returns the resulting set of partial belief states; and
- function $\mathsf{guess}(\mathcal{V}, C_k)$: guesses all possible truth assignments for the relevant interface w.r.t. C_k, i.e., for $\mathbf{B}_k \cap \mathcal{V}$.[5]

DMCS proceeds in the following way:

(a) check the cache for an appropriate partial belief state;

5. In order to relate beliefs in $\mathbf{B}_k$, $\mathcal{V}$ can either be a vector of sets, or variables in $\mathcal{V}$ are prefixed with context ids; for simplicity, we kept $\mathcal{V}$ as a set without further assumptions.

Algorithm 1: DMCS$(\mathcal{V}, hist)$ at $C_k = (L_k, kb_k, br_k)$

Input: $\mathcal{V}$: relevant interface, $hist$: visited contexts
Data: $c(k)$: static cache
Output: set of accumulated partial belief states

(a) **if** $c(k)$ *is not empty* **then return** $c(k)$
$\mathcal{S} := \emptyset$
(b) **if** $k \in hist$ **then** // cyclic: guess local beliefs w.r.t. $\mathcal{V}$
(c) $\quad \mathcal{S} := \mathsf{guess}(\mathcal{V}, C_k)$
else // acyclic: collect neighbor beliefs and add local ones
$\quad \mathcal{T} := \{(\epsilon, \ldots, \epsilon)\}$ and $hist := hist \cup \{k\}$
(d) $\quad$ **foreach** $i \in In(k)$ **do**
$\quad\quad$ **if** *for some* $T \in \mathcal{T}$, $T_i = \epsilon$ **then**
$\quad\quad\quad \mathcal{T} := \mathcal{T} \bowtie C_i.\mathsf{DMCS}(\mathcal{V}, hist)$
(e) $\quad$ **foreach** $T \in \mathcal{T}$ **do** $\mathcal{S} := \mathcal{S} \cup \mathsf{lsolve}(T)$
(f) $\quad c(k) := \mathcal{S}|_{\mathcal{V}}$
return $\mathcal{S}|_{\mathcal{V}}$

Algorithm 2: lsolve(S) at $C_k = (L_k, kb_k, br_k)$

Input: S: partial belief state $S = (S_1, \ldots, S_n)$
Output: set of locally acceptable partial belief states

$\mathbf{T} := \mathbf{ACC}_k(kb_k \cup \{head(r) \mid r \in app(br_k, S)\})$
return $\{(S_1, , \ldots, S_{k-1}, T_k, S_{k+1}, \ldots, S_n) \mid T_k \in \mathbf{T}\}$

(b) check for a cycle;

(c) if a cycle is detected, then guess partial belief states of the relevant interface of the context running DMCS;

(d) if no cycle is detected, but import from neighbor contexts is needed, then request partial belief states from all neighbors and join them;

(e) compute local belief states given the partial belief states collected from neighbors;

(f) cache the current (projected) partial belief state.

The next examples illustrate evaluation runs of DMCS for finding all partial equilibria with different MCS. We start with an acyclic run.

Example 7 Reconsider M from Example 4. Suppose the user invokes $C_1.\mathsf{DMCS}(\mathcal{V}, \emptyset)$, where $\mathcal{V} = \{a, b, c, f, g\}$, to trigger the evaluation process. Next, C_1 forwards in (d) requests to C_2 and C_3, which both call C_4. When called for the first time, C_4 calculates in (e) its own belief sets and assembles the set of partial belief states

$$\mathcal{S}_4 = \{(\epsilon, \epsilon, \epsilon, \{f, \neg g\}),\ (\epsilon, \epsilon, \epsilon, \{\neg f, g\})\}\,.$$

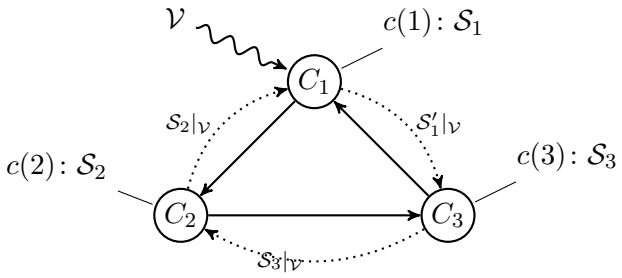


Figure 5: A cyclic topology

After caching $\mathcal{S}_4|_\mathcal{V}$ in (f), C_4 returns $\mathcal{S}_4|_\mathcal{V} = \mathcal{S}_4$ to one of the contexts C_2, C_3 whose request arrived first. On second call, C_4 simply returns $\mathcal{S}_4|_\mathcal{V}$ to the other context from the cache.

C_2 and C_3 next call lsolve (in (e)) two times each, which results in $\mathcal{S}_2 = \mathcal{S}$ resp. $\mathcal{S}_3 = \mathcal{T}$ with $\mathcal{S}, \mathcal{T}$ from Example 6.

$$\mathcal{S} = \{\, (\epsilon, \{b\}, \epsilon, \{\neg f, g\})\,,\ (\epsilon, \{\neg b\}, \epsilon, \{f, \neg g\})\,\} \text{ and}$$
$$\mathcal{T} = \{(\epsilon, \epsilon, \{\neg c, \neg d, e\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{c, d, \neg e\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{\neg c, \neg d, \neg e\}, \{f, \neg g\})\}\,.$$

Thus,

$$\begin{aligned}\mathcal{S}_2|_\mathcal{V} &= \{\, (\epsilon, \{b\}, \epsilon, \{\neg f, g\})\,,\ (\epsilon, \{\neg b\}, \epsilon, \{f, \neg g\})\,\} \text{ and}\\ \mathcal{S}_3|_\mathcal{V} &= \{(\epsilon, \epsilon, \{\neg c\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{c\}, \{\neg f, g\}),\ \ (\epsilon, \epsilon, \{\neg c\}, \{f, \neg g\})\}\,.\end{aligned}$$

C_1, after computing in (d)

$$\mathcal{S}_2|_\mathcal{V} \bowtie \mathcal{S}_3|_\mathcal{V} = \{(\epsilon, \{b\}, \{\neg c\}, \{\neg f, g\}),\ \ (\epsilon, \{b\}, \{c\}, \{\neg f, g\}),\ \ (\epsilon, \{\neg b\}, \{\neg c\}, \{f, \neg g\})\}$$

calls lsolve in (e) thrice to compute the final result:

$$\mathcal{S}_1|_\mathcal{V} = \{(\{a\}, \{b\}, \{c\}, \{\neg f, g\}),\ \ (\{\neg a\}, \{b\}, \{\neg c\}, \{\neg f, g\}),\ \ (\{\neg a\}, \{\neg b\}, \{\neg c\}, \{f, \neg g\})\}\,.$$

The next example illustrates the run of DMCS on a cyclic topology.

Example 8 Let $M = (C_1, C_2, C_3)$ be an MCS such that each L_i is an ASP logic, and

- $kb_1 = \emptyset$, $br_1 = \{a \leftarrow \text{not } (2 : b)\}$;
- $kb_2 = \emptyset$, $br_2 = \{b \leftarrow (3 : c)\}$; and
- $kb_3 = \emptyset$, $br_3 = \{c \vee d \leftarrow \text{not } (1 : a)\}$.

Figure 5 shows the cyclic topology of M. Suppose that the user sends a request to C_1 by calling C_1.DMCS$(\mathcal{V}, \emptyset)$ with $\mathcal{V} = \{a, b, c\}$. In step (d) of Algorithm 1, C_1 calls C_2.DMCS$(\mathcal{V}, \{1\})$, then context C_2 issues a call C_3.DMCS$(\mathcal{V}, \{1, 2\})$, thus C_3 invokes C_1.DMCS$(\mathcal{V}, \{1, 2, 3\})$. At this point, the instance of DMCS at C_1 detects a cycle in (b) and guesses the partial belief states

$$\mathcal{S}'_1 = \{(\{a\}, \epsilon, \epsilon), (\{\neg a\}, \epsilon, \epsilon)\}$$

for $\Sigma_1 \cap \mathcal{V}$. Then, following the dotted lines in Figure 5, the set $\mathcal{S}_1'|_{\mathcal{V}} = \mathcal{S}_1'$ is the return value for the request from C_3, which joins it with an initial empty belief state $(\epsilon, \epsilon, \epsilon)$, gives us $\mathcal{T}$ and then calls lsolve(T) for each $T \in \mathcal{T}$ in (e), resulting in

$$\mathcal{S}_3 = \{(\{\neg a\}, \epsilon, \{c, \neg d\}),\ \ (\{\neg a\}, \epsilon, \{\neg c, d\}),\ \ (\{a\}, \epsilon, \{\neg c, \neg d\})\}\,.$$

The next step of C_3 is to return $\mathcal{S}_3|_{\mathcal{V}}$ back to C_2, which will proceed as C_3 before. The result is the set of belief states

$$\mathcal{S}_2 = \{(\{\neg a\}, \{b\}, \{c\}),\ \ (\{\neg a\}, \{\neg b\}, \{c\}),\ \ (\{a\}, \{\neg b\}, \{\neg c\})\}\,,$$

which will be sent back to C_1 as $\mathcal{S}_2|_{\mathcal{V}}$. Notice that belief state $(\{\neg a\}, \{\neg b\}, \{c\})$ is inconsistent in C_1, but will be eventually eliminated once C_1 evaluates $\mathcal{S}_2|_{\mathcal{V}}$ with lsolve.

Next, C_1 will join $\mathcal{S}_2|_{\mathcal{V}}$ with $(\epsilon, \epsilon, \epsilon)$, which yields $S_2|_{\mathcal{V}}$, and then use this result to call lsolve. The union gives us

$$\mathcal{S}_1 = \{(\{\neg a\}, \{b\}, \{c\}),\ (\{a\}, \{\neg b\}, \{\neg c\})\}\quad,$$

which is also sent back to the user as final result.

Given an MCS $M = (C_1, \ldots, C_n)$ and a context C_k, using the recursive import interface of C_k, i.e., $\mathcal{V}^*(k)$, as the relevant interface is a safe (lower) bound for the correctness of Algorithm 1. In what follows, let M, C_k, and $\mathcal{V}^*(k)$ as above.

Theorem 1 (Correctness of DMCS with partial equilibrium) *For every $\mathcal{V} \supseteq \mathcal{V}^*(k)$, it holds that $S' \in C_k$.DMCS$(\mathcal{V}, \emptyset)$ iff M has some partial equilibrium S w.r.t. C_k such that $S' = S|_{\mathcal{V}}$.*

We can compute partial equilibria at C_k if we use $\mathcal{V}_{\mathbf{B}}$. This holds because using $\mathcal{V}_{\mathbf{B}}$ preserves all belief sets returned from step (e), as the projection at step (f) takes no effect.

Corollary 2 *S is a partial equilibrium of M w.r.t. C_k iff $S \in C_k$.DMCS$(V_{\mathbf{B}}, \emptyset)$.*

Under the assumption that M has a single root context C_1, i.e., such that $i \in IC(1)$ for all $2 \leq i \leq n$, DMCS computes equilibria.

Corollary 3 *If an MCS M has a single root context C_1, then S is an equilibrium of M iff $S \in C_1$.DMCS$(V_{\mathbf{B}}, \emptyset)$.*

An analysis of the algorithm yields the following upper bound on the computational complexity and communication activity.

Proposition 4 *Let $M = (C_1, \ldots, C_n)$ be an MCS. In each run of DMCS at a context C_k with an interface $\mathcal{V}$, it holds that*

(1) the total number of calls to lsolve is exponentially bound by $n \times |\mathcal{V}|$, i.e., $O(2^{n \times |\mathcal{V}|})$.

(2) the number of messages exchanged between contexts C_i, where $i \in IC(k)$, is bounded by $2 \cdot |E(k)|$, where $E(k) = \{(i, c_j) \mid i \in IC(k), r \in br_i, (c_j : p_j) \in B(r)\}$.

3.3 Discussion

Algorithm DMCS naturally proceeds "forward" in the import direction of context C_k. Thus, starting from there, it computes partial equilibria which cover C_k and contexts in its import closure. All other contexts will be ignored; in fact, they are unknown to all contexts in the closure. While partial equilibria may exist for C_k and its import closure, the whole MCS could have no equilibrium, because, e.g., (P1) contexts that access beliefs from C_k or its closure get inconsistent, or (P2) an isolated context or subsystem is inconsistent.

Enhancements of DMCS may deal with such situations: As for (P1), the context neighborhood may include both importing and supporting contexts. Intuitively, if C_i imports from C_j, then C_i must register to C_j. By carefully adapting DMCS, we can then solve (P1). However, (P2) remains; this needs knowledge about the global system topology.

A suitable assumption is that a manager $\mathcal{M}$ exists which every context C_i in the system can reach and ask whether some isolated inconsistent context or subsystem exists; if $\mathcal{M}$ confirms this, C_i's DMCS instance simply returns $\emptyset$, eliminating all partial equilibria.

To improve decentralization and information hiding, we can weaken the manager assumption by introducing *routers*. Instead of asking $\mathcal{M}$, a context C_i queries an assigned router $\mathcal{R}$, which collects topology information needed by C_i or looks up a cache. The information exchange between C_i and $\mathcal{R}$ is flexible, depending on the system setting, and could contain contexts that import information from C_i, or isolated and inconsistent contexts.

A further advantage of topological information is that C_i can recognize cyclic and acyclic branches upfront; the invocation order of the neighborhood can then be optimized, by starting with all acyclic branches before entering cyclic subsystems. The caching mechanism can be adapted for acyclic branches, as intermediate results are complete and the cache is meaningful even across different evaluation sessions.

In our setting, we are safe assuming that $\mathcal{V}^*(k) \subseteq \mathcal{V}$. But this is not needed if M resp. C_k's import closure has no *join-contexts*, i.e., contexts having at least two parents. If we have access to path information in M at each context, we could calculate $\mathcal{V}$ on the fly and adjust it during MCS traversal. In particular, for a tree- or ring-shaped M, we can restrict $\mathcal{V}$ to the *locally shared interface* between C_k and its import neighbors, i.e., restricting $\mathcal{V}$ to the bridge atoms of br_k. In presence of join-contexts, $\mathcal{V}$ must be made "big enough," e.g. using path information. Furthermore, join-contexts may be eliminated by virtually splitting them, if orthogonal parts of the contexts are accessed. This way, scalability to many contexts can be achieved.

Next, we present optimization techniques using topological information of the system.

4. Topology-Based Optimization Algorithm (DMCSOPT)

As a basic version, Algorithm DMCS uses no further metadata apart from the minimal information that each context must know: its interface with every neighboring context. There are scalability issues which can be tracked down to the following problems:

(1) contexts are unaware of context dependencies in the system beyond their neighbors, and thus treat all neighbors equally. Specifically, cyclic dependencies remain undetected until a context, seeing the invocation chain, requests models from a context in the chain. Furthermore, a context C_i does not know whether a neighbor C_j already requested models from another neighbor $C_{j'}$ which then would be passed to C_i; hence, C_i makes possibly a superfluous request to $C_{j'}$.

(2) a context C_i returns its local models combined with the results from all its neighbors. In case of multiple models, the result size can become huge as the system size and number of neighbors increases. In fact, this is one of the main performance obstacles.

In this section we address optimizations to increase the scalability of distributed MCS evaluation. Resorting to methods from graph theory, we aim at decomposing, pruning, and improved cycle breaking for dependencies in MCSs. Focusing on (1), we describe a decomposition method using biconnected components of inter-context dependencies. Based on it we can break cycles and prune acyclic parts ahead and create an acyclic query plan. To address (2), we foster a partial view of the system, which is often sufficient for a satisfactory answer, as a compromise between partial information and performance. We thus define a set of variables for each import dependency in the system to project the models in each context to a bare minimum such that they remain meaningful. In this manner, we can omit needless information and circumvent excessive model combinations.

We proceed as follows. After introducing a running example and a superficial explanation of optimization on it, we present the details of the techniques in Section 4.2. Section 4.3 introduces the notion of query plans, which is used in Section 4.4 to describe the algorithm DMCSOPT that intertwines decomposition and pruning with variable projection for performance gains in MCS evaluation.

4.1 Running Scenario

We first present a scenario in Example 9 as a running example for this section.

Example 9 (Scientists Group) A group of four scientists, Alice, Bob, Charlie, and Demi meets after a conference closing to arrange travel back home. The options are going by train or by car (which is slower); if they use the train, they should bring along some food. Alice as the group leader finally decides, based on the information she gets from Bob and Charlie.[6]

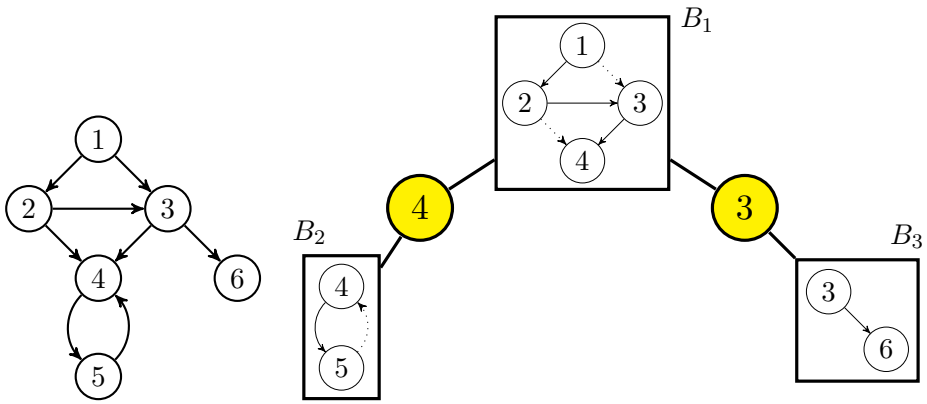
Alice prefers to go by car, but she would not object if Bob and Charlie want to go by train. Charlie has a daughter, Fiona; he does not mind either option, but if Fiona is sick he wants the fastest transport to get home. Demi just got married, and her husband, Eddie, wants her to be back soon, and even sooner if she would come soon; Demi tries to yield to her husband's plea.

Charlie is in charge of buying provisions if they go by train. He might choose either salad or peanuts; notably, Alice is allergic to nuts. The options for beverages are coke and juice. Bob is modest; he agrees to any choice of Charlie and Demi for transport but he dislikes coke. Charlie and Demi do not want to bother the others with their personal matters and just communicate their preferences, which is sufficient for reaching an agreement.

Example 10 The scenario in Example 9 can be encoded as an MCS $M = (C_1, \ldots, C_6)$, where Alice = 1, Bob = 2, etc in lexicographical order and all L_i are ASP logics. The knowledge bases kb_i and bridge rules br_i are as follows:

$$C_1\colon\ kb_1 = \left\{ \begin{array}{c} car_1 \leftarrow \text{not } train_1. \\ \bot \leftarrow nuts_1. \end{array} \right\} \text{ and } br_1 = \left\{ \begin{array}{l} train_1 \leftarrow (2 : train_2), (3 : train_3). \\ nuts_1 \leftarrow (3 : peanuts_3). \end{array} \right\};$$

6. Similar scenarios have already been investigated in the realm of multi-agent systems (on social answer set programming see, e.g., Buccafurri & Caminiti, 2008). We do not aim at introducing a new semantics for such scenarios; our example serves as a plain MCS showcase for the algorithms.



(a) Diamond-ring (b) Diamond-ring block tree

Figure 6: Topologies and decomposition of the scientist group example

$$C_2\colon\ kb_2 = \{\bot \leftarrow \text{not } car_2, \text{not } train_2.\} \text{ and } br_2 = \left\{ \begin{array}{c} car_2 \leftarrow (3:car_3), (4:car_4). \\ train_2 \leftarrow (3:train_3), (4:train_4), \\ \text{not } (3:coke_3). \end{array} \right\};$$

$$C_3\colon\ kb_3 = \left\{ \begin{array}{c} car_3 \vee train_3. \leftarrow \\ train_3 \leftarrow urgent_3. \\ salad_3 \vee peanuts_3 \leftarrow train_3. \\ coke_3 \vee juice_3 \leftarrow train_3 \end{array} \right\} \text{ and } br_3 = \left\{ \begin{array}{c} urgent_3 \leftarrow (6:sick_6). \\ train_3 \leftarrow (4:train_4) \end{array} \right\};$$

$$C_4\colon\ kb_4 = \{car_4 \vee train_4 \leftarrow\} \text{ and } br_4 = \{train_4 \leftarrow (5:sooner_5)\};$$

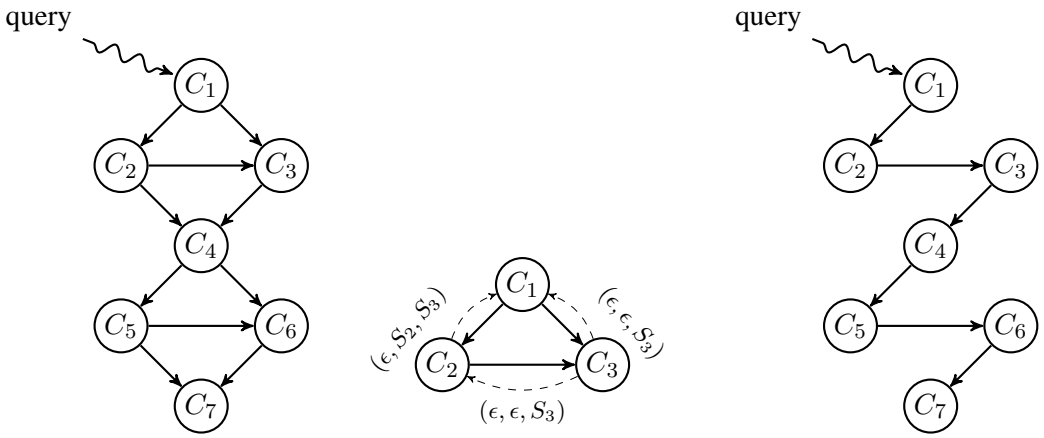
$$C_5\colon\ kb_5 = \{sooner_5 \leftarrow soon_5\} \text{ and } br_5 = \{soon_5 \leftarrow (4:train_4)\};$$

$$C_6\colon\ kb_6 = \{sick_6 \vee fit_6 \leftarrow\} \text{ and } br_6 = \emptyset.$$

The context dependencies of M are shown in Fig. 6a. M has three equilibria, namely:

- $(\{train_1\}, \{train_2\}, \{train_3, urgent_3, juice_3, salad_3\}, \{train_4\}, \{soon_5, sooner_5\}, \{sick_6\})$;
- $(\{train_1\}, \{train_2\}, \{train_3, juice_3, salad_3\}, \{train_4\}, \{soon_5, sooner_5\}, \{fit_6\})$; and
- $(\{car_1\}, \{car_2\}, \{car_3\}, \{car_4\}, \emptyset, \{fit_6\})$.

Example 11 Consider an MCS $M = (C_1, \ldots, C_7)$ with context dependencies as drawn in Figure 7a. When the user queries C_1 and just cares about the local belief sets in C_1, then in the evaluation process, C_4 can discard all local belief sets of C_5 and C_6 when answering to a call from C_2 or C_3. However, when C_1 calls C_2 (or C_3), the invoked context must carry local belief sets of C_4 in its answers to C_1. The reason is that belief sets of C_4 can cause *inconsistent joins* at C_1 for partial belief states returned from C_2 and C_3, while those of C_5 to C_7 contribute only directly to computing local belief sets at C_4. Note that belief sets of C_4 to C_7 play no role in determining the applicability of bridge rules in C_1.



(a) Original Topology (b) Triangle (c) Transitive Reduction

Figure 7: Topology of Example 11 (two stacked zig-zag diamonds)

Now, take a sub-system including C_1, C_2, and C_3, assuming that C_1 has bridge rules with atoms $(2 : p_2)$ and $(3 : p_3)$ in the body, and C_2 with atoms $(3 : p_3)$. That is, C_1 depends on both C_2 and C_3, while C_2 depends on C_3 (see Fig. 7b). A straightforward approach to evaluate this MCS asks at C_1 for the belief sets of C_2 and C_3. But as C_2 also depends on C_3, we would need another query from C_2 to C_3 to evaluate C_2 w.r.t. the belief sets of C_3. This shows evident redundancy, as C_3 will need to compute its belief sets twice. Simple caching strategies could mellow out the second belief state building at C_3; nonetheless, when C_1 asks C_3, the context will transmit its belief states back, thus consuming network resources.

Moreover, when C_2 asks for the PEs of C_3, it will receive a set of PEs that covers the belief sets of C_3 and in addition of all contexts in C_3's import closure. This is excessive from C_1's view, as it only needs to know about $(2 : p_2)$ and $(3 : p_3)$. However, C_1 needs the belief states of both C_2 and C_3 in reply of C_2: if C_2 only reports its own belief sets (which are consistent w.r.t. C_3), C_1 can't align the belief sets received from C_2 with those received from C_3. Realizing that C_2 also reports the belief sets of C_3, no call to C_3 must be made.

4.2 Decomposition of Nonmonotonic MCS

Based on the observations above, we present an optimization strategy that pursues two orthogonal goals: (i) to prune dependencies in an MCS and cut superfluous transmissions, belief state building, and joining of belief states; and (ii) to minimize content in transmissions. We start with defining the topology of an MCS.

Definition 8 (Topology) *The* topology *of an MCS $M = (C_1, \ldots, C_n)$ is the directed graph $G_M = (V, E)$, where $V = \{C_1, \ldots, C_n\}$ resp. $V = \{1, \ldots, n\}$ and $(i, j) \in E$ iff some rule in br_i has an atom $(j{:}p)$ in the body.*

The first optimization technique is made up of three graph operations. We get a coarse view of the topology by splitting it into *biconnected components*, which form a *tree representation* of the MCS. Then, edge removal techniques yield acyclic structures.

In the sequel, we will use standard terminology from graph theory (see Bondy & Murty, 2008); graphs are directed by default. We may view undirected graphs as directed graphs that have both edges (u, v), (v, u) for an undirected edge $\{u, v\}$.

For any graph G and edges $S \subseteq E(G)$, we denote by $G \backslash S$ the maximal subgraph of G having no edges from S. Suppose that $V' \subseteq V(G)$ is nonempty. Then the subgraph $G' = (V', E')$ of G with vertex set V' and edge set $E = \{(u, v) \in E(G) \mid u, v \in V(G)\}$ is the *subgraph induced* by V', denoted by $G[V']$. The induced subgraph $G[V \setminus V']$ is denoted by $G \backslash V'$; it results from G by deleting the vertices in V' together with their incident edges. If $V' = \{v\}$, we write $G \backslash v$ for $G \backslash \{v\}$.

Two vertices u and v of G are said to be *connected*, if there is a (directed) path from u to v in G, i.e., a sequence of vertices $u = v_1, v_2, \ldots, v_n = v$, such that $(v_i, v_{i+1}) \in E(G)$, for each $i = 1, \ldots, n-1$; the path is *trivial* if $n = 1$. For an undirected graph G, connectedness is an equivalence relation on $V(G)$. Thus there is a partition of $V(G)$ into nonempty subsets $V_1, V_2, \ldots, V_w$ such that two vertices u and v of G are connected iff both of them belong to the same set V_i. The subgraphs $G[V_1], G[V_2], \ldots, G[V_w]$ are called the *components* of G. If $w = 1$ (i.e., G has exactly one component), then G is *connected*; otherwise, G is *disconnected*.

A directed graph G is *strongly connected*, if for each vertices $u, v \in V(G)$ a path from u to v and vice versa exists. The *strongly connected components* of G are the subgraphs $G[V_1], \ldots, G[V_m]$ in the unique partition of the graph G into pairwise disjoint induced subgraphs (i.e., $V_i \cap V_j = \emptyset$) that are strongly connected.

Furthermore, a directed graph G is *weakly connected*, if turning all edges into undirected edges yields a connected graph. A vertex c of a weakly connected graph G is a *cut vertex*, if $G \backslash c$ is disconnected. A *biconnected graph* is a weakly connected graph without cut vertices.

A *block* in a graph G is a maximal biconnected subgraph of G. Given a set of blocks $\mathcal{B}$, the *union* of blocks in $\mathcal{B}$ is defined as $\bigcup \mathcal{B} = \bigcup_{B \in \mathcal{B}} B$, where the union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$.

Let $T(G) = (\mathcal{B} \cup \mathcal{C}, \mathcal{E})$ denote the undirected bipartite graph, called *block tree of graph* G, where

(i) $\mathcal{B}$ is the set of blocks of G,

(ii) $\mathcal{C}$ is the set of cut vertices of G,

(iii) and $(B, c) \in \mathcal{E}$ with $B \in \mathcal{B}$ and $c \in \mathcal{C}$ iff $c \in V(B)$.

Note that $T(G)$ is a forest for any graph G and a rooted tree if G is weakly connected.

Example 12 Consider the graph in Figure 7a. One can check that 4 is the only cut vertex and there are two blocks, viz. the subgraphs induced by $\{1, 2, 3, 4\}$ and $\{4, 5, 6, 7\}$.

The next example shows the block tree of our scenario in Example 9.

Example 13 The topology G_M of M in Example 10 is shown in Figure 6a. It has two cut vertices, namely 3 and 4; thus the block tree $T(G_M)$ (Figure 6b) contains the blocks B_1, B_2, and B_3, which are subgraphs of G_M induced by $\{1, 2, 3, 4\}$, $\{4, 5\}$, and $\{3, 6\}$, respectively.

A *topological sort* of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering.

Pruning. In acyclic topologies, like the triangle presented in Figure 7b, we can exploit a minimal graph representation to avoid unnecessary calls between contexts, namely, the transitive reduction of the graph G_M. Recall from Aho, Garey, and Ullman (1972) that a graph G^- is a *transitive reduction* of the directed graph G whenever the following two conditions are satisfied:

(i) there is a directed path from vertex u to vertex v in G^- iff there is a directed path from u to v in G, and

(ii) there is no graph with fewer edges than G^- satisfying condition (i).

Note that G^- is unique if G is acyclic. For instance, the graph in Figure 7c is the unique transitive reduction of the one in Figure 7a.

Ear decomposition Another essential part of our optimization strategy is to break cycles by removing edges. To this end, we use ear decompositions of cyclic graphs. A block may have multiple cycles that are not necessarily strongly connected; thus we first decompose blocks into their strongly connected components. Using Tarjan's algorithm (Tarjan, 1972) for this task, one gets as a byproduct a topological sort on the directed acyclic graph formed by the strongly connected components. This yield a sequence of nodes $r_1, \ldots, r_s$ that are used as entry points to each component. The next step is to break cycles.

An *ear decomposition* of a strongly connected graph G *rooted at a node* r is a sequence $P = \langle P_0, \ldots, P_m \rangle$ of subgraphs of G such that

(i) $G = P_0 \cup \cdots \cup P_m$,

(ii) P_0 is a simple cycle (i.e., has no repeated edges or vertices) with $r \in V(P_0)$, and

(iii) each P_i ($i > 0$) is a non-trivial path (without cycles) whose endpoint t_i is in $P_0 \cup \cdots \cup P_{i-1}$, but the other nodes are not.

Let $cb(G, P)$ be the set of edges containing (ℓ_0, r) of P_0 and the last edge (ℓ_i, t_i) of each P_i, $i > 0$. Here, ℓ_0 is the vertex belonging to the edge to the root node r in the simple cycle P_0.

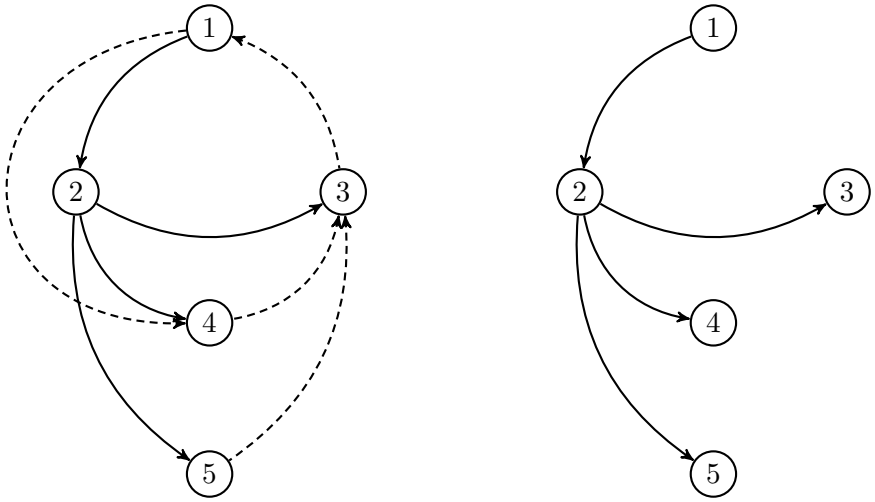
Example 14 Take, as an example, a strongly connected graph G in Figure 8a. An ear decomposition of G rooted at node 1 is $P = \langle P_0, P_1, P_2, P_3 \rangle$ where

$$V_{P_0} = \{1,2,3\}, E_{P_0} = \{(1,2),(2,3),(3,1)\}, \quad V_{P_1} = \{2,4,3\}, E_{P_1} = \{(2,4),(4,3)\},$$
$$V_{P_2} = \{2,5,3\}, E_{P_2} = \{(2,5),(5,3)\}, \quad V_{P_3} = \{1,4\}, E_{P_4} = \{(1,4)\}.$$

The last edges of P_i are dashed. They form the set $cb(G, P) = \{(3,1),(4,3),(5,3),(1,4)\}$. Removing these edges results in an acyclic topology as in Figure 8b.

Intuitively, ear decomposition is used to remove cycles from the original system M. On the resulting acyclic topology, algorithms for evaluating MCSs can be designed more conveniently. The trade off is that for any edge (ℓ, t) removed from M, context C_ℓ, despite being now a leaf context, has to guess values for variables from C_t. The following example shows the application of the optimization techniques above to our running scenario.

Example 15 (cont'd) Block B_1 of $T(G_M)$ is acyclic, and the transitive reduction gives B_1^- with edges $\{(1,2),(2,3),(3,4)\}$. B_2 is cyclic, and $\langle B_2 \rangle$ is the only ear decomposition rooted at 4; removing $cb(B_2, \langle B_2 \rangle) = \{(5,4)\}$, we obtain B_2' with edges $\{(4,5)\}$. B_3 is acyclic and already reduced. Fig. 6b shows the final result (dotted edges are removed).



(a) A strongly connected component (b) Acyclic topology

Figure 8: Ear decomposition example

The graph-theoretic concepts introduced here, in particular the transitive reduction of acyclic blocks and the ear decomposition of cyclic blocks, are used to implement the first optimization of MCS evaluation outlined above. Intuitively, in each block, we apply ear decomposition to get rid of cycles (with the trade-off of guessing), and then use transitive reduction to minimize communication. Given the transitive reduction B^- of an acyclic block $B \in \mathcal{B}$, and a total order on $V(B^-)$, one can evaluate the respective contexts in the reverse order of this total order for computing PEs at some context C_k: the first context simply computes its local belief sets which—represented as a set of partial belief states $\mathcal{S}_0$—constitutes an initial set of partial belief states $\mathcal{T}_0$. In each iteration step $i \geq 1$, $\mathcal{T}_i$ is computed by joining $\mathcal{T}_{i-1}$ with the local belief sets $\mathcal{S}_i$ of the considered context C_i. Given the final $\mathcal{T}_k$, we have that $\mathcal{T}_k|_{\mathcal{V}^*(k)}$ is the set of PEs at C_k (restricted to contexts in $V(B^-)$).

Refined recursive import. Next, we define the second part of our optimization strategy which handles minimization of information needed for transmission between two neighboring contexts C_i and C_j. For this purpose, we refine the notion of recursive import interface (Definition 7) in a context w.r.t. a particular neighbor and a given (sub-)graph.

Definition 9 *Given an MCS $M = (C_1, \ldots, C_n)$ and a subgraph G of G_M, for an edge $(i, j) \in E(G)$, the* recursive import interface of C_i to C_j w.r.t. G *is* $\mathcal{V}^*(i,j)_G = \mathcal{V}^*(i) \cap \bigcup_{\ell \in G|_j} \mathbf{B}_\ell$ *where $G|_j$ contains all nodes in G reachable from j.*[7]

Example 16 (cont'd) For the MCS in Example 10, we have $\mathcal{V}^*(1) = \{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4, sooner_5, sick_6\}$. When we focus on block B_1, the refined recursive import interface $\mathcal{V}^*(1,2)_{B_1^-}$ can be obtained by removing bridge atoms from contexts in the other blocks B_2 and B_3, yielding $\{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4\}$.

Algorithms. Algorithms 3 and 4 combine the optimization techniques outlined above. Intuitively, OptimizeTree takes as input a block tree T with 'parent cut vertex' c_p and 'root cut vertex' c_r. It traverses T in DFS manner and calls OptimizeBlock on every block. The call results are collected in a

7. Note that $\mathcal{V}^*(k)$ is defined in Definition 7.

Algorithm 3: OptimizeTree$(T = (\mathcal{B} \cup \mathcal{C}, \mathcal{E}), c_p, c_r)$

Input: T: block tree, c_p: identifies level in T, c_r: identifies level above c_p
Output: F: removed edges from $\bigcup \mathcal{B}$, v: labels for $(\bigcup \mathcal{B}) \backslash F$

$\mathcal{B}' := \emptyset, \quad F := \emptyset, \quad v := \emptyset$ // initialize siblings $\mathcal{B}'$ and return values
if $c_p = c_r$ **then**
  $\mathcal{B}' := \{B \in \mathcal{B} \mid c_r \in V(B)\}$
else
  $\mathcal{B}' := \{B \in \mathcal{B} \mid (B, c_p) \in \mathcal{E}\}$
foreach *sibling block* $B \in \mathcal{B}'$ **do** // sibling blocks B of parent c_p
  $E :=$ OptimizeBlock(B, c_p) // prune block
  $\mathcal{C}' := \{c \in \mathcal{C} \mid (B, c) \in \mathcal{E} \wedge c \neq c_p\}$ // children cut vertices of B
  $B' := B \backslash E$, $F := F \cup E$
(a) **foreach** *edge* (i, j) *of* B' **do** // setup interface of pruned B
    $v(i,j) := \mathcal{V}^*(i,j)_{B'} \cup \bigcup_{c \in \mathcal{C}'} \mathcal{V}^*(c_p)|_{\mathbf{B}_c} \cup \bigcup_{(\ell,t) \in E} \mathcal{V}^*(c_p)|_{\mathbf{B}_t}$
  foreach *child cut vertex* $c \in \mathcal{C}'$ **do** // accumulate children
(b) $(F', v') :=$ OptimizeTree$(T \backslash B, c, c_p)$
    $F := F \cup F'$, $v := v \cup v'$
return (F, v)

set F of removed edges; after all blocks have been processed, the final result of OptimizeTree is the pair (F, v) where v is a labeling for the remaining edges. OptimizeBlock takes a graph G and calls CycleBreaker for cyclic G, which decomposes G into its strongly connected components, creates an ear decomposition P for each component G_c, and breaks cycles by removing edges $cb(G_c, P)$. For the resulting acyclic subgraph of G, OptimizeBlock computes the transitive reduction G^- and returns all edges that have been removed from G. OptimizeTree continues by computing the labeling v for the remaining edges, building on the recursive import interface, but keeping relevant interface beliefs of child cut vertices and removed edges. Example 20 (Appendix B) illustrates Algorithms 3 and 4 with a detailed run on the MCS in Example 10.

Formally, the following property holds.

Proposition 5 *Given an MCS M and a context C_k such that k is a cut vertex in the topology G_M, OptimizeTree$(T(G_M), k, k)$ returns a pair (F, v) such that*

(i) the subgraph G of $G_M \backslash F$ induced by $IC(k)$ is acyclic, and

(ii) in any block B of G and for all $(i, j) \in E(B)$, it holds that $v(i, j) \supseteq \mathcal{V}^*(i, j)_B$.

Regarding the computational cost of computation, we obtain:

Proposition 6 *Given an MCS M and a context C_k such that k is a cut vertex in the topology G_M, OptimizeTree$(T(G_M), k, k)$ runs in polynomial (quadratic) time in the size of $T(G_M)$ resp. G_M.*

Algorithm 4: OptimizeBlock(G: graph, r: context id)

$F := \emptyset$
if G *is cyclic* **then**
(c) $\quad F :=$ CycleBreaker(G, r) // ear decomposition of strongly connected components
(d) Let G^- be the transitive reduction of $G \backslash F$
return $E(G) \setminus E(G^-)$ // removed edges from G

4.3 Query Plan

Given the topology of an MCS, we need to represent a stripped version of it that contains both the minimal dependencies between contexts and interface beliefs that need to be transferred between contexts. This representation will be a *query plan* that can be used for execution processing. Syntactically, query plans have the following form.

Definition 10 (Query Plan) *A* query plan of an MCS M w.r.t. context C_k *is any labeled subgraph* Π *of* G_M *induced by* $IC(k)$ *with* $E(\Pi) \subseteq E(G_M)$*, and edge labels* $v\colon E(G) \to 2^{\Sigma}$.

For any MCS M and context C_k of M, not every query plan is suitable for evaluating M; however, the following query plan is in fact effective.

Definition 11 (Effective Query Plan) *Given an MCS* M *and a context* C_k*, the* effective query plan *of* M *with respect to* C_k *is* $\Pi_k = (V(G), E(G)\backslash F, v)$ *where* G *is the subgraph of* G_M *induced by* $IC(k)$ *and* $(F, v) =$ OptimizeTree$(T(G_M), k, k)$.

We next use Π_k for MCS evaluation, and tacitly assume that query plans are effective.

4.4 Evaluation with Query Plans

We now present the algorithm DMCSOPT, which is based on DMCS but exploits the optimization techniques from above. The idea of DMCSOPT is as follows: we start with context C_k and traverse a given query plan Π_k by expanding the outgoing edges of Π_k at each context, like in a DFS, until a leaf context C_i is reached. The context simply computes its local belief sets, transforms all belief sets into partial belief states, and returns the result to its parent. If C_i has $(j : p)$ in bridge rules bodies but context C_j is not in the query plan (this means we broke a cycle by removing the last edge to C_j), all possible truth assignments to the import interface to C_j are considered.

The result of any context C_i is a set of partial belief states, which amounts to the join, i.e., the consistent combination, of its local belief sets with the results of its neighbors; the final result is obtained from C_k. To keep recomputation and recombination of belief states with local belief sets at a minimum, partial belief states are cached in every context.

Algorithm 5 shows our distributed algorithm, DMCSOPT, with its instance at context C_k. On input of the id c of a predecessor context (which the process awaits), it proceeds based on an (acyclic) query plan Π_r w.r.t. context C_r, i.e., the starting context of the system. The algorithm maintains in $cache(k)$ a cache at C_k (which is kept persistent).

- C_i.DMCSOPT(c): send id c to DMCSOPT at context C_i and wait for its result.
- guess$(\mathcal{V})$: guess all possible truth assignments for the interface beliefs $\mathcal{V}$.

Algorithm 5: DMCSOPT(c : context id of predecessor) at $C_k = (L_k, kb_k, br_k)$

Data: Π_r: query plan w.r.t. starting context C_r and label v, $cache(k)$: cache
Output: set of accumulated partial belief states

(a) **if** $cache(k)$ *is not empty* **then**
  $\mathcal{S} := cache(k)$
else
  $\mathcal{T} := \{(\epsilon, \ldots, \epsilon)\}$
(b) **foreach** $(k, i) \in E(\Pi_r)$ **do** $\mathcal{T} := \mathcal{T} \bowtie C_i$.DMCSOPT$(k)$ // neighbor belief sets
(c) **if** *there is* $i \in In(k)$ *s.t.* $(k, i) \notin E(\Pi_r)$ *and* $T_i = \epsilon$ *for* $T \in \mathcal{T}$ **then**
    $\mathcal{T} :=$ guess$(v(c, k)) \bowtie \mathcal{T}$ // guess for removed dependencies in Π_r
  $\mathcal{S} := \emptyset$
(d) **foreach** $T \in \mathcal{T}$ **do** $\mathcal{S} := \mathcal{S} \cup$ lsolve(T) // get local beliefs w.r.t. T
  $cache(k) := \mathcal{S}$
(e) **if** $(c, k) \in E(\Pi_r)$ *(i.e., C_k is non-root)* **then**
  return $\mathcal{S}|_{v(c,k)}$
else
  return $\mathcal{S}$

- lsolve(S) (Algorithm 2): given a partial belief state S, augment kb_k with all heads from bridge rules br_k applicable w.r.t. S (=: kb'_k), compute local belief sets by $\mathbf{ACC}(kb'_k)$, and merge them with S; return the resulting set of partial belief states.

The steps of Algorithm 5 are explained as follows:

(a)+(b) check the cache, and if it is empty get neighbor contexts from the query plan, request partial belief states from all neighbors and join them;

(c) if there are $(i : p)$ in the bridge rules br_k such that $(k, i) \notin E(\Pi_r)$, and no neighbor delivered the belief sets for C_i in step (b) (i.e., $T_i = \epsilon$), we have to call guess on the interface $v(c, k)$ and join the result with $\mathcal{T}$ (intuitively, this happens when edges have been removed from cycles);

(d) compute local belief states given the partial belief states collected from neighbors; and

(e) return the locally computed belief states and project them to the variables in $v(c, k)$ for non-root contexts; this is the point where we mask out parts of the belief states that are not needed in contexts lying in a different block of $T(G_M)$.

Theorem 7 shows that DMCSOPT is sound and complete.

Theorem 7 *Let C_k be a context of an MCS M, let Π_k be the query plan as in Definition 11 and let $\widehat{\mathcal{V}} = \{p \in v(k, j) \mid (k, j) \in E(\Pi_k)\}$. Then,*

(i) for each $S' \in C_k$.DMCSOPT(k), there exists a partial equilibrium S of M w.r.t. C_k such that $S' = S|_{\widehat{\mathcal{V}}}$; and

(ii) for each partial equilibrium S of M w.r.t. C_k, there exists an $S' \in C_k$.DMCSOPT(k) such that $S' = S|_{\widehat{\mathcal{V}}}$.

5. Streaming Equilibria (DMCS-STREAMING)

Algorithm DMCSOPT shows substantial improvements over DMCS; however, when the sizes of the local knowledge bases and the context interfaces increase, it also suffers from bottlenecks.

This stems from the way in which models are exchanged between contexts. Suppose context C_1 accesses several neighbors $C_2, \ldots, C_m$ under an acyclic information flow, and that each C_i, $i \geq n$, has n_i PEs. Before C_i computes in DMCS resp. DMCSOPT any local models, it must join all PEs from its neighbors; this may lead to $n_2 \times n_3 \times \cdots \times n_m$ many PEs, and each of them is an input to the local solver. This may not only take considerable time but also exhaust memory, even before local model computation starts.

Note however that if instead each neighbor would transfer just a portion of its PEs, then the computation at C_1 can avoid such a memory blowup. Moreover, this strategy also helps to reduce inactive running time at C_1 while waiting for all neighbors to return all PEs, as C_1 can already start local computing while the neighbors are producing more models.

In general, it is indispensable to trade more computation time, due to recomputations, for less memory if eventually *all* partial equilibria at C_1 shall be computed. This is the idea underlying a *streaming* evaluation method for distributed MCS. It is particularly useful when a user is interested in obtaining just *some* instead of all answers from the system, but also for other realistic scenarios where the current evaluation algorithm does not manage to output under resource constraints in practice any equilibrium at all.

In this section, we turn this idea into a concrete streaming algorithm DMCS-STREAMING for computing partial equilibria. Its main features are briefly summarized as follows:

- the algorithm is *fully distributed*, i.e., instances of its components run at every context and communicate, thus cooperating at the level of peers;
- when invoked at a context C_i, the algorithm streams (i.e. computes) $k \geq 1$ partial equilibria at C_i at a time; in particular, setting $k = 1$ allows for consistency checking of the MCS (sub-)system.
- issuing follow-up invocations one may compute the next k partial equilibria at context C_1 until no further equilibria exist; i.e., this evaluation scheme is complete.
- local buffers can be used for storing and exchanging local models (partial belief states) at contexts, avoiding the space explosion problem.

As this section mainly studies the streaming aspect of the algorithm, we simplify the presentation and omit the interface between contexts. The principles presented here can be applied for both DMCS and DMCSOPT by adapting the interface and pruning the topology at preprocessing time. Furthermore, we assume to work with acyclic MCSs. Treatment of cyclic cases can be easily achieved by adding guessing code to the solving component as in DMCS and DMCSOPT.

To the best of our knowledge, a similar streaming algorithm has neither been developed for the particular case of computing equilibria of a MCS, nor more generally for computing models of distributed knowledge bases. Thus, the results obtained here are not only of interest in the setting of heterogeneous MCS, but they are also relevant in general for model computation and reasoning over distributed (potentially homogeneous) knowledge bases like e.g. distributed SAT instances.

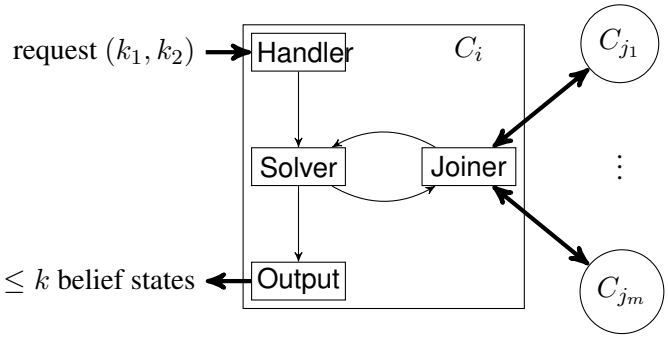


Figure 9: DMCS-STREAMING architecture

Algorithm 6: Handler(k_1, k_2: package range) at C_i

Output.$k_1 := k_1$, Output.$k_2 := k_2$,
Solver.$k_2 := k_2$, Joiner.$k := k_2 - k1 + 1$
call Solver

5.1 Basic Streaming Procedure

The basic idea is as follows: each pair of neighboring contexts can communicate in multiple rounds, and each request has the effect to receive at most k PEs. Each communication window of k PEs ranges from the k_1-th PE to the k_2-th ($= k_1 + k - 1$) PE. A parent context C_i requests from a child context C_j a pair (k_1, k_2) and will receive some time later a package of at most k PEs; receiving ϵ indicates that C_j has fewer than k_1 models. A parallelized version is discussed in Section 5.2.

Important subroutines of the new algorithm DMCS-STREAMING take care of receiving the requests from parents, receiving and joining answers from neighbors, local solving and returning results to parents. They are reflected in four components: Handler, Solver, Output, and Joiner (only active in non-leaf contexts); see Figure 9 for an architectural overview.

All components except Handler (shown in Algorithm 6) communicate using message queues: Joiner has j queues to store partial equilibria from j neighbors, Solver has one queue to hold joined PEs from Joiner, and Output has a queue to carry results from Solver. To bound space usage, each queue has a limit on the number of entries. When a queue is full (resp., empty), the enqueuing writer (resp., dequeuing reader) is automatically blocked. Furthermore, getting an element also removes it from the queue, which makes room for other PEs in the queue later. This property frees us from synchronization technicalities.

Algorithms 7 and 8 show how Solver and Joiner work. They use the following primitives:

- lsolve(S): works as lsolve in DMCS and DMCSOPT, but in addition may return only one answer at a time and may be able to tell whether there are models left. Moreover, we require that the results from lsolve are returned in a fixed order, regardless of when it is called. This property is the key to guarantee the correctness of our algorithm.

- *get_first*(ℓ_1, ℓ_2, k): send to each neighbor from c_{ℓ_1} to c_{ℓ_2} a request for the first k partial equilibria, i.e., $k_1 = 1$ and $k_2 = k$; if they all return some models, store them in the respective queues and return $true$; otherwise, return $false$ (some neighbor is inconsistent).

Algorithm 7: Solver() at C_i

```
Data: Input queue: q, maximal number of models: k_2
count := 0
while count < k_2 do
(a)   if C_i is a leaf then S := ∅
(b)   else call Joiner and pop S from q
      if S = ε then count := k_2
(c)   while count < k_2 do
          pick the next model S* from Isolve(S)
          if S* ≠ ε then
              push S* to Output.q
              count := count + 1
          else break
refresh() and push ε to Output.q
```

• *get_next*(ℓ, k): request the next k equilibria from neighbor C_{c_ℓ}; if C_{c_ℓ} sends back some models, store them in the queue q_ℓ and return $true$; otherwise, return $false$ as the neighbor already exhaustively returned its PEs from the previous request. Note that this subroutine needs to keep track of which range has been already asked for to what neighbor, by maintaining a set of counters. A counter w.r.t. a neighbor C_{c_ℓ} is initialized to 0 and increased each time *get_next*(ℓ, k) is called. When its value is t, the request to C_{c_ℓ} asks for the t'th package of k models, i.e., models in the range given by $k_1 = (t-1) \times k + 1$ and $k_2 = t \times k$. When *get_first*(ℓ_1, ℓ_2, k) is called, all counters in range $[\ell_1, \ell_2]$ are reset to 0.

• *refresh*(): reset all counters and flags of Joiner to their starting states, e.g., $first\_join$ to $true$, all counters to 0.

The process at each context C_i is triggered when a message from a parent, which contains the range (k_1, k_2) arrives at Handler. The latter notifies Solver to compute up to k_2 models and Output to collect those in the range (k_1, k_2) and return them to the parent. Furthermore, it sets the package size at Joiner to $k = k_2 - k_1 + 1$ in case C_i needs to query further neighbors (cf. Algorithm 6).

When Solver receives a notification from Handler, it first prepares the input for the local solver. If C_i is a leaf context, the input S gets the empty set assigned in Step (a); otherwise, Solver triggers Joiner (Step (b)) for input from neighbors. Fed with input from them, Isolve is used in Step (c) to compute at most k_2 results and send them to the output queue.

The Joiner, which is only activated for intermediate contexts as discussed, gathers partial equilibria from the neighbors in a fixed ordering and stores the joined, consistent input to a local buffer. It communicates just one input at a time to Solver upon request. The fixed joining order is guaranteed by always asking the first package of k models from all neighbors at the beginning in Step (d). In subsequent rounds, we begin with finding the first neighbor C_{c_ℓ} that can return further models (Step (e)), and reset the query to ask for first packs of k models from neighbors from C_{c_1} to $C_{c_{\ell-1}}$. When all neighbors run out of models in Step (f), the joining process ends and sends ϵ to Solver.

Note that while the above procedure guarantees that no models are missed, it can lead to consider the same combinations (inputs to Solver) multiple times. Using a cache helps to mitigate

Algorithm 8: Joiner() at C_i

Data: Queue $q_1, \ldots,$ queue q_j for $In(i) = \{c_1, \ldots, c_j\}$, buffer for partial equilibria: buf, flag $first\_join$

```
while true do
    if buf is not empty then
        pop S from buf, push S to Solver.q
        return
    if first_join then
(d)     if get_first(1, j, k) = false then
            push ε to Solver.q
            return
        else first_join := false
    else
        ℓ := 1
(e)     while get_next(ℓ, k) = false and ℓ ≤ j do ℓ := ℓ + 1
        if 1 < ℓ ≤ j then
            get_first(1, ℓ − 1, k)
(f)     else if ℓ > j then
            push ε to Solver.q
            return
    for S_1 ∈ q_1, ..., S_j ∈ q_j do add S_1 ⋈ ··· ⋈ S_j to buf
```

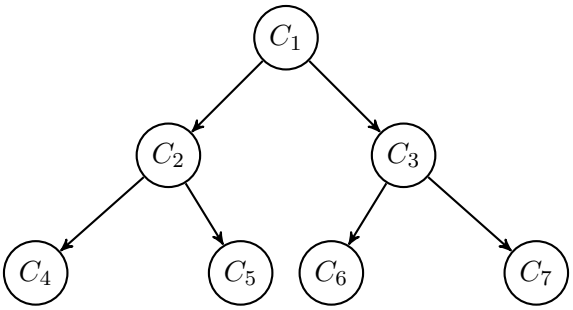


Figure 10: Binary tree MCS

recomputation, but as unlimited buffering again quickly exceeds memory limits, recomputation is an inevitable part of trading computation time for less memory.

The Output component simply reads from its queue until it receives ϵ or reaches k_2 models (cf. Algorithm 9). Upon reading, it throws away the first $k_1 - 1$ models and only keeps the ones from k_1 onwards. Eventually, if fewer than k_1 models have been returned by Solver, then Output will return ϵ to the parent.

Example 17 Let $M = (C_1, \ldots, C_n)$ be an MCS such that for a given integer $m > 0$, we have $n = 2^{m+1} - 1$ contexts, and let $\ell > 0$ be an integer. Let all contexts in M have ASP logics. For $i < 2^m$,

Algorithm 9: Output() at C_i

```
Data: Input queue: q, starting model: k_1, end model: k_2
buf := ∅ and count := 0
while count < k_1 do
    pick an S from Output.q
    if S = ε then count := k_2 + 1
    else count := count + 1
while count < k_2 + 1 do
    wait for an S from Output.q
    if S = ε then count := k_2 + 1
    else
        count := count + 1
        add S to buf
if buf is empty then
    send ε to parent
else
    send content of buf to parent
```

the context $C_i = (L_i, kb_i, br_i)$ has

$$kb_i = \{a_i^j \vee \neg a_i^j \leftarrow t_i \mid 1 \leq j \leq \ell\} \text{ and } br_i = \left\{ \begin{array}{l} t_i \leftarrow (2i : a_{2i}^j), \\ t_i \leftarrow (2i+1 : a_{2i+1}^j) \end{array} \;\middle|\; 1 \leq j \leq \ell \right\} \quad , \qquad (2)$$

and for $i \geq 2^m$, we let C_i have

$$kb_i = \{a_i^j \vee \neg a_i^j \mid 1 \leq j \leq \ell\} \text{ and } br_i = \emptyset \quad . \qquad (3)$$

Intuitively, M is a binary tree-shaped MCS with depth m and $\ell+1$ is the size of the alphabet in each context. Figure 10 shows such an MCS with $n = 7$ contexts and depth $m = 2$; the internal contexts have knowledge bases and bridge rules as in (2), while the leaf contexts are as in (3). The directed edges show the dependencies of the bridge rules. Such a system M has equilibria $S = (S_1, \ldots, S_n)$ with $S_i = \{a_i^k, t_i\}$, for $1 \leq k \leq \ell$.

To compute one PE of M using DMCS or DMCSOPT, one needs to transfer packages of 2^ℓ PEs from each context to its parent (as each context C_i computes all subsets of $\{a_i^1, \ldots, a_i^\ell\}$). Each intermediate context receives 2^ℓ results from each of its children, whose join leads to $2^{2\ell}$ inputs for lsolve; it invokes lsolve that often and only then returns its 2^ℓ models to the parent, which has to wait for this.

On the other hand, DMCS-STREAMING only needs to transfer a single PE between each pair of connected contexts, which is a significant saving. Indeed, consider e.g. $m = 1, \ell = 5$, i.e., $M = (C_1, C_2, C_3)$. Querying C_1 with package size $k = 1$ first causes the query to be forwarded to C_2 as a pair $k_1 = k_2 = 1$. As C_2 is a leaf context, it invokes the local solver and eventually gets five different models. However, it just returns one PE to C_1, say $(\epsilon, \{a_2^1\}, \epsilon)$. Note that t_2 is projected off as it is not among the atoms of C_2 accessed by C_1. The same happens at C_3, which we assume

to return $(\epsilon, \epsilon, \{a_3^2\})$ to C_1. At the root context C_1, the two PEs from its neighbors are consistently combined into $(\epsilon, \{a_2^1\}, \{a_3^2\})$. Feeding this to the local solver, C_1 obtains five models, and returns one of them, say $S = (\{a_1^1, t_1\}, \{a_2^1\}, \{a_3^2\})$.

The following proposition shows the correctness of our algorithm.

Proposition 8 *Let $M = (C_1, \ldots, C_n)$ be an MCS, $i \in \{1, \ldots, n\}$ and let $k \geq 1$ be an integer. On input $(1, k)$ to C_i.Handler, C_i.Output returns up to k different partial equilibria with respect to C_i, and in fact k if at least k such partial equilibria exist.*

5.2 Parallelized Streaming

As one might expect, the strategy of ignoring up to k_1 models and then collecting the next k is not likely to be the most effective. The reason is that each context uses only one Solver, which in general has to serve more than one parent, i.e., requests for different ranges of models of size k. When a new parent context requests models, we have to refresh the state of Solver and Joiner and redo from scratch. This is unavoidable, unless a context satisfies the specific property that only one parent can call it.

A way to address this problem is parallelization. The idea is to serve each parent with a suite of Handler, Joiner, Solver and Output. The basic interaction between units is still as shown in Figure 9, with the notable difference that each component now runs in an *individual thread*. The significant change is that Solver does not control Joiner but waits at its queue to get new input for the local solving process. The Joiner independently queries the neighbors, combines PEs from neighbors, and puts the results into the Solver queue.

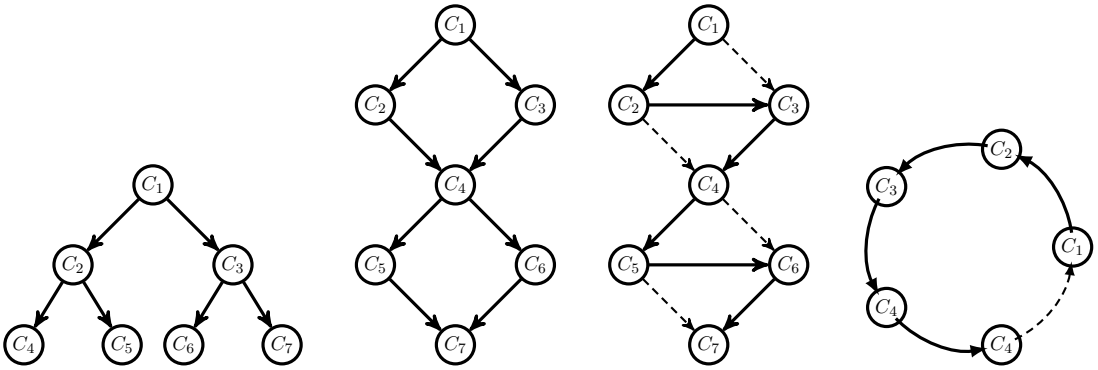
The effect is that we do not waste recomputation time for unused models. However, parallelization has its limits in practice. While DMCSOPT may run out of memory, unlimited parallel instances of the streaming algorithm can exceed the number of threads/processes that the operating system can support; this happens if contexts can reach others on many alternative paths, like in the stacked diamond topology: the number of threads is exponential in the number of connected contexts, which prohibits scaling to large system sizes. However, in real-world applications the number of paths might still be ok.

A compromise between the two extremes is a *bounded parallel* algorithm. The idea is to create a fixed-size pool of multiple threads and components to share among the contexts; when incoming requests cannot be served with the resources available, the algorithm continues with the basic streaming procedure. A realization remains for future work.

6. Experimental Evaluation

We have implemented the algorithms above using C++ in a system prototype called DMCS, which is available online.[8] For space reasons, we omit a detailed presentation and refer for it to the work of Bairakdar, Dao-Tran, Eiter, Fink, and Krennwallner (2010b), Dao-Tran (2014, ch. 7). Briefly, the main components of the global architecture are (i) a command-line frontend `dmcs` for the user to access the system; (ii) demons `daemon` which represent nodes that contain (a set of) contexts; and (iii) a manager `dmcsm` containing meta-information about the MCS (topology, interfaces) with

8. `http://www.kr.tuwien.ac.at/research/systems/dmcs`, `https://github.com/DistributedMCS/dmcs/`



(a) Binary Tree (T) (b) Diamond (D) (c) Zig-Zag (Z) (d) Ring (R)

Figure 11: Topologies for testing DMCS algorithms

a helper `dmcsgen` for generating configurations with optimized components. Contexts are implemented as groups of threads that communicate with each other through concurrent message queues. The system has two main command-line tools, viz. for running the algorihms and for test case generation, respectively. It allows to switch between different algorithms and modes by simply changing the command-line arguments.

We now turn to an experimental evaluation of DMCS under various aspects. Next we describe how the benchmarks were set up, and then we go into runs and results interpretation.

6.1 Benchmark Setup

The idea is to analyze strong and weak points of each algorithm with respect to different parameters, namely system topology, system size, local theory (i.e., knowledge base) size, and interface size. Specifically, we considered MCSs with topologies as in Figure 11, including:

Binary Tree (T): Binary trees grow balanced, i.e., every level except the last one is complete. With this topology, no edge needs to be removed to form the optimal topology; as every intermediate node is a cut-vertex, the import interface in the query plan is drastically reduced, leading to an extreme performance improvement.

(Stack of) Diamond(s) (D): a diamond consists of four nodes connecting as C_1 to C_4 in Figure 11b. A stack of diamonds combines multiple diamonds in a row, i.e., stacking m diamonds in a tower of $3m + 1$ contexts. Similar to Binary Tree, no edge is removed in constructing the query plan. W.r.t. this topology, every context connecting two diamonds is a cut-vertex. As such, the import interface in the query plan is refined after every diamond; this avoids significantly repetition of partial PEs in evaluation.

(Stack of) Zig-Zag Diamond(s) (Z): a zig-zag diamond is a diamond with a connection between the two middle contexts, as depicted by contexts C_2 to C_4 in Figure 11c. A stack of zig-zag diamonds is built as above. This topology is interesting as after removing two edges per block, the query plan turns into a linear topology.

Ring (R): ring (Figure 11d). The query plan removes the connection from context C_n to C_1 and then carries the interface between them all the way back to C_1. This topology requires guess-

Figure 12: Local theory structure

ing and checking in any DMCS algorithm; thus it is quite unpredictable which algorithm performs better in general.

The other quantitative parameters are represented as tuple $P = (n, s, b, r)$, where

- n is the system size (number of contexts),
- s is the local theory size (number of ground atoms in a local theory),
- b is the number of local atoms that can be used as bridge atoms in other contexts, in other words, the number of interface atoms, and
- r is the maximal number of bridge rules. The generator generates a bridge rule while iterating from 1 to r with 50% chance; hence on average $r/2$ bridge rules are generated. We allow bridge bodies of size 1 or 2.

A test configuration is formulated as $X/(n, s, b, r)$ where $X \in \{T, D, Z, R\}$ represents the topology and n, s, b, r are integers representing the quantitative (i.e., size-related) parameters. As we would like to run several instances over one configuration, the final formulation of a test instance is $X_i/(n, s, b, r)$, where i is the index of the test instance.

Inside each context, the local theories are structured as follows. Context C_i has s ground atoms indicated by $a_{i,1}, \ldots, a_{i,s}$. Rules are of the form $a_{i,j} \leftarrow \text{not } a_{i,k}$ where $k = j + 1$, if j is odd; otherwise, we randomly choose k to be $j - 1$ or $j + 1$ with a probability of 50% for each possibility. In case if $k > s$ then the rule does not exist. An example of a context with local theory size is 8 can be illustrated with the dependency graph as in Figure 12. Here, the bold arrows stand for the fixed rules while dashed arrows stands for the rules decided by randomization. The corresponding local theory of this figure is:

$$\left\{ \begin{array}{cccc} a_1 \leftarrow \text{not } a_2 & a_3 \leftarrow \text{not } a_4 & a_4 \leftarrow \text{not } a_5 & a_6 \leftarrow \text{not } a_7 \\ a_2 \leftarrow \text{not } a_1 & a_4 \leftarrow \text{not } a_3 & a_5 \leftarrow \text{not } a_6 & a_7 \leftarrow \text{not } a_8 \end{array} \right\}.$$

With this setting, a local context has 2^m answer sets, where $m \in [0, s/2]$.

Furthermore, one can obtain *deterministic contexts* (having just one answer set) by disallowing cycles in the structure of local theories.

6.2 Experiments

We conducted the experiments on a host system using 4-core Intel(R) Xeon(R) CPU 3.0GHz processor with 16GB RAM, running Ubuntu Linux 12.04.1. Furthermore, we used DLV [build BEN/Sep 28 2011 gcc 4.3.3] as a back-end ASP solver.

We ran a comprehensive set of benchmarks under the setup described in Section 6.1. As the parameter space $P = (n, s, b, r)$ is huge, we singled out in an initial probing phase the following values for the experiments:

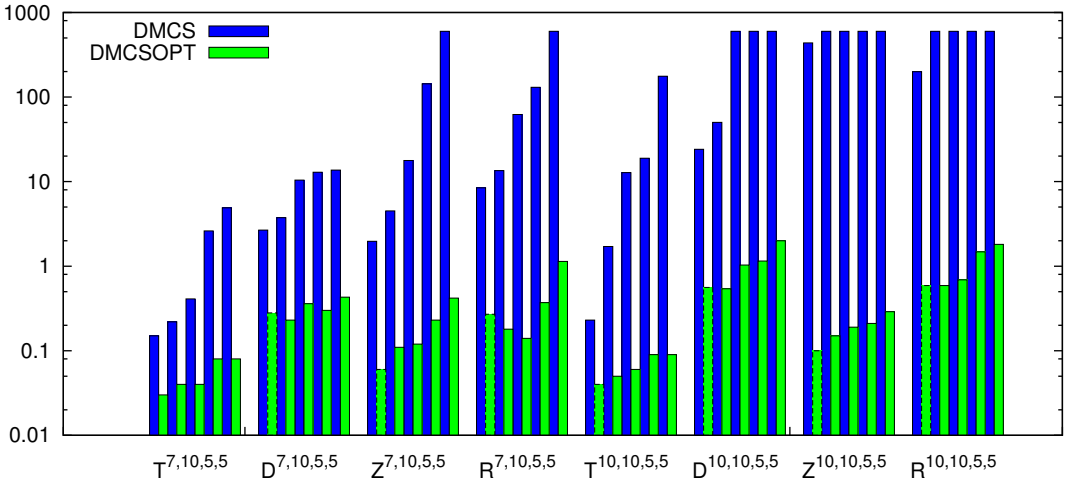


Figure 13: DMCS vs. DMCSOPT in *non-streaming* mode

- for the system size n, depending on the topology:

$$T\text{:}\quad n \in \{7, 10, 15, 31, 70, 100\} \qquad Z\text{:}\quad n \in \{4, 7, 10, 13, 25, 31, 70\}$$
$$D\text{:}\quad n \in \{4, 7, 10, 13, 25, 31\} \qquad R\text{:}\quad n \in \{4, 7, 10, 13, 70\}$$

- s, b, r are fixed to either $10, 5, 5$ or $20, 10, 10$, respectively.

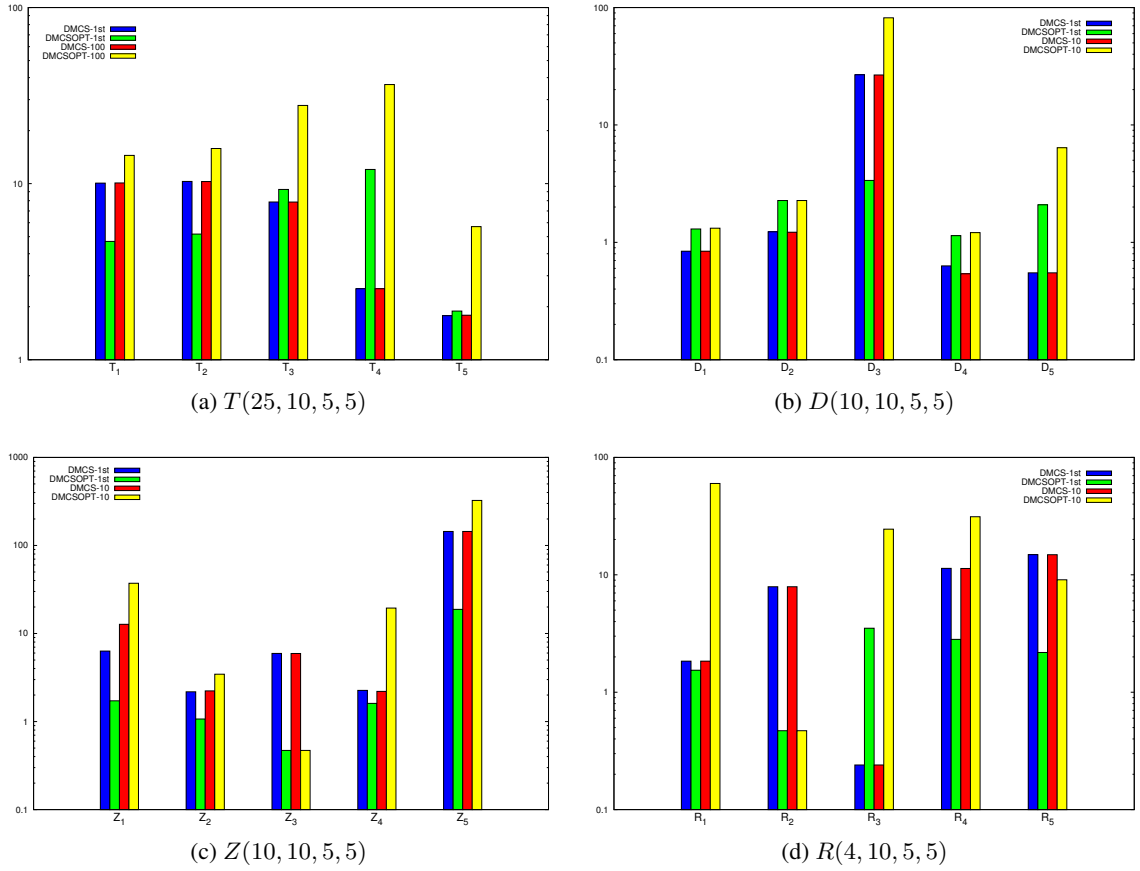
A combination of topology X and parameters $P = (n, s, b, r)$ is denoted by $X(n, s, b, r)$ or $X^{n,s,b,r}$ (used in figures). Each parameter setting has been tested on five instances. For each instance, we measured the total running time and the total number of returned partial equilibria on DMCS, DMCSOPT in non-streaming and streaming mode. For the latter mode, DMCS-STREAMING, we asked for k answers, where $k \in \{1, 10, 100\}$. This parameter also influences the size of packages transferred between contexts (at most k partial equilibria are transferred in one message). As in streaming mode, asking for more than one PE may require multiple rounds to get all answers, it is of interest to see how fast the first answers arrive compared to having all answers. We thus compared the running time of these tasks for $k = 10$ and $k = 100$.

6.3 Observations and Interpretations

Figures 13-17 summarize the results of our experiments. Run times are in seconds and timeout is 600 seconds. From these data, several interesting properties can be observed. We organize our analysis along the following aspects: (1) comparing DMCS and DMCSOPT, (2) comparing streaming and non-streaming mode, (3) effect of the package size, (4) role of the topologies, and (5) the behavior of the algorithms on deterministic contexts.

6.3.1 DMCS vs. DMCSOPT

Figure 13 shows the running time of DMCS and DMCSOPT for computing all partial equilibria, i.e., in *non-streaming mode*, of five instances of the respective parameter settings. Clearly DMCSOPT outperforms DMCS. This can be explained by the fact that when computing all answers, DMCS always produces more partial equilibria than DMCSOPT, as one PE returned by DMCSOPT can



(a) $T(25, 10, 5, 5)$ (b) $D(10, 10, 5, 5)$

(c) $Z(10, 10, 5, 5)$ (d) $R(4, 10, 5, 5)$

Figure 14: DMCS vs. DMCSOPT in *streaming* mode

be obtained from projecting many partial equilibria returned by DMCS on the imported interface. Furthermore, all intermediate results are transferred in one message, which makes no difference in terms of the number of communications between the algorithms. As such, DMCS must spend more time on processing possibly exponentially more input; hence, unsurprisingly, it is consistently slower than DMCSOPT.

However, the observation in *streaming* mode is different. Figure 14 shows the running time of DMCS and DMCSOPT in streaming mode to compute the first 100 respectively 10 *unique* partial equilibria for $T(25, 10, 5, 5)$ respectively $D(10, 10, 5, 5)$, $Z(10, 10, 5, 5)$ and $R(4, 10, 5, 5)$. On a first view, as DMCSOPT is consistently slower than DMCS, one might question the correctness of the results. However, they are not a surprise: again a PE returned by DMCSOPT should correspond to several PEs returned by DMCS. Hence, the batch of the first k unique answers in DMCS corresponds to only a smaller number of (few) unique answers in DMCSOPT.

Therefore, comparing DMCS and DMCSOPT in streaming mode by measuring the runtime to compute the first k answers is not fair. We thus took the time when both algorithms finished the first round of answers (denoted by DMCS-1st and DMCSOPT-1st in Figure 14). With this setting, we observed the following:

- on the majority of cases DMCSOPT finishes the first round faster than DMCS, however in about 40% of the instances, it is the other way around; this shows the effect of using the query plan;

• however, in some cases DMCS wins. This can be explained as follows. First of all, in streaming mode, we transfer only packages of k partial equilibria at a time; therefore, the effect of reducing the amount of total work to be done does not always apply as in the non-streaming mode. Furthermore, at every context, we compute k PEs and project them to the output interface before returning the results. According to this strategy, when a context C_i returns k_1 partial equilibria in non-streaming mode and k_2 partial equilibria in streaming to another context C_j, it might happen that k_2 is much smaller than k_1 and hence does not provide enough input for C_j to compute k PEs. Therefore, C_j will issue more requests to C_i asking for further packages of k PEs, e.g., $[k+1, 2k]$, $[2k+1, 3k]$, etc; and this costs DMCSOPT more time to even compute the first batch of PEs at the root context. Another approach is to compute always k unique partial equilibria before returning to a parent context. However, this strategy risks to compute even all local models before k unique partial equilibria can be found.

Overall, there is not much difference in running time when DMCSOPT is slower than DMCS, except for instance R_3 (Figure 14d). This however comes from a different reason: the cyclic topology with guess-and-check effects, which play a much more important role than choosing between DMCS and DMCSOPT (see Section 6.3.4).

6.3.2 STREAMING VS. NON-STREAMING DMCS

We now compare streaming and non-streaming for the same algorithm (DMCS resp. DMCSOPT).

Figure 15 shows the results for DMCS in (15a), and the results for DMCSOPT to compute the first 10 resp. 100 PEs with small systems/local knowledge bases in (15b) and with large systems/local theories in (15c). Excluding Ring (which behaves abnormally due to guess-and-check) one can see that:

• For DMCS, the streaming mode is definitely worth pursuing since DMCS in non-streaming mode times out in many cases (see also Figure 13), while in streaming mode we still could find some answers after a reasonable time.

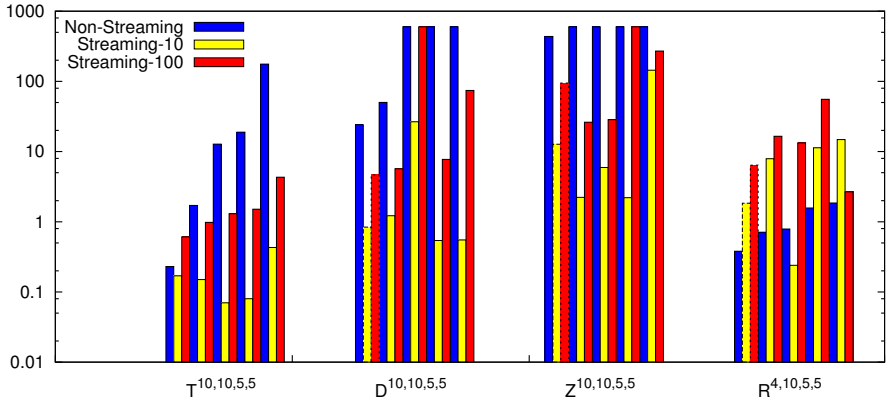
• For DMCSOPT, the situation is a bit different, as streaming loses against non-streaming on small instances. This is due to the recomputation that the streaming mode pays for transferring just chunks of partial equilibria between contexts; furthermore, there are duplications between answers. When one moves to larger systems and local knowledge bases, the streaming mode starts gaining back. However, it does not always win, as recomputation still significantly takes time in some cases.

Summing up, when the system is small enough, one should try the non-streaming mode as it avoids recomputation and duplication of PEs between different rounds of computation. But for large systems, streaming can rescue us from timing out. Even if we have to pay for recomputation, it still helps in cases when some but not all results are needed, e.g. in brave query answering (membership of the query in some PE).

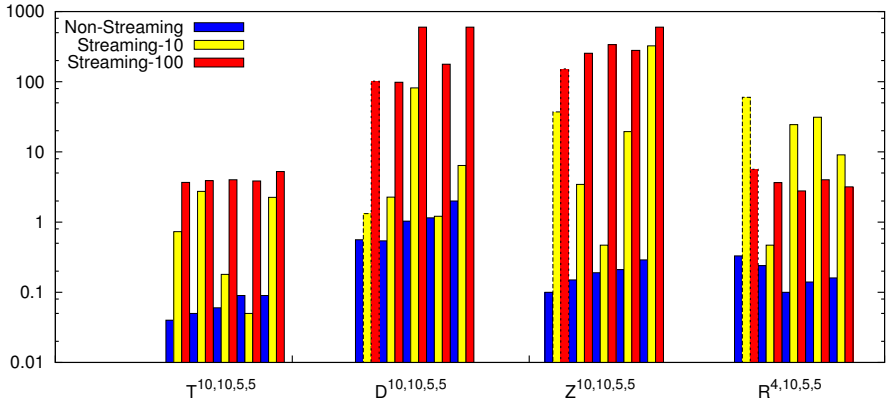
6.3.3 EFFECTS OF THE PACKAGE SIZE IN STREAMING MODE

The considerations above raise the question of the optimal number of PEs that should be transferred in return messages between contexts. We will analyze the experimental results on the streaming mode with package sizes 1, 10, and 100 to give some hints on this.

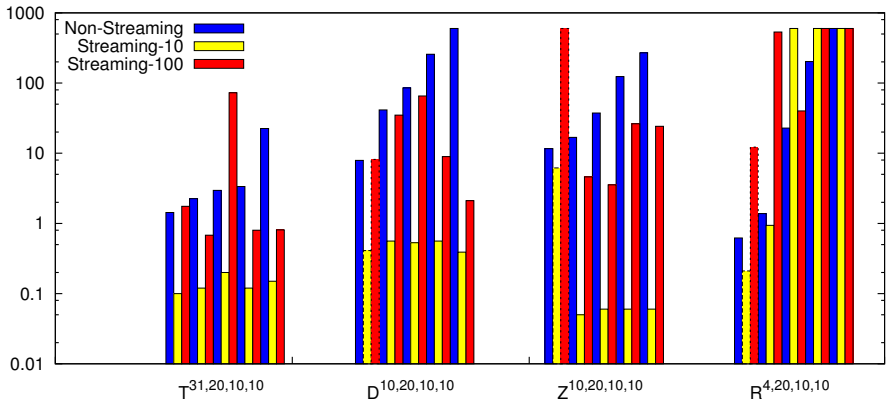
Figure 16 shows the average time to compute 1 PE of DMCSOPT in streaming mode with respect to three package sizes. One can see that transferring just a single PE to get the first answer



(a) DMCS



(b) DMCSOPT with *small* systems and local theories



(c) DMCSOPT with *large* systems and local theories

Figure 15: Non-streaming vs. streaming under DMCS and DMCSOPT

is acceptable in most cases, in particular if no guessing is needed. Moving from size 1 to a small package size like 10 here is sometimes better, as one can save communication time (sending once a package of 10 partial equilibria vs. sending ten times a package with a single PE). This setting (small package sizes like 10) will be more effective when communication is a big factor, which

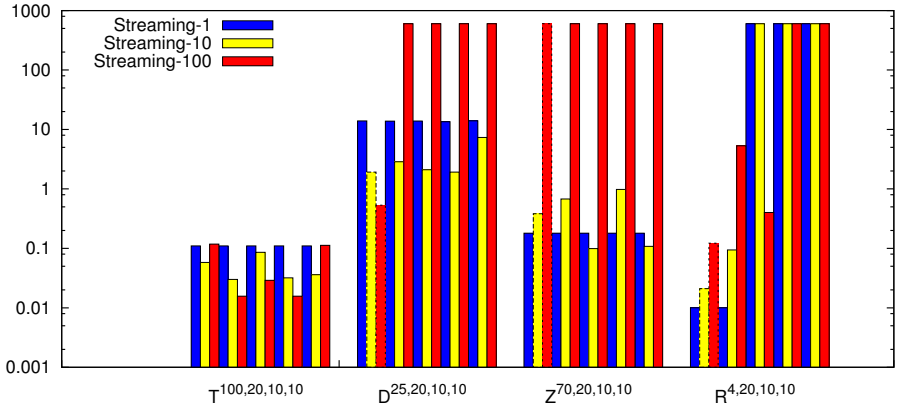


Figure 16: Average time of DMCSOPT to find one partial equilibrium in streaming mode, varying package size

happens in real applications where contexts are located at physically distributed nodes. In such cases, computing 10 partial equilibria should be faster than computing 1 PE in 10 consecutive times.

Furthermore, having package of size 1 is not safe in cases where guessing is applied, e.g., in $R_3(4, 20, 10, 10)$. For these cases, a large enough package size might help to cover the correct guess; but in general, there is no guarantee for such a coverage. To thoroughly solve this problem, one needs to apply conflict learning on the whole MCS evaluation.

Also, it is interesting to see that with package size 100, DMCSOPT usually times out. The reason is that there are many duplications and once DMCSOPT is stuck with a local search branch that promises fewer than 100 partial equilibria, the algorithm will lose time here without finding new unique answers and will eventually time out.

To find a good package size p with a specific setting (topology, system size, local theory size), one may run the system on a training set and apply binary search on p.

6.3.4 EFFECT OF TOPOLOGY

A quick glance over all plots in Figures 13–16 reveals the pattern that the algorithms, especially the optimizations, perform better on tree than on zigzag and diamond, depending on DMCS or DMCSOPT, and worst on ring.

The system topology plays an important role here. The aspects that affect the performance of the algorithms are (i) number of connections, (ii) the structure of block trees and cut vertices, and (iii) acyclicity vs. cyclicity.

Regarding (i), the topology introduces the number of connections based on the system size. Tree has fewer connections than Diamond and Zigzag, which reduces not only communication but also local solving time as fewer requests are made; and the performance of DMCS on these topologies proves this observation. If one follows this argument, then Ring must offer the best performance. However, this is actually not the case due to aspect (iii) that we will shortly analyze below.

Concerning (ii), tree can be ultimately optimized as every intermediate node is a cut vertex. Hence, when applying the query plan for DMCSOPT, we can strip off all beliefs in PEs sent from child contexts to a parent context. In other words, only local beliefs at a context C_i are needed to be transferred back to its parents. This drastically decreases the amount of information to be

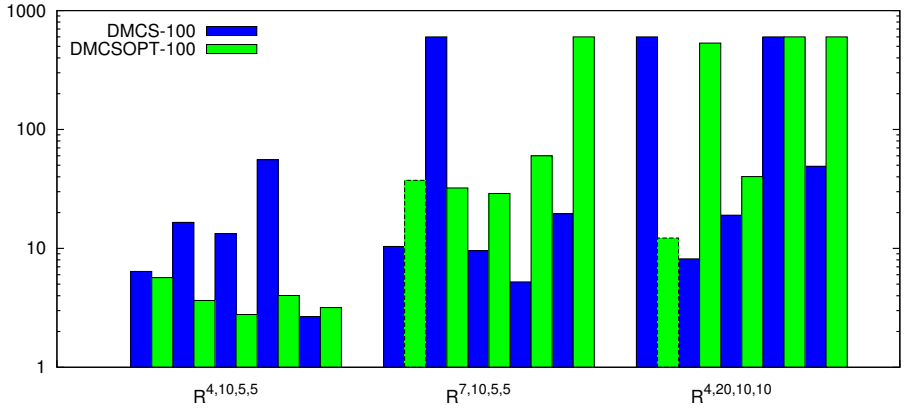


Figure 17: DMCS vs. DMCSOPT in streaming mode with package size 100 on ring

communicated, and more importantly, the number of calls to lsolve. Due to this special property, DMCSOPT performs extremely well on the tree topology, and scales to hundreds of contexts.

Comparing Diamond and Zigzag, they have the same number of cut vertices. However, Zigzag is converted to a linear topology with an optimal query plan (cf. Figure 11c), and therefore can be processed much faster than Diamond. In Figure 16, DMCSOPT scales on Zigzag to 70 contexts with an average time to compute one answer that still better than the one on diamond with 25 contexts.

Regarding (iii), Ring is a cyclic topology while the other topologies are acyclic. Hence each of the algorithms must do some guess-and-check at some context in the topology. Making the right guess is most important, even more important than reducing communication and calls to local solvers. The result of running DMCS and DMCSOPT on this topology (Figure 17) does not follow any pattern; it absolutely depends on a specific instance whether the above sequential guessing luckily arrives at the result. Therefore, we frequently see that DMCS outperforms DMCSOPT in streaming mode, as in such cases, guessing at the root context (after detecting the cycle) is more effective than guessing at the parent of the root context according to the optimal query plan.

Based on these observations, one can come up with a best strategy to evaluate different types of topologies. When dealing with MCSs of arbitrary topologies, it looks natural to decompose them into parts of familiar topologies for which efficient strategies are known, and to combine then these strategies to an overall evaluation method. Studying this is beyond the scope of this work and an interesting issue for future research.

6.3.5 BEHAVIOR ON DETERMINISTIC CONTEXTS

Above we considered our algorithms on MCSs consisting of possibly non-deterministic contexts, i.e., they can have more than one acceptable belief set per knowledge base. It is intriguing to see how the algorithms behave if all contexts always have exactly one accepted belief set per knowledge base; this might be because the underlying logic is genuinely "deterministic" and the accepted belief set clear (e.g., closure in classical logic) or among multiple candidates a particular belief set is chosen (in implementations typically the first or a "best" solution computed, e.g. in SAT solving or in ASP). We observed that:

- for non-cyclic topologies, there is no performance difference between DMCS and DMCSOPT, because the smaller interface used in DMCSOPT does not reduce the number of intermediate PEs transferred between contexts, as there is only one partial equilibrium computed at every context.

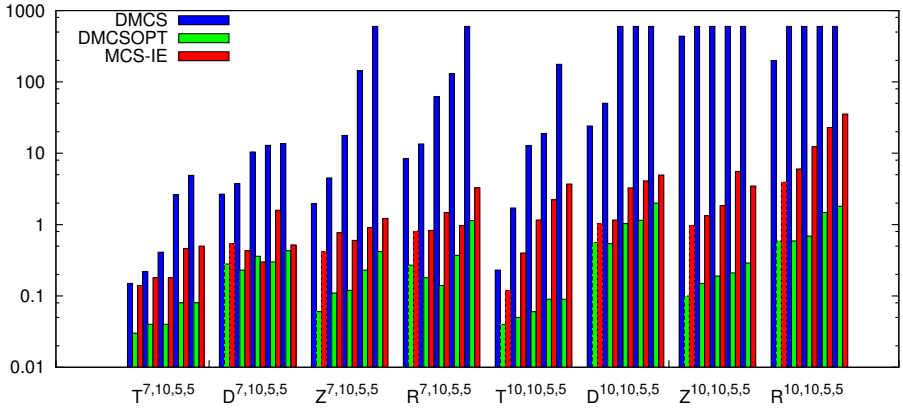


Figure 18: DMCS vs. DMCSOPT in streaming mode with package size 100 on ring

• for cyclic topology (Ring), guessing plays the main role. Hence it depends on the individual instance whether DMCS or DMCSOPT wins, like in the case of non-deterministic contexts (cf. Section 6.3.4).

• non-streaming mode is much faster than streaming (on both DMCS and DMCSOPT); this is reasonable as any request for further partial equilibria is redundant.

6.3.6 COMPARISON WITH MCS-IE AND P2P-DR

Systems close to DMCS are MCS-IE (Bögl et al., 2010)[9] and P2P-DR (Bikakis, Antoniou, & Hassapis, 2011). The former is a plugin of the dlvhex system and was originally developed to compute explanations for inconsistency in Multi-context Systems, but also includes a mode for computing equilibria of an MCS. However, MCS-IE was implemented with a centralized approach. Figure 18 presents the run time of DMCS, DMCSOPT in comparison with MCS-IE in computing all partial equilibria of the respective configurations. It shows that MCS-IE outperforms DMCS since it inherits a powerful decomposition technique from dlvhex; however, the decomposition based on topological information of DMCSOPT turns out to be more efficient, as it also localizes the interface beliefs to communicate between blocks of contexts, which is specific for MCS and is not exploited by the general decomposition technique in dlvhex.

P2P-DR supports distributed query answering for multi-context systems based on defeasible logic; for more details, see Section 7. We present here a comparison between DMCS and P2P-DR.

We converted our benchmark to P2P-DR's style by converting the local knowledge bases and bridge rules to defeasible local and meta rules, and added a fixed trust order between contexts. We then queried the root context with an atom appearing in one of the answers of DMCS-STREAMING with package size 10. It turned out that P2P-DR always found the answers in around 0.25 seconds, regardless of the tested instance. This behavior can be explained as follows. To find answers for a query atom, the algorithm of P2P-DR first evaluates the local theory. If it can determine the truth value of the query, it terminates; otherwise the algorithm consults neighbors to get further evidence for the reasoning. As our local knowledge base structure, when converted to P2P-DR's defeasible theories, allows for a local decision, the system works only on the local theory of the root context

9. http://www.kr.tuwien.ac.at/research/systems/mcsie/tut/

for every test case, thus results in almost constant execution time. Even when asking neighbours is necessary, P2P-DR in general may be much faster than DMCS, as the query answering process is inherently deterministic and in a low-complexity logic; in turn, the formalism is less expressive. A detailed study of this issue remains for future work.

6.3.7 SUMMARY

Summing up, the analysis of the experimental results shows that there is no clear winner among the algorithms (DMCS vs. DMCSOPT) under different running modes (streaming vs. non-streaming, with different package size) on different topologies. We can distill from it a guideline to choose the setup that fits specific instances in practice, including some issues open for further investigation, which can be briefly stated as follows:

- choose DMCSOPT over DMCS in non-streaming mode, except for cyclic topologies;
- in streaming mode, choose an appropriate package size carefully (e.g., doing a binary search on some training instances;
- decompose random topologies into parts whose topologies have effective strategies to evaluate, and study how to combine the strategies for the over all systems.

7. Related Work

In this section, we resume the discussion of related work. Starting with multi-context systems, we provide more details on the work by Roelofsen et al. (2004), Bikakis et al. (2011) and consider other work. We then move to related formalisms in SAT, CSP and ASP.

Roelofsen et al. (2004) described evaluation of monotone MCS with classical theories using SAT solvers for the contexts in parallel. They used a (co-inductive) fixpoint strategy to check MCS satisfiability, where a centralized process iteratively combines results of the SAT solvers. Apart from being not truly distributed, an extension to nonmonotonic MCS is non-obvious; furthermore, no caching technique was used.

Serafini, Borgida, and Tamilin (2005) and Serafini and Tamilin (2005) developed distributed tableaux algorithms for reasoning in distributed ontologies, which can be regarded as multi-context systems with special bridge rules. The algorithms serve to decide whether such a system is consistent, provided no cyclic context dependencies exist (in technical terms, the distributed TBox is acyclic); the DRAGO system (Serafini & Tamilin, 2005) implements this approach for OWL ontologies. Compared to ours, this work is tailored for a specific class of multi-context systems resp. knowledge bases, without nonmonotonic negation and cyclic dependencies (which are challenging); furthermore, it targets query answering rather than model building, which in a sense is a dual problem.

More related to our work as regards distributed evaluation is the the system P2P-DR of Bikakis et al. (2011). They developed a distributed algorithm for query evaluation in a multi-context system framework that is specifically based on (propositional) defeasible logic. In this framework, contexts are built using defeasible rules and can exchange literals via bridge rules, and a trust order between contexts is be supplied. Each knowledge base at a context has, in our terminology, a single accepted belief set which contains the literals concluded; the global system semantics is given in terms of a (unique) three-valued assignment to all literals, which can be determined using the algorithm: whether literal l is provably (not) a logical conclusion of the system, or whether this remains

open. Apart from being tailored to a particular logic and preference mechanisms for evaluating interlinked contexts, applying this algorithm to model building is not straightforward; in particular, as it produces unique belief sets, dealing with nondeterminism and multiple equilibria is not possible.

Our work on computing equilibria for distributed multi-context systems is clearly related to work on solving constraint satisfaction problems (CSP) and SAT solving in a distributed setting; Yokoo and Hirayama (2000) survey some algorithms for distributed CSP solving, which are usually developed for a setting where each node (agent) holds exactly one variable, the constraints are binary, communication is done via messages, and every node holds constraints in which it is involved. This is also adopted by later works (Gao, Sun, & Zhang, 2007) but can be generalized (Yokoo & Hirayama, 2000). In relation to the topology-based optimization techniques in Section 4, biconnected components are used by Baget and Tognetti (2001) to decompose CSP problems. The decomposition is used to localize the computation of a single solution in the components of undirected constraint graphs. Along the same lines, our approach is based on directed dependencies, which allows us to use a query plan for MCS evaluation.

The predominant solution methods in CSP are backtracking algorithms. Bessiere, Bouyakhf, Mechqrane, and Wahbi (2011) took them a step further with backtracking on a dynamic total ordering between agents guided by nogoods. Our approach, however, allows for cyclic dependency between contexts. Hirayama and Yokoo (2005) presented a suite of algorithms for solving distributed SAT (DisSAT), based on a random assignment and improvement flips to reduce conflicts. However, these algorithms are geared towards finding a single model, and an extension to streaming multiple (or all) models is not straightforward; for other works on distributed CSP and SAT, this is similar.

Finally, (distributed) SAT and CSP solving concerns monotonic systems (removal of clauses resp. constraints preserves satisfiability), while MCSs evaluation concerns nonmonotonic systems, even if all contexts were monotonic (e.g., clause sets); this makes efficient evaluation more difficult, as important structural properties of the search space cannot be exploited.

Adjiman, Chatalic, Goasdoué, Rousset, and Simon (2006) present a framework of peer-to-peer inference systems, where local theories of propositional clause sets share atoms and a special algorithm for consequence finding is available. As we pursue the dual problem of model building, applying it for our needs is not straightforward; furthermore, we are dealing with non-monotonic systems, while the peer-to-peer systems by Adjiman et al. are monotonic.

Moving to ASP, Pontelli, Son, and Nguyen's (2011) ASP-PROLOG shares with MCS the idea of integrating several knowledge bases, called modules, possibly under different semantics. However, they restricted module semantics to ASP and Prolog (that is, the least Herbrand model), and ASP-PROLOG pursues query answering instead of model building.

As for streaming, an answer set streaming algorithm for HEX-programs (which generalize ASP with external information access) was given by Eiter, Fink, Ianni, Krennwallner, and Schüller (2011). Despite some similarities to Algorithm DMCS-STREAMING, it is rather different: monolithic programs are syntactically decomposed into modules and answer sets computed in a modular fashion; it is not fully distributed and combines partial models from lower components to input for upper components straightforwardly; moreover, it may use exponential space in components.

8. Conclusion

We have considered distributed evaluation of Multi-context Systems (MCSs) that were introduced by Brewka and Eiter (2007) as a general formalism to interlink possibly nonmonotonic and heterogeneous knowledge bases. We have presented a suite of generic algorithms to compute the equilibria, i.e., the semantics of an MCS in a fully distributed manner, using local solvers for the knowledge bases at the contexts. It contains a basic algorithm DMCS, an advanced version DMCSOPT that uses topology-based optimizations, and a streaming variant DMCS-STREAMING for computing partial equilibria gradually. We believe that the underlying principles and techniques might be exploited in related contexts, and in particular for distributed evaluation of other non-monotonic knowledge base formalisms.

The algorithms have been implemented in a prototype system that is available as open source.[8] On top of this implementation, we have conducted comprehensive experiments to compare the performance of the algorithms and we gave an insight analysis on the results. It points out advantages, disadvantages as well as the time/memory trade off between the algorithms in different situations depending on parameters such as system topology, local interface and theory size, and number of equilibria desired by the user. Based on this, the user can choose the setting (algorithm and mode) that fits her need best for finding (partial) equilibria of an MCS. A more extensive treatment is given by Dao-Tran (2014).

Further work and open issues. Several issues remain for further investigation. One is further improvement of the algorithms. Here, the experimental results on the Ring topology strongly suggest to incorporate conflict learning, which proved to be valuable in ASP and SAT solving, to DMCS and DMCSOPT; we expect that cyclic topologies will benefit from a better guided guessing process. Another issue concerns further semantics and variants of MCSs. As for the former, grounded equilibria are considered by Dao-Tran (2014), which are akin to answer sets of logic programs and applicable to MCSs that satisfy certain algebraic conditions; they can be characterized like answer sets using an (adapted) loop formula approach (Lee & Lifschitz, 2003). Dealing with supported equilibria (Tasharrofi & Ternovska, 2014), however, is open.

Regarding MCS variants, managed MCSs (Brewka et al., 2011) generalize bridge rules to derive operations (commands) for a management function that is applied on the knowledge bases; it seems possible to generalize our algorithms to this setting, but an efficient realization is not straightforward. Another generalization of MCS concerns dynamic data: in areas like sensor networks, social networks, or smart city applications, data may change or even continuously arrive at nodes, which motivates reactive and stream processing for MCSs (Goncalves et al., 2014; Brewka et al., 2014). Last but not least, allowing contexts to evolve via interation with users or with changes in the environment is a valuable extention. Extending our algorithms to these settings is interesting but challenging.

Finally, extending this work to query answering over MCSs, where the user poses a query at a context and receives results derived from (partial) equilibria is another natural issue. As there is no need for building whole equilibria, better performance may be achieved.

Acknowledgments

This research has been supported by the Austrian Science Fund (FWF) projects P20841 and P26471.

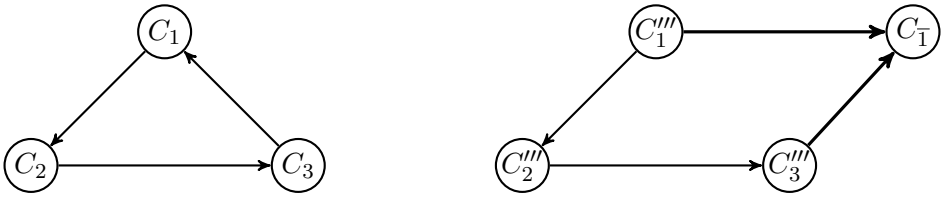


Figure 19: Introducing guess context(s)

We thank to the reviewers for pointing out corrections and their constructive suggestions which helped to improve the presentation of this work, and we thank Antonis Bikakis for providing us with the P2P-DR system for the experimental comparison.

Appendix A. Proofs

Proof of Theorem 1

To prove this theorem, we first prove the following Lemmas 9 and 10. The latter aims at simplifying the proof for the cyclic case, based on the notion of converting cyclic MCSs to acyclic ones.

Lemma 9 *For any context C_k and partial belief state S of an MCS $M = (C_1, \ldots, C_n)$,*

$$app(br_k, S) = app(br_k, S|_{\mathcal{V}}) \text{ for all } \mathcal{V}_{\mathbf{B}} \supseteq \mathcal{V} \supseteq \mathcal{V}^*(k).$$

Proof For any $r \in app(br_k, S)$, we have that for all $(c_i : p_i) \in B^+(r)$: $p_i \in S_{c_i}$ and for all $(c_j : p_j) \in B^-(r)$: $p_j \notin S_{c_j}$. We need to show that $p_i \in S_{c_i}|_{\mathcal{V}_{c_i}} \wedge p_j \notin S_{c_j}|_{\mathcal{V}_{c_j}}$. Indeed:

We have $\mathcal{V} \subseteq \mathcal{V}_{\mathbf{B}} \Rightarrow \mathcal{V}_{c_j} \subseteq \mathcal{V}_{\mathbf{B}_j} \Rightarrow S_{c_j}|_{\mathcal{V}_{c_j}} \subseteq S_{c_j}$. Therefore, $p_j \notin S_{c_j} \Rightarrow p_j \notin S_{c_j}|_{\mathcal{V}_{c_j}}$.

Now, assume that $p_i \notin S_{c_i}|_{\mathcal{V}_{c_i}}$. From the fact that $p_i \in S_{c_i}$, it follows that $p_i \notin \mathcal{V}_{c_i}$, hence $p_i \notin \mathcal{V}^*(k)$. But this is in contradiction with the fact that p_i occurs in some bridge rule body.

Therefore, $r \in app(br_k, S|_{\mathcal{V}})$. □

The next Lemma 10 is based on the following notions that convert cyclic MCSs to acyclic ones and show that they have corresponding equilibria. The intuition (illustrated in Figure 19 and Examples 18, 19) is to introduce an additional context $C_{\overline{k}}$ to take care of guessing for every cycle breaker C_k. Then, the bridge rules of C_k and its parents are modified to point to $C_{\overline{k}}$. We now formally realize this idea starting with a function ren that renames part of the bridge rules.

Definition 12 *Let C_k be a context in an MCS M, and let $\mathcal{V}$ be an interface for running DMCS. The* renaming function ren *is defined as follows:*

- *For an atom a:* $ren(a, k, \mathcal{V}) = \begin{cases} a^g & \text{if } a \in \mathbf{B}_k \cap \mathcal{V} \\ a & \text{otherwise} \end{cases}$
- *For a context index c:* $ren(c, k, \mathcal{V}) = \begin{cases} \overline{c} & \text{if } c \in \{1, \ldots, n\} \\ c & \text{otherwise} \end{cases}$
- *For a bridge atom $(c_i : p_i)$:* $ren((c_i : p_i), k, \mathcal{V}) = (ren(c_i, k, \mathcal{V}) : ren(p_i, k, \mathcal{V}))$

- *For a bridge body* $B = \{(c_1 : p_1) \dots (c_j : p_j)\}$*:*

$$ren(B, k, \mathcal{V}) = \{ren((c_i : p_i), k, \mathcal{V}) \mid (c_i : p_i) \in B\}$$

- *For a bridge rule* $r = head(r) \leftarrow B(r)$*:*

$$ren(r, k, \mathcal{V}) = head(r) \leftarrow ren(B(r), k, \mathcal{V})$$

- *For a set of bridge rules* br*:* $ren(br, k, \mathcal{V}) = \{ren(r, k, \mathcal{V}) \mid r \in br\}$
- *For a context* $C_i = (L_i, kb_i, br_i)$ *in* M*:* $ren(C_i, k, \mathcal{V}) = (L_i, kb_i, ren(br_i, k, \mathcal{V}))$.

Example 18 Let us slightly modify the MCS $M = (C_1, C_2, C_3)$ from Example 8 as follows:

- $kb_1 = \{e \vee \neg e\}$, $br_1 = \{a \leftarrow (1 : e), \text{not } (2 : b)\}$;
- $kb_2 = \emptyset$, $br_2 = \{b \leftarrow (3 : c)\}$; and
- $kb_3 = \emptyset$, $br_3 = \{c \vee d \leftarrow \text{not } (1 : a)\}$.

Applying function ren to contexts C_1 and C_3 results in the following bridge rules wrt. an interface $\mathcal{V} = \{a, b, c, e\}$:

- $ren(br_1, 1, \mathcal{V}) = \{a \leftarrow (\overline{1} : e^g), \text{not } (2 : b)\}$,
- $ren(br_3, 1, \mathcal{V}) = \{c \vee d \leftarrow \text{not } (\overline{1} : a^g)\}$.

For two contexts C_i and C_j, the former is called a parent of the latter with respect to an interface $\mathcal{V}$, denoted by $parent(C_i, C_j, \mathcal{V})$ iff there exists a bridge rule $r \in br_i$ such that there exists $(c : p) \in B(r)$ and $p \in \mathbf{B}_j \cap \mathcal{V}$.

A set of contexts $\{C_{c_1}, C_{c_2}, \dots, C_{c_\ell}\}$ of an MCS M is called a cycle w.r.t. an interface $\mathcal{V}$ iff

$$parent(C_{c_\ell}, C_1, \mathcal{V}) \wedge \bigwedge_{1 \le i \le \ell - 1} parent(C_{c_i}, C_{c_{i+1}}, \mathcal{V})$$

holds. One can pick an arbitrary context in this set to be its cycle-breaker. Given an MCS M, there are several ways to choose a (finite) set of its contexts to be cycle-breakers. In Algorithm DMCS, Step (d) practically establishes the cycle-breakers based on the order that elements in $In(k)$ are iterated. For the next definition, we are interested in this particular set of cycle-breakers.

Definition 13 *Given an MCS* $M = (C_1, \dots, C_n)$*, let* $\mathcal{CB}_M^r = \{C_{c_1}, \dots, C_{c_j}\}$ *be the set of cycle-breakers for* M *based on the application of* DMCS *on* M *starting from context* C_r*. The conversion of* M *to an* equal *acyclic* $M^\star$ *based on* $\mathcal{CB}_M^r$ *and an interface* $\mathcal{V}$ *is done as follows:*

Let $C_i' = (L_i, kb_i, br_i') = \begin{cases} ren(C_i, i, \mathcal{V}) & \text{if } C_i \in \mathcal{CB}_M^r \\ C_i & \text{otherwise} \end{cases}$

Let $C_i'' = (L_i, kb_i, br_i'') = \circ_{C_k \in \mathcal{CB}_M} ren(C_i', k, \mathcal{V})$[10]

Let $C_i''' = (L_i, kb_i, br_i''')$ *where* $br_i''' = \begin{cases} br_i'' \cup \{a \leftarrow (\bar{i} : a^g) \mid a \in \mathbf{B}_i \cap \mathcal{V}\} & \text{if } C_i \in \mathcal{CB}_M^r \\ br_i'' & \text{otherwise} \end{cases}$

For each $C_j \in \mathcal{CB}_M^r$*, introduce* $C_{\bar{j}} = (L_{\bar{j}}, kb_{\bar{j}}, br_{\bar{j}})$ *where* $br_{\bar{j}} = \emptyset$ *and* $kb_{\bar{j}} = \{a^g \vee \neg a^g \mid a \in \mathbf{B}_j \cap \mathcal{V}\}$*. Then* $M^\star = (C_1''', \dots, C_n''', C_{\overline{c_1}}, \dots, C_{\overline{c_j}})$.

10. The order of composing function ren with different parameters k does not matter here.

Example 19 (cont'd) Let M be the MCS from Example 18 and $\mathcal{CB}^r_M = \{C_1\}$. Then, the conversion in Definition 13 gives $M^\star = (C_1''', C_2''', C_3''', C_{\overline{1}})$, where:

- $kb_1 = \{e \vee \neg e\}$, $br_1''' = \{a \leftarrow (\overline{1} : e^g), \text{not}\,(2 : b). \quad a \leftarrow (\overline{1} : a^g). \quad e \leftarrow (\overline{1} : e^g).\}$;
- $kb_2 = \emptyset$, $br_2''' = \{b \leftarrow (3 : c)\}$;
- $kb_3 = \emptyset$, $br_3''' = \{c \vee d \leftarrow \text{not}\,(\overline{1} : a^g)\}$; and
- $kb_{\overline{1}} = \{e^g \vee \neg e^g. \quad a^g \vee \neg a^g.\}$, $br_{\overline{1}} = \emptyset$.

Lemma 10 *Let M be an MCS and $M^\star$ be its conversion to an acyclic MCS as in Definition 13. Then the equilibria of M and $M^\star$ are in 1-1 correspondence.*

Proof (Sketch) Let (R_1) and (R_2) be the runs of DMCS on M and $M^\star$, respectively. Due to the selection of $\mathcal{CB}^r_M$ to construct $M^\star$, both (R_1) and (R_2) have the same order visiting the contexts, except that when (R_1) revisits a cycle-breaker $C_k \in \mathcal{CB}^r_M$, its counterpart (R_2) visits $C_{\overline{k}}$. At these corresponding locations:

- (R_1) calls guess$(\mathcal{V}, C_k)$ at Step (c), and
- (R_2) calls lsolve$(\{\epsilon, \ldots, \epsilon\})$ at Step (e) since $C_{\overline{k}}$ is a leaf context.

The construction of the local knowledge base of $C_{\overline{k}}$ gives us exactly the guess on C_k. Furthermore, these guesses are passed on to the parent contexts of $C_{\overline{k}}$ and then later on unified by the additional bridge rules $a \leftarrow (\overline{k} : a^g)$ introduced in br_k'''. Therefore, the belief combinations (Step (d)) done at C_k are executed on the same input on both runs (R_1) and (R_2). The correspondence of equilibria hence follows. □

Proof (Theorem 1) Thanks to Lemma 10, we now need to prove Theorem 1 only for the acyclic case and automatically get the result for the cyclic case.

($\Rightarrow$) We start by showing soundness of DMCS. Let $S' \in C_k$.DMCS$(\mathcal{V}, \emptyset)$ such that $\mathcal{V} \supseteq \mathcal{V}^*(k)$. We show now that there is a partial equilibrium S of an acyclic M w.r.t. C_k such that $S' = S|_{\mathcal{V}}$. We proceed by structural induction on the topology of M.

Base case: C_k is a leaf with $In(k) = \emptyset$ and $br_k = \emptyset$ and $k \notin hist$. This means that (d) is not executed, hence, in (e), lsolve runs exactly once on $(\epsilon, \ldots, \epsilon)$, and we get as result the set of all belief states $\mathcal{S} = \text{lsolve}((\epsilon, \ldots, \epsilon)) = \{(\epsilon, \ldots, \epsilon, T_k, \epsilon, \ldots, \epsilon) \mid T_k \in \mathbf{ACC}_k(kb_k)\}$. We now show that $S' \in \mathcal{S}|_{\mathcal{V}}$. Towards a contradiction, assume that there is no partial equilibrium $S = (S_1, \ldots, S_n)$ of M w.r.t. C_k such that $S' = S|_{\mathcal{V}}$. From $In(k) = \emptyset$, we get that $IC(k) = \{k\}$, thus the partial belief state $(\epsilon, \ldots, \epsilon, T_k, \epsilon, \ldots, \epsilon) \in \mathcal{S}$ (where $T_k \in \mathbf{ACC}_k(kb_k)$) is a partial equilibrium of M w.r.t. C_k. Contradiction.

Induction step: assume context C_k has import neighborhood $In(k) = \{i_1, \ldots, i_m\}$ and

$$\begin{aligned} \mathcal{S}^{i_1} &= C_{i_1}.\text{DMCS}(\mathcal{V}, hist \cup \{k\}), \\ &\vdots \\ \mathcal{S}^{i_m} &= C_{i_m}.\text{DMCS}(\mathcal{V}, hist \cup \{k\}). \end{aligned}$$

Then by the induction hypothesis, for every $S'^{i_j} \in \mathcal{S}^{i_j}$, there exists a partial equilibrium S^{i_j} of M w.r.t. C_{i_j} such that $S^{i_j}|_{\mathcal{V}} = S'^{i_j}$.

Let $\mathcal{S} = C_k.\mathsf{DMCS}(\mathcal{V}, hist)$. We need to show that for every $S' \in \mathcal{S}$, there is a partial equilibrium of M w.r.t. C_k such that $S' = S|_{\mathcal{V}}$. Indeed, since $In(k) \neq \emptyset$, Step (d) is executed; let

$$\mathcal{T} = \mathcal{S}^{i_1} \bowtie \cdots \bowtie \mathcal{S}^{i_m}$$

be the result of combining partial belief states from calling DMCS at $C_{i_1}, \ldots, C_{i_m}$. Furthermore, by Step (e), we have that $\mathcal{S} = \mathcal{S}^\star|_{\mathcal{V}}$ where $\mathcal{S}^\star = \bigcup\{\mathsf{lsolve}(S) \mid S \in \mathcal{T}\}$. Eventually, $S' \in \mathcal{S}|_{\mathcal{V}}$. Since every DMCS at $C_{i_1}, \ldots, C_{i_m}$ returns its partial equilibria w.r.t. C_{i_j} projected to $\mathcal{V}$, we have that every $T \in \mathcal{T}$ is a partial equilibrium w.r.t. C_{i_j} projected to $\mathcal{V}$. M is acyclic and we have visited all contexts from $In(k)$, thus by Lemma 9 we get that for every $T \in \mathcal{T}$, $app(br_k, T)$ gives us all applicable bridge rules r regardless of $T_j = \epsilon$ in T, for $j \notin In(k)$. Hence, for all $T \in \mathcal{T}$, $\mathsf{lsolve}(T)$ returns only partial belief states, where each component is projected to $\mathcal{V}$ except the kth component. As every $T \in \mathcal{T}$ preserves applicability of the rules by Lemma 9, we get that for every $S' \in \mathcal{S}|_{\mathcal{V}}$, there exists a partial equilibrium S of M w.r.t. C_k such that $S' = S|_{\mathcal{V}}$.

($\Leftarrow$) We give now a proof for completeness of DMCS by structural induction on the topology of an acyclic M. Let $S = (S_1, \ldots, S_n)$ be a partial equilibrium of M w.r.t. C_k and let $S' = S|_{\mathcal{V}}$. We show now that $S' \in C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$.

Base case: C_k is a leaf context. Then, when executing $C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$, Step (d) is ignored and Step (e) is called with input $(\epsilon, \ldots, \epsilon)$, and $\mathsf{lsolve}((\epsilon, \ldots, \epsilon))$ gives us all belief sets $\mathcal{S}$ of C_k. As S is an equilibrium of M w.r.t. C_k, $S \in \mathcal{S}$; hence, $S' = S|_{\mathcal{V}}$ will be returned from $C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$.

Induction case: suppose that the import neighborhood of context C_k is $In(k) = \{i_1, \ldots, i_m\}$. Let the restriction of S to every context $C_{i_j} \in In(k)$ be denoted by S^{i_j}, where:

$$S^{i_j} = (S'_1, \ldots, S'_n) \text{ where } S'_\ell = \begin{cases} S_\ell & \text{if } \ell \in IC(i_j) \\ \epsilon & \text{otherwise} \end{cases}$$

Informally speaking, this restriction keeps only belief sets of the contexts reachable from C_{i_j} and sets those of non-reachable contexts to ϵ. By the induction hypothesis, $S^{i_j}|_{\mathcal{V}}$ is computed by $C_{i_j}.\mathsf{DMCS}(\mathcal{V}, \emptyset)$ for all $i_j \in In(k)$. We will show that $S|_{\mathcal{V}}$ is computed by $C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$.

Indeed, because we are considering an acyclic M, it holds that $S^{i_j}|_{\mathcal{V}}$ is also returned from a call $C_{i_j}.\mathsf{DMCS}(\mathcal{V}, \{k\})$, as k plays no role in further calls from C_{i_j} to its neighbors. This means that after step (d), $\mathcal{T}$ contains a $T = S_{i_1} \bowtie \ldots \bowtie S_{i_m}$ where S_{i_j} appears at position i_j in S.

Since S is a partial equilibrium of M w.r.t. C_k, we have that $S_k \in \mathbf{ACC}_k(kb_k \cup \{head(r) \mid r \in app(br_k, S)\})$. Furthermore, by choosing $\mathcal{V} \supseteq \mathcal{V}^*(k)$, Lemma 9 tells us that the applicability of bridge rules is preserved under the projection of belief sets to $\mathcal{V}$. This gives us that $S_k \in \mathsf{lsolve}(T)$ in step (e), and hence $S' = S|_{\mathcal{V}}$ is returned from $C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$. □

Proof of Proposition 4

(1) For a context C_k, let the number of calls to its local solver be denoted by $c(k)$. This number is calculated during the computation of $\mathcal{T}$ in Step (d), and it is bounded by the maximal number of combined partial belief sets from its neighbors. Formally speaking:

$$c(k) \leq \Pi_{i \in In(k)} 2^{|\mathcal{V} \cap \mathbf{B}_i|} \leq 2^{|In(k)| \times |\mathcal{V}|} \leq 2^{n \times |\mathcal{V}|}.$$

Hence for the whole MCS, the upper bound of calls to lsolve in a run of DMCS is

$$c = \Sigma_{1 \leq k \leq n} c(k) \leq n \times 2^{n \times |\mathcal{V}|}$$

(2) For a context C_k of an MCS $M = (C_1, \ldots, C_n)$, the set $E(k)$ contains all dependencies from contexts C_i for $i \in IC(k)$. We visit all $(i, j) \in E(k)$ exactly twice during DFS-traversal of M: once when calling C_j.DMCS$(\mathcal{V}, hist)$ at C_i, and once when retrieving $\mathcal{S}|_{\mathcal{V}}$ from C_j in C_i. Furthermore, the caching technique in Step (a) prevents recomputation on already visited nodes, thus prevents recommunication in the subtree of any visited node. The claim hence follows. □

Proof of Proposition 5

Item (i) is trivial to see since CycleBreaker is applied in Algorithm 4. To prove item (ii), let us look at two cases in which an edge (ℓ, t) is removed from the original topology at Step (a) of Algorithm 3:

- (ℓ, t) is removed by CycleBreaker: this causes that certain nodes in the graph cannot reach t via ℓ. However, the interface that C_t provides is already attached to $v(i, j)$ via $\mathcal{V}^*(c_p)|_{\mathbf{B}_t}$.
- (ℓ, t) is removed by transitive reduction: this does not change the reachability of t from other nodes; therefore, the interface that C_t provides is already included in $\mathcal{V}^*(i, j)_{B'}$.

This argument gives us property (ii). □

Proof of Proposition 6

First, we estimate the complexity to compute $v(i, j)$ in loop (a).

$$v(i, j) := \mathcal{V}^*(i, j)_{B'} \cup \bigcup_{c \in \mathcal{C}'} \mathcal{V}^*(c_p)|_{\mathbf{B}_c} \cup \bigcup_{(\ell,t) \in E} \mathcal{V}^*(c_p)|_{\mathbf{B}_t}$$

On the one hand, the refined recursive import $\mathcal{V}^*(i, j)'_B$ is defined as (Definition 9):

$$\mathcal{V}^*(i, j)'_B = \{\mathcal{V}^*(i) \cap \bigcup_{\ell \in B'|_j} \mathbf{B}_\ell\}$$

where $B'|_j$ contains all nodes reachable from j.

On the other hand, since all sets of possible beliefs in different contexts are disjoint, we have that

$$\bigcup_{c \in \mathcal{C}'} \mathcal{V}^*(c_p)|_{\mathbf{B}_c} \cup \bigcup_{(\ell,t) \in E} \mathcal{V}^*(c_p)|_{\mathbf{B}_t} = \mathcal{V}^*(c_p)|_{\bigcup_{c \in \mathcal{C}'} \mathbf{B}_c \cup \bigcup_{(\ell,t) \in E} \mathbf{B}_t}$$

Since the recursive import interface for a node k is defined as $\mathcal{V}^*(k) = \bigcup_{i \in IC(k)} \mathcal{V}(i)$, the expression to compute $v(i, j)$ is in the end a combination of set intersection, union, and projection. With an implementation of sets using hash set, that is, look up takes $O(1)$, these operators can be implemented in linear time. Therefore, $v(i, j)$ can be computed in linear time in the total number of beliefs of contexts in the system.

Given G_M, the block tree graph $T(G_M)$ can be constructed in linear time (Vats & Moura, 2010). Ear-decomposition (Step (c)) can also be done in linear time (Valdes, Tarjan, & Lawler,

1982). Transitive reduction (Step (d)) can be computed in quadratic time with respect to the number of edges in the block.

OptimizeTree$(T(G_M), k, k)$ iterates through all blocks. Assume that we have m blocks $B_1 \ldots$, B_m, and each B_i contains n_i edges, where $n = \Sigma_{i=1}^{m} n_i$ is the total number of edges in the original graph. Let t_i be the time to process block B_i. Then the bound of the total processing time can be assessed as follows:

$$t = \sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} n_i^2 \leq (\sum_{i=1}^{m} n_i)^2 = n^2.$$

Therefore, if we ignore loop (a), OptimizeTree can be done in quadratic time in the size of the original input, i.e., the size of G_M. □

Proof of Theorem 7

To prove this, we need Proposition 11 to claim that partial equilibria returned from DMCS and DMCSOPT are in correspondence. But first, we need the following supportive notion.

Definition 14 *Let C_k be a context of an MCS M, and let Π_k be the query plan as in Definition 11. For each block B of Π_k, the* block interface *of B, whose root vertex is c_B, is*

$$\mathcal{V}_B = \{p \in v(i,j) \mid (i,j) \in E(B)\} \cup \mathbf{B}_{c_B}.$$

Let C_i be a context in B. The self-recursive import interface of C_i in B *is*

$$\mathcal{V}^*(i)_B = \mathbf{B}_i \cup \bigcup_{(i,\ell) \in E(\Pi_k)} \mathcal{V}^*(i,\ell)_B.$$

Proposition 11 *Let C_k be a context of an MCS M, let Π_k be the query plan as in Definition 11 in which C_k belongs to block B of Π_k and let $\mathcal{V} = \bigcup_{B \in \Pi_k} \mathcal{V}_B$. Then,*

(i) for each $S' \in$ DMCSOPT(k) called from C_c where $(c,k) \in E(\Pi_k)$ or $c = k$, there exists a partial equilibrium $S \in C_k$.DMCS$(\mathcal{V}, \emptyset)$ such that $S' = S|_{\mathcal{V}^*(c,k)_B}$ if $(c,k) \in E(\Pi_k)$ or $S' = S|_{\mathcal{V}^*(k)_B}$ if $c = k$;

(ii) for each $S \in C_k$.DMCS$(\mathcal{V}, \emptyset)$, there exists some $S \in$ DMCSOPT(k) called from C_c such that $S' = S|_{\mathcal{V}^*(c,k)_B}$ if $(c,k) \in E(\Pi_k)$ or $S' = S|_{\mathcal{V}^*(k)_B}$ if $c = k$.

A detailed proof for Proposition 11 is given in the next section, we now give a proof for Theorem 7.

Proof (Theorem 7) (i) Let $S' \in C_k$.DMCSOPT(k) be a result from DMCSOPT. By Proposition 11 (i) for $c = k$, there exists an $S'' \in C_k$.DMCS$(\mathcal{V}, \emptyset)$ such that $S' = S''|_{\mathcal{V}^*(k)_B}$, where we choose $\mathcal{V} = \bigcup_{B \in \Pi_k} \mathcal{V}_B$. Note that $\mathcal{V}^*(k) \subseteq \mathcal{V}$ as $\mathcal{V}$ collects all bridge atoms from all blocks, which might contain blocks not reachable from k. By Theorem 1, there exists a partial equilibrium S of M such that $S'' = S|_{\mathcal{V}}$. Thus, we have that

$$\begin{array}{rcll} S' & = & (S|_{\mathcal{V}})|_{\mathcal{V}^*(k)_B} & \\ & = & S|_{\mathcal{V}^*(k)_B} & \text{because } \mathcal{V}^*(k)_B \subseteq \mathcal{V} \\ & = & S|_{\widehat{\mathcal{V}}} & \text{because } \widehat{\mathcal{V}} \subseteq \mathcal{V}^*(k)_B \end{array}$$

(ii) Let S be a partial equilibrium of M. By Theorem 1, there exists $S'' \in C_k$.DMCS$(\mathcal{V}, \emptyset)$ such that $S'' = S|_{\mathcal{V}}$ where we choose $\mathcal{V} = \bigcup_{B \in \Pi_k} \mathcal{V}_B$. As above, $\mathcal{V}^*(k) \subseteq \mathcal{V}$. By Proposition 11 (ii) for $c = k$, there exists $S' \in C_k$.DMCSOPT(k) such that $S' = S''|_{\mathcal{V}^*(k)_B}$. As above, we have that $S' = S|_{\widehat{\mathcal{V}}}$. □

Proof of Proposition 11

To support the proof of Proposition 11, we need the following lemmas.

Lemma 12 *Assume context C_k has import neighborhood $In(k) = \{i_1, \ldots, i_m\}$, no (k, i_j) is removed from the original topology by* OptimizeBlock(B, c_B)*, and*

$$\begin{array}{rclcrcl} \mathcal{S}'^{i_1} & = & \textit{DMCSOPT}(k) \text{ at } C_{i_1} & & \mathcal{S}^{i_1} & = & C_{i_1}.\textit{DMCS}(\mathcal{V}_B, \emptyset) \\ & \vdots & & & & \vdots & \\ \mathcal{S}'^{i_m} & = & \textit{DMCSOPT}(k) \text{ at } C_{i_m} & & \mathcal{S}^{i_m} & = & C_{i_m}.\textit{DMCS}(\mathcal{V}_B, \emptyset) \end{array}$$

such that for every partial equilibrium $S' \in \mathcal{S}'^{i_j}$, there exists $S \in \mathcal{S}^{i_j}$ such that $S' = S|_{\mathcal{V}^*(k,i_j)_B}$.

Let $\mathcal{T}' = \mathcal{S}'^{i_1} \bowtie \ldots \bowtie \mathcal{S}'^{i_m}$ and $\mathcal{T} = \mathcal{S}^{i_1} \bowtie \ldots \bowtie \mathcal{S}^{i_m}$. Then, for each $T' \in \mathcal{T}'$, there exists $T \in \mathcal{T}$ such that $T' = T|_{\mathcal{V}_{input}(1,m)}$ with $\mathcal{V}_{input}(\ell_1, \ell_2) = \bigcup_{j=\ell_1}^{\ell_2} \mathcal{V}^*(k, i_j)_B$.

Proof We prove by induction on the number of neighbors in $In(k)$.

Base case: $In(k) = \{i\}$, the claim trivially holds.

Induction case: $In(k) = \{i_1, \ldots, i_\ell\}$, $\mathcal{U}' = \mathcal{S}'^{i_1} \bowtie \ldots \bowtie \mathcal{S}'^{i_{\ell-1}}$, $\mathcal{U} = \mathcal{S}^{i_1} \bowtie \ldots \bowtie \mathcal{S}^{i_{\ell-1}}$, and for each $U' \in \mathcal{U}'$, there exists $U \in \mathcal{U}$ such that $U' = U|_{\mathcal{V}_{input}(1,\ell-1)}$. We need to show that for each $T' \in \mathcal{U}' \bowtie \mathcal{S}'^{i_\ell}$, there exists a $T \in \mathcal{U} \bowtie \mathcal{S}^{i_\ell}$ such that $T' = T|_{\mathcal{V}_{input}(1,\ell)}$.

Assume that the opposite holds, i.e., there exists $T = U' \bowtie S'$ where $U' \in \mathcal{U}'$ and $S' \in \mathcal{S}'^{i_\ell}$, and for all $U \in \mathcal{U}$, $S \in \mathcal{S}^{i_\ell}$ such that $U' = U|_{\mathcal{V}_{input}(1,\ell-1)}$ and $S' = S|_{\mathcal{V}^*(k,i_\ell)_B}$, we have that $U \bowtie S$ is void.

This means there exists a context C_t reachable from C_k by two different ways, one via i_ℓ and the other via one of $i_1, \ldots, i_{\ell-1}$ such that $U_t \neq \epsilon$, $S_t \neq \epsilon$, $U_t \neq S_t$, and either

(i) $U'_t = \epsilon$ or $S'_t = \epsilon$, or

(ii) $U'_t = S'_t \neq \epsilon$

Case (i) cannot happen because C_t is reachable from C_k, hence $\mathcal{V}_{input}(1, \ell-1) \cap \mathbf{B}_t \neq \emptyset$ and $\mathcal{V}^*(k, i_\ell) \cap \mathbf{B}_t \neq \emptyset$.

Concerning case (ii), we have that $U_t|_{\mathcal{V}_{input}(1,\ell-1)} = S_t|_{\mathcal{V}^*(k,i_\ell)} \neq \epsilon$, hence there exists $a \in U_t \setminus U_t|_{\mathcal{V}_{input}(1,\ell-1)}$ and $a \notin S_t|_{\mathcal{V}^*(k,i_\ell)}$. This means that $\mathcal{V}_{input}(1, \ell-1) \cap \mathbf{B}_t \neq \mathcal{V}^*(k, i_\ell) \cap \mathbf{B}_t$.

However, from Definition 9 of recursive import interface, we have that $\mathcal{V}^*(k, i_x)_B = \mathcal{V}^*(k) \cap \bigcup_{\ell \in B|_k} \mathbf{B}_\ell$, where $B|_{i_x}$ contains all nodes in B reachable from i_x. It follows that $\mathcal{V}^*(k, i_\ell)$ and $\mathcal{V}^*(k, i_j)$ for any $1 \leq j \leq \ell - 1$ that reaches t, share the same projection to $\mathbf{B}_t$, hence $\mathcal{V}_{input}(1, \ell - 1) \cap \mathbf{B}_t = \mathcal{V}^*(k, i_\ell) \cap \mathbf{B}_t$.

We reach a contradiction, and therefore Lemma 12 is proved. □

Lemma 13 *The join operator $\bowtie$ has the following properties, given arbitrary belief states S, T, U with the same size: (i) $S \bowtie S = S$ (ii) $S \bowtie T = T \bowtie S$ (iii) $S \bowtie (T \bowtie U) = (S \bowtie T) \bowtie U$. These properties also hold for sets of belief states.*

Proof The first two properties are trivial to prove. We will prove associativity.

Let $R = S \bowtie (T \bowtie U)$ and $W = (S \bowtie T) \bowtie U$. Consider doing the joins from left to right. At each position i ($1 \leq i \leq n$), R_i and W_i are determined by locally comparing S_i, T_i and U_i. If we

$S_i = \epsilon$	$T_i = \epsilon$	$U_i = \epsilon$	$S_i = T_i$	$T_i = U_i$	$U_i = S_i$	R_i	W_i
Y	Y	Y	Y	Y	Y	ϵ	ϵ
Y	Y	N	Y	N	N	U_i	U_i
Y	N	Y	N	N	Y	T_i	T_i
Y	N	N	N	Y	N	T_i	T_i
			N	N	N	void	void
N	Y	Y	N	Y	N	S_i	S_i
N	Y	N	N	N	Y	S_i	S_i
			N	N	N	void	void
N	N	Y	Y	N	N	S_i	S_i
			N	N	N	void	void
N	N	N	Y	Y	Y	S_i	S_i
			Y	N	N	void	void
			N	Y	N	void	void
			N	N	N	void	void

Table 1: Possible cases when joining at position i

reach inconsistency, the process terminates and $void$ is returned; otherwise, we conclude the value for R_i, W_i and continue to the next position. The final join is returned if position n is processed without any inconsistency.

All possible combination of S_i, T_i, and W_i are shown in Table 1. One can see that we always have the same outcome for R_i and W_i. Therefore, we have in the end either $R = W$ or both are $void$. This concludes that the join operator $\bowtie$ is commutative. □

Lemma 14 *Let C_i and C_j be two contexts in M such that they are in the same block after executing* OptimizeTree *and there is a directed path from C_i to C_j. Suppose that $\mathcal{S}^i = \mathsf{DMCSOPT}(k)$ at C_i and $\mathcal{S}^j = \mathsf{DMCSOPT}(k)$ at C_j. Then $\mathcal{S}^i = \mathcal{S}^i \bowtie \mathcal{S}^j$.*

Proof The use of cache in DMCSOPT does not change the result and can be disregarded, i.e., we can assume without loss of generality that $cache(k) = \emptyset$ in DMCSOPT. Indeed, $cache(k)$ is filled with the result of the computation when it is empty (i.e., when C_k is accessed the first time), and is after that never changed and DMCSOPT just returns $cache(k)$, i.e., the value of the computation with empty $cache(k)$.

Under the above assumption, Lemma 14 can be proven by taking any path $C_i = C_{p_1}, \ldots, C_{p_h} = C_j$ that connects C_i to C_j, and arguing that for each index $\ell \in \{1, \ldots, h\}$, it holds that $\mathcal{S}^{p_\ell} = \mathcal{S}^{p_\ell} \bowtie \mathcal{S}^j$ $(\star)$. Indeed, we can show this by an induction on the path.

Base case: $\ell = h$, statement $(\star)$ holds as we have $\mathcal{S}^{p_h} \bowtie \mathcal{S}^j = \mathcal{S}^j \bowtie \mathcal{S}^j = \mathcal{S}^j$ by identity (Lemma (13), (i)).

Induction case: consider $\ell < h$, and suppose we already established by the induction hypothesis that $\mathcal{S}^{p_{\ell+1}} = \mathcal{S}^{p_{\ell+1}} \bowtie \mathcal{S}^j$.

Now by definition of $\mathcal{S}^{p_\ell}$ and DMCSOPT, it holds that $\mathcal{S}^{p_\ell} = \mathsf{lsolve}(\mathcal{T})$[11] and $\mathcal{T}$ is, by the statements (b) and (c), of the form $\mathcal{T} = \mathcal{S}^{p_{\ell+1}} \bowtie \mathcal{T}'$; this holds because there is an edge $(p_\ell, p_{\ell+1})$ in E, and because $\bowtie$ is commutative and associative (Lemma (13), (ii) and (iii)). By the induction hypothesis, we get

$$\mathcal{T} = S^{p_{\ell+1}} \bowtie \mathcal{T}' = (\mathcal{S}^{p_{\ell+1}} \bowtie \mathcal{S}^j) \bowtie \mathcal{T}' = \mathcal{S}^j \bowtie (S^{p_{\ell+1}} \bowtie \mathcal{T}'),$$

that is, $\mathcal{T}$ is of the form $\mathcal{S}^j \bowtie \mathcal{T}''$.

Next, $\mathsf{lsolve}(\mathcal{T})$ does not change the value of any component of any interpretation I in $\mathcal{T}$ that is defined in $\mathcal{S}^j$; that is, $\mathsf{lsolve}(\mathcal{T}) \bowtie \mathcal{S}^j = \mathsf{lsolve}(\mathcal{T})$. This means $\mathcal{S}^{p_\ell} = \mathsf{lsolve}(\mathcal{T}) = \mathsf{lsolve}(\mathcal{T}) \bowtie \mathcal{S}^j = \mathcal{S}^{p_\ell} \bowtie \mathcal{S}^j$, which proves statement $(\star)$ holds for ℓ.

Eventually, we get for $\ell = 1$ that $\mathcal{S}^i = \mathcal{S}^{p_1} = \mathcal{S}^{p_1} \bowtie \mathcal{S}^j = \mathcal{S}^i \bowtie \mathcal{S}^j$. □

Based on Lemma 14, we have the following result.

Lemma 15 *Assume the import neighborhood of context C_k is $In(k) = \{i_1, \ldots, i_m\}$, and that $\mathcal{S}^{i_j} = \mathsf{DMCSOPT}(k)$ at C_{i_j}, $1 \leq j \leq m$. Furthermore, suppose that edge (k, i_j) was removed by the optimization process ($1 \leq j \leq m$), and that C_{i_ℓ} is a neighbor of C_k such that there exists a path from k to i_j through i_ℓ in the optimized topology. Then $\mathcal{S}^{i_\ell} = \mathcal{S}^{i_\ell} \bowtie \mathcal{S}^{i_j}$; in other words, the input to DMCSOPT at C_k is not affected by the removal of (k, i_j).*

Proof Since C_{i_j} and C_{i_ℓ} are direct children of C_k, it follows that they belong to the same block. Therefore, by Lemma 14 we have that $\mathcal{S}^{i_\ell} = \mathcal{S}^{i_\ell} \bowtie \mathcal{S}^{i_j}$. □

Proof (Proposition 11) We proceed by structural induction on the block tree of an MCS M. First, we consider the case where the topology of M is a single block B. In this case, the interface passed to DMCS is $\mathcal{V} = \mathcal{V}_B$.

Base case: C_k is a leaf. Then we now compare a call $\mathsf{DMCSOPT}(k)$ at C_k and $C_k.\mathsf{DMCS}(\mathcal{V}, \emptyset)$, where $\mathcal{V} = \mathcal{V}^*(k)_B = \mathbf{B}_k$. Algorithm 1 returns local belief sets of C_k projected to $\mathcal{V}$ and Algorithm 5 returns plain local belief sets, the claim follows as $\mathcal{V} = \mathcal{V}^*(k)_B = \mathbf{B}_k$.

Induction case: Assume the import neighborhood of context C_k is $In(k) = \{i_1, \ldots, i_m\}$, and

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{S}'^{i_1} & = & \mathsf{DMCSOPT}(k) \text{ at } C_{i_1} & \mathcal{S}^{i_1} & = & C_{i_1}.\mathsf{DMCS}(\mathcal{V}_B, \emptyset) \\ & \vdots & & & \vdots & \\ \mathcal{S}'^{i_m} & = & \mathsf{DMCSOPT}(k) \text{ at } C_{i_m} & \mathcal{S}^{i_m} & = & C_{i_m}.\mathsf{DMCS}(\mathcal{V}_B, \emptyset) \end{array}$$

such that for every partial equilibrium $S' \in \mathcal{S}'^{i_j}$, there exists $S \in \mathcal{S}^{i_j}$ such that $S' = S|_{\mathcal{V}^*(k,i_j)_B}$.

There are two cases. First, no edge (k, i_j) is removed by the optimization procedure. Then, by Lemma 12, we have the correspondence between the input to DMCSOPT and DMCS at C_k.

On the other hand, assume that an edge (k, i_j) was removed by the optimization process. The removal can be from either transitive reduction or ear decomposition. In the former case, Lemma 15 shows that the input to C_k is not affected by the removal of this edge. For the latter case, the removal can be one of three possibilities as illustrated in Figure 20, assuming that context C_1 gets called:

(i) $(6, 1)$, the last edge of the simple cycle $P_0 = \{1, 2, 3, 4, 5, 6\}$

11. With abuse of notation, we write $\mathsf{lsolve}(\mathcal{T})$ for $\bigcup_{T \in \mathcal{T}} \mathsf{lsolve}(T)$

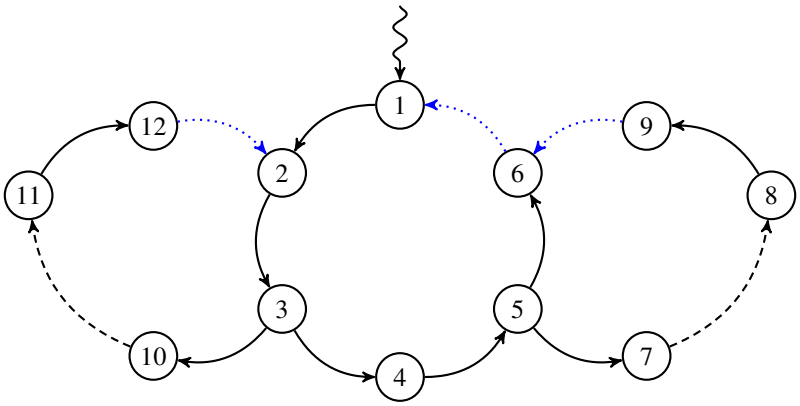


Figure 20: Possible cycle breakings

(ii) $(9, 6)$, the last edge of path $P_1 = \{5, 7, 8, 9, 6\}$

(iii) $(12, 2)$, the last edge of path $P_2 = \{3, 10, 11, 12\}$

Cases (i) and (iii) differ from case (ii) in the sense that a cycle will be recognized by DMCS while for case (ii), no cycle is detected along the corresponding path.

Now, consider when (k, i_j) is removed in situations similar to cases (i) and (iii), DMCSOPT will issue a guess at Step (c) of Algorithm 5 on $v(k, i_j)$, which includes $\mathcal{V}^*(c_B)|_{\mathbf{B}_{i_j}} = \mathcal{V}_B \cap \mathbf{B}_{i_j}$. On the other hand, DMCS will recognize the cycle at C_{i_j} and issue a guess on $\mathcal{V}_B \cap \mathbf{B}_{i_j}$ at Step (c) of Algorithm 1. Therefore, the guess is fed equally to C_k.

When (k, i_j) is removed in situations similar to case (ii), all guesses of C_k on the interface from C_{i_j} will be eventually filtered when being combined with the local belief states computed by C_{i_j}, at the starting node of the path containing (k, i_j) as the last edge (in the ear decomposition). In Figure 20, this is node 5.

In all cases, we have that whenever there is an input T' into Isolve in DMCSOPT(k) called by C_c, there is an input T to Isolve in C_k.DMCS$(\mathcal{V}_B, \emptyset)$. Therefore, the claim on the output holds.

Now that Proposition 11 holds for a single leaf block, one can see that the upper blocks only need to import the interface beliefs from the cut vertices (also the root contexts of the lower blocks). Under the setting of $\mathcal{V} = \bigcup_{B \in \Pi_k} \mathcal{V}_B$, results from DMCSOPT and DMCS projected to the interface of the cut vertices are identical. Therefore, the upper blocks receive the same input regarding the interfaces of the cut vertices in running both algorithms. And therefore the final results projected to $\mathcal{V}^*(k)_B$ are in the end the same. □

Proof of Proposition 8

Note that the components Handler and Output simply take care of the communication part of DMCS-STREAMING. Output makes sure that the models sent back to the invokers are in correspondence with the request that Handler got. The other routines Joiner and Solver are the main components that play the role of Step (b) and (d) in Algorithm 5, respectively.

$$
\begin{array}{lccccccccc}
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,1} & \bowtie & \mathcal{T}_{m,1} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,1} & \bowtie & \mathcal{T}_{m,p_m} \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,2} & \bowtie & \mathcal{T}_{m,1} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,2} & \bowtie & \mathcal{T}_{m,p_m} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & \mathcal{T}_{m,1} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & \mathcal{T}_{m,p_m} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,1} & \bowtie & \mathcal{T}_{m,1} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & \mathcal{T}_{m,p_m} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,p_1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_1} & \bowtie & \mathcal{T}_{m,1} \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,p_1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & \mathcal{T}_{m,p_m} \\
= & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,1} & \bowtie & F(m,m) \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & F(m,m) \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,1} & \bowtie & F(m,m) \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & F(m,m) \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,p_1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & \mathcal{T}_{m-1,p_{m-1}} & \bowtie & F(m,m) \\
= & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,1} & \bowtie & \cdots & \bowtie & F(m-1,m) & & \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & F(m-1,m) & & \\
 & \cdots & & & & & & \cdots & & \\
 & \mathcal{T}_{1,p_1} & \bowtie & \mathcal{T}_{2,p_2} & \bowtie & \cdots & \bowtie & F(m-1,m) & & \\
= & [\mathcal{T}_{1,1} \bowtie F(2,m)] & & & \cup & \cdots & \cup & [\mathcal{T}_{1,p_1} \bowtie F(2,m)] & & \\
= & F(1,m). & & & & & & & &
\end{array}
$$

Table 2: Accumulation of Joiner

To prove the correctness of DMCS-STREAMING, we just need to show that the input to lsolve is complete in the sense that if Step (e) of Algorithm 8 is exhaustively executed, the full join of partial equilibria from the neighboring contexts is delivered.

Formally, assume that the current context's import neighborhood is $\{1, 2, \ldots, m\}$. Assume that for neighbor C_i where $1 \leq i \leq m$, the full partial equilibria are $\mathcal{T}_i$ and the returned packages of size k are denoted by $\mathcal{T}_{i,1}, \ldots, \mathcal{T}_{i,p_i}$, that is, $\mathcal{T}_i = \mathcal{T}_{i,1} \cup \ldots \cup \mathcal{T}_{i,p_i}$. For the correctness of the algorithm, we assume that $\mathcal{T}_{i,1}, \ldots, \mathcal{T}_{i,p_i}$ is a fixed partition of $\mathcal{T}_i$. This is possible when, for example, lsolve always returns answers in a fixed order. We need to show that the accumulation of the join by Algorithm 8 is actually $\mathcal{T}_1 \bowtie \ldots \bowtie \mathcal{T}_m$.

Indeed, each possible join $\mathcal{T}_{1,i_1} \bowtie \mathcal{T}_{2,i_2} \bowtie \ldots \bowtie \mathcal{T}_{m,i_m}$ is considered by Joiner, which performs a lexicographical traversal of all suitable combinations. Formally speaking, let $F(p, q)$, where $q < q$, denote the join result of neighbors from p to q, that is, $F(p, q) = \mathcal{T}_p \bowtie \mathcal{T}_{p+1} \bowtie \ldots \bowtie \mathcal{T}_q$. According to the lexicographical order, we have that the accumulation of Joiner is $\bigcup_{j=1}^{p_1}[\mathcal{T}_{1,j} \bowtie F(2, m)] = F(1, m)$ as demonstrated in Table 2.

This shows that the input to lsolve is complete. Hence, DMCS-STREAMING is correct. □

Appendix B. Detailed Run of OptimizeTree

Example 20 We illustrate now the call OptimizeTree$(T = (\mathcal{B} \cup \mathcal{C}, \mathcal{E}), c_p, c_r)$ for the block set $\mathcal{B} = \{B_1, B_2, B_3\}$, $B_1 = \{1, 2, 3, 4\}$, $B_2 = \{4, 5\}$, $B_3 = \{3, 6\}$, $\mathcal{C} = \{1, 3, 4\}$, $\mathcal{E} = \{(B_1, 1), (B_2, 4), (B_3, 3)\}$, and $c_p = c_r = 1$.

From the local knowledge bases presented in Example 10, we have:

$$\mathbf{B}_1 = \{car_1, train_1, nuts_1\} \qquad \mathbf{B}_4 = \{car_4, train_4\}$$
$$\mathbf{B}_2 = \{car_2, train_2\} \qquad \mathbf{B}_5 = \{soon_5, sooner_5\}$$
$$\mathbf{B}_3 = \{car_3, train_3, salad_3, peanuts_3, coke_3, juice_3, urgent_3\} \qquad \mathbf{B}_6 = \{fit_6, sick_6\}$$

Since $c_p = c_r$, we start with $\mathcal{B}' = \{B_1\}$. We have $F = v = \emptyset$.

Now we call OptimizeBlock$(B_1, 1)$. Since B_1 is acyclic, only the transitive reduction is applied. We get $B_1^- = (\{1, 2, 3, 4\}, \{(1, 2), (2, 3), (3, 4)\})$. The subroutine returns $E = \{(1, 3), (2, 4)\}$.

The child cut vertices of B_1 are $\mathcal{C}' = \{3, 4\}$; we update F to $\{(1, 3), (2, 4)\}$.

Next, we update the label of all edges (i, j) in B_1^-. But before this, let us enumerate the recursive import interfaces, starting from the import interface, for every node from 1 to 6:

$$\mathcal{V}(1) = \{train_2, train_3, peanuts_3\} \qquad \mathcal{V}(3) = \{train_4, sick_6\} \qquad \mathcal{V}(4) = \{sooner_5\}$$
$$\mathcal{V}(2) = \{car_3, coke_3, train_3, car_4, train_4\} \qquad \mathcal{V}(5) = \{train_4\} \qquad \mathcal{V}(6) = \emptyset$$

$$\mathcal{V}^*(1) = \{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4, sooner_5, sick_6\}$$
$$\mathcal{V}^*(2) = \{train_3, car_3, coke_3, train_4, car_4, sooner_5, sick_6\}$$
$$\mathcal{V}^*(3) = \{train_4, sooner_5, sick_6\}$$
$$\mathcal{V}^*(4) = \{train_4, sooner_5\}$$
$$\mathcal{V}^*(5) = \{train_4, sooner_5\}$$
$$\mathcal{V}^*(6) = \emptyset$$

Now, let us compute $\mathcal{V}^*(1, 2)_{B_1} = \mathcal{V}^*(1) \cap \bigcup_{\ell \in B_1|_2} \mathbf{B}_\ell$. We have that $B_1|2 = \{3, 4\}$, thus

$$\mathcal{V}^*(1, 2)_{B_1} = \mathcal{V}^*(1) \cap (\mathbf{B}_3 \cup \mathbf{B}_4) = \{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4\}$$

Similarly, with $B_1|_3 = B_1|_4 = \{4\}$, we have:

$$\mathcal{V}^*(2, 3) = \mathcal{V}^*(2) \cap \mathbf{B}_3 = \{car_4, train_4\}$$
$$\mathcal{V}^*(3, 4) = \mathcal{V}^*(3) \cap \mathbf{B}_4 = \{train_4\}$$

The removed edges and updated labels to be stored respectively in F and v for block B_1 can be summarized as:

$$\begin{aligned}
F &= \{(1,3),(2,4)\} \\
v(1,2) &= \mathcal{V}^*(1,2) \cup \mathcal{V}^*(1)|_{\mathbf{B}_3} \cup \mathcal{V}^*(1)|_{\mathbf{B}_4} = \left\{ \begin{matrix} train_2, train_3, peanuts_3, \\ car_3, coke_3, car_4, train_4 \end{matrix} \right\} \\
v(2,3) &= \mathcal{V}^*(2,3) \cup \mathcal{V}^*(1)|_{\mathbf{B}_3} \cup \mathcal{V}^*(1)|_{\mathbf{B}_4} = \{train_3, peanuts_3, car_3, coke_3, car_4, train_4\} \\
v(3,4) &= \mathcal{V}^*(3,4) \cup \mathcal{V}^*(1)|_{\mathbf{B}_3} \cup \mathcal{V}^*(1)|_{\mathbf{B}_4} = \{train_3, peanuts_3, car_3, coke_3, car_4, train_4\}
\end{aligned}$$

Next, we call OptimizeTree$(T \setminus B_1, 3, 1)$ and OptimizeTree$(T \setminus B_1, 4, 1)$, which eventually process blocks B_2 and B_3 in the same manner as above. The two calls respectively return:

$$\begin{aligned}
F' &= \{(5,4)\} & F'' &= \emptyset \\
v'(4,5) &= \{sooner_5\} & v''(3,6) &= \{train_4, sick_6\}
\end{aligned}$$

Combining all results together, OptimizeTree$(T, 1, 1)$ returns as the set of removed edges

$$F = \{(1,2),(3,4),(5,4)\}$$

and as updated labels v for the remaining edges in the blocks

$$\begin{aligned}
v(1,2) &= \{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4\} \\
v(2,3) &= \{train_3, peanuts_3, car_3, coke_3, car_4, train_4\} \\
v(3,4) &= \{train_3, peanuts_3, car_3, coke_3, car_4, train_4\} \\
v(4,5) &= \{sooner_5\} \\
v(3,6) &= \{train_4, sick_6\}
\end{aligned}$$

References

Adjiman, P., Chatalic, P., Goasdoué, F., Rousset, M.-C., & Simon, L. (2006). Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *J. Artif. Intell. Res.*, *25*, 269–314.

Aho, A. V., Garey, M. R., & Ullman, J. D. (1972). The Transitive Reduction of a Directed Graph. *SIAM J. Comput.*, *1*(2), 131–137.

Analyti, A., Antoniou, G., & Damásio, C. V. (2011). MWeb: A principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Log.*, *12*(2), 17.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2003). *The Description Logic Handbook*. Cambridge University Press.

Baget, J.-F., & Tognetti, Y. S. (2001). Backtracking through biconnected components of a constraint graph. In Nebel, B. (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pp. 291–296. Morgan Kaufmann.

Bairakdar, S. E.-D., Dao-Tran, M., Eiter, T., Fink, M., & Krennwallner, T. (2010a). Decomposition of distributed nonmonotonic multi-context systems. In Janhunen, T., & Niemelä, I. (Eds.), *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, Vol. 6341 of *Lecture Notes in Computer Science*, pp. 24–37. Springer.

Bairakdar, S. E.-D., Dao-Tran, M., Eiter, T., Fink, M., & Krennwallner, T. (2010b). The DMCS solver for distributed nonmonotonic multi-context systems. In Janhunen, T., & Niemelä, I. (Eds.), *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, Vol. 6341 of *Lecture Notes in Computer Science*, pp. 352–355. Springer.

Bessiere, C., Bouyakhf, E., Mechqrane, Y., & Wahbi, M. (2011). Agile asynchronous backtracking for distributed constraint satisfaction problems. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pp. 777–784.

Bikakis, A., & Antoniou, G. (2010). Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, *22*(11), 1492–1506.

Bikakis, A., Antoniou, G., & Hassapis, P. (2011). Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowl. Inf. Syst.*, *27*(1), 45–84.

Bögl, M., Eiter, T., Fink, M., & Schüller, P. (2010). The MCS-IE system for explaining inconsistency in multi-context systems. In *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, Vol. 6341 of *Lecture Notes in Computer Science*, pp. 356–359. Springer.

Bondy, A., & Murty, U. S. R. (2008). *Graph Theory*, Vol. 244 of *Graduate Texts in Mathematics*. Springer.

Brewka, G., Eiter, T., Fink, M., & Weinzierl, A. (2011). Managed multi-context systems. In Walsh, T. (Ed.), *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pp. 786–791. AAAI Press/IJCAI.

Brewka, G., Ellmauthaler, S., & Pührer, J. (2014). Multi-context systems for reactive reasoning in dynamic environments. In Ellmauthaler, S., & Pührer, J. (Eds.), *Proceedings of the International Workshop on Reactive Concepts in Knowledge Representation (ReactKnow) 2014*, pp. 23–30. Tech.Rep. 1, Computer Science Institute, Univ. Leipzig, ISSN 1430-3701.

Brewka, G., & Eiter, T. (2007). Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pp. 385–390. AAAI Press.

Brewka, G., Eiter, T., & Fink, M. (2011). Nonmonotonic Multi-Context Systems: A Flexible Approach for Integrating Heterogeneous Knowledge Sources. In Balduccini, M., & Son, T. C. (Eds.), *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, Vol. 6565 of *Lecture Notes in Computer Science*, pp. 233–258. Springer.

Brewka, G., Roelofsen, F., & Serafini, L. (2007). Contextual default reasoning. In Veloso, M. M. (Ed.), *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 268–273.

Buccafurri, F., & Caminiti, G. (2008). Logic programming with social features. *Theory and Practice of Logic Programming*, *8*(5-6), 643–690.

Dao-Tran, M. (2014). *Distributed Nonmonotonic Multi-Context Systems: Algorithms and Efficient Evaluation*. Ph.D. thesis, Faculty of Informatics, Vienna University of Technology, Austria.

Dao-Tran, M., Eiter, T., Fink, M., & Krennwallner, T. (2010). Distributed nonmonotonic multi-context systems. In Lin, F., Sattler, U., & Truszczynski, M. (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press.

Dao-Tran, M., Eiter, T., Fink, M., & Krennwallner, T. (2011). Model streaming for distributed multi-context systems. In Mileo, A., & Fink, M. (Eds.), *2nd International Workshop on Logic-based Interpretation of Context: Modeling and Applications*, Vol. 738 of *CEUR Workshop Proceedings*, pp. 11–22.

Eiter, T., Fink, M., Ianni, G., Krennwallner, T., & Schüller, P. (2011). Pushing efficient evaluation of hex programs by modular decomposition. In Delgrande, J. P., & Faber, W. (Eds.), *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, Vol. 6645 of *Lecture Notes in Computer Science*, pp. 93–106. Springer.

Eiter, T., Ianni, G., Schindlauer, R., & Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pp. 90–96.

Faltings, B., & Yokoo, M. (2005). Introduction: Special issue on distributed constraint satisfaction. *Artif. Intell.*, *161*(1-2), 1–5.

Fink, M., Ghionna, L., & Weinzierl, A. (2011). Relational information exchange and aggregation in multi-context systems. In Delgrande, J. P., & Faber, W. (Eds.), *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, 16-19 May, 2011*, Vol. 6645 of *Lecture Notes in Computer Science*, pp. 120–133. Springer.

Gao, J., Sun, J., & Zhang, Y. (2007). An improved concurrent search algorithm for distributed CSPs. In *Australian Conference on Artificial Intelligence*, pp. 181–190.

Gelfond, M., & Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, *9*(3/4), 365–386.

Ghidini, C., & Giunchiglia, F. (2001). Local models semantics, or contextual reasoning=locality+compatibility. *Artif. Intell.*, *127*(2), 221–259.

Giunchiglia, F. (1992). Contextual Reasoning. *Epistemologia, Special Issue on I Linguaggi e le Macchine*, *345*, 345–364.

Giunchiglia, F., & Serafini, L. (1994). Multilanguage hierarchical logics or: How we can do without modal logics. *Artif. Intell.*, *65*(1), 29–70.

Goncalves, R., Knorr, M., & Leite, J. (2014). Evolving multi-context systems. In Schaub, T., Friedrich, G., & O'Sullivan, B. (Eds.), *Proceedings of the 21st Eureopean Conference on Artificial Intelligence, ECAI'2014, Prague, Czech Republic, August 18-23, 2014*. IOS Press.

Hirayama, K., & Yokoo, M. (2005). The distributed breakout algorithms. *Artif. Intell.*, *161*(1–2), 89–115.

Homola, M. (2010). *Semantic Investigations in Distributed Ontologies*. Ph.D. thesis, Comenius University, Bratislava, Slovakia.

Lee, J., & Lifschitz, V. (2003). Loop formulas for disjunctive logic programs. In Palamidessi, C. (Ed.), *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, Lecture Notes in Computer Science, pp. 451–465. Springer.

McCarthy, J. (1993). Notes on formalizing context. In Bajcsy, R. (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pp. 555–562. Morgan Kaufmann.

Pontelli, E., Son, T., & Nguyen, N.-H. (2011). Combining answer set programming and prolog: The ASP-PROLOG system. In Balduccini, M., & Son, T. (Eds.), *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, Vol. 6565, pp. 452–472. Springer Berlin Heidelberg.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, *13*, 81–132.

Roelofsen, F., Serafini, L., & Cimatti, A. (2004). Many hands make light work: Localized satisfiability for multi-context systems. In de Mántaras, R. L., & Saitta, L. (Eds.), *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pp. 58–62. IOS Press.

Serafini, L., Borgida, A., & Tamilin, A. (2005). Aspects of distributed and modular ontology reasoning. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pp. 570–575. AAAI Press.

Serafini, L., & Tamilin, A. (2005). Drago: Distributed reasoning architecture for the semantic web. In Gómez-Pérez, A., & Euzenat, J. (Eds.), *The Semantic Web: Research and Applications, Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29 - June 1, 2005, Proceedings*, Lecture Notes in Computer Science, pp. 361–376. Springer.

Tarjan, R. E. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, *1*(2), 146–160.

Tasharrofi, S., & Ternovska, E. (2014). Generalized multi-context systems.. In Baral, C., Giacomo, G. D., & Eiter, T. (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press.

Valdes, J., Tarjan, R. E., & Lawler, E. L. (1982). The recognition of series parallel digraphs. *SIAM J. Comput.*, *11*(2), 298–313.

Vats, D., & Moura, J. M. F. (2010). Graphical models as block-tree graphs. *CoRR*, *abs/1007.0563*.

Velikova, M., Novák, P., Huijbrechts, B., Laarhuis, J., Hoeksma, J., & Michels, S. (2014). An Integrated Reconfigurable System for Maritime Situational Awareness. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pp. 1197–1202.

Yokoo, M., & Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, *3*(2), 185–207.