FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Distributed Nonmonotonic Multi-Context Systems: Algorithms and Efficient Evaluation

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Minh Dao-Tran
Matrikelnummer 0727429

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O. Univ. Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Dipl.-Ing. Dr. techn. Michael Fink

Diese Dissertation haben begutachtet:

(O. Univ. Prof. Dipl.-Ing.
Dr. techn. Thomas Eiter)

(Prof. Dr. Tran Cao Son)

Wien, 10.02.2014

(Minh Dao-Tran)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Distributed Nonmonotonic Multi-Context Systems: Algorithms and Efficient Evaluation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Minh Dao-Tran
Registration Number 0727429

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O. Univ. Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Dipl.-Ing. Dr. techn. Michael Fink

The dissertation has been reviewed by:

(O. Univ. Prof. Dipl.-Ing. Dr. techn. Thomas Eiter)

(Prof. Dr. Tran Cao Son)

Wien, 10.02.2014

(Minh Dao-Tran)

Erklärung zur Verfassung der Arbeit

Minh Dao-Tran
Donaufelderstrasse 91/2/233, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

From the bottom of my heart, I want to express my gratefulness to my supervisors: Thomas Eiter and Michael Fink. It is such a great honour to have Thomas as a supervisor for both of my Master and PhD theses, and Michael for the latter. Every meeting with them is an adventure. Under their sharp technical eyes and the ability to immediately hit the crucial points of the problems, I can learn lots of new things, from technical details to methodology; and especially, get encouragement to carry out the on-going work in better ways. I want to thank Thomas Krennwallner. We not only had fun working on the theoretical side, but also enjoyed implementing the DMCS system together. My technical skills, especially C++ and Emacs, were strongly affected by his style. As a colleague, a friend, Thomas is always open to help me and others on not only work but also everyday issues. I must say that I was very lucky to be in a team with brilliant people whom working with brought me an experience of life.

Major thanks to the Austrian Science Fund (FWF) for granting Project P20841 that funded my research.

I want to thank Katrin Seyr, who helped me with the biggest problem outside DMCS. Without her aid, I am not sure whether I can be here writing these final lines of my thesis. Katrin also taught me a lesson that one can overcome all obstacles or bad luck to live happily.

I want to thank colleagues/friends from KBS and DBAI (listed in alphabetic order of last names): Cristina Feier, Sarah Gaggl, Nysret Musliu, Magdalena Ortiz, Johannes Oetsch, Joerg Puehrer, Christoph Redl, Vadim Savenkov, Patrik Schneider, Peter Schueller, Mantas Simkus, Daria Stepanova, Antonius Weinzierl, Magdalena Widl, Stefan Woltran, Gouhui Xiao, who have been sharing the pressure of doing a PhD with me, and having fun of it in everyday procrastinating conversations.

I want to thank Eva Nedoma and Matthias Schloegel for all of their administrative and technical supports. Eva never says no to any of my request, from translating a notification in German to filling out any long application form. And when you ask Matthias for help with network issue or even a keyboard, he is always there.

Finally, I want to thank my parents for raising me, bringing me the opportunities to a better education and giving me a good background for my development. I want to thank my brother who is here with me and helps when I need. And thanks to my little family with my wife and my little son, you are the motivation for all what I have been doing, you are home where I want to be with after a long/stressful working day. You are my everything.

Abstract

Heterogeneous Nonmonotonic Multi-Context Systems (MCSs) are a generalization of a series of works on formalizing contexts in AI dating back in the 80s by John McCarthy. As such, they are a formalism for representing systems consisting of multiple (possibly nonmonotonic) knowledge-based systems (contexts). Knowledge between contexts is exchanged via bridge rules, a form of rules that allow to augment knowledge at a context depending on whether certain beliefs are accepted (not accepted) at certain contexts. Although virtually all formalizations of systems of multiple contexts are inherently targeted for distributedness, no truly distributed algorithms for evaluating their semantics exist, due to several obstacles: (i) the semantic abstraction of contexts to belief sets hinders interference with the local evaluation processes in contexts; (ii) information hiding and security aspects disable access to the context theories themselves, merely interfaces to the belief sets are provided; (iii) the complete system topology might be unknown to a context, which hinders decomposed evaluation; and (iv) the bridge rules of two contexts may refer to each other, thus creating cyclic systems that must be handled with care.

Coping with these challenges, we aim at designing and realizing truly distributed algorithms for evaluating MCSs. Deliberately approaching the issues, we started with a basic algorithm that mainly concentrates on the distributed aspects, i.e., dealing with transferring messages and breaking cycles. Then, we looked into possible optimizations at the meta level which preprocesses the global topology of the system to reduce information exchange. Taking one more step forward, we investigated gradually evaluating MCS where not all results are returned at once but in a streaming fashion. On an additional explorative branch, we designed an algorithm to configure dynamic MCSs into the original ones.

As the theoretical results of this thesis, we came up with notions to support evaluation of MCSs, such as contexts' import neighborhood, import closure, import interface, partial belief states and equilibria, loop formulas for MCSs, decomposition of MCSs, etc. Based on these notions, different algorithms to evaluate MCSs were proposed, namely DMCS, DMCSOPT, and DMCS-STREAMING. The first two correspond to the basic and topology-optimized evaluation. The last one introduces a new strategy in which both DMCS and DMCSOPT can be deployed to compute partial equilibria in a streaming fashion.

As the empirical results of the thesis, we have realized all proposed algorithms in a prototype implementation. Furthermore, we did a thorough experimental evaluation to compare the implemented algorithms on different aspects. And by analyzing the experimental results, we give a brief guideline on choosing the appropriate algorithm and running mode in a particular situation, determined by the parameter setting.

Beside the main results, a number of interesting future works were also revealed from experiences gained within the thesis project, including but not limited to conflicts learning for the algorithms, query answering in MCSs, and a potential for distributed heterogeneous stream reasoning.

Kurzfassung

Heterogene nichtmonotone Multi-Kontext Systeme (MCSs) sind eine Generalisierung einer Reihe von Arbeiten von John McCarthy über die Formalisierung von Kontexten in der Künstlichen Intelligenz, die bis in die 80er Jahre reichen. Sie sind ein Formalismus zur Repräsentierung von Systemen von mehreren (möglicherweise nichtmonotonen) wissensbasierten Systemen (Kontexte). Das Wissen zwischen Kontexten wird mittels Brückenregeln (bridge rules) ausgetauscht. Diese Form von Regeln erlaubt, Wissen eines Kontextes zu erweitern, je nachdem, ob gewisse Überzeugungen (nicht) bei gewissen Kontexten akzeptiert werden. Obwohl nahezu alle Formalisierungen von Systemen mit mehreren Kontexten grundsätzlich auf Verteilte Systeme abzielen, existieren keine echten verteilten Algorithmen zur Evaluierung ihrer Semantiken aufgrund mehrerer Hürden: (i) die semantische Abstraktion der Kontexte mit Hilfe von Überzeugungsmengen (belief sets) verhindert den Zugriff in den lokalen Evaluationsprozess von Kontexten; (ii) Informationsabgrenzung und Sicherheitsaspekte sperren the Zugriff auf die Kontext Theorien; (iii) die komplette Systemtopologie könnte einem partikulären Kontext unbekannt sein, was wiederum das aufgeteilte Evaluieren verhindert; und (iv) die Brückenregeln zweier Kontexte können aufeinander verweisen, was ein zyklisches System erzeugt, welches mit Vorsicht handzuhaben ist.

Um diese Herausforderungen zu meistern, zielen wir auf das Entwerfen und Realisieren von echten verteilten Algorithmen zur Evaluierung von MCSs. Durch bewusste Annäherung an diese Probleme, haben wir mit einem einfachem Algorithmus angefangen, der sich hauptsächlich auf verteilte Aspekte konzentriert, das heisst, dem Übertragen von Nachrichten und dem Brechen von Zyklen. Danach untersuchten wir mögliche Optimierungen auf der Meta-Ebene, welche die globale Topologie des Systems aufbereiten, um den Informationsaustausch zu reduzieren. Einen Schritt vorausgehend, erforschen wir die stufenweise Auswertung von MCS, bei denen nicht alle Ergebnisse auf einmal zurückgegeben werden, sondern auf kontinuierliche Weise. Auf einem weiteren explorativen Zweig entwickelten wir einen Algorithmus zur Konfiguration dynamischer MCSs auf ursprüngliche MCS.

Als theoretische Resultate dieser Dissertation entwickelten wir Konzepte, die die Evaluierung von MCSs unterstützen, wie etwa Import Nachbarschaft, Import Abschluss, Import Schnittstelle, partielle Überzeugungsmengen und Gleichgewichte, Schleifenformeln für MCSs, Zerlegung von MCSs, etc. Basierend auf diesen Konzepten, schlugen wir unterschiedliche Algorithmen zur Evaluierung von MCSs vor, nämlich DMCS, DMCSOPTsowie DMCS-STREAMING. Die ersten beiden korrespondieren zur einfachen und Topologie-optimierten Evaluierung. Der letzte Algorithmus stellt eine neue Strategie vor, in welcher sowohl DMCS als auch DMCSOPT eingesetzt werden kann zur Berechnung partieller Gleichgewichte auf kontinuierlicher Weise.

Die empirischen Resultate der Dissertation umfassen die Realisierung aller vorgeschlagenen Algorithmen in prototypischen Implementierungen. Weiters haben wir eine sorgfältige experimentelle Evaluierung durchgeführt, um die implementierten Algorithmen auf unterschiedlichen Aspekten zu vergleichen. Bei der Analyse der experimentellen Resultaten haben wir eine kurze Richtlinie entwickelt, um den geeigneten Algorithmus und Laufmodus in einer bestimmten Situation zu wählen, welcher durch eine Parameterkonfiguration bestimmt wird.

Neben den Hauptresultaten haben wir eine Anzahl interessanter zukünftiger Studien ausgearbeitet, die aus den Erfahrungen des Dissertationsprojekt gewonnen wurden, einschliesslich, aber nicht beschränkt auf Konfliktlernen für die Algorithmen, Anfragebeantwortung von MCSs, sowie Potentiale für verteilte heterogene Datenstromsschlussfolgerung.

Contents

Part I

Basic Notions

CHAPTER 1

Introduction

1.1 Motivation

In recent years, there has been an increasing interest in systems comprising multiple knowledge bases. This approach allows one to deploy a wide range sophisticated applications, including but not limited to data integration, multi-agent systems, argumentation, or project costs and time management as an example of real-life applications, where regulations such as constraints like working laws, holiday restrictions, etc. are kept in different knowledge bases like an ontology of personal costs, rules that compute the work amount for work packages, personal timekeeping, central administration data, local preferences, and so on.

The rise of distributed systems and the World Wide Web fostered this development, and to date, several formalisms are available to accommodate multiple, possibly distributed knowledge bases. Well-known formalisms are distributed SAT solving [62], distributed constraint satisfaction [47,96], distributed ontologies in different flavors [63], MWeb [4], and different approaches on Multi-Context Systems [19,21,56,58,78]; among these, *Heterogeneous Nonmonotonic Multi-Context System* (MCSs) [19] is of our special interest. As a generalization of previous proposals on Multi-Context Systems, MCSs brought in a powerful formalism in which one can specify systems whose contexts hold different knowledge representation and reasoning powers, ranging from simple, monotonic one such as querying to a database, to more sophisticated one such as ontology reasoning, or even with nonmonotonicity like disjunctive logic programs under the answer set semantics. On top of these distributed and heteregeneous knowledge bases, bridge rules are a uniform way to interlink the contexts, in a possibly *cyclic* manner. A bridge rule updates the local knowledge base at a context when certain beliefs are concluded to be true/false in other contexts; hence, influences a context to derive new beliefs based on remote information. The semantics of MCSs is given in terms of equilibria, which are intuitively states in which every context announces a "local model" that is conformant with those local models of other contexts, obeying the knowledge imported by bridge rules. The following simple example illustrates this idea.

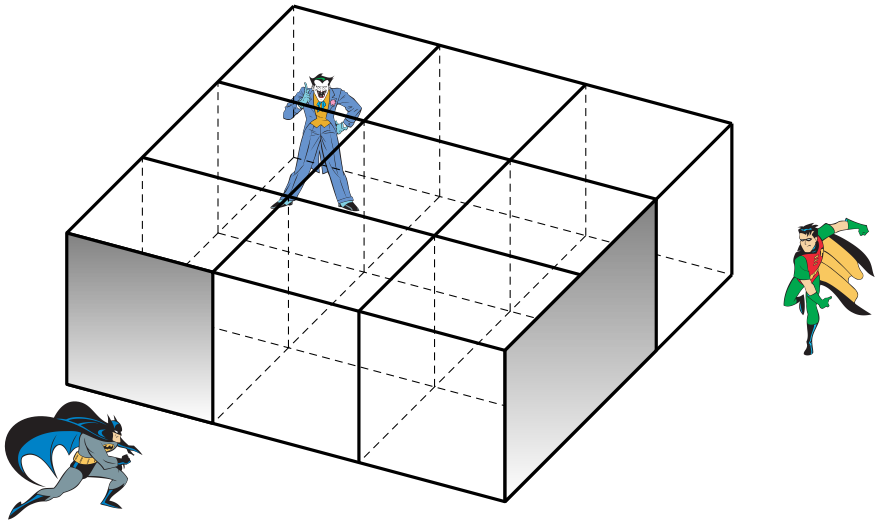


Figure 1.1: The magic box example

Example 1 (Inspired by the Magic-Box example [56]) Suppose that Batman and Robin were chasing Joker and finally reached an area that is partially visible to both. Furthermore, assume that Robin is currently wounded and cannot distinguish the distance to an object. Under these conditions and their positions as in Figure 1.1, neither Batman nor Robin can tell the exact position of Joker, as Batman only make sure that Joker is not in columns 2 or 3, while Robin can only tell that Joker is on row 1. However, if they exchange their above partial knowledge then Joker's position can be exactly located at row 1, column 1.

To formalize this scenario, each of Batman and Robin can be modelized as a context whose local knowledge bases contain their partial knowledges about Joker's position, which are then exchanged by bridge rules. A single equilibrium in this case gives the answer to Joker's position. For a concrete encoding, we refer the reader to Example 8 in Chapter 2.

Although virtually all formalisations of MCSs are inherently targeted for distributed systems, no truly distributed algorithms for MCSs exist. In [19], the authors gave an encoding to evaluate equilibria using dlvhex [43]. While this approach inherits the capability to deal with heterogeneity from dlvhex, it is totally centralized. In another attempt, [86] proposed an algorithm for checking satisfiability of homogeneous, monotonic MCS with a centralized control accesses contexts in parallel, thus is not truly distributed.

The lack of distributed algorithms for MCSs is due to several obstacles:

(1) to give semantics for MCSs, the local models at every context are abstracted to a uniform, general notion called *belief sets*. This view however hinders the interference of algorithms that evaluate the whole system at a global level with the local semantics and evaluation process at each context;

(2) with the purpose of gearing MCSs towards real life applications, e.g., to model information exchange between companies where certain levels of information hiding and security are required, the formalism fosters this feature by allowing only necessary part of the information, identified as interfaces, to be transferred between contexts. For the global algorithm, this prevents a context to get more insight about its neighbors for potential optimization, for example, to learn conflicts across contexts;

(3) the complete system topology might be unknown to a context, which disables decomposing the system for more efficient, modular evaluation;

(4) last but not least, the bridge rules might form a cyclic information interlinking through a group of contexts. In such cases, even though their local theories only require acyclic evaluation, the global evaluation must carefully handle the cycles.

In this thesis, we will address these challenges and develop solutions towards efficient evaluation of MCSs. Before going into the goals in detail, we give a brief view of state of the art in fields most related to the topic, namely Answer Set Programming (ASP) and other theoretical developments regarding Multi-Context Systems.

1.2 State of the Art

Reasoning with logic programming under the answer set semantics

The abstract model of MCSs allows one to locate any kind of reasoning power at a context. To realize this theoretical result in a practical implementation, the first decision one has to make is to choose a suitable formalism that on the one hand, guarantees a certain level of expressiveness to represent sophisticated scenarios, and on the other hand, has efficient implementations available. Our choice for this first step is ASP [55], a recently emerged tool for declarative knowledge representation and reasoning, because it fulfills the needs above.

ASP is a formalism that allows one to represent problems by logic programs (sets of finite logic rules), and it defines the semantics of the programs as sets of models in which all rules are satisfied; as such, each model is a solution to the original problem. The traditional ASP formalism does not allow function symbols in the programs to guarantee termination but it is still powerful because of supported expressive constructs such as negation as failure in rule bodies, disjuction in rule heads. These constructs, together with the possibility of having cyclic dependencies between predicates, are suitable for handling incomplete, inconsistent information, i.e., non-monotonicity, as well as for expressing non-determinism. From the complexity point of view, ASP can represent problems and reasoning tasks having complexity up to Σ_2^P.

For such expressibility, ASP is a suitable tool to serve as a host language for advanced reasoning tasks. ASP solvers can be used as underlying engines for processing such dedicated tasks; those were implemented can be listed as planning [37], diagnostic reasoning [36], computing updates of nonmonotonic knowledge bases represented as logic programs [40], or the semantics of inheritance programs [24]. The increasing interest in ASP is also documented by

the formation of the *Working Group on Answer Set Programming (WASP)*,[1] supported by the European Commission from 2002 to 2005.

Further advanced features have been added to modern ASP, for example, function symbols [10, 16, 27, 28, 44, 45, 92], paracoherent ASP [39], open ASP [60, 61], modularity [30, 66], ASP with external information [43], reactive, online ASP [51], etc. However, in this thesis, we stick to traditional ASP as there are efficient implementations available for these tasks. This is also the second reason for us to choose ASP to work with.

Indeed, one of the reasons contributed to the success of ASP in knowledge representation and reasoning is the availability of a number of effective ASP solvers which have been developing for more than 15 years. One can name the most successful engines such as clasp,[2] DLV,[3] and Smodels[4] which exploit model-building on the grounded programs with advanced search techniques such as conflict learning, unfounded sets to gain performance. Besides, there are alternative, new approaches for evaluate ASP programs such as using SAT solvers by translation via loop formulas [76], or grounding-on-the-fly ASP [34, 72, 83] that saves memory during run time. With this diversity of developing ASP, one can expect more and more efficient engines in the near future. This development process is also encouraged by the ASP competition which is co-organized biennial with LPNMR, a crucial conference for Logic Programming and Nonmonotonic Reasoning.

Multi-context systems

The problem of context has a long tradition in different areas of Artificial Intelligence (AI). But the issue of formalizing contexts has become widely discussed only in the late 80s, when J. McCarthy proposed the formalization of context as a crucial step toward the solution of the problem of generality [77], and was later elaborated in his note [78]. Intuitively, an axiom only holds in certain contexts and does not hold in more general ones; therefore, contexts are needed in representing/formalizing common sense knowledge.

Under McCarthy's supervision, Guha proposed in his PhD thesis [59] a first formalization of contexts a long the lines suggested in [78]. These works were the starting point of Propositional Logic of Context (PLC) [25] by Buvac and Manson, in which contexts are treated as first class objects (i.e., the logical language must contain terms for contexts, and the interpretation domain contains objects for contexts), and two main contextual reasoning mechanisms are those of entering and exiting a context. Following a different approach, Giunchiglia proposed to formalize contexts based on the problem of locality [57], which emphasizes more on formalizing contextual reasoning than on formalizing contexts as first class objects. Giunchiglia and Serafini then proposed Multi-Context Systems (MCSs) as a proof-theoretical framework for contextual reasoning [58], and Serafini and Bouquet in their comparison [89] presented that MCSs are more general than PLC and a more adequate formalizations of contexts.

The above proposals share a property that contexts are based on classical, monotonic reasoning, i.e., the acquisition of new information is based on the presence of other information

[1] `http://www.kr.tuwien.ac.at/research/projects/WASP/`
[2] `http://www.cs.uni-potsdam.de/clasp/`
[3] `http://www.dlvsystem.com/dlvsystem/index.php/Home`
[4] `http://www.tcs.hut.fi/Software/smodels/`

only. However, in many natural situations, new information is obtained due to a lack/absence of other information. This motivated to generalize MCSs with a non-monotonic reasoning capability. Roelofsen and Serafini made a first step a long this idea in [85] by adding default negation to a rule based MCS and thus combining contextual and default reasoning; however, this approach has a serious weakness regarding skeptical reasoning. Brewka et al. [21] later proposed a syntactical counterpart of the approach in [85], called Contextual Default Logic, as a contextual variant of Reiter's Default Logic [84]. This proposal, on the one hand, paves the way to remedy the weakness above, and on the other hand, is closer to standard ways of representing non-monotonic inference and closer to implementation.

Nevertheless, the formalizations in [21, 85] are homogeneous, in the sense that local theories at every context in a system are of the same type. In practical distributed applications, this assumption normally does not hold; for examples, when different companies join and share information via an MCS, they would prefer to keep their own internal structures and communicate via a uniform interface, rather than making significant changes internally. Inspired by this observation, a higher abstract level of MCSs was proposed in [19], namely a framework that allows both non-monotonicity and heterogeneity regarding local theories at the contexts. Intuitively, according to this generic framework:[5]

- contexts are constructed based on a notion of logic in a very broad sense: basically only sets and functions on sets are taken into account;
- the unit elements in the logics are abstracted in terms of beliefs, and models are abstracted to belief sets;
- interlinking of information between contexts is done via a uniform means called bridge rules, which support negation as failure;
- the semantics of MCSs is defined in terms of equilibria, which are intuitively stably interlinked local models.

As already mentioned in Section 1.1, due to the generality of MCSs, several obstacles are observed that challenge efforts to realize this theoretical results in terms of practical implementation. In this thesis, we take this challenge and give solutions to the problem. The detail goals and main results of the thesis are presented next.

1.3 Goals of the Thesis, Main Results, and Structure

The main goal of this thesis is to *develop effcient meta algorithms for evaluating Multi-Context Systems in a truly distributed way, and to realize the algorithms in terms of prototype implementation.*

In more concrete terms, the main aspects considered in the thesis are the following:

[5] For the details on the formalization, we refer the reader to Section 2.4.

Meta algorithms and optimization techniques for evaluating MCSs in a distributed manner. We first develop a basic algorithm for evaluating MCSs with emphasis on being truly distributed [31]. As a basic version, we deal with obstacles (1)-(4) in a generic way: contexts just exchange belief sets and the call history between each other; no further information is used. Belief sets are a uniform representation of local results obtained from the local solving processes at different nodes, which are based on different semantics. This allows us to uniformly deal with local results and concentrate on the global aspects of the algorithm. At this level, the semantics of the system is represented by belief states (which are sequences of belief sets, each corresponds to a local context) and we need to consistently combine those from neighboring contexts before starting the local solving process at any non-leaf context. Regarding the call history, it is used to detect cycles. When a context gets a request and sees its own identifier in the call history, this means a cycle was detected and the context is responsible for breaking the cycle by means of guessing. The checking part will be done later on the same context, along the returning path, by making use of the combination operator of belief states mentioned above.

By limiting what a context should know about the system and what is transferred between them, we obtain a truly distributed algorithm for evaluating MCSs but at the same time suffer some scalability issues. To enhance the performance in an optimized version of the algorithm, we reveal more meta-level information to contexts, namely the topological dependency of the system for decomposing it into a block tree and the interface between contexts for optimizing the data transfer between blocks [8]. The former breaks the cycles in advance, while the latter reduces a significant amount of duplicated local evaluation. Both techniques show remarkable improvement in performance compared to the basic version.

Nevertheless, as the first two algorithms aim at computing all equilibria of an MCS, they can not escape from scability issues as well as memory consumption when local contexts produce exponentially many local models, which is often the case in ASP. Therefore, computing models in a streaming way was investigated for a more pratical usage [33]. The idea here is not to return all local models from a context to its parent in one shot but rather to gradually return them in small portions. This way, a memory blowup can be avoided, and contexts can also process in parallel instead of inactively waiting for all answers from all neighbors. This approach is more user-friendly as one can observe the answers gradually with acceptable interval rather than waiting for long for all answers at once. More importantly, it meets certain pratical tasks that do not require to compute all answers, such as consistency checking.

Configuration of Dynamic MCS. We also look into an extended setting: dynamic MCS [32], where the linking between contexts is not fixed at design time and is only decided at run time, before evaluating the equilibria. For this setting, we first extend the MCS formalism to handle "dynamic" rules, and then formalize the notion of *context substitution* to bind dynamic MCSs to the original/static ones. The substitution uses quality matching from schematic beliefs to local beliefs, which is assessed by a similarity function. Based on these building blocks, we design a truly distributed algorithm to config dynamic MCSs, that is, computing the binding.

Prototype Implementation and Experimental Evaluation. As the practical aspect of the thesis, we realized the proposed algorithms in a prototype implementation [9]. To assess the effect

of the optimization techniques, we set up a benchmarking system and did thorough experiments with automatically generated data. The test results confirm our expectation of the optimization techniques in general (details can be found in Chapter 8):

- the decomposition technique absolutely improves the performance in non-streaming mode;
- the streaming mode is definitely worth pursuing: there are cases where the non-streaming mode times out while the streaming mode can still find some first answers;
- in streaming mode, it is very important to choose the package size of each returned portion of the whole answer;
- the system topology plays an important role as some optimization techniques show drastic improvement in some specific topologies.

However, there are also special/rare cases when some optimization pushes so hard that the performance is actually not as good as using no optimization. These interesting observations can be input for further investigation to enhance our system.

Thesis Organization. The remainder of this thesis is organized as follows. In Chapter 2, we give a formal introduction to fields related to the thesis, namely Answer Set Programming, Loop Formulas, and Multi-Context Systems. The main contributions, which are presented in Chapters 3 to 7, are devided into two main parts. In the first part, Chapters 3 to 6, we develop algorithms for evaluating MCSs, including a basic distributed algorithm, a topology-based optimization algorithm, and a streaming algorithm; and for configuring dynamic MCSs. In the second part, we present important aspects of the prototype implementation in Chapter 7 and a thorough experiments of the proposed algorithms in Chapter 8. Finally, the results of the thesis are summarized in Chapter 9, together with comparison to related works, and an outlook to interesting future works that come from experiences working on this thesis.

CHAPTER 2

Preliminaries

This Chapter provides a more technical view of the underlying machineries used in this thesis. Firstly, we introduce the basics and principles of Answer-Set Programming. Loop Formulas are then described as a bridge to connect ASP to SAT solving. Finally, we outline Multi-Context Systems, a framework for distributed heterogeneous reasoning, which is the main topic of the thesis.

2.1 Declarative Logic Programming

In computer science, programming languages can be categorized into two big programming concepts, namely *imperative programming* and *declarative programming*. On the one hand, an imperative program comprises of a sequence of commands for the computer to perform, hence focuses on *how* to solve a problem by an algorithm. Such imperative programming languages are Fortran, C, C++, Java,... On the other hand, a declarative program concentrates on representing *what* are the properties of the desired solution. Therefore, programmers using purely declarative programming usually do not need to know how the solver process their programs. Further classifications in declarative programming bring us functional programming, logic programming, and constraint programming with LISP, Haskell, and variants of Prolog as typical languages.

Compared to imperative programs, declarative programs are usually more concise and more powerful in terms of reasoning abilities. Any programer who has tried to solve the Hamiltonian Cycle problem in C++ or Java can easily verify that he/she must use a deep-first-search implemented in a recursive way, and it cannot be written in just 6 lines like our example in Section 1.2.

Hereafter, we will focus on Declarative Logic Programming. A programmer using this paradigm needs to specify in his/her logic program the relationships in the domain of discourse obeying the syntax of a language, and then gets the output through the semantics of the program.

There are logic programming languages such as Prolog which are not purely declarative. Evaluating such a program depends on the order of rules in the program and the order of atoms

in rules; therefore, makes it not very comprehensible and not easy to modify. A Prolog program does not guarantee termination. Moreover, Prolog provides extra-logical features to control the execution of the program, e.g., the cut rule "!" which does not have a logical meaning.

A different paradigm of logic programming introduced by Gelfond and Lifschitz is Answer-Set Programming [54], which is purely declarative and guarantees termination. We have introduced ASP in Section 1.2; next, we will breifly present its syntax and semantics in a more technical way. For a detail tutorial, we refer the reader to [42].

2.2 Logic Programs under the Answer-Set Semantics

Syntax of answer-set programs

Let $\mathcal{P}, \mathcal{C}, \mathcal{V}$ be disjoint sets of predicate, constant, and variable symbols from a first-order vocabulary Φ, respectively, where $\mathcal{V}$ is infinite and $\mathcal{P}$ and $\mathcal{C}$ are finite. Assume that elements from $\mathcal{C}$ and $\mathcal{P}$ are string constants that begin with a lowercase letter or double-quoted, and elements from $\mathcal{C}$ can also be integer numbers; elements from $\mathcal{V}$ begin with an uppercase letter. A *term* is either a constant or a variable. Given a predicate $p \in \mathcal{P}$, an *atom* is defined as $p(t_1, \ldots, t_k)$, where k is called the arity of p and each $t_1, \ldots, t_k$ is a term. Atoms of arity 0 are called *propositional atoms*.

A *classical literal* (or simply *literal*) l is an atom a or a negated atom $\neg a$, where "$\neg$" is the symbol for true (classical) negation. Its *complementary* literal is $\neg a$ (resp., a). A *negation as failure literal* (or *NAF-literal*) is a literal l or a default-negated literal $\mathrm{not}\, l$. It evaluates to *true* if l cannot be proved, i.e., either l is false or we do not know whether l is true or false.

A *rule r* is an expression of the form

$$a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \mathrm{not}\ b_{m+1}, \ldots, \mathrm{not}\ b_n\,, \qquad k \geq 0\,, m \geq n \geq 0\,, \tag{2.1}$$

where $a_1, \ldots, a_k, b_1, \ldots, b_n$ are classical literals. We say that $a_1, \ldots, a_k$ is the *head* of r while the conjunction $b_1, \ldots, b_m, \mathrm{not}\ b_{m+1}, \ldots, \mathrm{not}\ b_n$ is the *body* of r. We use $H(r)$ to denote r's head literals, and $B(r)$ to denote the set of all its body literals $B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ are called the positive and negative body, respectively. A rule r without head literals (i.e., $k = 0$) is an *integrity constraint*. A rule r with exactly one head literal (i.e., $k = 1$) is a *normal rule*. If the body of r is empty (i.e., $m = n = 0$), then r is a *fact*, and we often omit "$\leftarrow$".[1] An *extended disjunctive logic program* (EDLP, or simply *program*) P is a finite set of rules r of the form (2.1).

Programs without disjunction in the heads of rules are called *extended logic programs* (ELPs). A program P without NAF, i.e., for all $r \in P, B^-(r) = \emptyset$ is called a *positive logic program*. If, additionally, no strong negation occurs in P, i.e., the only form of negation is default negation in rule bodies, then P is called *normal logic program* (NLP). The generalization of an NLP by allowing default negation in the heads of rules is called *generalized logic program* (GLP). Additional program classes of logic programming with the corresponding restrictions on the rules in a program a summarized in Table 2.1.

[1] In this thesis, we will use both forms "$a \leftarrow$" and "a." to denote that a is a fact in a logic program.

Name	restriction
definite Horn	$k = 1, n = m$
Horn	$k \leq 1, n = m$
normal	$k \leq 1$
definite	$k \geq 1, n = m$
positive	$n = m$
disjunctive	no restriction

Table 2.1: Program classes

Example 2 The following set of rules comprises a logic program:

$$P = \left\{ \begin{array}{rcl} \mathit{flies}(X) & \leftarrow & \mathit{bird}(X), \text{not } \neg \mathit{normal}(X). \\ \mathit{bird}(X) & \leftarrow & \mathit{penguin}(X). \\ \neg \mathit{normal}(X) & \leftarrow & \mathit{penguin}(X). \\ & & \mathit{penguin}(\mathit{tweety}). \\ & & \mathit{bird}(\mathit{joe}). \end{array} \right\}$$

In this program, the first rule is a normal rule encoding a default inference "*birds normally fly*." The next two are positive rules saying that penguins are birds and penguins are abnormal, respectively. Finally, there are two facts about *tweety* and *joe*; the former is a penguin while the latter is known to be a bird.

Intuitively, one should be able to conclude from this program that *joe* flies, *tweety* is a bird but it does not fly. This is accomplished by the semantics of Answer-Set programs presented next.

Semantics of answer-set programs

The semantics of extended disjunctive logic programs is defined via variable-free programs. Hence, we first define the *ground instantiation* of a program.

The *Herbrand universe* of a program P, denoted HU_P, is the set of all constant symbols $C \subseteq \mathcal{C}$ appearing in P. If there is no such constant symbol, then $HU_P = \{c\}$, where c is an arbitrary constant symbol from Φ. Terms, atoms, literals, rules, programs, etc. are *ground* iff they do not contain any variables. The *Herbrand base* of a program P, denoted HB_P, is the set of all ground literals that can be constructed from the predicate symbols appearing in P and the constant symbols in HU_P. A *ground instance* of a rule $r \in P$ is obtained from r by replacing every variable that occurs in r by a constant symbol in HU_P. We use $ground(P)$ to denote the set of all ground instances of rules in P.

Example 3 Take the program P from Example 2, its ground version is

$$ground(P) = \left\{ \begin{array}{rcl} \mathit{flies}(tweety) & \leftarrow & bird(tweety), \text{not } \neg normal(tweety). \\ bird(tweety) & \leftarrow & penguin(tweety). \\ \neg normal(tweety) & \leftarrow & penguin(tweety). \\ \mathit{flies}(joe) & \leftarrow & bird(joe), \text{not } \neg normal(joe). \\ bird(joe) & \leftarrow & penguin(joe). \\ \neg normal(joe) & \leftarrow & penguin(joe). \\ & & penguin(tweety). \\ & & bird(joe). \end{array} \right\}$$

The semantics for EDLPs is defined first for positive ground programs. A set of literals $X \supseteq HB_P$ is *consistent* iff $\{p, \neg p\} \subsetneq X$ for every atom $p \in HB_P$. An *interpretation I* relative to a program P is a consistent subset of HB_P. We say that a set of literals S *satisfies* a rule r if $H(r) \cap S \neq \emptyset$ whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$. A *model* of a positive program P is an interpretation $I \subseteq HB_P$ such that I satisfies all rules in P. An *answer set* of a positive program P is the least model of P w.r.t. set inclusion.

To extend this definition to programs with negation as failure, we recall the *Gelfond-Lifschitz transform* (also often called the *Gelfond-Lifschitz reduct*) from a program P relative to an interpretation $I \subseteq HB_P$, denoted P^I, as the ground positive program obtained from $ground(P)$ by

(i) deleting every rule r such that $B^-(r) \cap I \neq \emptyset$, and

(ii) deleting the negative body from every remaining rule.

An *answer set* of a program P is an interpretation $I \subseteq HB_P$ such that I is an answer set of P^I.

Example 4 Consider $ground(P)$ from Example 3 and an interpretation $I = \{penguin(tweety), bird(tweety), \neg normal(tweety), bird(joe), \mathit{flies}(joe)\}$. The reduct of P wrt. I is

$$P^I = \left\{ \begin{array}{rcl} bird(tweety) & \leftarrow & penguin(tweety). \\ \neg normal(tweety) & \leftarrow & penguin(tweety). \\ \mathit{flies}(joe) & \leftarrow & bird(joe). \\ bird(joe) & \leftarrow & penguin(joe). \\ \neg normal(joe) & \leftarrow & penguin(joe). \\ & & penguin(tweety). \\ & & bird(joe). \end{array} \right\}$$

One can check that I satisfies all rules of P^I and is minimal wrt set inclusion. Therefore, I is an answer set of P.

A constraint is used to eliminate "unwanted" models from the result, since its head is implicitly assumed to be *false*. A model that satisfies the body of a constraint is hence dismissed from the set of answer sets.

The main reasoning tasks associated with EDLPs under the answer-set semantics are the following:

- decide whether a given program P has an answer set (*consistency checking*);
- given a program P and a ground formula ϕ, decide whether ϕ holds in every (resp., some) answer set of P (*cautious* (resp., *brave*) *reasoning*);
- given a program P and an interpretation $I \subseteq HB_P$, decide whether I is an answer set of P (*answer-set checking*); and
- compute the set of all answer sets of a given program P.

Answer-set solvers

With a lot of efforts put into implementing Answer-Set Solvers during the last two decades, there have been successful implementations that can solve problems at sizes comparable to practical needs. We list in the following some well-known solvers:

- ASPeRix[2] escapes from the preliminary phase of rule instantiation by integrating it in the search process [72] (called grounding on the fly) while other solvers choose the pre-grounding approach, i.e., ground the program first and then compute answer sets of the grounded instance. An important benefit of this technique is to avoid the bottleneck of instantiation phase arising from some problems because of the huge amount of memory needed to ground all rules of a program, even if these rules are not really useful in certain cases. The solver [73] is at a very preliminary state of development but experimental results already show good performances for definite, stratified and almost stratified programs.
- ASSAT[3] uses a technique called *Loop Formulas* [76] (described in details in Section 2.3) to translate an answer set program into a propositional clausal theory (SAT instance), and then it just needs to feed the theory to a SAT solver for computing answer sets. Since the SAT community has been developing many successful solvers, some are deployed in practical applications, this approach is definitely worth it as the solver can exploit all the best results from the SAT solving area with the effort of providing the translation.
- clasp[4] is a part of the Potassco project,[5] which contains bundles tools for Answer Set Programming developed at the University of Potsdam. clasp combines the high-level modeling capacities of ASP with state-of-the-art techniques from the area of Boolean constraint solving. The primary clasp algorithm relies on conflict-driven nogood learning [52], a technique that proved very successful for SAT. As the outcome, clasp has been the most powerful ASP solver; it won the two latest ASP competitions in 2009 and 2011.
- DLV[6] is a state-of-the-art answer set solver which has been developed at the University of Calabria and the Vienna University of Technology for over more than a decade.

[2] `http://www.info.univ-angers.fr/pub/claire/asperix/`
[3] `http://assat.cs.ust.hk/`
[4] `http://www.cs.uni-potsdam.de/clasp/`
[5] `http://potassco.sourceforge.net/`
[6] `http://www.dlvsystem.com/dlvsystem/index.php/Home/`

The system [74] has a richer language than extended disjunctive logic programs, and supports additional constructs (e.g., aggregates, weak constraints) some of which increase the expressivity. DLV supports certain built-in predicates (e.g. bounded integer arithmetic and comparisons), and offers a range of front-ends for specific KR tasks (e.g., planning or diagnosis), as well an interface to databases. The engine has been extended in many directions leading to a family of systems that support different purposes, including dlv-ex, dlvhex, OntoDLV, dlv-db, and dlt.

- Smodels[7] allows for the computation of answer sets for normal logic programs. It has GNT [65][8] as an extended prototype version for the evaluation of disjunctive logic programs. Smodels is another extension to pure answer-set programming allowing to minimize/maximize over sets of predicates. During model computation Smodels does not compute only optimal answer sets, but first evaluates an arbitrary model and then incrementally only returns "better" answer sets, such that the last answer set found by Smodels is the optimal one. Similar to DLV, Smodels allows for a restricted form of integer arithmetics and lexicographic comparison predicates.

2.3 Loop Formulas

Logic programming with answer-set semantics and propositional logic are closely related. It has been shown in [80, 97] that there is a local and modular translation from clauses to logic program rules such that the models of a set of clauses and the answer sets of its corresponding logic program are in one-to-one correspondence.

The other direction (from logic program rules to clauses) is however more difficult and interesting. When such a translation in this direction is available, one can exploit all the state-of-the-art SAT solvers to do the hard work of computing the models of the clauses, and then convert the models to the answer sets of the corresponding logic program. There have been several proposals regarding this approach such as [11] for normal logic programs which uses a quadratic number of propositional variables, or [64] for the class of 2-literal programs which does not require any extra variable.

In [76], the authors proposed a translation for normal logic programs which does not need extra variables but might introduce in the worst case exponentially many clauses. It is motivated by the relationship between the answer-set semantics and the completion semantics. Basically, every answer set of a logic program is also a model of the completion of that program, but the other way only holds for ''tight" programs, i.e., those without positive cycles between atoms. To make it possible for the general case, one must treat positive cycles carefully so that no unfounded model is allowed.

We start presenting this approach by recalling the notion of completion. In the following, given a logic program P, $atom(P)$ denotes the set of atoms appearing in P. For a set of atoms A, let $\neg.A = \{\neg a \mid a \in A\}$ and, as usual, $\bigvee F = \bigvee_{f \in F} f$ and $\bigwedge F = \bigwedge_{f \in F} f$ (note that $\bigvee \emptyset = \bot$ and $\bigwedge \emptyset = \top$.

[7] http://www.tcs.hut.fi/Software/smodels/

[8] http://www.tcs.hut.fi/Software/gnt/

Definition 1 ([76]) *Given a normal logic program P, i.e., a set of rules of the form* (2.1) *where $k \leq 1$, its* completion*, written $Comp(P)$, is the union of the constraints in P and the Clark completion [29] of the set of rules in P, i.e., the following sentences:*

- *for each $p \in atom(P)$, let $r_1, \ldots, r_\ell$ be all the rules whose heads are $\{p\}$, then*

$$p \equiv \bigwedge B^+(r_1) \wedge \bigwedge \neg.B^-(r_1) \wedge \ldots \wedge \bigwedge B^+(r_\ell) \wedge \bigwedge \neg.B^-(r_\ell)$$

  is in $Comp(P)$. If $\ell = 0$, then the equivalence is $p \equiv \bot$, which is equivalent to $\neg p$.

- *if $r = \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$ is a constraint in P then $\bigvee \neg.B^+(r) \vee \bigvee B^-(r)$ is in $Comp(P)$.*

Example 5 Consider the following simple program:

$$P = \left\{ \begin{array}{lll} a \leftarrow b. & b \leftarrow a. & a \leftarrow \text{not } c. \\ c \leftarrow d. & d \leftarrow c. & c \leftarrow \text{not } a. \end{array} \right\}$$

Then $Comp(P) = \{a \equiv (b \vee \neg c), c \equiv (\neg a \vee d), b \equiv a, d \equiv c\}$. According to the answer set semantics, P has two models $\{a, b\}$ and $\{c, d\}$. However, $Comp(P)$ has three models, namely the two above and additionally $\{a, b, c, d\}$.

Notice that the last model in $Comp(P)$ is unfoundedly deduced since a, b (and also c, d) depend on each other in a cyclic way. To eliminate such unfounded models, we need the notions of positive cycles and loop formulas.

Given a logic program P, the *positive dependency graph* of P is denoted by $G_P = (V, E)$, where $V = atom(P)$ and $(p, q) \in E$ if there is a rule r such that $H(r) = \{p\}$ and $q \in B^+(r)$. Informally, an edge from p to q means that p positively depends on q. From this dependency, loops are defined as follows.

Definition 2 ([76]) *Given a finite normal logic program P that may contain constraints, a non-empty subset L of $atom(P)$ is called a* loop *of P, if for any p and q in L, there is a path of length > 0 from p to q in the positive dependency graph of P, G_P, such that all the vertices in the path are in L.*

This means that if L is non-empty, and not a singleton, then it is a loop iff the subgraph of G induced by L is *strongly connected*, i.e., in this subgraph, every vertex is reachable from every other vertex. If L is a singleton, say $\{p\}$, then it is a loop iff there is an edge from p to itself in G_P.

The program P in Example 5 has two loops, namely $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$.

Given a logic program P and a loop L in P, the following two sets of rules are associated with them:

$$\begin{array}{rcl} R^+(L, P) & = & \{r \mid r \in P, H(r) \cap L \neq \emptyset, B^+(r) \cap L \neq \emptyset\}, \\ R^-(L, P) & = & \{r \mid r \in P, H(r) \cap L \neq \emptyset, B^+(r) \cap L = \emptyset\}. \end{array}$$

In the following, when the program P is clear from the context, we will write $R^+(L, P)$ as $R^+(L)$, and $R^-(L, P)$ as $R^-(L)$.

Intuitively, $R^+(L)$ contains rules *in the loop*, and they give rises to edges connecting vertices in L in the positive dependency graph G_P; on the other hand, $R^-(L)$ contains those rules about atoms in L that are *out of the loop*. For instance, for the program P in Example 5 and its two loops L_1, L_2, we have that:

$$R^+(L_1) = \{a \leftarrow b.\ \ b \leftarrow a.\} \quad R^-(L_1) = \{a \leftarrow \text{not } c.\}$$
$$R^+(L_2) = \{c \leftarrow d.\ \ d \leftarrow c.\} \quad R^-(L_1) = \{c \leftarrow \text{not } a.\}$$

For a loop L in a logic program P, one can observe that $\emptyset$ is the only answer set of $R^+(L)$. Therefore, an atom in the loop cannot be in any answer set unless it is derived using some other rules, i.e., those from $R^-(L)$. This motivates the definition of *loop formulas*.

Definition 3 *Let P be a logic program and L a loop in it. Then, the* loop formula *associated with L (under P), denoted by $miLF(L, P)$, or simply $LF(L)$ when P is clear from the context, is the following implication:*

$$\left(\bigvee L\right) \supset \left(\bigvee_{r \in R^-(L)} B(r)\right)$$

Example 6 Consider again the program P in Example 5. With two loops L_1 and L_2 as described above, we have that $LF(L_1) = (a \vee b) \supset \neg c$ and $LF(L_2) = (c \vee d) \supset \neg a$. Adding these two loop formulas to $Comp(P)$ will eliminate the model $\{a, b, c, d\}$, and the remaining models, namely $\{a, b\}$ and $\{c, d\}$ are exactly the answer sets of P.

The following theorem shows the correctness of this translation.

Theorem 1 *[76] Let P be a logic program, $Comp(P)$ its completion, and LF the set of loop formulas associated with the loops of P. We have that for any set of atoms, it is an answer set of P iff it is a model of $Comp(P) \cup LF$.*

We have presented the basic loop formulas for normal logic programs. This approach has gained interest from the KR community and there have been following works covering further aspects, namely:

- a generalization of Theorem 1 to cover disjunctive programs, programs with nested expressions in [48],
- exploration of a model-theoretic counterpart of loop formulas [67],
- explanation of the blow up of number of loop formulas [75],
- development of loop formulas for circumscription [69], disjunctive logic programs [68], for non-ground programs [70], and for first-order stable semantics [71].

2.4 Multi-Context Systems

Formalization of multi-context systems

This Section summarizes the formalization of Heterogeneous Nonmonotonic Multi-Context Systems (MCSs) proposed in [19], which serves as the base of this thesis. 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 4 ([19]) *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*. We assume that each element of* $\mathbf{KB}_L$ *is a set.*
2. $\mathbf{BS}_L$ *is the set of possible belief sets,*
3. $\mathbf{ACC}_L : \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ *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 is generic as it captures well-known logics (defined over a signature Σ) such as:

- Default logic [84]:
  - $\mathbf{KB}$: the set of default theories based on Σ,
  - $\mathbf{BS}$: the set of deductively closed set of Σ-formulas,
  - $\mathbf{ACC}(kb)$: the set of kb's extensions.
- Normal logic programs under answer set semantics [55]:
  - $\mathbf{KB}$: the set of normal logic programs over Σ,
  - $\mathbf{BS}$: the set of sets of atoms over Σ,
  - $\mathbf{ACC}(kb)$: the set of kb's answer sets.
- Propositional logic under the closed world assumption [18]:
  - $\mathbf{KB}$: the set of propositional formulas over Σ,
  - $\mathbf{BS}$: the set of deductively closed set of propositional Σ-formulas, (i.e., $Cn(S) = S$),
  - $\mathbf{ACC}(kb)$: the (singleton set containing the) set of kb's consequences under closed world assumption, i.e., $\mathbf{ACC}(\varphi) = Cn(CWA(\varphi))$, where $CWA(\varphi) = \{\varphi\} \cup \{\neg p \mid p \in \Sigma \wedge \varphi \not\models p\}$.

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

Definition 5 ([19]) *Let $L = \{L_1, \ldots, L_n\}$ be a set of logics. An L_k-bridge rule over L, $1 \leq k \leq n$, is of the form*

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \text{not}\,(c_{j+1} : p_{j+1}), \ldots, \text{not}\,(c_m : p_m) \tag{2.2}$$

where $1 \leq r_k \leq n$, p_k is an element of some belief set of L_{r_k}, and for each $kb \in \mathbf{KB}_k$, it holds that $kb \cup \{s\} \in \mathbf{KB}_k$.

Here, $head(r)$ is used to denote the head of a bridge rule r. Bridge rules refer in their bodies to other contexts and can thus add information to a context based on what is believed or disbelieved in other contexts. In contrast to Giunchiglia's (monotonic) multi-context systems [57], there is no single, global set of bridge rules in MCSs. Instead, each context knows only its own bridge rules. This can be emphasized by adding the local context identifier to the heads of the bridge rules when necessary. Now that the means for connecting contexts is available, MCSs can be formally defined.

Definition 6 ([19]) *A* multi-context system $M = (C_1, \ldots, C_n)$ *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 is a knowledge base (an element of $\mathbf{KB}_i$), and br_i is a set of L_i-bridge rules over $\{L_1, \ldots, L_n\}$.*

Example 7 ([20]) As a simple example, consider $M = (C_1, C_2)$, where the contexts are different views of a paper by its co-authors A_1 and A_2 who reason in different logics. In C_1, we have Classical Logic as L_1, and

- the knowledge base $kb_1 = \{unhappy \supset revision\}$,
- the bridge rules $br_1 = \{unhappy \leftarrow (2 : work)\}$.

Intuitively, if A_1 is unhappy with the paper, then she wants a revision, and if A_2 finds that the paper needs more work, then A_1 feels unhappy.

In C_2, we have Answer Set Programming as L_2, and

- the knowledge base $kb_2 = \{accepted \leftarrow good, \text{not}\ \neg accepted\}$,
- the bridge rules $br_2 = \{work \leftarrow (1 : revision); \quad good \leftarrow \text{not}\,(1 : unhappy)\}$.

Intuitively, A_2 thinks that the paper, if good, is usually accepted; moreover, she infers that more work is needed if A_1 wants a revision, and the paper is good if there is no evidence that A_1 is unhappy.

Example 8 The scenario from Example 1 can be formalized by a multi-context system $M = (C_1, C_2)$, where we have in both contexts L_1, L_2 as Answer Set Programming, and:

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

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

where

- $R = \left\{ \begin{array}{rcl} joker\_in & \leftarrow & at\_row(X). \\ joker\_in & \leftarrow & at\_col(X). \\ at\_row(X) & \leftarrow & joker\_in, row(X), \text{not } \neg at\_row(X). \\ \neg at\_row(X) & \leftarrow & joker\_in, row(X), at\_row(Y), X \neq Y. \\ at\_col(X) & \leftarrow & joker\_in, col(X), \text{not } \neg at\_col(X). \\ \neg at\_col(X) & \leftarrow & joker\_in, col(X), at\_col(Y), X \neq Y. \end{array} \right\}$
- $F = \{row(1).\ row(2).\ row(3).\ col(1).\ col(2).\ col(3).\}$
- $F_1 = \{\neg see\_col(2).\ \neg see\_col(3).\}$
- $F_2 = \{see\_row(1).\}$

Intuitively, C_1 formalizes Batman's knowledge about the magic box and C_2 formalizes that of Robin, respectively. The facts in F represent the magic box of size 3×3. F_1 and F_2 show what Batman and Robin see, respectively: Batman sees that Joker is neither in columns 2 nor 3, while Robin sees that Joker is on row 1. The rules in R make sure that there is only one fact of each predicate at_row and at_col holds in each local model of each context when the fact $joker\_in$ is set to true, i.e., Joker's position is uniquely determined in both Batman and Robin's mind. The rest of local rules in kb_1 and kb_2 determine whether Joker is in and if yes, his position based on what can be seen locally by Batman, Robin, and from the other. The latter is communicated by bridge rules in br_1, br_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 (5) is applicable in S, if $p_i \in S_{c_i}$, for $1 \leq i \leq j$, and $p_k \in S_{c_k}$, for $j+1 \leq k \leq m$. Denote by by $app(R, S)$ the set of bridge rules $r \in R$ that are applicable in S. Then,

Definition 7 ([20]) *A belief state $S = (S_1, \ldots, S_n)$ of a multi-context system M is an* equilibrium *iff $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, $\quad 1 \leq i \leq n$.*

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 9 ([20]) Reconsidering $M = (C_1, C_2)$ from Example 7, we find that M has two equilibria, namely:

- $E_1 = (Cn(\{unhappy, revision\}), \{work\})$, and
- $E_2 = (Cn(\{unhappy \supset revision\}), \{good, accepted\})$,

where $Cn(\cdot)$ is the set of all classical consequences.

As for E_1, the bridge rule of C_1 is applicable in E_1, and $Cn(\{unhappy, revision\})$ is the (single) acceptable belief set of $kb_1 \cup \{unhappy\}$; the first bridge rule of C_2 is applicable in E_1, but not the second; clearly, $\{work\}$ is the single answer set of $kb_2 \cup \{work\}$.

As for E_2, the bridge rule of C_1 is not applicable in E_2, and $Cn(\{unhappy \supset revision\}) = Cn(kb_1)$; now the second bridge rule of C_2 is applicable but not the first, and $\{good, accepted\}$ is the single answer set of $kb_2 \cup \{good\}$.

Example 10 Consider the multi-context system M from Example 8. This MCS has a 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 ideed 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, without communication, cannot accomplish this task.

Example 11 Let $M = (C_1, C_2, C_3, C_4)$ be an MCS such that all L_i are ASP logics, with alphabets $\mathcal{A}_1 = \{a\}$, $\mathcal{A}_2 = \{b\}$, $\mathcal{A}_3 = \{c, d, e\}$, $\mathcal{A}_4 = \{f, g\}$. 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.

Similar to answer sets of modular logic programs [41], equilibria suffer from groundedness problems due to cyclic justifications: the bridge rules might be applied unfoundedly. In Example 7, the equilibrium E_1 can be concluded because of the cycle

$$(1 : unhappy) \rightarrow (1 : revision) \rightarrow (2 : work) \rightarrow (1 : unhappy),$$

which is the only cyclic justification for $unhappy$ in E_1. More specifically, if any of the formula in the above cycle is evaluated to $true$ then all other formulas in this cycle are also evaluated to $true$, hence one has E_1 as an equilibrium of the MCS in Example 7.

To overcome this, [19] proposed *grounded equilibria*. The latter are defined in terms of a GL-reduct which transforms $M = (C_1, \ldots, C_n)$, given a belief state $S = (S_1, \ldots, S_n)$, into another MCS $M^S = (C_1^S, \ldots, C_n^S)$ that behaves nomotonically, such that a unique minimal equilibrium exists; if it coincides with S, we have groundedness.

Formally, $C_i^S = (L_i, red_i(kb_i, S), br_i^S)$, where $red_i(kb_i, S)$ maps kb_i and S to a monotonic core of L_i and br_i^S is the GL-reduct of br_i wrt S, i.e., it contains $s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j)$ for each rule of form (2.2) in br_i such that $p_k \notin S_{r_k}$ for $k = j + 1, \ldots, m$. In addition, the following reducibility conditions are assumed:

(i) $red_i(kb_i, S_i)$ is antimonotonic in S_i,

(ii) S_i is acceptable for kb_i iff $\mathbf{ACC}_i(red_i(kb_i, S_i)) = \{S_i\}$, and

(iii) $red_i(kb_i, S) \cup H_i = red_i(kb_i \cup H, S)$, for each $H_i \subseteq \{head(r) \mid r \in br_i\}$

Grounded equilibria are then defined as follows.

Definition 8 ([19]) *A belief state $S = (S_1, \ldots, S_n)$ is a* grounded equilibrium *of M iff S is the unique minimal equilibrium of M^S, where minimality is componentwise wrt $\subseteq$.*

Example 12 ([41]) Consider again Example 9: naturally, $red_1(kb_1, S)$ is identity and $red_2(kb_2, S)$ is the GL-reduct. Then, E_1 is not a grounded equilibrium of M because M^{E_1} has a single minimal equilibrium $(Cn(unhappy \supset revision), \emptyset) \neq E_1$. On the other hand, one can check that E_2 is indeed a grounded equilibrium of M.

Another semantics of MCS, the well-founded semantics, was also introduced and the details can be found in [19]. In this thesis, the general equilibria semantics is of our main concern and the main purpose is to design and implement efficient algorithms to compute equilibria of an MCS.

Centralized evaluation of multi-context systems

The computation of equilibria for a given MCS has been realized by a declarative implementation using HEX-programs [43] which can be evaluated using the dlvhex system.[9] HEX-programs extend disjunctive logic programs with access to external information by means of so-called external atoms.

Focusing on ground (variable-free) HEX-programs, we say that an *ordinary atom* is a predicate $p(c_1, \ldots, c_n)$ where p, and $c_1, \ldots, c_n$ are constants. An external atom is of the form

$$\&g[\boldsymbol{v}](\boldsymbol{w})$$

where $\boldsymbol{v}$, and $\boldsymbol{w}$ are fixed length lists of constants, and $\&g$ is an external predicate name. Intuitively, an external atom provides a way for deciding the truth value of $g(\boldsymbol{w})$ in an external source which is accessed providing the extension of predicates $\boldsymbol{v}$ as input.

A HEX rule r is of the form

$$\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \text{not } \beta_{m+1}, \ldots, \text{not } \beta_n \tag{2.3}$$

$m, k \geq 0$, where all α_i are ordinary atoms and all β_j are ordinary or external atoms. As usual, a rule r is a constraint, if $k = 0$. Furthermore, a HEX-program (or program) is a finite set of HEX rules. The semantics of HEX-programs is defined considering interpretations I over the ordinary Herbrand base HB_P of a program P and a set of constants C. An interpretation I satisfies an external atom $\alpha = \&g[\boldsymbol{v}](\boldsymbol{w})$ (denoted $I \models \alpha$), iff $f_{\&g}(I, \boldsymbol{v}, \boldsymbol{w}) = 1$, where $\boldsymbol{v} \in C^n$ and $\boldsymbol{w} \in C^m$ and $f_{\&g}$ is a (fixed) function $f_{\&g} \colon 2^{HB_P} \times C^{n+m} \rightarrow \{0, 1\}$, representing the (semantics of the)

[9] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

corresponding external source. For an ordinary atom α, a rule r, or a program P , the satisfaction relation $I \models \alpha$ (respectively $I \models r$ or $I \models P$) is defined as usual.

The *FLP-reduct* [46] of P wrt. I is the set $fP^I \subseteq P$ of all rules r of form (2.3) in P such that $I \models \beta_i$, for all $i \in \{1, \ldots, m\}$ and $I \not\models \beta_j$, for all $j \in \{m+1, \ldots, n\}$. Eventually, I is an *answer set* of P, if I is $\subseteq$-minimal model of fP^I.[10]

For a more detailed account of HEX and its relation to MCS, see [35].

Concerning the purpose of evaluating MCS via HEX-programs, given an MCS M, we assemble a program $P(M)$ for computing equilibria of M as follows, where $1 \leq i \leq n$. An arbitrary truth assignment to beliefs is guessed:

$$a_i(p) \vee \overline{a}_i(p). \qquad \text{for all } p \in \Sigma_i \tag{2.4}$$

Each bridge rule of form (2.2) is evaluated by the corresponding HEX rules, wrt. the guess:

$$b_i(s) \leftarrow a_{c_1}(p_1), \ldots, a_{c_j}(p_j), \text{not } a_{c_{j+1}}(p_{j+1}), \ldots, \text{not } a_{c_m}(p_m). \tag{2.5}$$

Finally, constraints ensure that answer sets of the program correspond to equilibria:

$$\leftarrow \text{not } \&con\_out_i[a_i, b_i](). \tag{2.6}$$

Given an interpretation I, let $A_i^I = \{p \mid a_i(p) \in I\}, 1 \leq i \leq n$, denote a belief set for context C_i (corresponding to the guess on Σ_i in (2.4)), and let $B_i^I = \{s \mid b_i(s) \in I\}$ denote the set of bridge rule heads, from bridge rules br_i, which are applicable wrt. the guessed belief state. Each external atom in (2.6) represents $\mathbf{ACC}_i$: it returns true iff context C_i accepts a belief set upon input of B_i^I, which corresponds to the guessed A_i^I. Formally, we define $f_{\&con\_out_i}(I, a_i, b_i) = 1$ iff there exists an $S \in \mathbf{ACC}_i(kb_i \cup B_i^I)$ such that $S = A_i^I$.

Proposition 2 ([20]) *Answer sets I of $P(M)$ correspond 1-1 to equilibria S^I of M, where $I \rightleftharpoons S^I = (S_1^I, \ldots, S_n^I)$ and $S_i^I = \{p \mid a_i(p) \in I\}, i = 1, \ldots, n$.*

For further details on a concrete implementation we refer the reader to the MCS-IE system [15]. This thesis pursues 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 Part II of the thesis.

[10] For a program P without external atoms, the answer sets of P coincide with the ones of [54].

Part II

Algorithms for Multi-Context Systems

CHAPTER 3

Basic Distributed Algorithm and Realization with Loop Formulas

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

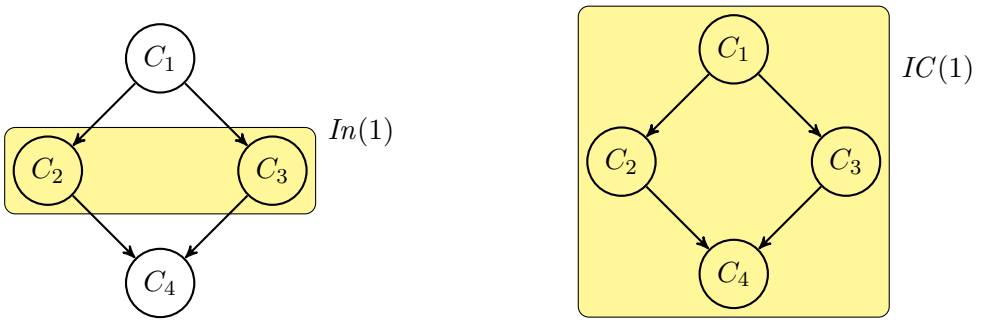
Besides the basic algorithm that deals with general MCSs, we take a closer look into systems whose contexts are built upon ASP logics. This setting allows for a special treatment of bridge rules by compiling them into the local knowledge bases using loop formulas. As such, one can exploit off-the-shelf SAT solvers to compute local belief sets at each context, and need to make just one call to this local solving process instead of multiple calls with respect to multiple inputs provided by children contexts. The trade-off is that the single call needs to guess for bridge atoms from the compiled-away bridge rules, which will be analyzed in Chapter 8.

3.1 Basic Algorithm for Multi-Context Systems

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.

Basic notions

Let us start defining some concepts required. The notion of import closure formally captures reachability.



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

Figure 3.1: Import neighborhood and Import closure

Definition 9 (Import Closure) *Let $M = (C_1, \ldots, C_n)$ be an MCS. The* import neighborhood of a context C_k *is the set*

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

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

Alternatively, we can constructively characterize

$$IC(k) = \{k\} \cup \bigcup_{j \geq 0} IC^j(k) \ ,$$

where $IC^0(k) = In(k)$, and $IC^{j+1}(k) = \bigcup_{i \in IC^j(k)} In(i)$. Note that the import closure of any context is finite, i.e., for an MCS $M = (C_1, \ldots, C_n)$ and C_k from M, $|IC(k)| \leq n$.

Example 13 Consider M in Example 11. 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 3.1).

Based on the import closure we define partial equilibria.

Definition 10 (Partial Belief States and Equilibria)
Let $M = (C_1, \ldots, C_n)$ be an MCS, and let $\epsilon \notin \bigcup_{i=1}^{n} \mathbf{BS}_i$. A partial belief state *of M is a sequence $S = (S_1, \ldots, S_n)$, such that $S_i \in \mathbf{BS}_i \cup \{\epsilon\}$, for $1 \leq i \leq n$.*

A partial belief state $S = (S_1, \ldots, S_n)$ of M is a partial equilibrium of M wrt. a context C_k *iff $i \in IC(k)$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and if $i \notin IC(k)$, then $S_i = \epsilon$, for all $1 \leq i \leq n$.*

As an aside, $IC(k)$ essentially defines a subsystem M' of M that is connected by bridge rules. We use partial equilibria of M instead of equilibria of M' to keep the original MCS M intact. Our view is similar to unnamed attributes in a relational database; essentially, we reference contexts in an MCS by position as in standard equilibria. Alternative representations of equilibria

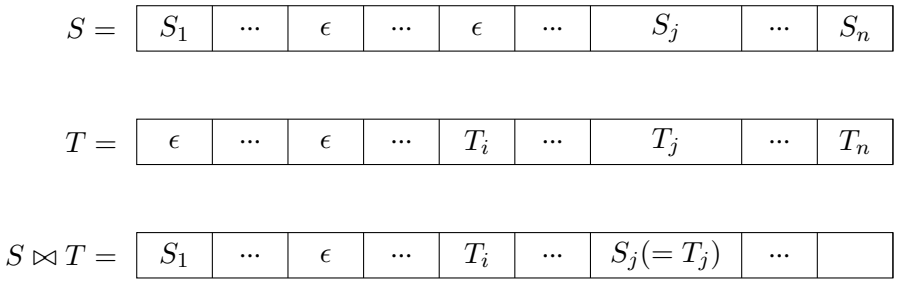


Figure 3.2: Joining Partial Belief States

for subsystems are possible but would prohibit to easily talk about the initial M without additional mappings from M' to M. Thus, for the purpose here it is more convenient to use the reference-by-position approach.

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

(i) $U_i = S_i$, if $T_i = \epsilon$ or $S_i = T_i$,

(ii) $U_i = T_i$, if $T_i \neq \epsilon$ and $S_i = \epsilon$,

for all $1 \leq i \leq n$. Figure 3.2 illustrates this operator. Note that $S \bowtie T$ is void, if some S_i, T_i are from $\mathbf{BS}_i$ but different. The *join* of two sets $\mathcal{S}$ and $\mathcal{T}$ of partial belief states is then naturally defined as $\mathcal{S} \bowtie \mathcal{T} = \{S \bowtie T \mid S \in \mathcal{S}, T \in \mathcal{T}\}$.

Example 14 Consider two sets of partial belief states:

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

Their join is given by

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

The basic algorithm

Given an MCS M and a starting context C_k, we aim at finding all partial equilibria of M wrt. C_k in a distributed way. To this end, we design an algorithm DMCS, whose instances run independently at each context node 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 such 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., partial equilibria projected to a relevant portion of the signature of the import closure of the starting context. This can be exploited for specific tasks like, e.g., local query answering or consistency checking. When computing projected partial equilibria, 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. Moreover, it is assumed that the topology of the overall MCS is not known at context nodes (in Chapter 4, we present optimized algorithms when such topology information is given). The idea is as follows: starting from context C_k, we visit the import closure of C_k by expanding the import neighborhood at each context like in a depth-first search, maintaining the set of visited contexts in a set $hist$, until a leaf context is reached, or a cycle is detected by noticing the presence of the current context in $hist$. A leaf context simply computes its local belief sets, transforms all belief sets into partial belief states, and returns this result to its parent (invoking context, Figure 3.3a). In case of a cycle (Figure 3.3c), the context detecting the cycle, say C_i, must also break it, by (i) guessing belief sets for the "export" interface of C_i, (ii) transforming the guesses into partial belief states, and (iii) returning them to the invoking context.

The results of intermediate contexts are partial belief states, which can be joined, i.e., consistently combined, with partial belief states from their neighbors; an intermediate context returns its local belief sets, joined with the results from its neighbors, as final result (Figure 3.3b).

For computing projected partial equilibria, the algorithm offers a parameter V, the *relevant interface*. Given a (partial) belief state S and set $V \subseteq \Sigma$ of variables, the *restriction of S to V*, denoted $S|_V$, is given by the (partial) belief state $S' = (S_1|_V, \ldots, S_n|_V)$, where $S_i|_V = S_i \cap V$ if $S_i \neq \epsilon$, and $\epsilon|_V = \epsilon$; the restriction of a set of (partial) belief states $\mathcal{S}$ to V is $\mathcal{S}|_V = \{S|_V \mid S \in \mathcal{S}\}$.

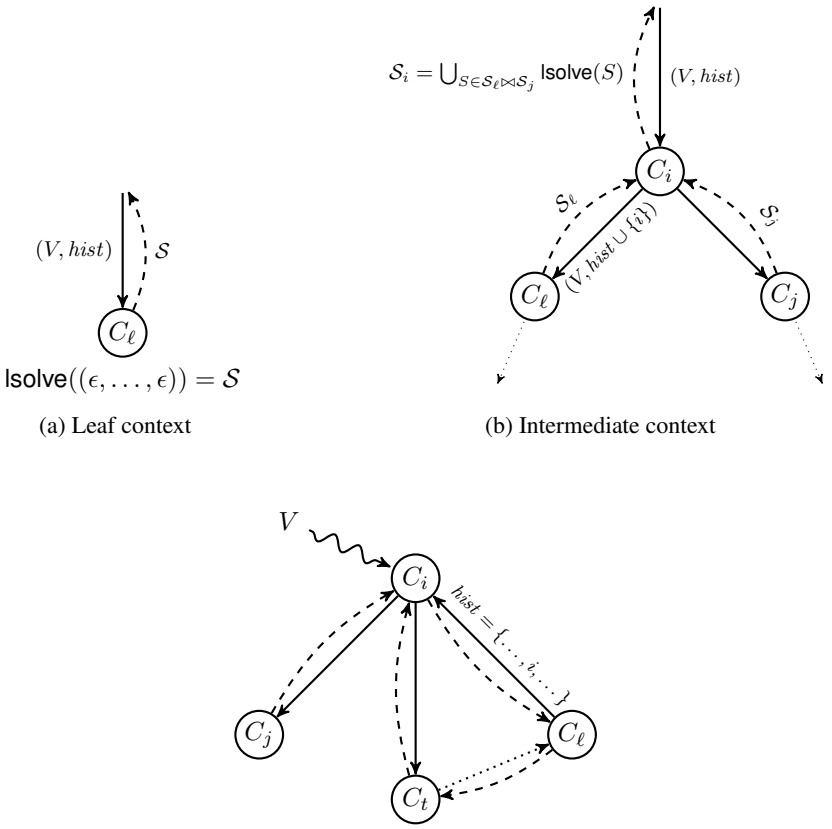
Let $V(k) = \{p_i \mid (c_i : p_i) \in B(r), r \in br_k\}$ denote the *import interface* of context C_k. By $V^*(k) = \bigcup_{i \in IC(k)} V(i)$, the *recursive import interface* of C_k, we refer to the interface of the import closure of C_k.

Given a context C_k, we have two extremal cases: (1) $V = V^*(k)$ and (2) $V = \Sigma$. In Case (1), DMCS basically checks for consistency on the import closure of C_k by computing partial equilibria projected to interface variables only. In Case (2), the algorithm computes partial equilibria wrt. C_k. Between these two, by providing a fixed interface 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 re-computation and the 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 $(V, hist)$, upon which it starts the computation outlined in Algorithm 3.1. This process also serves the purpose of keeping the cache $c(k)$ persistent. We write $C_i.\mathsf{DMCS}(V, hist)$ to specify that we send $(V, hist)$ to the process at context C_i and wait for its return message.

Algorithm 3.1 uses the following primitives:



(a) Leaf context

(b) Intermediate context

(c) Cycle breaking

Figure 3.3: Basic Distributed Algorithm - Casewise

- function $\mathsf{lsolve}(S)$ (Algorithm (c)): augments the knowledge base kb of the current context with the heads of bridge rules in br that are applicable wrt. 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 $\mathsf{guess}(V, C_k)$: guesses all possible truth assignments for the relevant interface wrt. C_k, i.e., for $\Sigma_k \cap V$.[1]

DMCS proceeds in the following way:

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

(b) check for a cycle;

[1] In order to relate variables to context signatures, V can either be a vector of sets, or variables in V are prefixed with context ids; for simplicity, we kept V as a set without further assumptions.

Algorithm 3.1: DMCS$(V, hist)$ at $C_k = (L_k, kb_k, br_k)$

Input: 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 wrt. V
(c) $\quad \mathcal{S} := \mathsf{guess}(V, C_k)$
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**
$\quad\quad\quad \mathcal{T} := \mathcal{T} \bowtie C_i.\mathsf{DMCS}(V, hist)$
(e) $\quad$ **foreach** $T \in \mathcal{T}$ **do** $\mathcal{S} := \mathcal{S} \cup \mathsf{lsolve}(T)$
(f) $\quad c(k) := \mathcal{S}|_V$
return $\mathcal{S}|_V$

Algorithm 3.2: 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 $\{(S_1, , \ldots, S_{k-1}, T_k, S_{k+1}, \ldots, S_n) \mid T_k \in \mathbf{T}\}$

(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 imported 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 15 Reconsider M from Example 11. Suppose the user invokes $C_1.\mathsf{DMCS}(V, \emptyset)$, where $V = \{a, b, c, f, g\}$, 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

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

After caching $\mathcal{S}_4|_V$ in (f), C_4 returns $\mathcal{S}_4|_V = \mathcal{S}_4$ to one of the contexts C_2, C_3 whose request arrived first. On second call, C_4 simply returns to the other context $\mathcal{S}_4|_V$ 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 14.

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

Thus,

$$\mathcal{S}_2|_V = \{\, (\epsilon, \{b\}, \epsilon, \{\neg f, g\})\,,\ (\epsilon, \{\neg b\}, \epsilon, \{f, \neg g\})\,\} \text{ and}$$
$$\mathcal{S}_3|_V = \left\{ \begin{array}{l} (\epsilon, \epsilon, \{\neg c\}, \{\neg f, g\}), \\ (\epsilon, \epsilon, \{c\}, \{\neg f, g\}), \\ (\epsilon, \epsilon, \{\neg c\}, \{f, \neg g\}) \end{array} \right\} \quad .$$

C_1, after computing in (d)

$$\mathcal{S}_2|_V \bowtie \mathcal{S}_3|_V = \left\{ \begin{array}{l} (\epsilon, \{b\}, \{\neg c\}, \{\neg f, g\}), \\ (\epsilon, \{b\}, \{c\}, \{\neg f, g\}), \\ (\epsilon, \{\neg b\}, \{\neg c\}, \{f, \neg g\}) \end{array} \right\}$$

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

$$\mathcal{S}_1|_V = \left\{ \begin{array}{l} (\{a\}, \{b\}, \{c\}, \{\neg f, g\}), \\ (\{\neg a\}, \{b\}, \{\neg c\}, \{\neg f, g\}), \\ (\{\neg a\}, \{\neg b\}, \{\neg c\}, \{f, \neg g\}) \end{array} \right\} \quad .$$

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

Example 16 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 3.4 shows the cyclic topology of M. Suppose that the user sends a request to C_1 by calling C_1.DMCS$(V, \emptyset)$ with $V = \{a, b, c\}$.

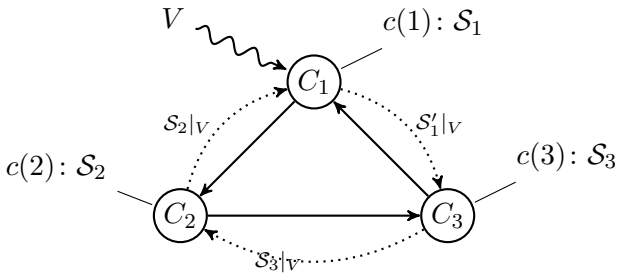


Figure 3.4: A cyclic topology

In step (d) of Algorithm 3.1, C_1 calls $C_2.\mathsf{DMCS}(V, \{1\})$, then context C_2 issues a call $C_3.\mathsf{DMCS}(V, \{1, 2\})$, thus C_3 invokes $C_1.\mathsf{DMCS}(V, \{1, 2, 3\})$. 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 $\Sigma_1 \cap V$. Then, following the dotted lines in Figure 3.4, the set $\mathcal{S}_1'|_V = \mathcal{S}_1'$ is the return value for the request from C_3, who joins it with an initial empty belief state $(\epsilon, \epsilon, \epsilon)$, gives us $\mathcal{T}$ and then calls $\mathsf{Isolve}(T)$ for each $T \in \mathcal{T}$ in (e), resulting in

$$\mathcal{S}_3 = \left\{ \begin{array}{l} (\{\neg a\}, \epsilon, \{c, \neg d\}), \\ (\{\neg a\}, \epsilon, \{\neg c, d\}), \\ (\{a\}, \epsilon, \{\neg c, \neg d\}) \end{array} \right\} .$$

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

$$\mathcal{S}_2 = \left\{ \begin{array}{l} (\{\neg a\}, \{b\}, \{c\}), \\ (\{\neg a\}, \{\neg b\}, \{c\}), \\ (\{a\}, \{\neg b\}, \{\neg c\}) \end{array} \right\} ,$$

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

Next, C_1 will join $\mathcal{S}_2|_V$ with $(\epsilon, \epsilon, \epsilon)$, which yields $S_2|_V$, and then use this result to call Isolve. 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., $V^*(k)$, as the relevant interface is a safe (lower) bound for the correctness of Algorithm 3.1. In what follows, let M, C_k, and $V^*(k)$ as above.

Theorem 3 (Correctness of DMCS with partial equilibrium) *For all $V \supseteq V^*(k)$, $S' \in C_k.\mathsf{DMCS}(V, \emptyset)$ iff there exists a partial equilibrium S of M wrt. C_k such that $S' = S|_V$.*

To prove this theorem, we first prove the following Lemmas 4 and 5. The latter aims at simplifying the proof for the cyclic case, based on the notion of converting cyclic MCSs to acyclic ones.

Lemma 4 *For any context C_k and partial belief state S of an MCS $M = (C_1, \ldots, C_n)$, $app(br_k, S) = app(br_k, S|_V)$ for all $V_\Sigma \supseteq V \supseteq V^*(k)$.*

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 $p_i \in S_{c_i}|_{V_{c_i}} \wedge p_j \notin S_{c_j}|_{V_{c_j}}$. Indeed:

We have $V \subseteq V_\Sigma \Rightarrow V_{c_j} \subseteq V_{\Sigma_j} \Rightarrow S_{c_j}|_{V_{c_j}} \subseteq S_{c_j}$. Therefore, $p_j \notin S_{c_j} \Rightarrow p_j \notin S_{c_j}|_{V_{c_j}}$.

Now, assume that $p_i \notin S_{c_i}|_V$. From the fact that $p_i \in S_{c_i}$, it follows that $p_i \notin V_{c_i}$, hence $p_i \notin V^*(k)$. But this is in contradiction with the fact that p_i appears in the body of a bridge rule.

Therefore, $r \in app(br_k, S|_V)$. □

The next Lemma 5 is based on the following notions that convert cyclic MCSs to acyclic ones and show that they have corresponding equilibria. The idea is to introduce an additional context C_k^g for every cycle breaker C_k and to modify the bridge rules of C_k as well as its parent contexts. We start with a function ren that renames part of bridge rules.

Definition 11 *Let C_k be a context in an MCS M, and let V be an interface for running DMCS. The* renaming function ren *is defined as follows:*

- *For an atom a:* $ren(a, k, V) = \begin{cases} a^g & \text{if } a \in \Sigma_k \cap V \\ a & \text{otherwise} \end{cases}$
- *For a context index c:* $ren(c, k, V) = \begin{cases} \overline{c} & \text{if } c \in \{1, \ldots, n\} \\ c & \text{otherwise} \end{cases}$
- *For a bridge atom $(c_i : p_i)$:* $ren((c_i : p_i), k, V) = (ren(c_i, k, V) : ren(p_i, k, V))$
- *For a bridge body $B = \{(c_1 : p_1) \ldots (c_j : p_j)\}$:*

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

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

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

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

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

A set of contexts $\{C_{c_1}, C_{c_2}, \ldots, C_{c_\ell}\}$ of an MCS M is called a cycle wrt. an interface V iff

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

One can pick an arbitrary context in this set to be its cycle-breaker. Given an MCS M, there are several way 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 12 *Given an MCS $M = (C_1, \ldots, C_n)$ and let $\mathcal{CB}_M^r = \{C_{c_1}, \ldots, C_{c_j}\}$ 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 V is done as follows:*

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

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

Let $C_i''' = (L_i, kb_i, br_i''')$ *where* $br_i''' = \begin{cases} br_i'' \cup \{a \leftarrow (\overline{i}\colon a^g)\} & \text{if } C_i \in \mathcal{CB}_M^r \\ br_i'' & \text{otherwise} \end{cases}$

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

Lemma 5 *Let M be an MCS and $M^\star$ be its conversion to an acyclic MCS as in Definition 12. 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 $\mathcal{CB}_M^r$ 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 $C_k \in \mathcal{CB}_M^r$, its counterpart (R_2) visits $C_{\overline{k}}$. At these corresponding locations:

- (R_1) calls guess(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 two runs (R_1) and (R_2). The correspondence of equilibria hence follows. □

Proof (Theorem 3) Thanks to Lemma 5, we now need to prove Theorem 3 only for the acyclic case and automatically get the result for the cyclic case.

($\Rightarrow$) We start by showing soundness of DMCS. Let $S' \in C_k.\mathsf{DMCS}(V, \emptyset)$ such that $V \supseteq V^*(k)$. We show now that there is a partial equilibrium S of an acyclic M w.r.t. C_k such that $S' = S|_V$. 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 $\mathcal{S} = \mathsf{lsolve}((\epsilon, \ldots, \epsilon)) = \{(\epsilon, \ldots, \epsilon, T_k, \epsilon, \ldots, \epsilon) \mid T_k \in \mathbf{ACC}_k(kb_k)\}$. We now show that $S' \in \mathcal{S}|_V$. 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 $S' = S|_V$. From $In(k) = \emptyset$, we get that $IC(k) = \{k\}$, thus the partial belief state $(\epsilon, \ldots, \epsilon, T_k, \epsilon, \ldots, \epsilon) \in \mathcal{S}$ is a partial equilibrium of M w.r.t. C_k. Contradiction.

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

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

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 $S^{i_j}|_V = S'^{i_j}$.

Let $\mathcal{S} = C_k.\mathsf{DMCS}(V, hist)$. 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 $S' = S|_V$. 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, let $\mathcal{S} = \bigcup\{\mathsf{lsolve}(S) \mid S \in \mathcal{T}\}$ be the result of executing Step (e). Eventually, $S' \in \mathcal{S}|_V$. Since every DMCS at $C_{i_1}, \ldots, C_{i_m}$ returns its partial equilibria w.r.t. C_{i_j} projected to V, we have that every $T \in \mathcal{T}$ is a partial equilibrium w.r.t. C_{i_j} projected to V. M is acyclic and we have visited all contexts from $In(k)$, thus by Lemma 4 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 V except the kth component. As every $T \in \mathcal{T}$ preserves applicability of the rules by Lemma 4, we get that for every $S' \in \mathcal{S}|_V$, there exists a partial equilibrium S of M w.r.t. C_k such that $S' = S|_V$.

($\Leftarrow$) 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 $S' = S|_V$. We show now that $S' \in C_k.\mathsf{DMCS}(V, \emptyset)$.

Base case: C_k is a leaf context. Then in executing $C_k.\mathsf{DMCS}(V, \emptyset)$, 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 wrt. C_k, $S \in \mathcal{S}$; hence, $S' = S|_V$ will be returned from $C_k.\mathsf{DMCS}(V, \emptyset)$.

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, $S^{i_j}|_V$ is computed by $C_{i_j}.\mathsf{DMCS}(V, \emptyset)$ for all $i_j \in In(k)$. We will show that $S|_V$ is computed by $C_k.\mathsf{DMCS}(V, \emptyset)$.

Indeed, because we are considering an acyclic M, it holds that $S^{i_j}|_V$ is also returned from a call $C_{i_j}.\mathsf{DMCS}(V, \{k\})$, 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 wrt. 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 $V \supseteq V^*(k)$, Lemma 4 tells us that the applicability of bridge rules is preserved under the projection of belief sets to V. This gives us that $S_k \in \mathsf{lsolve}(T)$ in step (e), and hence $S' = S|_V$ is returned from $C_k.\mathsf{DMCS}(V, \emptyset)$. □

We can compute partial equilibria at C_k if we use V_Σ. This holds because using V_Σ preserves all belief sets returned from step (e), as the projection at step (f) takes no effect.

Corollary 6 *S is a partial equilibrium of M wrt. C_k iff $S \in C_k.\mathsf{DMCS}(V_\Sigma, \emptyset)$.*

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. (Disconnected contexts in M can be always connected to a new root context using simple bridge rules. An MCS with a self-loop context can also be converted to a single root context using Definition 12.)

Corollary 7 *S is an equilibrium of the MCS M iff $S \in C_1.\mathsf{DMCS}(V_\Sigma, \emptyset)$ for a single root context C_1.*

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

Proposition 8 *In a run of DMCS with an interface V:*

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

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

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

$$c(k) \leq \Pi_{i \in In(k)} 2^{|V \cap \Sigma_i|} \leq 2^{|In(k)| \times |V|} \leq 2^{n \times |V|}$$

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

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

(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 $C_j.\mathsf{DMCS}(V, hist)$ at C_i, and once when retrieving $\mathcal{S}|_V$ 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. □

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 the existence of a manager $\mathcal{M}$ that is reachable from every context C_i in the system, which can ask $\mathcal{M}$ whether some isolated inconsistent context or subsystem exists. If $\mathcal{M}$ affirms, C_i's DMCS simply returns $\emptyset$, eliminating all partial equilibria.

In an attempt to improve improving decentralization and information encapsulation, we can weaken the manager assumption by introducing *routers*. Instead of asking the manager, a context C_i queries an assigned router $\mathcal{R}$, which collects the necessary topology information for C_i or makes a cache look-up. 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, and 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 $V^*(k) \subseteq V$. But this is not needed if M resp. the import closure of C_k has no *join-contexts*, i.e., contexts which have at least two parents. If we have access to path information in M at each context, we could calculate V on the fly and change it accordingly during MCS traversal. In particular, for tree-shaped or ring topology of M, we can restrict V to the *locally shared interface* between C_k and its import neighbors, i.e., restricting V to the bridge atoms of br_k. In presence of join-contexts, 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.

In Chapter 4, we will present optimization techniques when topological information of the system is available.

3.2 Realization with Loop Formulas

Algorithm DMCS incorporates in step (e) via lsolve the bridge rules br_k into the local knowledge base kb_k, given belief input from a belief state T, and then computes the belief sets; this is done for all $T \in \mathcal{T}$.

In certain settings, it is possible to compile br_k into kb_k, yielding some kb'_k, such that the belief sets of kb'_k are precisely the possible belief sets T_k in the return value of any $\mathsf{lsolve}(S)$; hence, the for-loop in step (e) can be replaced by a *single* join $\mathcal{S} := \mathcal{T} \bowtie \mathcal{B}_k|_V$, where $\mathcal{B}_k$ are the acceptable belief sets of kb'_k, properly converted to partial belief states.

For example, this is possible for classical logics L_k (assuming that contexts are not self-referential), or ASP logics. This is because there are well-known transformations of ASP programs P into equivalent classical theories $\phi(P)$, such that the answer sets of P are given by the classical models of $\phi(P)$, which hinge on *loop formulas* (Section 2.3).

In this section, we develop loop formulas for MCS, by which bridge rules can be compiled into a local classical theory. In fact, we combine this with a loop formula transformation of ASP programs into classical theories; this enables us to obtain particular equilibria satisfying groundedness. Roughly, we adapt the notion of support formulas in such a way that also bridge rules have an effect on loops (e.g., $a \leftarrow (1 : c)$ on the loop above), but we distinguish *local support* and *bridge support*.

Loop formulas for MCS

We assume that in $M = (C_1, \ldots, C_n)$, all logics L_i are ASP logics with $\Sigma_i = \mathcal{A}_i$ and $\Sigma = \mathcal{A}$. Furthermore, we assume that all heads of bridge rules are (disjunctive) facts (this is no loss of generality). This allows us to adapt disjunctive loop formulas to encode bridge rules as classical theories.

Notice that in the definition of bridge rules in MCSs, the head of a bridge rule can be an element of the knowledge base of the logic. In the particular contexts that we adopt here, such a head can be (i) a disjunctive rule, (ii) a set of disjunctive rules, (iii) or even any propositional formula that has an equivalent representation to a set of disjunctive rules. One can see that the two last cases are reducible to the first one. Moreover, when having a disjunctive rule in the head of a bridge rule, it is possible to replace this head by a fresh auxiliary atom and put the disjunctive rule, with body increased with the auxiliary atom, into the knowledge base of the context. Therefore, in the sequel, it is enough to consider only bridge rules whose heads are disjunctive facts.

Let $\neg.A = \{\neg a \mid a \in A\}$ and, as usual, $\bigvee F = \bigvee_{f \in F} f$ and $\bigwedge F = \bigwedge_{f \in F} f$ (note that $\bigvee \emptyset = \bot$ and $\bigwedge \emptyset = \top$).

For any ASP rule r, we then define

$$\kappa(r) = \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \supset \bigvee H(r) \ ,$$

and for any set R of ASP rules, $\kappa(R) = \bigwedge_{r \in R} \kappa(r)$.

The *support formula of a set $A \subseteq \mathcal{A}$ w.r.t. an ASP rule r* is

$$\varepsilon(A, r) = \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge \bigwedge \neg.(H(r) \setminus A) \ ,$$

and w.r.t. any set R of ASP rules, $\varepsilon(A, R) = \bigvee_{r \in R} \varepsilon(A, r)$.[2]

To build support formulas w.r.t. a bridge rule r of form (2.2), we convert it to an ASP rule $\ell(r)$ by replacing $(c_k : p_k)$ with p_k, $1 \leq k \leq m$; for any bridge rule set R, we let $\ell(R) = \{\ell(r) \mid r \in R\}$.

We identify the *support rules* and the *external support rules* of a set of ASP rules R w.r.t. a set $A \subseteq \mathcal{A}$ as

$$SR(A, R) = \{r \in R \mid H(r) \cap A \neq \emptyset\} \text{ and}$$
$$ER(A, R) = \{r \in R \mid H(r) \cap A \neq \emptyset, B^+(r) \cap A = \emptyset\} \ ,$$

respectively (note the $SR(A, R)$ are not minimizing).

We next define necessary dependency relations. In a set R of ASP rules, we say that a *depends on* b, denoted $a \to b$, if $a \in H(r)$ and $b \in B^+(r)$ for some rule $r \in R$. The set of *dependencies in context C_i* is then the set of all pairs $a \to_i b$ such that $a \to b$ in $kb_i \cup \ell(br_i)$.

Based on this, we define the dependency graph of an MCS and loops for contexts and MCS.

Definition 13 (Dependency Graph and Loops) *The* dependency graph of a multi-context system $M = (C_1, \ldots, C_n)$ *is the directed graph* $G = (\mathcal{A}, \bigcup_{1 \leq i \leq n} \to_i)$.

A loop of C_i (resp., M) *is any set $\mathcal{L} \subseteq \mathcal{A}_i$ (resp., $\mathcal{L} \subseteq \mathcal{A}$) of atoms such that the subgraph of G induced by $\mathcal{L}$ is strongly connected.*

Note that each singleton $\{a\}$ is a loop. Figure 3.5 illustrates the dependency graph of the system in Example 11, in which singleton loops are hidden for simplification. A non-trivial loop here is $\{c, d\}$.

Example 17 Consider $M = (C_1)$ and $M' = (C_1')$, where

$$kb_1 = \left\{ \begin{array}{l} a \leftarrow b \\ b \leftarrow a \end{array} \right\}, br_1 = \left\{ \begin{array}{l} a \leftarrow (1 : b) \\ b \leftarrow (1 : a) \end{array} \right\},$$
$$kb_1' = \left\{ \begin{array}{l} c \leftarrow d \\ d \leftarrow c \end{array} \right\}, br_1' = \left\{ \begin{array}{l} c \leftarrow \text{not } (1 : d) \\ d \leftarrow \text{not } (1 : c) \end{array} \right\} \ .$$

The loops of C_1 are $\{a\}$, $\{b\}$, and $\{a, b\}$, and those of C_1' are $\{c\}$, $\{d\}$, and $\{c, d\}$. Moreover, $\{a, b\}$ is also the only loop of M (at the global level), while $\{c, d\}$ is not that of M'.

[2] [48] calls $\varepsilon(A, r)$ the *external* support formula, which is not entirely true in our setting.

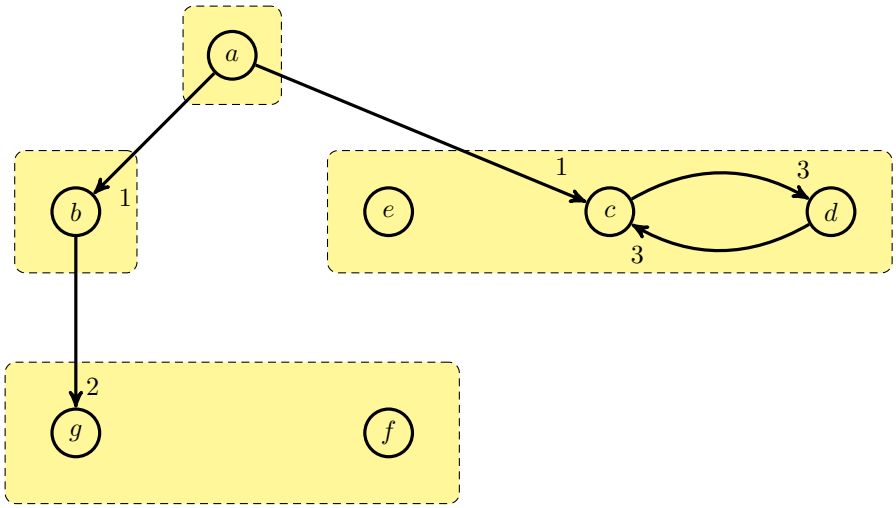


Figure 3.5: Dependency Graph of the MCS in Example 11

Next we define the local loop formula of a context.

Definition 14 (Local Loop Formulas) *Let $\mathcal{L}$ be a loop of context C_i. Then the* loop formula for $\mathcal{L}$ w.r.t. C_i *is*

$$\lambda(\mathcal{L}, C_i) = \left(\bigvee \mathcal{L}\right) \supset \varepsilon(\mathcal{L}, ER(\mathcal{L}, kb_i) \cup SR(\mathcal{L}, \ell(br_i))) \ .$$

Furthermore, the loop formula of context C_i *is the conjunction $\lambda(C_i) = \bigwedge_{\mathcal{L}} \lambda(\mathcal{L}, C_i)$ of all loops $\mathcal{L}$ of C_i.*

Example 18 Continuing Example 17, $\mathcal{L}_1 = \{a, b\}$ has the loop formula (i) $\lambda(\mathcal{L}_1, C_1) = a \vee b \supset b \vee a$. Indeed, both rules of br_1 are support rules of C_1 w.r.t. $\mathcal{L}_1$, which has no external support rules in kb_1; thus $\lambda(\mathcal{L}_1, C_1) = a \vee b \supset \varepsilon(\mathcal{L}_1, \ell(br_1))$, and $\varepsilon(\mathcal{L}_1, \ell(br_1)) = b \vee a$. Similarly, $\mathcal{L}'_1 = \{c, d\}$ has the loop formula (ii) $\lambda(\mathcal{L}'_1, C'_1) = c \vee d \supset \neg d \vee \neg c$.

Experts on loop formulas will notice that (i) is weaker than the loop formula $a \vee b \supset \bot$ of the program $\{a \leftarrow b;\ b \leftarrow a\}$, and admits $\{a, b\}$ as a model of the translation; this complies with the MCS semantics. Similarly, (ii) is weaker than $c \vee d \supset \bot$ but can eliminate the model $\{c, d\}$. Finally, we have $\lambda(\{a\}, C_1) = a \supset b$, $\lambda(\{b\}, C_1) = b \supset a$, $\lambda(\{c\}, C'_1) = c \supset d \vee \neg d$, and $\lambda(\{d\}, C'_1) = d \supset c \vee \neg c$.

The intuition behind the translation is that we distinguish between supports at the local and bridge levels. At the local level, support for a loop must come from the outside of the loop (hence called external support); while at the bridge level, the loop can support itself. When the same loop $\mathcal{L}$ without external support appears in both local and global levels, the self-support from SR remedies the empty support from ER. This way, we get the equilibrium semantics of MCSs. We now can transform the whole MCS into a formula.

Definition 15 (MCS loop transformation) *Given the multi-context system $M = (C_1, \ldots, C_n)$ with ASP logics, let*

$$\pi(C_i) = \lambda(C_i) \wedge \kappa(kb_i) \wedge \kappa(\ell(br_i)) \ \ , 1 \leq i \leq n, \textit{ and}$$
$$\pi(M) = \bigwedge_{i=1}^{n} \pi(C_i) \ \ .$$

Here, $\pi(C_i)$ describes the belief sets of C_i, depending on valuations of the atoms in bridge rule bodies; $\pi(M)$ just aligns descriptions. The next result shows that this transformation correctly captures the equilibria of M.

Theorem 9 *The equilibria of any M with ASP logics correspond one-to-one to the models of the formula $\pi(M)$.*

Note that by this theorem, consistency of M (i.e., existence of some equilibrium) maps to a distributed SAT problem.

Proof ($\Rightarrow$) Let $S = (S_1, \ldots, S_n)$ be an equilibrium of the MCS M. We have that S_i is an answer set of

$$R_i = kb_i \cup \{head(r) \mid r \in app(br_i, S)\}.$$

We show now that $\mathbf{I} = \bigcup_{1 \leq i \leq n} S_i$ is a model of $\pi(M)$ by showing that, for all $1 \leq i \leq n$,

(i) $\mathbf{I} \models \kappa(kb_i)$,

(ii) $\mathbf{I} \models \kappa(\ell(br_i))$, and

(iii) $\mathbf{I} \models \lambda(C_i)$.

Part (i): From S_i being an answer set of R_i, we get $S_i \models r$ for all $r \in kb_i$. It follows that $S_i \models \kappa(r)$, hence $S_i \models \kappa(kb_i)$, and finally, we get that $\mathbf{I} \models \kappa(kb_i)$.

Part (ii): Take an arbitrary bridge rule $r \in br_i$ of form (2.2):

- If $r \notin app(br_i, S)$, then either there exists $(c_h : p_h)$ in r such that $p_h \notin S_{c_h}$ for $1 \leq h \leq j$, or there exists $(c_k : p_k)$ in r such that $p_k \in S_{c_k}$ for $j+1 \leq k \leq m$. Since the sets Σ_i are pairwise disjoint, we get that $\mathbf{I}$ does not satisfy the antecedent of $\kappa(\ell(r))$, that is, $\bigwedge B^+(\ell(r)) \wedge \bigwedge \neg . B^-(\ell(r))$.

- If $r \in app(br_i, S)$, then for all $(c_h : p_h)$ for $1 \leq h \leq j$, $p_i \in S_{c_i}$, and for all $(c_k : p_k)$ for $j+1 \leq k \leq m$, $p_k \notin S_{c_k}$. As the sets Σ_i are pairwise disjoint, we have that $\mathbf{I}$ satisfies the antecedent of $\kappa(\ell(r))$. Furthermore, since r is applicable in S, $head(r)$ was added to R_i to determine S_i, thus $\mathbf{I} \models head(r)$ and so we have that $\mathbf{I}$ satisfies the consequent of $\kappa(\ell(r))$.

In both cases we can derive that $\mathbf{I} \models \kappa(\ell(r))$ for all $r \in br_i$, and eventually $\mathbf{I} \models \kappa(\ell(br_i))$.

Part (iii): Now take an arbitrary loop $\mathcal{L} \subseteq \mathcal{A}_i$ of C_i. We have to show that

$$\mathbf{I} \models \left(\bigvee \mathcal{L}\right) \supset \varepsilon(\mathcal{L}, ER(\mathcal{L}, kb_i) \cup SR(\mathcal{L}, \ell(br_i))) \tag{3.1}$$

holds. If $\mathcal{L} \cap S_i = \emptyset$, then (3.1) holds vacuously. Otherwise, $\mathbf{I} \models \bigvee \mathcal{L}$, so we have to show that $\mathbf{I}$ satisfies the consequent of (3.1), i.e., for some $\mathbf{r} \in ER(\mathcal{L}, kb_i) \cup SR(\mathcal{L}, \ell(br_i))$, we get

$$\mathbf{I} \models \bigwedge B^+(\mathbf{r}) \wedge \bigwedge \neg.B^-(\mathbf{r}) \wedge \bigwedge \neg.(H(\mathbf{r}) \setminus \mathcal{L}) \ . \tag{3.2}$$

Recall that S_i is an answer set of R_i, it is a minimal model of $R_i^{S_i}$ under subset inclusion. We obtain that $T = S_i \setminus \mathcal{L}$ is not a model of $R_i^{S_i}$. Hence, there exists an $\bar{r} \in R_i^{S_i}$ such that $T \models B(\bar{r})$ and $T \not\models H(\bar{r})$, thus $B^+(\bar{r}) \subseteq T$ and $B^-(\bar{r}) \cap T = \emptyset$.

We have that there is a rule $r \in R_i$ such that $\bar{r}$ is the reduced rule r by the GL-reduct. Since $T \subseteq S_i$, we get $S_i \models B(r)$, as $S_i \not\models B^-(r)$ follows from $\bar{r} \in R_i^{S_i}$. This means that $S_i \models H(r)$, hence $S_i \cap H(r) \neq \emptyset$. Since $T \not\models H(\bar{r})$, we also get that $T \cap H(r) = \emptyset$. Thus, $\mathcal{L} \cap H(r) \neq \emptyset$.

We obtain two cases for $r \in R_i$:

- $r \in kb_i$: from $B^+(\bar{r}) \subseteq T$ we get $B^+(r) \cap \mathcal{L} = \emptyset$, and from $\mathcal{L} \cap H(r) \neq \emptyset$ we can then conclude $r \in ER(\mathcal{L}, kb_i)$. We have to show that $S_i \models \varepsilon(\mathcal{L}, r)$, that is
  - $S_i \models \bigwedge B^+(r)$,
  - $S_i \models \bigwedge \neg.B^-(r)$, and
  - $S_i \models \neg.(H(r) \setminus \mathcal{L})$.

  The first two items hold by $S_i \models B(r)$. The last item holds since $T \cap H(r) = \emptyset$, hence $(S_i \setminus \mathcal{L}) \cap H(r) = S_i \cap (H(r) \setminus \mathcal{L}) = \emptyset$. Since $r \in ER(\mathcal{L}, kb_i)$, we set $r = \mathbf{r}$ and obtain that (3.2) is true, hence also (3.1) holds.

- $r \in \{head(r') \mid r' \in app(br_i, S)\}$: there exists a rule $r' \in app(br_i, S)$ of form (2.2) such that $r = head(r')$ and for all $(c_i : p_i)$ in r', $1 \leq i \leq j$, we have $p_i \in S_{c_i}$, and for all $(c_k : p_k)$ in r', $j+1 \leq k \leq m$, we have $p_k \notin S_{c_k}$. From $\mathcal{L} \cap H(r) \neq \emptyset$ we have that $\mathcal{L} \cap H(head(r')) \neq \emptyset$ and thus $\ell(r') \in SR(\mathcal{L}, \ell(br_i))$. We have to show for each S_i that $S_i \models \varepsilon(\mathcal{L}, \ell(r'))$, that is
  - $S_i \models \bigwedge B^+(\ell(r'))$,
  - $S_i \models \bigwedge \neg.B^-(\ell(r'))$, and
  - $S_i \models \neg.(H(\ell(r')) \setminus \mathcal{L})$.

  Since the sets Σ_i are pairwise disjoint, the first two items hold as $S_i \models B(r)$ and $\mathbf{I}$ is then a model of the body of $\ell(r')$. The last one holds since $T \cap H(head(r')) = \emptyset$, hence $(S_i \setminus \mathcal{L}) \cap H(head(r')) = S_i \cap (H(head(r')) \setminus \mathcal{L}) = \emptyset$. Since $\ell(r') \in SR(\mathcal{L}, \ell(br_i))$, we set $\ell(r') = \mathbf{r}$ and obtain that (3.2) is true, hence also (3.1) holds.

We have shown that (i)–(iii) holds, and as a result, we get that $\mathbf{I}$ is a model of $\pi(M)$.

($\Leftarrow$) Let $\mathbf{I}$ be a model of $\pi(M)$. We can create a belief state $S = (S_1, \ldots, S_n)$, where each $S_i = \mathbf{I}|_{\Sigma_i}$, and show that S is an equilibrium of M (note that the sets S_i are pairwise disjoint as the sets Σ_i are pairwise disjoint). We have to show that each $S_i \in \mathbf{ACC}_i(kb_i \cup H_i)$, where $H_i = \{head(r) \mid r \in app(br_i, S)\}$, i.e., each S_i is an answer set of $kb_i \cup H_i$.

We show the following:

(i) S_i is a model of $(kb_i \cup H_i)^{S_i} = kb_i^{S_i} \cup H_i$ and

(ii) S_i is minimal.

Part (i): By $\mathbf{I} \models \kappa(kb_i)$ we immediately get that $S_i \models \kappa(kb_i)$ (since the sets Σ_i are pairwise disjoint), hence $S_i \models kb_i$ and also $S_i \models kb_i^{S_i}$.

Moreover, we have that $\mathbf{I} \models \kappa(\ell(br_i))$, that is, for all $r_\ell \in \ell(br_i)$ we have $\mathbf{I} \models \kappa(r_\ell)$. Let $r \in br_i$ of form (2.2) such that $r_\ell = \ell(r)$. We have two cases:

- $\mathbf{I} \not\models \bigwedge B^+(r_\ell)$ or $\mathbf{I} \not\models \bigwedge \neg.B^-(r_\ell)$: there exists a $p_i \in B^+(r_\ell)$ such that $p_i \notin \mathbf{I}$ or a $p_k \in B^-(r_\ell)$ such that $p_k \in \mathbf{I}$. Since p_i, p_k are uniquely determined (as follows from our disjoint language assumption), we have that there exists $(c_i : p_i)$ in r, $1 \leq i \leq j$, or $(c_k : p_k)$ in r, $j+1 \leq k \leq m$. Thus, from our construction of S, we obtain $p_i \notin S_{c_i}$ or $p_k \in S_{c_k}$, and so we have that $r \notin app(br_i, S)$. We conclude that $head(r) \notin H_i$.

- $\mathbf{I} \models \bigwedge B^+(r_\ell)$ and $\mathbf{I} \models \bigwedge \neg.B^-(r_\ell)$ and $\mathbf{I} \models \bigvee H(r_\ell)$: for some $p_i \in B^+(r_\ell)$, $p_k \in B^-(r_\ell)$, we have $p_i \in \mathbf{I}$ and $p_k \notin \mathbf{I}$. Moreover, there exists a $p_h \in H(r_\ell)$ such that $p_h \in \mathbf{I}$. As the sets Σ_i are pairwise disjoint, all p_i, p_k, p_h are uniquely determined, i.e., for all of p_i, p_k, p_h we have $(c_i : p_i)$ in r $(1 \leq i \leq j)$, $(c_k : p_k)$ in r $(j+1 \leq k \leq m)$, and some $p_h \in H(s)$, where $s = head(r)$. By our construction of S, we obtain that all $p_i \in S_{c_i}$, and all $p_k \notin S_{c_k}$, thus $r \in app(br_i, S)$, and so is $head(r) \in H_i$. Since there exists a $p_h \in \mathbf{I}$, we obtain that $p_h \in S_i$, and so $S_i \models head(r)$.

To sum up, $S_i \models H_i$ and $S_i \models kb_i^{S_i}$, therefore (i) holds.

Part (ii): Assume that there is a model $T_i \subset S_i$ of $kb_i^{S_i} \cup H_i$.

We first show that (1) for every $a \in S_i \setminus T_i$, there exists a rule r such that $a \in H(r)$ and $H(r) \cap T_i = \emptyset$. Indeed, assume that no such rule exists; this means that for every rule r such that $a \in H(r)$, $H(r) \cap T_i \neq \emptyset$. Suppose that $H(r) = \{a, b, d_1, \ldots, d_k\}$ and $b \in T_i$. Now consider the singleton loop $\mathcal{L} = \{a\}$, it must hold that

$$\mathbf{I} \models a \supset \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge \bigwedge \neg.(H(r) \setminus \{a\})$$

In other words:

$$\mathbf{I} \models a \supset \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge (\neg b \wedge \neg d_1 \wedge \ldots \wedge \neg d_k) \tag{3.3}$$

Since $a \in S_i$, $b \in S_i$, and all signatures are pairwise disjoint, (3.3) does not hold, claim (1) follows.

Under this observation, we now construct a loop as follows. Pick $a \in S_i \setminus T_i$, and set $\mathcal{L} = \{a\}$. For all rules r where $a \in H(r)$ and $H(r) \cap T_i = \emptyset$,i.e., $T_i \not\models H(r)$, since $T_i \models r$, we have that $T_i \not\models B^+(r)$. If $B^+(r) = \emptyset$, r is a fact and we come to a contradiction; Part (ii) is proved. Otherwise, we collect all $b \in B^+(r)$ such that $b \notin T_i$ into $\mathcal{L}$. Since we work with a finite Herbrand base and never encounter a fact, we will eventually end up with $\mathcal{L} = \{a_1, \ldots, a_k\}$ in which all members depend on each other, a loop. Note that $\mathcal{L} \subset S_i$ and $\mathcal{L} \not\subseteq T_i$.

For this loop, we have that

$$\mathbf{I} \models \left(\bigvee \mathcal{L}\right) \supset \varepsilon(\mathcal{L}, ER(\mathcal{L}, kb_i) \cup SR(\mathcal{L}, \ell(br_i))) \tag{3.4}$$

Since $\mathcal{L} \subseteq S_i$, it must be the case that $\mathbf{I} \models \varepsilon(\mathcal{L}, ER(\mathcal{L}, kb_i) \cup SR(\mathcal{L}, \ell(br_i)))$. We try to find a rule r satisfying that $\mathbf{I} \models \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge \bigwedge \neg.(H(r) \setminus \mathcal{L})$. It can be either (a) $r \in ER(\mathcal{L}, kb_i)$ or (b) $r \in SR(\mathcal{L}, \ell(br_i))$.

Case (a): $r \in ER(\mathcal{L}, kb_i)$ means that $H(r) \cap \mathcal{L} \neq \emptyset$ and $B^+(r) \cap \mathcal{L} = \emptyset$. Similar to proving claim (1), one can show that $H(r) \cap T_i \neq \emptyset$, i.e., $T_i \not\models H(r)$; hence $T_i \not\models B^+(r)$. This means that there exists some $b \in B^+(r)$ such that $b \notin T_i$. By the construction of $\mathcal{L}$ and the fact that $H(r) \cap \mathcal{L} \neq \emptyset$, we have that $b \in \mathcal{L}$; therefore $B^+(r) \cap \mathcal{L} \neq \emptyset$, a contradiction.

Case (b): Similar to proving claim (1), one can show that $H(r) \cap T_i \neq \emptyset$; therefore, $H(r) \notin H_i$. On the other hand, since $\mathbf{I} \models \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge \bigwedge \neg.(H(r) \setminus \mathcal{L})$, we have that $\mathbf{I} \models \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r)$, which makes sure that $r \in app_i(br_i \cup S)$, thus $r \in H_i$, a contradiction to the previous observation.

We have shown that such a rule r satisfying that $\mathbf{I} \models \bigwedge B^+(r) \wedge \bigwedge \neg.B^-(r) \wedge \bigwedge \neg.(H(r) \setminus \mathcal{L})$ can not be found, hence $\mathbf{I}$ does not satisfy the loop formula wrt. $\mathcal{L}$. This completes our part of proving the minimality of S_i. Therefore, each S_i is an answer set of $kb_i \cup H_i$ and hence S is an equilibrium of M. □

Example 19 For M and M' from Example 17, we have

$$\pi(M) = (b \supset a) \wedge (a \supset b) \wedge (a \vee b \supset a \vee b), \text{ and}$$

$$\begin{aligned}\pi(M') = {} & (c \supset d) \wedge (d \supset c) \wedge (\neg c \supset d) \wedge (\neg d \supset c) \wedge \\ & (c \supset d \vee \neg d) \wedge (d \supset c \vee \neg c) \ \wedge (c \vee d \supset \neg c \vee \neg d) \ .\end{aligned}$$

Clearly, $\pi(M)$ has the models $\emptyset$ and $\{a, b\}$, while it can be checked that $\pi(M')$ has no model.

Example 20 Let us reconsider M from Example 11. We have

$\pi(C_1)$: $\kappa(C_1) = b \wedge c \supset a$ and $\lambda(C_1) = a \supset b \wedge c$

$\pi(C_2)$: $\kappa(C_2) = g \supset b$ and $\lambda(C_2) = b \supset g$

$\pi(C_3)$: $\kappa(C_3) = (d \supset c) \wedge (c \supset d) \wedge (\neg f \supset c \vee e)$ and

$$\begin{aligned}\lambda(C_3) = {} & (c \supset d \vee (\neg e \wedge \neg f)) \wedge (d \supset c) \wedge \\ & (c \vee d \supset (\neg e \wedge \neg f)) \wedge (e \supset (\neg f \wedge \neg c))\end{aligned}$$

$\pi(C_4)$: $\kappa(C_4) = f \vee g$ and $\lambda(C_4) = (f \supset \neg g) \wedge (g \supset \neg f)$.

The formula $\pi(M) = \pi(C_1) \wedge \cdots \wedge \pi(C_4)$ has three models, namely $\{a, b, c, d, g\}$, $\{b, e, g\}$, and $\{f\}$. They correspond to the three projected equilibria of M shown in Example 15.

Loop formulas for grounded equilibria

As we already saw in Section 2.4, equilibria lack groundedness in general, as cyclic bridge rules might be applied unfoundedly as for example $a \leftarrow (1 : b)$ and $b \leftarrow (1 : a)$ in Example 17. To overcome this, grounded equilibria were proposed in [19] for certain MCS's, in which bridge rules intuitively act under ASP semantics. Our transformation π can be adapted to capture grounded equilibria. We restrict here to *normal* ASP logics L_i, i.e., $\mathbf{KB}_i$ is the set of all normal (disjunction-free) ASP programs (this ensures a technical reducibility condition for L_i). Adapting Definitions 14 and 15, we define global loop formulas.

Definition 16 (Global Loop Formulas) *Let $\mathcal{L}$ be a loop of MCS $M = (C_1, \ldots, C_n)$. The* loop formula for $\mathcal{L}$ w.r.t. M *is*

$$\lambda(\mathcal{L}, M) = \left(\bigvee \mathcal{L}\right) \supset \varepsilon(\mathcal{L}, \bigcup_{i=1}^{n} ER(\mathcal{L}, kb_i \cup \ell(br_i))) \ ,$$

and the loop formula of M *is the conjunction $\lambda(M) = \bigwedge_{\mathcal{L}} \lambda(\mathcal{L}, M)$ for all loops $\mathcal{L}$ of M. Furthermore, we let*

$$\pi_{\mathbf{GE}}(M) = \lambda(M) \wedge \bigwedge_{i=1}^{n} (\kappa(kb_i) \wedge \kappa(\ell(br_i))) \ .$$

We then can show that $\pi_{\mathbf{GE}}$ captures grounded equilibria.

Theorem 10 *The grounded equilibria of any M with normal ASP logics correspond one-to-one to the models of $\pi_{\mathbf{GE}}(M)$.*

Proof (Sketch) Consider an MCS $M = (C_1, \ldots, C_n)$ and a belief state $S = (S_1, \ldots, S_n)$. Given that the logics at all contexts C_i are normal ASP, we have that $red_i(kb_i, S)_i$ are GL-reducts. Let $P = \bigcup_{i=1}^{n}(kb_i \cup \ell(br_i))$. One can show that the GL-reduct of P wrt. S is:

$$P^S = \bigcup_{i=1}^{n}(red_i(kb_i, S_i) \cup \ell(br_i)^S)$$

which is in one-to-one correspondence to $M^S = (C_1^S, \ldots, C_n^S)$, where $C_i^S = (L_i, red_i(kb_i, S_i), br_i^S)$. The only difference is that the renamed bridge rules $\ell(br_i)$ are used in P^S while the original ones are in M^S, but the correspondence is guaranteed due to the assumption of disjoint signatures between contexts.

On the other hand, the global loop formula $\pi_{\mathbf{GE}}(M)$ in Definition 16 is exactly the loop formula for P. Using P as an intermediate step to connect M and $\pi_{\mathbf{GE}}(M)$, one can show the one-to-one correspondence between the models of M and its global loop formula. □

Example 21 The MCS M in Example 17 has

$$\pi_{\mathbf{GE}}(M) = (a \supset b) \wedge (b \supset a) \wedge (a \vee b \supset \bot) \ ,$$

as the external support rules $ER(\{a, b\}, kb_1 \cup \ell(br_1)) = \emptyset$. The only model of $\pi_{\mathbf{GE}}(M)$ is $\emptyset$, which corresponds to the grounded equilibrium of M.

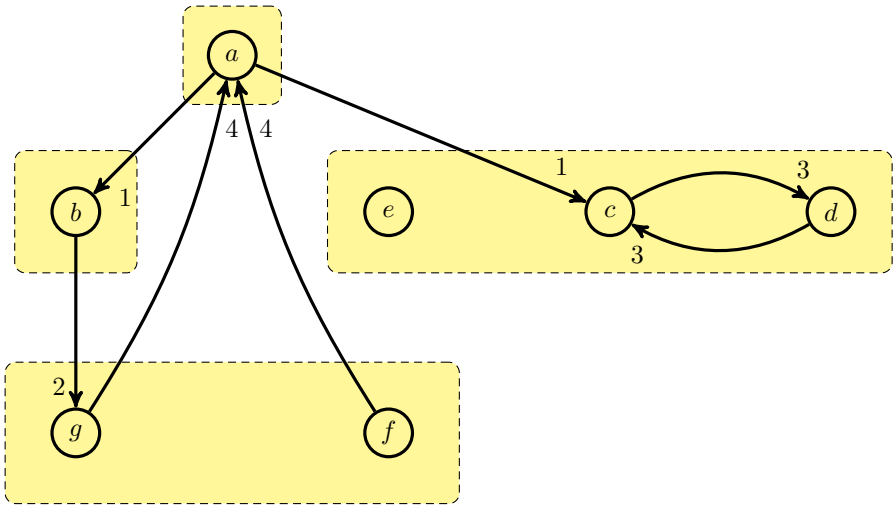


Figure 3.6: Dependency Graph of the MCS in Example 22

Example 22 Let us modify the MCS in Example 11 to M' so that $br_4 = \{f \vee g \leftarrow (1 : a)\}$ and $kb_4 = \emptyset$. This creates a global loop $\mathcal{L} = \{a, b, g\}$ as shown in Figure 3.6, and the corresponding loop formula is

$$\lambda(M') = a \vee b \vee g \supset \bot$$

since there is no external support rule for this loop. Furthermore, compared to Example 20, we have that $\kappa(C_4) = \top$ and $\lambda(C_4) = (f \supset a \wedge \neg g) \wedge (g \supset a \wedge \neg f)$. With these changes, the models of $\pi_{\mathbf{GE}}(M)$ are $\{c, d\}$ and $\{e\}$, which are in correspondence with the grounded equilibria of M': $(\emptyset, \emptyset, \{c, d\}, \emptyset)$ and $(\emptyset, \emptyset, \{e\}, \emptyset)$.

Note that DMCS cannot be run straightforwardly on $\pi_{\mathbf{GE}}(M)$, as the formulas in $\pi_{\mathbf{GE}}(M)$ are intermingled and prohibit a clear context separation. Intuitively, this is due to the encoded groundedness check. We can overcome this easily by extending M' for $\pi(M)$ above to $M'' = (C_0, C'_1, \ldots, C'_n)$, where $C_0 = (L_0, \pi_{\mathbf{GE}}(M), br_0)$ has propositional logic L_0 and $br_0 = \bigcup_{i=1}^{n} \{a \leftarrow (i : a) \mid a \in \mathcal{A}_i\}$; intuitively, C_0 filters out grounded equilibria. We then run DMCS on M'' at C_0.

Algorithm for SAT-based MCS

With the notion of loop formulas for MCS, bridge rules can be compiled away into the corresponding local theory. Hence, there is no need to evaluate bridge rules and one can make just a single call to lsolve, instead of invoking lsolve for every combined input from neighbor contexts. On the other hand, we have to pay the price of guessing for all bridge atoms in the original bridge rules in this single call, and later filter out unsatisfiable guesses by combining local belief sets with belief states returned from the neighbors.

In this section, we present an adaptation of Algorithm DMCS to the loop formulas-based setting. First, we replace in M each context C_i with $C'_i = (L'_i, \pi(C_i)', br')$, where L'_i is proposi-

Algorithm 3.3: DMCS$-\mathbf{SAT}(V, hist)$ at C_k

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

(a) **if** $\exists(V', \mathcal{S}) \in c(k)$ *such that* $V \subseteq V'$ **then return** $\mathcal{S}|_V$
$V' := V \cup \Sigma_k$
(b) $\mathbf{T} := \mathbf{SAT}(\pi(C_k)')$
$\mathcal{S} := \{(res_k(T,1), \ldots, res_k(T,n)) \mid T \in \mathbf{T}\}$
(c) **if** $k \notin hist$ **then**
  $\mathcal{T} := \{(\epsilon, \ldots, \epsilon)\}$ and $hist := hist \cup \{k\}$
(d) **foreach** $i \in In(k)$ **do**
    if *for some* $T \in \mathcal{T}$, $T_i = \epsilon$ **then**
      $\mathcal{T} := \mathcal{T} \bowtie C_i.\mathsf{DMCS}(V, hist)$
(e) $\mathcal{S} := \mathcal{S} \bowtie_V^k \mathcal{T}$
  update-cache$(c(k), V', \mathcal{S})$
return $\mathcal{S}|_V$

tional logic, $\pi(C_i)'$ is a renaming of $\pi(C_i)$ such that variables in different contexts are disjoint, and br' contains the bridge rules $\{a_i \leftarrow (j : a_j); \neg a_i \leftarrow (j : \neg a_j)\}$ for every renamed original atom a occurring in both $\pi(C_i)$ and $\pi(C_j)$, $i \neq j$. Applying the algorithm to M', we obtain the equilibria of M.

This respective algorithm DMCS$-\mathbf{SAT}$, shown in Algorithm 3.3 implements exactly the above idea. It first checks for the cache in step (a) and returns if the partial belief states had been computed already. Otherwise, step (b) makes a call to a SAT solver with the loop formulas of the context. The algorithm then converts each SAT model from this call into a belief state shape and stores it in $\mathcal{S}$. Then, if no cycle is detected (step (c)), we make further calls to the neighbors and combine them in step (d). After that, the consistent belief states from neighbors are combined with the local guesses (step (e)) and the cache is updated. Note that when a cycle is detected, we do not need to perform any additional handling, as step (b) already takes care of guessing for bridge atoms from the neighbor contexts. This algorithm uses the following primitives:

- $res_k(A, i) = \begin{cases} A|_{\Sigma_i} & \text{if } i \in In(k) \cup \{k\} \\ \epsilon & \text{otherwise,} \end{cases}$

  which restricts a belief set A over Σ to a belief set $A' \in \mathbf{BS}_i$ s.t. i is in the import neighborhood of C_k or $i = k$.

- For combining partial belief states $S = (S_1, \ldots, S_n)$ and $T = (T_1, \ldots, T_n)$, we define their *join* $S \bowtie_V^k T$ as the partial belief state $(U_1, \ldots, U_n)$ with (i) $U_k = S_k$, if $T_k = \epsilon \vee S_k = T_k|_V$, and (ii) $U_i = S_i$, $i \neq k$, if $T_i = \epsilon \vee S_i = T_i$, and (iii) $U_i = T_i$, if $T_i \neq \epsilon \wedge S_i = \epsilon$,

for all $1 \leq i \leq n$. Note that $S \bowtie^k_V T$ is void, if some S_i, T_i are from $\mathbf{BS}_i$ but different. The *join* of two sets $\mathcal{S}$ and $\mathcal{T}$ of partial belief states is then naturally defined as

$$\mathcal{S} \bowtie^k_V \mathcal{T} = \{S \bowtie^k_V T \mid S \in \mathcal{S}, T \in \mathcal{T}\}.$$

Theorem 9 may be generalized to contexts with extensions of ASP logics that have loop formula characterizations, like those in [48,66]. Furthermore, we have developed such a characterization for modular logic programs [30], which feature modules akin to imperative programs and have increased expressiveness.

Note that this loop formula characterization of equilibria may lead in the worst case to an exponential blow-up in the size of the MCS. This is not surprising, as standard loop formulas [48, 68, 76] also face this situation, and [75] show that this unavoidable, under the widely believed assumption from computational complexity theory that polynomial time computations cannot be simulated with small propositional formulas. A remedy would be to encode bridge rules in answer set programs. Note however, that some ASP solvers like ASSAT rely internally on loop formulas and SAT solving techniques for model search; thus the expected performance gain from a short ASP encoding might not always surface in practice.

CHAPTER 4

Topology-based Optimized Algorithm

In Chapter 3, we introduced a generic algorithm DMCS to compute partial equilibria of an MCS under an assumption of the availability of solvers at each local context. As a basic version, we did not take into account any further metadata rather than the minimal information that a context must know: the interface with every neighboring context. As one can see in Chapter 8, experiments for an instantiation of DMCS with answer set programming contexts revealed some 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 each neighbor in a generic way. Specifically, cyclic dependencies remain undetected until a context, seeing the invocation chain, requests models from a context in the chain. Furthermore, a context C_k does not know whether a neighbor C_i already requests models from another neighbor C_j which then would be passed to C_k; hence, C_k makes possibly a superfluous request to C_j.

(2) a context C_i returns the combination of its local models with the models received from all neighboring contexts. As contexts may have multiple models, the number of models can become huge as the size of the system respectively neighbors increases. In fact, this is one of the main performance obstacles.

In this Chapter, we address the issue of optimization; there is an urgent need for this in order 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 multi-context systems. Focusing on (1), we describe a decomposition method using biconnected components of inter-context dependencies. Based on this we can break cycles and prune acyclic parts before evaluating the system and create an acyclic query plan. To address (2), we foster a partial view of the system, which is often sufficient to reach a satisfactory answer. This way we can make 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 the bare minimum such that they continue to be meaningful. In this manner, we can omit needless information and circumvent excessive model combinations.

The Chapter will begin with motivating examples for the optimizing techniques, which are presented in details in Section 4.2. And finally, Section 4.3 describes algorithm DMCSOPT, which intertwines decomposition and pruning with variable projection to gain some performance for MCS evaluation.

4.1 Motivating Scenario

We first present a scenario in Example 23 as a running example for this Chapter. The corresponding encoding is given in Example 24. Then, Example 25 analyzes some observations as the first hints to our optimization techniques.

Example 23 (Scientists Group) A group of four scientists, Alice, Bob, Charlie, and Deni, just finished their conference visit and are now arranging a trip back home. They can choose between going by train or by car (which is usually slower than the train); and if they use the train, they should bring along some food. Moreover, Charlie and Deni have additional information from home that might affect their decision.

Charlie has a daughter, Fiona. He is fine with either transportation option, but if Fiona is sick then he wants to use the fastest transport to get home. Deni just got married, and her husband, Eddie, wants her to come back as soon as possible. He urges her to try to come home even sooner, while Deni tries to yield to her husband's plea.

If they go by train, Charlie is responsible for buying provisions. He might choose either salad or peanuts. The options for beverages are coke or juice. Bob is a modest person as long as he gets home. He agrees to any choice that Charlie and Deni select for vehicle but he dislikes coke. Alice is the leader of the group and prefers to go by car, but if Bob and Charlie want to go by train then she would not object. A problem is that Alice is allergic to nuts.

Charlie and Deni do not want to bother the group with their personal circumstances and communicate just their preferences, which is sufficient for reaching an agreement. Alice decides which option to take based on the information she gets from Bob and Charlie.

For further references, we map the name Alice, Bob, and so on to 1, 2, . . ., 6, respectively.

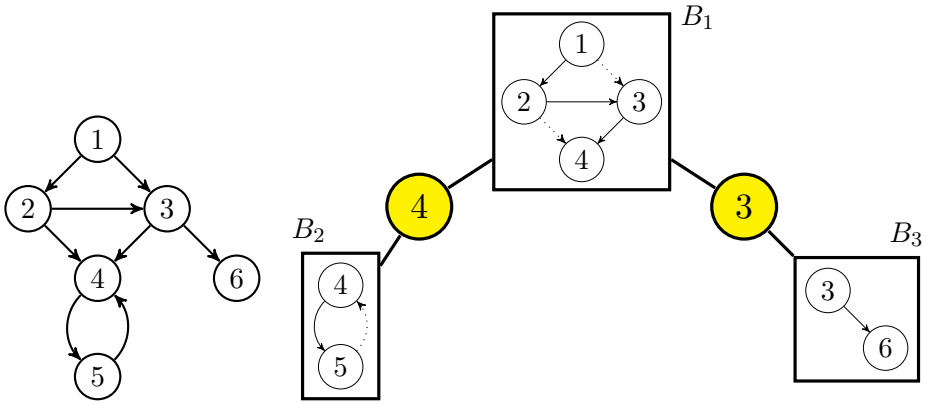
An important note is that similar scenarios have already been investigated in the realm of multi-agent systems (see, e.g., [22] on social answer set programming). We do not aim at introducing a new semantics for such scenarios; our example is meant to be a plain showcase application of MCS. We stress that MCS have potential as a host for KR formalisms, just like answer set programs have; however, here we concentrate on efficient MCS evaluation.

Example 24 The scenario in Example 23 can be encoded as an MCS $M = (C_1, \ldots, C_6)$, where all L_i are ASP logics and

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

$$C_2: \quad kb_2 = \{\bot \leftarrow \text{not } car_2, \text{not } train_2.\}$$

$$br_2 = \left\{ \begin{array}{l} car_2 \leftarrow (3 : car_3), (4 : car_4). \\ train_2 \leftarrow (3 : train_3), (4 : train_4), \text{not } (3 : coke_3). \end{array} \right\}$$



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

Figure 4.1: Topologies and Decomposition of Scientist Group Example

$$C_3: \quad kb_3 = \left\{ \begin{array}{rr} car_3 \vee train_3. \leftarrow & train_3 \leftarrow urgent_3. \\ salad_3 \vee peanuts_3 \leftarrow train_3. & coke_3 \vee juice_3 \leftarrow train_3 \end{array} \right\}$$

$$br_3 = \left\{ \begin{array}{c} urgent_3 \leftarrow (6 : sick_6). \\ train_3 \leftarrow (4 : train_4) \end{array} \right\}$$

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

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

$$C_6: \quad kb_6 = \{sick_6 \vee fit_6 \leftarrow\} \text{ and } br_6 = \emptyset$$

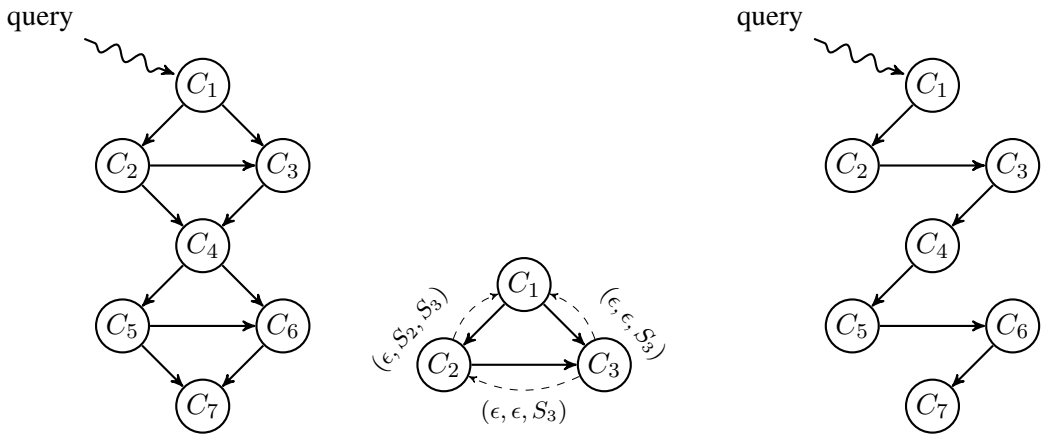
The context dependencies of M are shown in Fig. 4.1a. M has three equilibria, namely:

– $(\{train_1\}, \{train_2\}, \{train_3, urgent_3, juice_3, salad_3\}, \{train_4\}, \{soon_5, sooner_5\}, \{sick_6\})$;

– $(\{train_1\}, \{train_2\}, \{train_3, juice_3, salad_3\}, \{train_4\}, \{soon_5, sooner_5\}, \{fit_6\})$; and

– $(\{car_1\}, \{car_2\}, \{car_3\}, \{car_4\}, \emptyset, \{fit_6\})$.

Example 25 Consider an MCS $M = (C_1, \ldots, C_7)$ with dependencies between contexts as outlined in Figure 4.2a. 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 from C_5 and C_6 when returning to an invocation from C_2 or C_3. However, when C_1 calls C_2 (or C_3), the invoked context must carry local belief sets from C_4 in its answers to C_1. The reason is that belief sets from C_4 can cause inconsistent joins at C_1 for partial belief states returned from C_2 and C_3, while those from C_5 to C_7 only directly contribute to computing local belief sets at C_4. Note that all belief sets from 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 4.2: Topology of Example 25 (two stacked zig-zag diamonds)

Now, if we just take a subset of the system including C_1, C_2, and C_3 into account, assuming that C_1 has bridge rules with atoms of form $(2 : p_2)$ 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. 4.2b). A straightforward approach to evaluate this modified MCS is to ask in 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 that there is some evident redundancy in this approach, as C_3 will need to compute its belief sets twice. Simple caching strategies could mellow out the second belief state building in C_3; nonetheless, when C_1 asks C_3, the context will transmit back its belief states, thus consuming network resources.

Moreover, when C_2 asks for the partial equilibria of C_3, it will receive a set of partial equilibria that covers the belief sets of C_3 and in addition all contexts in the import closure $IC(3)$. This is excessive from the view of C_1, as it only needs to know the truth of $(2 : p_2)$ 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), then C_1 has no chance to 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 in the above Section, we present an optimization strategy which 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 information in transmissions.

Graph-theoretic concepts

We start with defining the topology of an MCS.

Definition 17 (Topology) *The* topology *of an MCS $M = (C_1, \ldots, C_n)$ is the directed graph $G_M = (V, E)$, where $V = \{1, \ldots, n\}$ 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 [17]); graphs are directed by default.

For any graph G and set $S \subseteq E(G)$ of edges, we denote by $G \backslash S$ the maximal subgraph of G that has no edges from S. For a vertex $v \in V(G)$, we denote by $G \backslash v$ the subgraph of G induced by $V(G) \backslash \{v\}$.

A graph is weakly connected if replacing every directed edge by an undirected edge 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. 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 rooted tree for any weakly connected graph G; for arbitrary graphs, it is a forest.

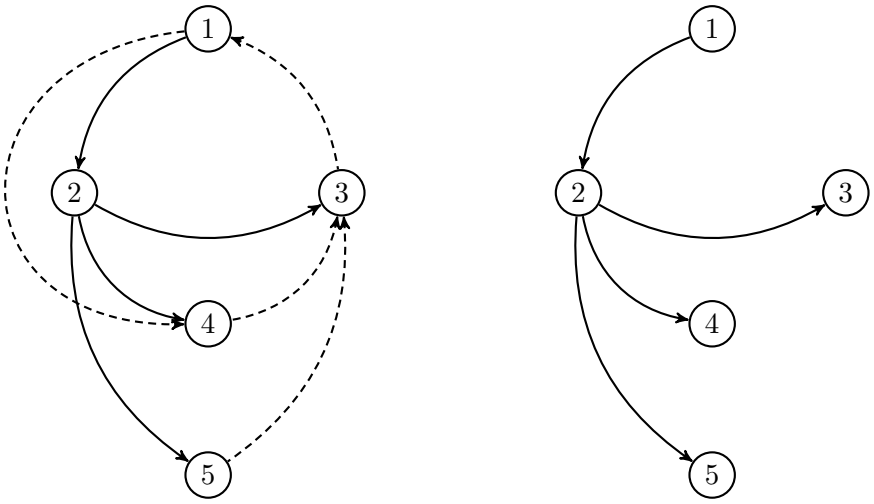
Example 26 Consider the graph in Figure 4.2a. One can check that 4 is the only cut vertex and there are two blocks of this graph: the subgraphs induced by $\{1, 2, 3, 4\}$ and $\{4, 5, 6, 7\}$.

The next example shows the block tree of our scenario in Example 23.

Example 27 The topology G_M of M in Example 24 is shown in Figure 4.1a. It has two cut vertices, namely 3 and 4; thus the block tree $T(G_M)$ (Figure 4.1b) 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.

Pruning

In acyclic topologies, like the triangle presented in the previous section, we can exploit a minimal graph representation to avoid unnecessary calls between contexts, namely, the transitive reduction of the graph G_M. Recall that the *transitive reduction* of a directed graph G is the graph G^- with the smallest set of edges (with respect to set inclusion) whose transitive closure



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

Figure 4.3: Ear Decomposition Example

equals the one of G. Note that G^- is unique if G is acyclic. For instance, the graph in Figure 4.2c is the unique transitive reduction of the one in Figure 4.2a.

Another essential part of our optimization strategy is to break cycles by removing edges from topologies. To this end, we use ear decompositions of cyclic graphs. A block may have multiple cycles which are not necessarily strongly connected, thus we first decompose cyclic blocks into their strongly connected components. The topological sort of these components 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) from P_0 and the last edge (ℓ_i, t_i) from each P_i, $i > 0$.

Example 28 Take, as an example, a strongly connected graph G in Figure 4.3a. An ear decomposition of G rooted at node 1 is $P = \langle P_0, P_1, P_2, P_3 \rangle$ where

– $V_{P_0} = \{1, 2, 3\}$, $E_{P_0} = \{(1, 2), (2, 3), (3, 1)\}$

– $V_{P_1} = \{2, 4, 3\}$, $E_{P_1} = \{(2, 4), (4, 3)\}$

– $V_{P_2} = \{2, 5, 3\}$, $E_{P_2} = \{(2, 5), (5, 3)\}$

– $V_{P_3} = \{1, 4\}$, $E_{P_4} = \{(1, 4)\}$

The last edges of P_i are dashed. Together 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 4.3b.

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

Example 29 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. 4.1b shows the final result (dotted edges are removed).

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 $V(B^-)$ that extends B^-, one can evaluate the respective contexts in reverse order for computing partial equilibria 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 any iterative Step i, $\mathcal{T}_{i-1}$ is updated by joining it with the local belief sets $\mathcal{S}_i$ of the context under consideration. Given $\mathcal{T}_k$ (after updating with $\mathcal{S}_k$) for context C_k, it holds that $\mathcal{T}_k|_{V^*(k)}$ is the set of partial equilibria 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 in a context w.r.t. a particular neighbor, and a given (sub-)graph.

Definition 18 *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* $V^*(i, j)_G = \{V^*(i) \cap \bigcup_{\ell \in G|_j} \Sigma_\ell\}$ *where $G|_j$ contains all nodes in G reachable from j.*

Example 30 Consider the MCS in Example 24, we have that the recursive import interface $V^*(1) = \{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4, sooner_5, sick_6\}$.

When we take into account just the block B_1, the refined recursive import interface $V^*(1, 2)_{B_1^-}$ can be archieved by removing bridge atoms from contexts that belong to other blocks B_2 and B_3, which results in $\{train_2, train_3, peanuts_3, car_3, coke_3, car_4, train_4\}$

Intuitively, if a context is a cut vertex c in G_M, one can drop all entries S_i ($i \neq c$) from the partial belief states computed at c, and pass this result to the parent block of c in $T(G_M)$, without compromising the computation of compatible (restricted) belief sets at the parent. Recursive import interfaces w.r.t. blocks in G_M reflect this property, which can be exploited for minimizing the information transmitted.

Algorithms

Algorithms 4.1 and 4.2 combine the optimization techniques outlined above. Intuitively, algorithm OptimizeTree takes a block tree T as input together with parent cut vertex c_p and root cut vertex c_r. It traverses T in a DFS-way and calls OptimizeBlock on every block. The result of the latter calls are removed edges F; after all blocks have been processed, the final result of OptimizeTree is a pair of all edges removed from blocks in T, and a labeling v for the remaining edges. OptimizeBlock takes a graph G and calls subroutine 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 (or if G was already acyclic), OptimizeBlock computes the transitive reduction G^-. All edges removed from G are returned. OptimizeTree continues computing the labeling v for the remaining edges, building on the recursive import interface, but keeping relevant interface variables of child cut vertices and removed edges. It can be shown that:

Proposition 11 *For any context C_k in an MCS 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 $v(i, j) \supseteq V^*(i, j)_B$.

Proof Item (i) is trivial to see since CycleBreaker is applied in Algorithm 4.2. 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 4.1:

- (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 $V^*(c_p)|_{\Sigma_t}$.
- (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 $V^*(i, j)_{B'}$.

This argument gives us property (ii). □

The following proposition shows the computational complexity of our algorithm.

Proposition 12 *For any context C_k in an MCS M, OptimizeTree$(T(G_M), k, k)$ runs in time polynomial (quadratic) in the size of $T(G_M)$ resp. G_M.*

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

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

On the one hand, the refined recursive import $V^*(i,j)'_B$ is defined as (Definition 18):

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

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

On the other hand, since all signatures disjoint, we have that

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

Since the recursive import interface for a node k is defined as $V^*(k) = \bigcup_{i \in IC(k)} V(i)$, the expression to compute $v(i,j)$ is in the end a combination of set intersection, union, and projection. With an implemetation 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 size of the signatures of contexts in the system.

Given G_M, the block tree graph $T(G_M)$ can be constructed in linear time [94]. Ear-decomposition (Step (c)) can also be done in linear time [93]. 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:

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

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. □

Given the topology of an MCS, we need to represent a stripped version of it which contains both the minimal dependencies between contexts and interface variables 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 19 (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 20 (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$(T(G_M), k, k)$.

Algorithm 4.1: 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
    $v(i,j) := V^*(i,j)_{B'} \cup \bigcup_{c \in \mathcal{C}'} V^*(c_p)|_{\Sigma_c} \cup \bigcup_{(\ell,t) \in E} V^*(c_p)|_{\Sigma_t}$
  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)

Algorithm 4.2: OptimizeBlock$(G\colon$ graph$, r\colon$ context id$)$

$F := \emptyset$
if G *is cyclic* **then**
(c) $F :=$ 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

In the next section, we show how to use Π_k for MCS evaluation. As it is clear from the context that we will use effective query plans, we omit the word effective and use query plans instead.

4.3 Evaluation with Query Plans

Given an MCS M and a starting context C_k, we aim at finding all projected partial equilibria of M w.r.t. C_k in a distributed way. To this end, we design an algorithm called DMCSOPT that is based on the algorithm DMCS in [31], but exploits properties of the optimization techniques described above. As a by-product, we obtain a simplification, because explicit cycle breaking is not needed. At each context node, an instance of DMCSOPT runs independently and commu-

nicates with other instances for exchanging sets of partial belief states. This provides a method for distributed model building, such that DMCSOPT can be deployed to any MCS where appropriate solvers for the respective context logics are available. The main feature of DMCSOPT is that it computes projected partial equilibria based on a query plan. This can be exploited for specific tasks like, e.g., local query answering or consistency checking. When computing projected partial equilibria, 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 idea is as follows: we start with context C_k and traverse a given query plan by expanding the outgoing edges of that plan at each context, like in a depth-first search, until a leaf context is reached. A leaf context C_i simply computes its local belief sets, transforms all belief sets into partial belief states, and returns this result to its parent. If the leaf C_i contains $(j : p)$ in bodies of bridge rules such that there is no context C_j to visit 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 4.3 shows our distributed algorithm, DMCSOPT, with its instance at a context C_k that runs in a background process (or daemon in Unix). 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 a cache $cache(k)$ at C_k, which is kept persistent by the background process. It uses the following helper functions:

- C_i.DMCSOPT(c): send id c to DMCSOPT at context C_i and wait for its result.
- guess(V): guess all possible truth assignments for the interface variables V.
- lsolve(S) (Algorithm (c)): 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 4.3 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 had been removed from cycles);

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

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

Algorithm 4.3: 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** $\mathcal{T} := \mathcal{T} \bowtie C_i.\mathsf{DMCSOPT}(k)$ // 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**
    $\mathcal{T} := \mathsf{guess}(v(c, k)) \bowtie \mathcal{T}$ // guess for removed dependencies in Π_r
  $\mathcal{S} := \emptyset$
(d) **foreach** $T \in \mathcal{T}$ **do** $\mathcal{S} := \mathcal{S} \cup \mathsf{lsolve}(T)$ // 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 $\mathcal{S}$

Theorem 14 shows that DMCSOPT is sound and complete. To prove it, we need Proposition 13 to claim that partial equilibria returned from DMCS and DMCSOPT are in correspondence. But first, we need the following supportive notion.

Definition 21 *Let C_k be a context of an MCS M, and let Π_k be the query plan as in Definition 20. For each block B of Π_k, the* block interface *of B, whose root vertex is c_B, is*

$$V_B = \{p \in v(i, j) \mid (i, j) \in E(B)\} \cup \Sigma_{c_B}.$$

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

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

Proposition 13 *Let C_k be a context of an MCS M, let Π_k be the query plan as in Definition 20 in which C_k belongs to block B of Π_k and let $V = \bigcup_{B \in \Pi_k} V_B$. 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$(V, \emptyset)$ such that $S' = S|_{V^*(c,k)_B}$ if $(c, k) \in E(\Pi_k)$ or $S' = S|_{V^*(k)_B}$ if $c = k$;

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

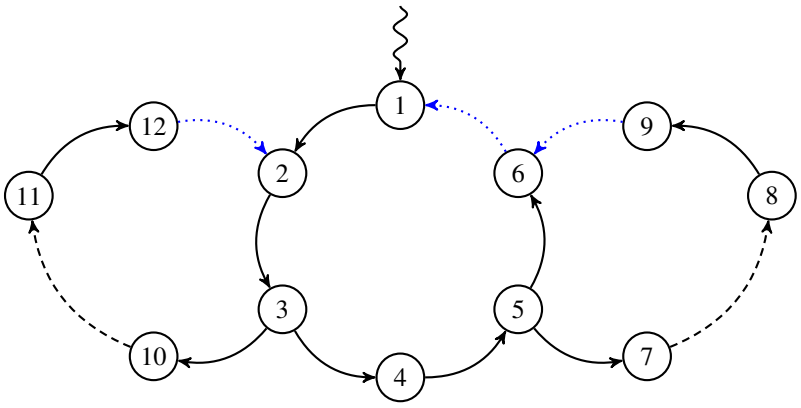


Figure 4.4: Possible cycle breakings

For the detailed proof of Proposition 13, we refer the reader to Section 4.4.

Theorem 14 *Let C_k be a context of an MCS M, let Π_k be the query plan as in Definition 20 and let $\widehat{V} = \{p \in v(k,j) \mid (k,j) \in E(\Pi_k)\}$. Then,*

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

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

Proof (i) Let $S' \in C_k.\mathsf{DMCSOPT}(k)$ be a result from DMCSOPT. By Proposition 13 (i) for $c = k$, there exists an $S'' \in C_k.\mathsf{DMCS}(V, \emptyset)$ such that $S' = S''|_{V^*(k)_B}$, where we choose $V = \bigcup_{B \in \Pi_k} V_B$. Note that $V^*(k) \subseteq V$ as V collects all bridge atoms from all blocks, which might contain blocks not reachable from k. By Theorem 3, there exists a partial equilibrium S of M such that $S'' = S|_V$. Thus, we have that

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

(ii) Let S be a partial equilibrium of M. By Theorem 3, there exists $S'' \in C_k.\mathsf{DMCS}(V, \emptyset)$ such that $S'' = S|_V$ where we choose $V = \bigcup_{B \in \Pi_k} V_B$. As above, $V^*(k) \subseteq V$. By Proposition 13 (ii) for $c = k$, there exists $S' \in C_k.\mathsf{DMCSOPT}(k)$ such that $S' = S''|_{V^*(k)_B}$. As above, we have that $S' = S|_{\widehat{V}}$. □

4.4 Proof of Proposition 13

To support the proof of Proposition 13, we need the following lemmas.

Lemma 15 *Assume that the import neighborhood of context C_k is $In(k) = \{i_1, \ldots, i_m\}$, no (k, i_j) is removed from the original topology by* OptimizeBlock(B, c_B)*, and*

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

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

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 $T' = T|_{V_{input}(1,m)}$ with $V_{input}(\ell_1, \ell_2) = \bigcup_{j=\ell_1}^{\ell_2} V^*(k, i_j)_B$.

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 $U' = U|_{V_{input}(1,\ell-1)}$. 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 $T' = T|_{V_{input}(1,\ell)}$.

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 $U' = U|_{V_{input}(1,\ell-1)}$ and $S' = S|_{V^*(k,i_\ell)_B}$, 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 $V_{input}(1, \ell-1) \cap \Sigma_t \neq \emptyset$ and $V^*(k, i_\ell) \cap \Sigma_t \neq \emptyset$.

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

However, from Definition 18 of recursive import interface, we have that $V^*(k, i_x)_B = V^*(k) \cap \bigcup_{\ell \in B|_k} \Sigma_\ell$ where $B|_{i_x}$ contains all nodes in B reachable from i_x, it follows that $V^*(k, i_\ell)$ and $V^*(k, i_j)$ for any $1 \leq j \leq \ell - 1$ that reaches t, share the same projection to Σ_t, hence $V_{input}(1, \ell-1) \cap \Sigma_t = V^*(k, i_\ell) \cap \Sigma_t$.

We reach a contradiction, and therefore Lemma 15 is proved. □

Lemma 16 *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) $S \bowtie (T \bowtie U) = (S \bowtie T) \bowtie U$.

$S_i = \epsilon$	$T_i = \epsilon$	$U_i = \epsilon$	$S_i = T_i$	$T_i = U_i$	$U_i = S_i$	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 4.1: Possible cases when joining at position ith

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 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 4.1. 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 17 *Let C_i and C_j be two contexts in M such that they are in the same block after* OptimizeTree *and there is a directed path from C_i to C_j, and that*

$\mathcal{S}^i$ = DMCSOPT(k) *at* C_i;
$\mathcal{S}^j$ = 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 17 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 (16), (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 $\mathcal{S}^{p_\ell} = \mathsf{lsolve}(\mathcal{T})$[1] 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 (16), (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 17, we have the following result.

Lemma 18 *Assume that the import neighborhood of context C_k is $In(k) = \{i_1, \ldots, i_m\}$, and*

$$\begin{aligned} \mathcal{S}^{i_1} &= \textit{DMCSOPT}(k) \textit{ at } C_{i_1} \\ &\vdots \\ \mathcal{S}^{i_m} &= \textit{DMCSOPT}(k) \textit{ at } C_{i_m}. \end{aligned}$$

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 it holds that $\mathcal{S}^{i_\ell} = \mathcal{S}^{i_\ell} \bowtie \mathcal{S}^{i_j}$. 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 directed children of C_k, it follows that they belong to the same block. Therefore, by Lemma 17 we have that $\mathcal{S}^{i_\ell} = \mathcal{S}^{i_\ell} \bowtie \mathcal{S}^{i_j}$. □

Proof (Proposition 13) 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 $V = V_B$.

Base case: C_k is a leaf. Then we now compare a call DMCSOPT(k) at C_k and C_k.DMCS$(V, \emptyset)$, where $V = V^*(k)_B = \Sigma_k$. Algorithm 3.1 returns local belief sets of C_k projected to V and Algorithm 4.3 returns plain local belief sets, the claim follows as $V = V^*(k)_B = \Sigma_k$.

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

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

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

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

There are two cases. First, no edge (k, i_j) is removed by the optimization procedure. Then, by Lemma 15, 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 18 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 4.4, assuming that context C_1 gets called:

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

(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 why 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 4.3 on $v(k, i_j)$, which includes $V^*(c_B)|_{\Sigma_{i_j}} = V_B \cap \Sigma_{i_j}$. On the other hand, DMCS will recognize the cycle at C_{i_j} and issue a guess on $V_B \cap \Sigma_{i_j}$ at Step (c) of Algorithm 3.1. 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 4.4, this is node 5.

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

Now that Proposition 13 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 $V = \bigcup_{B \in \Pi_k} V_B$, 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 $V^*(k)_B$ are in the end the same. □

CHAPTER 5

Streaming Algorithm

In Chapter 4, we presented optimized algorithms for preprocessing MCSs, which take the topology information of an MCS into account and first partition the global systems into a tree of blocks. Inside each block, we apply advanced decomposition techniques, namely ear decomposition and transitive reduction. The former aims at removing cycles from the original topology, making it easier for communicating between contexts when computing equilibria. The latter, on the other hand, takes as input the acyclic topology and cuts off as many connections as possible while guaranteeing reachability, to minimize communication. As a side effect, the interface between contexts must be updated accordingly to compensate the deleted connection links. Moreover, the decomposition of the whole topology into a block tree helps to keep local information within each block and therefore reduces a significant amount of information transferred in the global system.

This preprocessing builds up a query plan which is then used as input for our new algorithm DMCSOPT for evaluating equilibria of an MCSs. As one can see in Chapter 8, DMCSOPT shows substantial improvements compared to the basic algorithm DMCS regarding the number of contexts in an MCS it can handle. However, the experimental results also reveal limitations that DMCSOPT encountered, i.e., when the sizes of the local signatures at each context and the interface between contexts increase, DMCSOPT also suffers from bottlenecks.

These limitations stem from the way in which models are exchanged between contexts. For example, suppose context C_1 accesses information from several other contexts $C_2, \ldots, C_m$, called its neighbors. Consider a simple setting where the information flow is acyclic, meaning that none of the neighbors (directly or indirectly) accesses information from C_1. Furthermore, assume that $n_2, \ldots, n_m$ are the numbers of partial equilibria that exist at the neighbors, respectively. Intuitively, a partial equilibrium at a context is an equilibrium of the subsystem induced by information access. By the current approach for distributed evaluation, all the partial equilibria are returned to the parent context C_1.

Before any local model computation can take place at the parent, it needs to join, i.e., to properly combine the partial equilibria obtained from its neighbors. This may result in $n_2 \times n_3 \times \cdots \times n_m$ partial models to be constructed (each one providing a different input for local

model computation) which may not only require considerable computation time but also exhaust memory resources. In fact, memory can be exhausted before local model computation at C_1 has even been initiated, i.e., before any (partial) equilibrium is obtained.

Note however that if instead of the above procedure each neighbor would transfer back just a portion of its partial equilibria, 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 models as it 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 chapter, we pursue the idea sketched above and turn it into a concrete algorithm for computing partial equilibria of a distributed MCS in a streaming fashion. 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;
- upon invocation 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.

Note that as this chapter mainly studies the streaming aspect of the algorithm, we simplify the presentation by not mentioning 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 in this chapter 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.

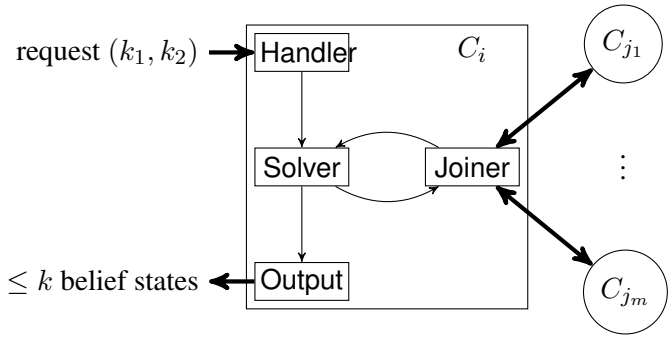


Figure 5.1: DMCS-STREAMING Architecture

5.1 Basic Streaming Procedure

Given an MCS M, a starting (root) context C_r, and an integer k, we aim at finding at most k partial equilibria of M w.r.t. C_r in a *distributed* and *streaming* way. While the aspect of distributedness has been investigated in the DMCS and DMCSOPT algorithms, adding the streaming aspect is not easy as one needs to take care of the communication between contexts in a nontrivial way. In this section, we present our algorithm DMCS-STREAMING which allows for gradual streaming of partial equilibria between contexts. This section describes a basic version of the algorithm, which concentrates on transferring packages of k equilibria with one return message. The system design is extendable to a parallel version, whose idea is discussed in Section 5.2.

In DMCSOPT, the reply to a request of a parent contains *all* partial equilibria from one context. This means that communication between contexts is synchronous—one request gets exactly one answer. While this is the easiest way to send solutions, it is very ineffective with larger MCS instances, as a small increase in the size of the alphabet may force the creation of many (partial) equilibria, which in turn may exceed memory limitations. The goal of this work is to develop an algorithm which allows for asynchronous communication for belief state exchange, i.e., one request for a bounded number of k (partial) equilibria may result in *at most* k solutions. This way we can restrict memory needs and evaluate multi-context systems that could not be handled by the algorithms in Chapter 4.

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 partial equilibria. Each communication window of k partial equilibria ranges from the k_1-th partial equilibrium to the k_2-th $(= k_1 + k - 1)$. A parent context C_i requests from a child context C_j a pair (k_1, k_2), and then receives at a future time point a package of at most k partial equilibria. Receiving ϵ indicates that C_j has fewer than k_1 models.

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 5.1 for an architectural overview.

All components except Handler (shown in Algorithm 5.1) communicate using message queues: Joiner has j queues to store partial equilibria from j neighbors, Solver has one queue to

Algorithm 5.1: 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

Algorithm 5.2: 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** $S := \emptyset$
(b) **else** call Joiner and pop S from q

if $S = \epsilon$ **then** $count := k_2$

(c) **while** $count < k_2$ **do**
pick the next model $S^\star$ from lsolve(S)
if $S^\star \neq \epsilon$ **then**
push $S^\star$ to Output.q
$count := count + 1$
else break

refresh() and push ϵ to Output.q

hold joined equilibria from Joiner, and Output has a queue to carry results from Solver. As our purpose is 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 partial equilibria to be stored in the queue later. This property frees us from synchronization technicalities.

Algorithms 5.2 and 5.3 show how the two main components 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 all neighbors in this range return some models then store them in the respective queues and return $true$; otherwise, return $false$ as one of the neighbor is inconsistent.

- *get_next*(ℓ, k): pose a request asking for the next k equilibria from neighbor C_{c_ℓ}; if C_{c_ℓ} sends back some models, then store them in the queue q_ℓ and return $true$; otherwise, return $false$ as the neighbor already exhaustively returned its partial equilibria from the previous request. Note that this subroutine needs to keep track of which range has been already asked for to which neighbor by maintaining a set of counters. A counter wrt. a neighbor C_{c_ℓ} is initialized to 0 and is 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 arrives at the Handler, which contains the range (k_1, k_2). Then Handler notifies Solver to compute up to k_2 models, and Output to collect the ones in the requested 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 5.1).

When receiving a notification from Handler, Solver first prepares the input for its local solver. If C_i is a leaf context then the input S simply is the empty set assigned in Step (a); otherwise, Solver has to trigger Joiner (Step (b)) for input from neighbors. With input fed from neighbors, the subroutine Isolve is used in Step (c) to compute at most k_2 results and send them to the output queue.

The Joiner, 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 reaches its end and sends ϵ to Solver.

Note that while proceeding as above guarantees that no models are missed, it can in general lead to multiple considerations of combinations (inputs to Solver). Using a fixed size cache might mitigate these effects of recomputation, but since limitless 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 5.4). 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 31 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.

Algorithm 5.3: 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
```

For $i < 2^m$, the context $C_i = (L_i, kb_i, br_i)$ has,

$$kb_i = \{a_i^j \vee \neg a_i^j \leftarrow t_i \mid 1 \leq j \leq \ell\} \text{ and} \tag{5.1}$$
$$br_i = \left\{ \begin{array}{llll} t_i \leftarrow (2i : a_{2i}^1) & \cdots & t_i \leftarrow (2i : a_{2i}^\ell) \\ t_i \leftarrow (2i+1 : a_{2i+1}^1) & \cdots & t_i \leftarrow (2i+1 : a_{2i+1}^\ell) \end{array} \right\} ,$$

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

$$kb_i = \{a_i^j \vee \neg a_i^j \mid 1 \leq j \leq \ell\} \text{ and } br_i = \emptyset \ . \tag{5.2}$$

Intuitively, M is a binary tree-shaped MCS with depth m and $\ell + 1$ is the size of the alphabet in each context. Figure 5.2 shows such an MCS with $n = 7$ contexts and depth $m = 2$; the internal contexts have knowledge bases and bridge rules as in (31), while the leaf contexts are as in (5.1). 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 equilibrium of M using either DMCS or DMCSOPT, one needs to transfer packages of 2^ℓ partial equilibria from each context to its parent (because each context C_i

Algorithm 5.4: Output() at C_i

Data: Input queue: q, starting model: k_1, end model: k_2

$buf := \emptyset$ and $count := 0$
while $count < k_1$ **do**
  pick an S from Output.q
  if $S = \epsilon$ **then** $count := k_2 + 1$
  else $count := count + 1$

while $count < k_2 + 1$ **do**
  wait for an S from Output.q
  if $S = \epsilon$ **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

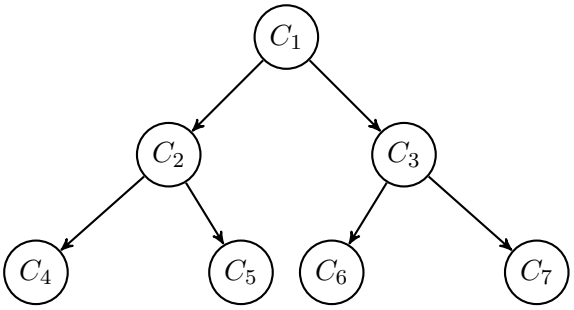


Figure 5.2: Binary tree MCS

computes all subsets of $\{a_i^1, \ldots, a_i^\ell\}$, where ℓ was chosen as an input of this Example). Each intermediate context receives two times 2^ℓ results from 2 children, have the join of $2^{2\ell}$ input to feed into lsolve. It then invokes lsolve these many times and only then can it return the whole 2^ℓ models to the parent. In the mean time, the corresponding parent has to wait for this input.

On the other hand, Algorithm DMCS-STREAMING only needs to transfer a single model between each pair of connected context, which is a significant saving. The following explanation gives the details. For simplicity, we choose $m = 1, \ell = 5$, i.e., $M = (C_1, C_2, C_3)$. Querying C_1 with a package size of $k = 1$ first causes the query to be forwarded to C_2 in terms of a pair $k_1 = k_2 = 1$. As a leaf context, C_2 invokes the local solver and eventually gets five different

models. However, it just returns one partial equilibrium back to C_1, e.g., $(\epsilon, \{a_2^1\}, \epsilon)$. Note that t_2 is projected away since it does not appear among the atoms of C_2 accessed in bridge rules of C_1. The same happens at C_3 and we assume that it returns $(\epsilon, \epsilon, \{a_3^2\})$ to C_1. At the root context C_1, the two single partial equilibria from its neighbors are consistently combined into $(\epsilon, \{a_2^1\}, \{a_3^2\})$. Taking this as an input to the local solving process, C_1 can eventually compute 5 answers, but in fact just returns one of them to the user, e.g., $S = (\{a_1^1, t_1\}, \{a_2^1\}, \{a_3^2\})$.

The following proposition shows the correctness of our algorithm.

Proposition 19 *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.*

Proof 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 4.3, respectively.

To prove the correctness of DMCS-STREAMING, we just need to show that the input to Isolve is complete in the sense that if Step (e) of Algorithm 5.3 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, Isolve always returns answers in a fixed order. We need to show that the accumulation of the join by Algorithm 5.3 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 5.1.

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

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

Table 5.1: Accumulation of Joiner

5.2 Parallelized Streaming

As the reader may have anticipated, 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.

Another possibility to circumvent this problem is parallelization. The idea is to serve each parent with a set of the Handler, Joiner, Solver and Output components. In this respect, the basic interaction between each unit is still as shown in Figure 5.1, with the notable difference that each component now runs in an *individual thread*. The significant change is that Solver does not control Joiner but rather waits at its queue to get new input for the local solving process. The Joiner independently queries the neighbors, combines partial equilibria from neighbors, and puts the results into the Solver queue.

The effect is that we do not waste recomputation time for unused models. However, in practice, unlimited parallelization also faces a similar problem of exhausting resources as observed in DMCSOPT. While DMCSOPT runs out of memory with instances whose local theories are large, unlimited parallel instances of the streaming algorithm can exceed the number of threads/processes that the operating system can support, e.g., in topologies that allow contexts to reach other contexts using alternative paths such as the diamond topology. In such situations, the number of threads generated is exponential in the number of pairs of connected contexts, which prohibits scaling to large system sizes.

A compromise between the two extreme approaches is to have a *bounded parallel* algorithm. The underlying idea is to create a fixed-size pool of multiple threads and components, and when incoming requests cannot be served with the available resources, the algorithm continues with the basic streaming procedure, i.e., to share computational resources (the components in the system) between different parents at the cost of recomputation and unused models. This approach is targeted for future work.

CHAPTER 6

Dynamic Multi-Context Systems

From Chapters 3 to 5, we have presented a series of algorithms for computing partial equilibria of a Multi-Context System with respect to a starting context. The common feature of all these algorithms is that they all operate on *static* systems, meaning that the interlinks between contexts are fixed before evaluation, by the *grounded* context identifiers specified in bridge rules.

However, a characteristics that comes with many distributed application scenarios is that the environment is open, at least to some extent, meaning that participating knowledge sources and their contents may change over time and are not known a priori. This is in contrast with the static nature of current MCS in the sense that participating contexts and the corresponding information exchange need to be fixed completely at design time. Thus, atoms in bridge rules always point to a particular belief from a concrete context. This is prohibitive to formalizing systems where part of the behavior is instantiated at run-time only.

In this Chapter, we address the above shortcoming of the MCS formalism concerning open environments of information exchange, that is when at design time the concrete knowledge sources participating in an information exchange are not known. Intuitively, what is needed to cope with such scenarios is a formalism for information exchange which is closer towards a peer-to-peer (P2P) approach, where so-called peers can at any time join or leave the system dynamically [1]. To this end, we present *Dynamic Nonmonotonic Multi-Context Systems*, which consist of schematic contexts that may leave some information interlinkage open at design time; this linkage is established by a configuration step at run time, in which concrete contexts and information imports between them are wired.

More specifically, this Chapter contributes the following:

- We formalize dynamic multi-context systems, which extend the MCS formalism with so-called *schematic bridge rules*. Intuitively, schematic bridge rules may contain place holders that can range over both context identifiers and beliefs. Their semantics is defined via suitable notions of substitution and binding, where a *context substitution* maps context holders to concrete contexts and a binding maps schematic belief atoms to adequate concrete beliefs. To take into account that a perfectly matching belief might not

exist, we use (unless exact substitution is forced) a 'similarity'-based binding of beliefs in which schematic beliefs are bound to 'similar' beliefs, which is assessed by a similarity function. To determine such beliefs, we foresee a matchmaking component as an oracle which returns on a call a list with similar beliefs. More precisely, it provides simple term substitutions according to an underlying similarity measure.

- We consider the problem of finding an instantiation of a dynamic multi-context system, starting from a specific context, i.e., a concrete "configuration" of the (open) system. To solve it, we first present a basic algorithm for computing configurations of dynamic MCS. As the number of configurations can be very large in general, we then consider different heuristics to generate 'good' ones, which take topological structure and/or different criteria of qualities of individual matches (bindings) into account. The algorithm is fully distributed, i.e., instances run at different contexts, and the configurations are found by local computations plus communication.

Using dynamic MCS, a broader range of application scenarios can be modeled which require the flexibility of taking changing context into account. In particular, group formation to satisfy information needs of heterogeneous components, with possible selection among different alternatives, can be readily expressed.

The remainder of this Chapter is structured as follows. The next Section presents a motivating scenario, which is used as a running example in this Chapter. Section 6.2 introduces basic notions for Dynamic MCS, where we provide their formal definition and semantics in terms of instantiation to ordinary MCS. Section 6.3 recaptures the idea of binding dynamic MCSs to original ones. Finally, Section 6.4 contains then the description of our basic distributed configuration algorithm and discussions of refinements such as, e.g., different heuristics to drive the configuration.

6.1 Motivating Scenario

We first present in this section a motivating scenario for dynamic MCS. Example 32 describes the scenario informally. Then, the encoding for local knowledge bases is shown in Example 33. Finally, Example 34 gives the encoding of a static setting of the scenario.

Example 32 At the beginning of each semester, students in a group (including Alice, Bob, and Carol) need to choose courses from their curriculum. For each possible course, the students have three possible decisions, namely *select*, *hesitate*, and *eliminate* (in decreasing order). Intuitively, there is a potential for selecting a course if one finds it interesting. However, if the lecturer is known to be hard to please, they fear that it might be tough (or impossible) to get good marks and potentially eliminate the course. If there are reasons for both selecting and eliminating—or none—they are then in the state of hesitation, which dominates the other two potential decisions.

Moreover, the final decision of each student is supported by the decisions of their friends. If some friend gives a positive (resp., negative) opinion about a particular course, and no other friend shares an opposite opinion, then the group will adjust their final decision accordingly.

According to this strategy, the students do not specify in advance for a course which friends they will consult. This depends on the friends they will meet at the course orientation meeting. While attending the orientation meeting and exchanging opinions, every student in the group finally comes up with a list of courses that conforms with the choices of their colleagues.

For example, Alice may believe that if Bob hesitates or selects a course, then this is a positive sign, because he is very cautious; on the other hand, if Bob eliminates a course, then this is a negative sign. But she has a different opinion about Carol's choice, namely she is encouraged only when Carol selects the course and is discouraged otherwise. Carol, who is a bit more careful, might only accept that Bob's selection (resp., elimination) of the course as a positive (resp., negative) hint, i.e., she has no bias when Bob is hesitating. Finally, Bob may interpret the opinions of the two girls in the same way as Carol does with his, i.e., mapping selections to be positive, elimination to be negative, and having no preference w.r.t. hesitance.

Regarding the course on Answer Set Programming, Bob finds it interesting, but Alice has the impression that the professor is very demanding; they ask Carol, who has no special opinion about it. One of the outcomes of the discussion is that Bob and Carol will select this course while Alice hesitates.

The current MCS setting is sufficient to formalize the last part of the discussion between Alice, Bob, and Carol (Example 34), but lacks dynamicity to formalize the general setting.

Example 33 We will now model Example 32 as an answer set program. Let R_i be a set of the following rules:

$$
\begin{array}{ll}
s_i \leftarrow ps_i, \text{not } h_i, \text{not } e_i & e_i \leftarrow pe_i, \text{not } h_i, \text{not } inc_i \\
s_i \leftarrow ph_i, \text{not } ps_i, \text{not } h_i, \text{not } e_i, inc_i & e_i \leftarrow ph_i, dec_i \\
h_i \leftarrow ps_i, \text{not } e_i, dec_i & ps_i \leftarrow inter_i \\
h_i \leftarrow ph_i, \text{not } inc_i, \text{not } dec_i & pe_i \leftarrow hprof_i \\
h_i \leftarrow pe_i, \text{not } ph_i, inc_i & ph_i \leftarrow ps_i, pe_i \\
 & ph_i \leftarrow \text{not } ps_i, \text{not } pe_i
\end{array}
$$

The atoms have the following meaning: s_i, h_i, and e_i stand for the three decisions: select, hesitate, and eliminate, respectively. Similarly, ps_i, ph_i, and pe_i stand for the potential to select, hesitate, and eliminate a course, resp. A course is interesting if $inter_i$ is true, and a professor is hard to please if $hprof_i$ is true. The atoms inc_i and dec_i mean that a student inclines and declines to select a course, respectively.

The program

$$P_1 = R_1 \cup \{hprof_1\}$$

has one answer set $\{e_1, pe_1, hprof_1\}$, the program

$$P_2 = R_2 \cup \{inter_2\}$$

noindent has the answer set $\{s_2, ps_2, inter_2\}$, while $P_3 = R_3$ has the answer set $\{h_3, ph_3\}$.

Intuitively, P_1 represents Alice's mind. She thinks that the professor is hard to please ($hprof_1$), hence she potentially eliminates (pe_1) the course and will eliminate it (e_1) if no more support information is provided. On the other hand, P_2 represents Bob's mind. He is really

interested in the course ($inter_2$) and selects it (s_2) based on his potential of selecting the course (ps_2). Carol, modeled by P_3, adds no personal view about the course. She is currently hesitating (h_3, ph_3) in taking the course; her final decision can change depending on decisions of other friends.

Example 34 Let $M' = (C_1, C_2, C_3)$ be an MCS such that all L_i are ASP logics, with alphabets $\mathcal{A}_i = \{s_i, h_i, e_i, ps_i, ph_i, pe_i, inter_i, hprof_i, inc_i, dec_i\}$, $1 \leq i \leq 3$. Suppose $kb_i = P_i$, with P_i taken from Example 33, and

$$br_1 = \left\{ \begin{array}{lcl} inc_1 & \leftarrow & (2:s_2), \text{not}\,(3:h_3), \text{not}\,(3:e_3), \text{not}\,(1:dec_1) \\ inc_1 & \leftarrow & (2:h_2), \text{not}\,(3:h_3), \text{not}\,(3:e_3), \text{not}\,(1:dec_1) \\ dec_1 & \leftarrow & (2:e_2), \text{not}\,(3:s_3), \text{not}\,(1:inc_1) \end{array} \right\},$$

$$br_2 = \left\{ \begin{array}{lcl} inc_2 & \leftarrow & (1:s_1), \text{not}\,(3:e_3), \text{not}\,(2:dec_2) \\ dec_2 & \leftarrow & (3:e_3), \text{not}\,(1:s_1), \text{not}\,(2:inc_2) \end{array} \right\}, \text{ and}$$

$$br_3 = \left\{ \begin{array}{lcl} inc_3 & \leftarrow & (2:s_2), \text{not}\,(1:e_1), \text{not}\,(3:dec_3) \\ dec_3 & \leftarrow & (1:e_1), \text{not}\,(2:s_2), \text{not}\,(3:inc_3) \end{array} \right\}$$

One can check that

$$S = (\{h_1, pe_1, hprof_1, inc_1\}, \{s_2, ps_2, inter_2\}, \{s_3, ph_3, inc_3\})$$

is an equilibrium of M'. Intuitively, M' models the discussion between Alice (C_1), Bob (C_2), and Carol (C_3). Comparing this to Example 33, the decision of Bob influences those of Alice and Carol, as Alice now hesitates about the course even though having the potential of eliminating it, while Carol decided to select the course although she was hesitating about it before.

Next, we turn to define Dynamic MCS, which can be used to formalize the general setting of the motivating scenario and similar situations satisfactorily.

6.2 Basic Notions for Dynamic Nonmonotonic Multi-Context Systems

Let $\mathcal{V}_{ctx}$ be a vocabulary of *context holders*,[1] and let $\Sigma = \bigcup \Sigma_i$ be a set of (possibly shared) *signatures*. Unless stated otherwise, elements from $\mathcal{V}_{ctx}$ (resp., Σ) are denoted with first letter in upper case (resp., lower case). Furthermore, we define the set $\Sigma_{@}$ (resp., $\Sigma_{\sim}$) of exact (resp., similar) *schematic beliefs* as the set of symbols $@[p]$ (resp., $[p]$) for all p in Σ. Let $bel(@[p]) = bel([p]) = p$ be a function for extracting the belief symbol from a schematic belief.

Definition 22 *A* dynamic multi-context system $M = \{C_1, \ldots, C_n\}$ *is a set of* schematic contexts $C_i = (L_i, kb_i, sbr_i)$*, where*

[1] We use the term 'holder' rather than 'variable' to avoid confusion with variables as introduced for *relational MCS* [49].

- $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ *is a logic based on a signature* Σ_i*,*
- $kb_i \in \mathbf{KB}_i$ *is a knowledge base, and*
- sbr_i *is a set of* L_i-schematic-bridge rules *(*s-bridge rules *for short) of the form*

$$s \leftarrow B(r), \chi(r) \tag{6.1}$$

with $B(r) = (X_1 : P_1), \ldots, (X_j : P_j), \mathrm{not}\,(X_{j+1} : P_{j+1}), \ldots, \mathrm{not}\,(X_m : P_m)$*, where*

(1) each $sb_\ell = (X_\ell, P_\ell)$*,* $1 \leq \ell \leq m$*, is a* schematic bridge atom *(*s-bridge atom *for short) in which* $X_\ell \in M \cup \mathcal{V}_{ctx}$ *either refers to a context in* M *or is a context holder, and* $P_\ell \in \Sigma \cup \Sigma_@ \cup \Sigma_\sim$ *is either a belief (i.e.,* $P_\ell \in \Sigma$*) or a schematic belief (*$P_\ell \in \Sigma_@ \cup \Sigma_\sim$*); and*

(2) $\chi(r) = Y_{1_1} \neq Y_{1_2}, \ldots Y_{k_1} \neq Y_{k_2}$ *is a (possibly empty) list of inequality atoms* $Y_{i_1} \neq Y_{i_2}$ *(*$1 \leq i \leq k$*) where* Y_{i_1}, Y_{i_2} *are two different context holders from* $X_1, \ldots, X_m$*.*

For simplicity, we assume that context holders in rules are standardized apart, i.e., there exist no two context holders with the same name in two different rules, as they can be bound to different contexts.

Example 35 A group of n students in Example 32 can be modeled as a dynamic MCS $M = \{C_1, \ldots, C_n\}$, where, for each $C_i = (L_i, kb_i, sbr_i) \in M$, L_i is an ASP logic, $kb_i = R \cup F_i$, with R from Example 33 and $F_i \subseteq \{inter_i, hprof_i\}$, and sbr_i is the following set of schematic bridge rules:

$$sbr_i = \left\{ \begin{array}{lcl} inc_i & \leftarrow & (X_i : [pos_i]), \mathrm{not}\,(Y_i : [neg_i]), \mathrm{not}\,(i : dec_i), X_i \neq Y_i \\ dec_i & \leftarrow & (Z_i : [neg_i]), \mathrm{not}\,(T_i : [pos_i]), \mathrm{not}\,(i : inc_i), Z_i \neq T_i \end{array} \right\}.$$

The first rule expresses that student i should be inclined to take a course, if some student in the group has a positive opinion and some other student does not have a negative opinion, and student i herself is not declining to take the course. The second rule is similar, but for declining the course.

Here, the context holders set is $\mathcal{V}_{ctx} = \{X_i, Y_i, Z_i, T_i\}$, and the local signature at each context C_i is $\Sigma_i = \mathcal{A}_i \cup \{pos_i, neg_i\}$ with $\mathcal{A}_i$ taken from Example 34. We use here only similar schematic beliefs, namely $[pos_i]$ and $[neg_i]$.

Dynamic MCS differ from original MCS in the sense that s-bridge atoms in general are not specifically bound to some beliefs of other dynamic contexts in the system, but rather represent a collection of possibilities to point to different beliefs in other contexts. From a topological point of view, such a high-level representation incurs numerous dependencies between dynamic contexts in general. However, most of these dependencies are not reflected in intended instantiations, which provides evidence not to aim at defining equilibria of dynamic MCS in a direct way. Hence, for defining semantics one rather considers how to bind them to original MCS. We consider such bindings next, starting with the notion of binding a schematic bridge atom to ordinary bridge atoms based on potential matches.

A *context substitution* is a map $\sigma\colon (M \cup \mathcal{V}_{ctx}) \to M$ such that for every inequality atom $Y_{i_1} \neq Y_{i_2}$ occurring in bridge rules of a context $C \in M$, it holds that $\sigma(Y_{i_1}) \neq \sigma(Y_{i_2})$. For a context C_k, we denote by $\sigma|_{C_k}$ the *restriction* of σ to C_k, i.e., the subset of σ containing only maps from a context holder appearing in an s-bridge rule in C_k. Due to the assumption of standardization of context holders, the set of restrictions of σ to all individual contexts in M is a partitioning of σ.

The application of a context substitution σ to an s-bridge atom $sb = (X : P)$ is $\sigma(sb) = (C_j : P)$ where $P \in \Sigma_@ \cup \Sigma_\sim \cup \Sigma_j$, and either $X = C_j$ or $X \in \mathcal{V}_{ctx}$ satisfying $(X \mapsto C_j) \in \sigma$. Intuitively, the application of a context substitution is responsible for instantiating a potential context holder of an s-bridge atom.

Example 36 Let $\sigma = \{X_1 \mapsto C_2, Y_1 \mapsto C_3\}$ be a context substitution. Then the application of σ to $sb_1 = (X_1 : [pos_1])$ and $sb_2 = (Y_1 : [neg_1])$ is $sb'_1 = \sigma(sb_1) = (C_2 : [pos_1])$ and $sb'_2 = \sigma(sb_2) = (C_3 : [neg_1])$, respectively.

Let $f_M\colon \Sigma \times \Sigma \to [0, 1]$ be a function measuring the similarity between beliefs in an MCS M, where higher similarity of beliefs p and q is reflected by a larger value of $f_M(p, q)$. In particular, $f_M(p, q) = 1$ means that p and q are considered to have highest similarity (especially, if they are identical) and $f_M(p, q) = 0$ that p and q are completely dissimilar. We do not commit to a particular function f_M here, which may depend on the application; in what follows, we just assume that some such function f_M has been fixed and is available, for instance consider similarity of terms as defined by WordNet [79] or different types of matches on Larks [91] specifications.

Definition 23 *Given an MCS M, a similarity function f_M, and a threshold t, a* term substitution *from C_i to C_j in M w.r.t. f_M and t, denoted by $\eta^t_M(C_i, C_j)$, is a relation $\eta^t_M(C_i, C_j) = \{(a, b) \mid a \in \Sigma_i, b \in \Sigma_j, f_M(a, b) > t\}$.*

By η^t_M we denote the collection of all pairwise term substitutions in M. The *density* of M w.r.t. η^t_M is $d_{\eta^t_M} = |\{(C_i, C_j) \mid \eta^t_M(C_i, C_j) \neq \emptyset\}|$. In the sequel, we pick a default value $t = 0$; furthermore, we use η instead of η^0_M when M is clear from the context.

The application of a term substitution η to an s-bridge atom $sb = (C_j : P)$ in context C_i, denoted by $\eta(sb)$, is defined by

(i) if $P = a$, then $\eta(sb) = \{(C_j : a)\}$ if $a \in \Sigma_j$, and $\emptyset$ otherwise;

(ii) if $P \in \Sigma_@$, then $\eta(sb) = \{(C_j : b) \mid (bel(P), b) \in \eta(C_i, C_j), f_M(bel(P), b) = 1\}$;

(iii) if $P \in \Sigma_\sim$, then $\eta(sb) = \{(C_j : b) \mid (bel(P), b) \in \eta(C_i, C_j)\}$.

Intuitively, the application of a term substitution to an s-bridge atom only applies to s-bridge atoms with instantiated context holders, and then collects all possible substitutions for the schematic belief P.

f_M	s_1	h_1	e_1	s_2	h_2	e_2	s_3	h_3	e_3
pos_1	0.0	0.0	0.0	**0.9**	**0.6**	0.0	**0.7**	0.0	0.0
neg_1	0.0	0.0	0.0	0.0	0.0	**0.8**	0.0	**0.6**	**0.7**
pos_2	**0.7**	0.0	0.0	0.0	0.0	0.0	**0.6**	0.0	0.0
neg_2	0.0	0.0	**0.7**	0.0	0.0	0.0	0.0	0.0	**0.6**
pos_3	**0.6**	0.0	0.0	**0.8**	0.0	0.0	0.0	0.0	0.0
neg_3	0.0	0.0	**0.7**	0.0	0.0	**0.8**	0.0	0.0	0.0

Table 6.1: Interesting part of similarity function

Example 37 Continue with Example 36, suppose that we have a similarity function f_M whose interesting part is described in Table 6.1; and for the rest, f_M takes value 1 if the two parameters are identical and 0 otherwise.

The values of f_M are taken in conformity with the scenario in Example 32, e.g., Alice trusts the *select* and *hesitate* decisions of Bob as a positive sign at a measurement of 0.9 and 0.6, respectively. She considers Bob *eliminating* the course as a negative sign of 0.8. Hence, $f_M(pos_1, s_2) = 0.9$, $f_M(pos_1, h_2) = 0.6$, and $f_M(neg_1, e_2) = 0.8$. On the other hand, Alice is encouraged only when Carol selects the course, but with less confidence as $f_M(pos_1, s_3) = 0.7$; and she interprets other choices from Carol as discouragement with $f_M(neg_1, h_3) = 0.5$, and $f_M(neg_1, e_3) = 0.7$. The next rows in Table 6.1 show the opinions of Bob and Carol about the decisions of the others. Note that they do not take *hesitance* into account.

The term substitutions from C_1 to C_2 and C_3 w.r.t. f_M are $\eta(C_1, C_2) = \{(pos_1, s_2), (pos_1, h_2), (neg_1, e_2)\}$, and $\eta(C_1, C_3) = \{(pos_1, s_3), (neg_1, h_3), (neg_1, e_3)\}$, respectively. The applications of these substitution to sb'_1 and sb'_2 are $\eta(sb'_1) = \{(C_2 : s_2), (C_2 : h_2)\}$ and $\eta(sb'_2) = \{(C_3 : e_3)\}$.

Based on σ and η, the notion of a bridge substitution is simply defined by their composition.

Definition 24 *Let σ be a context substitution of M. The* bridge substitution *θ for an s-bridge atom sb w.r.t. σ is $\theta(sb) = \eta(\sigma(sb))$.*

Thus, intuitively, the bridge substitution of an s-bridge atom is done in two steps. First, one uses σ to instantiate the context holders, and then η takes effect to instantiate the schematic beliefs.

Example 38 The bridge substitution of the s-bridge atom sb_1 from Example 36 w.r.t. σ from the same example and η from Example 37 is $\theta(sb_1) = \eta(\sigma(sb_1)) = \{(C_2 : s_2), (C_2 : h_2)\}$. Similarly, we have $\theta(sb_2) = \{(C_3 : h_3), (C_3 : e_3)\}$.

Let us now turn to bridge rules. Given a schematic bridge rule r of form (6.1) in a context C_i, a context substitution σ is called a substitution of r iff there exist bridge substitutions θ_ℓ w.r.t. σ for all schematic bridge atoms sb_ℓ in $B(r)$, i.e., for $1 \leq \ell \leq m$, such that $\theta_\ell(sb_\ell) \neq \emptyset$. The bindings of r w.r.t. σ are defined as the set of bound rules $r\sigma$ where each bound rule is obtained by replacing $sb_\ell = (X : P)$ in $B(r)$ with

(i) some bridge atom $(C_i : b)$ such that $(C_i : b) \in \theta_\ell(sb_\ell)$, if $sb_\ell \in B^+(r)$; and

(ii) the sequence of negated bridge atoms $\mathrm{not}\,(C_i : b_1), \ldots, \mathrm{not}\,(C_i : b_k)$ such that $\{(C_i : b_1), \ldots, (C_i : b_k)\} = \theta_\ell(sb_\ell)$; if $sb_\ell \in B^-(r)$.

The size m of $r\sigma$ is determined by $m = \Pi_{sb_\ell \in B^+(r)} |\theta_\ell(sb_\ell)|$.

Example 39 Continuing our example, pick r as the first s-bridge rule from Example 35 and consider it in context C_1 representing Alice's mind. Furthermore, regard the context substitution σ from Example 36. Taking the term substitutions of Example 37 into account, $r\sigma$ consists of two bound rules, namely the first two rules from br_1 in Example 34.

Given a set R of s-bridge rules of a context C and a context substitution σ, the binding of R w.r.t. σ is defined as $R\sigma = \bigcup_{r \in R} r\sigma$. Then, the binding of a context C w.r.t. a context substitution σ is given by $C\sigma = (kb, sbr\sigma)$.

Definition 25 *Given a dynamic MCS M and a context substitution σ, the set $M\sigma = \{C_1\sigma, \ldots, C_\ell\sigma\}$, is a* binding *of M with respect to a context C_k iff*

1. $\{C_1, \ldots, C_\ell\} \subseteq M$
2. *σ is a substitution for all s-bridge rules in all contexts $C_1, \ldots, C_\ell$, and*
3. $\{C_1, \ldots, C_\ell\} = \bigcup_{C_j \in \{C_1 \ldots C_\ell\}} \{C \mid r \in sbr_j\sigma \wedge (C : a) \in B(r)\} \cup \{C_k\}$.

Intuitively, a binding of M with respect to a substitution σ and a context C_k consists of a subset of the contexts of M, which must contain C_k (hence condition 1 and 3), which is properly instantiated by θ (condition 2) and, moreover, *closed* in the sense that every selected context, except for C_k, is used for instantiating bridge rules of other chosen contexts (condition 3).

The notions of belief state and equilibrium are then inherited from ordinary MCS.

Definition 26 *A* belief state *of $M\sigma$ is a sequence $S = (S_1, \ldots, S_\ell)$ of belief sets, one S_i for each $C_i\sigma$. Such a belief state is an* equilibrium *of M w.r.t. C_k and σ iff for all $1 \leq i \leq \ell$, it holds that $S_i \in \mathbf{ACC}_i(kb_i \cup head(r) \mid r \in app(sbr_i\sigma, S))$.*

The quality of a binding is simply the average of the similarities of all matches used in the binding.

Definition 27 *The quality of an MCS binding $M\sigma$ is*

$$\frac{1}{|\mathcal{U}|} \cdot \sum_{(a,b) \in \mathcal{U}} f_M(a, b)$$

where $\mathcal{U}$ is the set of matches used in the binding, i.e., $\mathcal{U} = \{(a, b) \mid a = bel(P) \wedge sb = (X : P) \in C_i, 1 \leq i \leq \ell \wedge (C_j : b) \in \theta(sb), 1 \leq j \leq \ell\}$, where $\theta = \eta \circ \sigma$.

We now give a detailed example on binding from a dynamic MCS to an original one (parts of the details below can be found in previous examples).

Example 40 Take an instance of the general dynamic MCS in Example 35 with $n = 3$: $M = (C_1, C_2, C_3)$, where

- for each $C_i = (L_i, kb_i, sbr_i)$, where $1 \leq i \leq 3$, L_i is an ASP logic;
- the local knowledges are $kb_1 = R_1 \cup \{hprof_1\}$, $kb_2 = R_2 \cup \{inter_2\}$, and $kb_3 = R_3$, where R_i is taken from Example 33;
- the schematic bridge rules are taken from Example 35, where $1 \leq i \leq 3$:

$$sbr_i = \left\{ \begin{array}{lcl} inc_i & \leftarrow & (X_i : [pos_i]), \text{not } (Y_i : [neg_i]), \text{not } (i : dec_i), X_i \neq Y_i \\ dec_i & \leftarrow & (Z_i : [neg_i]), \text{not } (T_i : [pos_i]), \text{not } (i : inc_i), Z_i \neq T_i \end{array} \right\}.$$

Take the following context substitution:

$$\sigma = \left\{ \begin{array}{llll} X_1 \mapsto C_2, & Y_1 \mapsto C_3, & Z_1 \mapsto C_2, & T_1 \mapsto C_3, \\ X_2 \mapsto C_1, & Y_2 \mapsto C_3, & Z_2 \mapsto C_3, & T_2 \mapsto C_1, \\ X_3 \mapsto C_2, & Y_3 \mapsto C_1, & Z_3 \mapsto C_1, & T_3 \mapsto C_2 \end{array} \right\}.$$

Applying σ on the schematic bridge atoms, we first get:

$$\begin{array}{ll} \sigma(X_1 : [pos_1]) = (C_2 : [pos_1]) & \sigma(Y_1 : [neg_1]) = (C_3 : [neg_1]) \\ \sigma(Z_1 : [neg_1]) = (C_2 : [neg_1]) & \sigma(T_1 : [pos_1]) = (C_3 : [pos_1]) \\ \sigma(X_2 : [pos_2]) = (C_1 : [pos_2]) & \sigma(Y_2 : [neg_2]) = (C_3 : [neg_2]) \\ \sigma(Z_2 : [neg_2]) = (C_3 : [neg_2]) & \sigma(T_2 : [pos_2]) = (C_1 : [pos_2]) \\ \sigma(X_3 : [pos_3]) = (C_2 : [pos_3]) & \sigma(Y_3 : [neg_3]) = (C_1 : [neg_3]) \\ \sigma(Z_3 : [neg_3]) = (C_1 : [neg_3]) & \sigma(T_3 : [pos_3]) = (C_2 : [pos_3]) \end{array}$$

Now, take the similarity function f_M from Table 6.1, the term substitution with a threshold $t = 0.5$ between contexts in M can be shown as follows:

$$\begin{array}{ll} \eta(C_1, C_2) = \{(pos_1, s_2), (pos_1, h_2), (neg_1, e_2)\} & \eta(C_1, C_3) = \{(pos_1, s_3), (neg_1, e_3)\} \\ \eta(C_2, C_1) = \{(pos_2, s_1), (neg_2, e_1)\} & \eta(C_2, C_3) = \{(pos_2, s_3), (neg_2, e_3)\} \\ \eta(C_3, C_1) = \{(pos_3, s_1), (neg_2, e_1)\} & \eta(C_3, C_2) = \{(pos_3, s_2), (neg_2, e_2)\} \end{array}$$

Combining σ and η, we get the following bridge atoms from s-bridge atoms:

$$\theta(X_1 : [pos_1]) = \{(C_2 : s_2), (C_2 : h_2)\} \qquad \theta(Y_1 : [neg_1]) = \{(C_3 : h_3), (C_3 : e_3)\}$$
$$\theta(Z_1 : [neg_1]) = \{(C_2 : e_2)\} \qquad \theta(T_1 : [pos_1]) = \{(C_3 : s_3)\}$$
$$\theta(X_2 : [pos_2]) = \{(C_1 : s_1)\} \qquad \theta(Y_2 : [neg_2]) = \{(C_3 : e_3)\}$$
$$\theta(Z_2 : [neg_2]) = \{(C_3 : e_3)\} \qquad \theta(T_2 : [pos_2]) = \{(C_1 : s_1)\}$$
$$\theta(X_3 : [pos_3]) = \{(C_2 : s_2)\} \qquad \theta(Y_3 : [neg_3]) = \{(C_1 : e_1)\}$$
$$\theta(Z_3 : [neg_3]) = \{(C_1 : e_1)\} \qquad \theta(T_3 : [pos_3]) = \{(C_2 : s_2)\}$$

With these bridge atoms, one can form the following bridge rules (Example 34):

$$sbr_1\sigma = \left\{ \begin{array}{lcl} inc_1 & \leftarrow & (2 : s_2), \text{not}\,(3 : h_3), \text{not}\,(3 : e_3), \text{not}\,(1 : dec_1) \\ inc_1 & \leftarrow & (2 : h_2), \text{not}\,(3 : h_3), \text{not}\,(3 : e_3), \text{not}\,(1 : dec_1) \\ dec_1 & \leftarrow & (2 : e_2), \text{not}\,(3 : s_3), \text{not}\,(1 : inc_1) \end{array} \right\},$$

$$sbr_2\sigma = \left\{ \begin{array}{lcl} inc_2 & \leftarrow & (1 : s_1), \text{not}\,(3 : e_3), \text{not}\,(2 : dec_2) \\ dec_2 & \leftarrow & (3 : e_3), \text{not}\,(1 : s_1), \text{not}\,(2 : inc_2) \end{array} \right\}, \text{and}$$

$$sbr_3\sigma = \left\{ \begin{array}{lcl} inc_3 & \leftarrow & (2 : s_2), \text{not}\,(1 : e_1), \text{not}\,(3 : dec_3) \\ dec_3 & \leftarrow & (1 : e_1), \text{not}\,(2 : s_2), \text{not}\,(3 : inc_3) \end{array} \right\}.$$

Then, $M\sigma = (C_1\sigma, C_2\sigma, C_3\sigma)$ is a binding of M with respect to C_1. One can check that all conditions in Definition 25 are satisfied. The quality of this binding is

$$\frac{0.9 + 0.6 + 0.7 + 0.8 + 0.6 + 0.7 + 2 \times (0.7 + 0.6 + 0.8 + 0.7)}{14} = 0.707.$$

6.3 From Dynamic to Ordinary Multi-Context Systems

Recapturing the idea of binding a dynamic MCS M to an original one, starting from a context C_{root}, one needs

1. to know all potential neighbors C_j for a context C_i and the term substitutions $\eta(C_i, C_j)$ between them;

2. a strategy to start from C_{root} and to expand the system by first determining a context substitution σ for each context term in the s-bridge rules of C_{root}, and then continuing the process at each neighbor, until a closed system is obtained;

3. some decision criteria to guide the process to come up with a most suitable substitution to bind M.

Task (1) is in fact matching beliefs from different contexts. This problem shares similarities with the *matchmaking problem* in Multi-Agent Systems (MAS), which has been widely considered [82, 91]. Our work here is not doing matchmaking but rather using the matchmaker

as a building block to configure the inter-linkage between contexts in a dynamic MCS to form ordinary ones. As such, we assume that there exists a *matchmaker* MatchMaker which, upon a call MatchMaker(P, C_i) from a context C_i, returns a set of potential neighbors such that

- if P is a schematic variable in $\Sigma_@$, then N is the set of context names C_j where the term substitution $\eta(C_i, C_j)$ contains at least one pair $(bel(P), a)$ with $f_M(bel(P), a) = 1$;
- if P is a schematic variable in $\Sigma_\sim$, then N is the set of context names C_j where the term substitution $\eta(C_i, C_j)$ is nonempty;
- if $P = p$ is an atom from Σ, then N is the set of all contexts C_j such that $p \in \Sigma_j$.

Further queries to the matchmaker such as MatchMaker(C_i, C_j) can give back $\eta(C_i, C_j)$ and/or the value of f_M for the pairs of atoms from this term substitution. This information is used for calculating the quality of the system after instantiating.

The main problems that we solve here are those in (2) and (3). Concerning (2), we present a backtracking algorithm to enumerate all possible context substitutions σ, in a distributed, peer-to-peer like setting. This means that each context, knowing only its potential neighbors by asking the matchmaker, can only locally choose the matches for its own s-bridge atoms, which consequently decides its real neighbors in the resulting MCS, and then has to ask these neighbors to continue the configuration (hence our algorithm is called lconfig).

The process starts at C_{root} and continues in a Depth-First Search (DFS) manner, carrying along the context substitution σ built up so far, until for all chosen contexts their s-bridge atoms are bound.

Regarding (3), we propose general methods to compare the outcome of different substitutions on two main aspects, namely

(Q1) the matching quality of the bound rules, and

(Q2) the topological quality of the resulting MCS.

These methods can be seen as heuristics that can be plugged into the basic version of lconfig to get the context substitutions returned ordered by quality.

For clarity and simplicity, in the sequel, we first present the very basic version of lconfig with generic possibilities for optimization. We then briefly go through such possibilities, where we choose some interesting ones to discuss in more detail and suggest potential realizations of them.

6.4 Multi-Context Systems Configuration

Basic algorithm

The question is now how one can actually compute substitutions as sketched above. We present a basic configuration algorithm which computes concrete bindings for a dynamic MCS. We

Algorithm 6.1: lconfig(C_{root}, R, σ) at C_k

Input: C_{root}: root context, R: set of s-bridge rules, σ: context substitution
Output: context susbstitution for C_k
Data: $obuf_r$ for every $r \in R$: substitutions for r

if $R = \emptyset$ **then**
(a) $\quad \mathcal{C}_{new} := \mathsf{get\_contexts}(\sigma|_{C_k}) \setminus (\mathsf{get\_contexts}(\sigma \setminus \sigma|_{C_k}) \cup \{C_{root}\})$
$\quad$ **if** $\mathcal{C}_{new} \neq \emptyset$ **then return** invoke_neighbors$(C_{root}, \mathcal{C}_{new}, \sigma)$
$\quad$ **else return** $\{\sigma\}$
(b) **else**
(c) $\quad$ pick r from R
$\quad obuf_r := \mathsf{bind\_rule}(\chi(r), B(r), \sigma)$
$\quad ctx\_sub := \emptyset$
$\quad$ **while** $obuf_r \neq \emptyset$ **do**
(d) $\quad\quad$ pick σ' from $obuf_r$
$\quad\quad obuf_r := obuf_r \setminus \{\sigma'\}$
$\quad\quad ctx\_sub := ctx\_sub \cup \mathsf{lconfig}(C_{root}, R \setminus \{r\}, \sigma')$
$\quad$ **return** ctx_sub

start with a particular context in the system and gradually invoke some neighbors to get further solutions.[2]

Given a dynamic MCS M and a starting context C_{root}, the algorithm lconfig presented in this section aims at enumerating all possible context substitutions that can lead to a binding for M, in a distributed way. It mutually calls an algorithm invoke_neighbors and makes use of the following primitives:

- a function get_contexts(σ), which takes a context substitution σ (containing substitutions of form $X \mapsto C$) as input and returns the set of contexts C used in σ.

- a DFS subroutine bind_rule, which given an s-bridge rule r as input consults the matchmaker MatchMaker and returns all context substitutions for the non-ordinary s-bridge atoms of r.

The algorithm lconfig has several parameters: the context C_{root} where the configuration started, the set R of s-bridge rules left to be bound, and the context substitution σ built up so far.

Intuitively, in a context C_k, lconfig first utilizes bind_rule in a DFS manner to enumerate all possible context substitutions for the s-bridge atoms in sbr_k (Step (b)). When this is done, in Step (a) it only refers to newly chosen contexts via a set $\mathcal{C}_{new}$ and calls invoke_neighbors to get the context substitutions of all members in $\mathcal{C}_{new}$.

[2] In centralized settings, one might instead compute substitutions by making use of more standard declarative solvers, e.g., such as ASP solvers with external information access, for example, the dlvhex system (`http://www.kr.tuwien.ac.at/research/systems/dlvhex/`).

Algorithm 6.2: invoke_neighbors(C_{root}, N, σ) at C_k

Input: C_{root}: root context, N: set of neighbors, σ: context substitution
Output: context substitutions for all neighbors of C_k

(e) **if** $N = \emptyset$ **then return** $\{\sigma\}$
else
(f) pick C_j from N
$obuf_{C_j} := C_j.\mathsf{lconfig}(C_{root}, sbr_j, \sigma)$
$ctx\_sub := \emptyset$

while $obuf_{C_j} \neq \emptyset$ **do**
(g) pick σ' from $obuf_{C_j}$
$obuf_{C_j} := obuf_{C_j} \setminus \{\sigma'\}$
(h) $N' := N \setminus (\mathsf{get\_contexts}(\sigma') \cup \{C_j\})$
$ctx\_sub := ctx\_sub \cup \mathsf{invoke\_neighbors}(C_{root}, N', \sigma')$

return ctx_sub

The algorithm invoke_neighbors has the same parameters C_{root} and σ as lconfig, and carries in addition a set N of newly chosen neighbors of C_k where local configuration needs to be done. The algorithm first picks a neighbor C_j and calls lconfig at this context (Step (f)) to get all context substitutions updated with local substitutions for C_j, stored in $obuf_{C_j}$. Then, in Step (g), it picks each substitution from $obuf_{C_j}$ and continues invoking the remaining contexts in N. Note that in Step (h), the set of remaining neighbors to invoke is recomputed in N', as some of the contexts in N might already be chosen by the call to C_j and thus they are already invoked.

When all invocations of neighbors have finished, the substitution computed at this point is returned and is treated by lconfig either as an intermediate result for the context that invoked it, or as the final result for the user.

Example 41 Take the setting from Example 39 and run bind_rule over the rule body with a starting empty substitution. The call is bind_rule$(B, \emptyset)$ in which $B = \{(X_1 : [pos_1]), (Y_1 : [neg_1])\}$. Assume that the first s-bridge atom chosen at Step (i) is $sb = (X_1 : [pos])$. A call MatchMaker$([pos_1], C_1)$ to the matchmaker returns $N = \{C_2, C_3\}$. The routine then tries all possibilities to bind sb and works recursively to bind the rest of the body. For example, if it chooses to bind X_1 to C_2, then the next call will be bind_rule$(\{(Y_1 : [neg_1])\}, \{X_1 \mapsto C_2\})$ which returns $\{\{X_1 \mapsto C_2, Y_1 \mapsto C_3\}\}$ as the set of all context substitutions in which X_1 is mapped to C_2. The binding continues with $X_1 \mapsto C_3$ and in the end, we get two context substitutions, namely $\{X_1 \mapsto C_2, Y_1 \mapsto C_3\}$ and $\{X_1 \mapsto C_3, Y_1 \mapsto C_2\}$.

Example 42 This example illustrates the run of lconfig and invoke_neighbors on a dynamic MCS from Example 35 with a pool of $n = 3$ contexts. Starting from C_1 we can pick one s-bridge rule from the non-empty set of s-bridge rules at (c), say the first one from Example 35. According

Algorithm 6.3: bind_rule(I, B, σ) at C_k

Input: I: set of inequality atoms, B: set of s-bridge atoms, σ: context substitution
Output: substitutions for B

(i) **if** $\exists a = (X : P)$ *non-ordinary in* B **then**
  $N :=$ MatchMaker(P, C_k) // N: set of potential neighbors
  if $\nexists (X \mapsto C)$ *in* σ **then**
    $dup := \{C_i \in N \mid (Y \mapsto C_i) \in \sigma \wedge (X \neq Y) \in I\}$
    $N := N \setminus dup$
    $ctx\_sub := \emptyset$
    while $N \neq \emptyset$ **do**
(j) choose a context C_j from N
      $N := N \setminus \{C_j\}$
      $ctx\_sub := ctx\_sub \cup$ bind_rule$(I, B \setminus \{a\}, \sigma \cup \{X \mapsto C_j\})$
    return ctx_sub
  else if $\sigma(X) \in N$ **then**
    return bind_rule$(I, B \setminus \{a\}, \sigma)$
  else return $\emptyset$
else return $\{\sigma\}$

to Example 41, the subroutine bind_rule returns a set of two possible context substitutions. Let us pick $\sigma = \{X_1 \mapsto C_2, Y_1 \mapsto C_3\}$ from this set and continue calling lconfig for the last rule. This gives two possible extensions of σ, one of which extends σ to $\{X_1 \mapsto C_2, Y_1 \mapsto C_3, Z_1 \mapsto C_2, T_1 \mapsto C_3\}$.

Having this context substitution carried to the next recursive call of lconfig, we reach the point where $R = \emptyset$, get $\mathcal{C}_{new} = \{C_2, C_3\}$, and continue calling lconfig at C_2 or C_3. The algorithm proceeds and in the end, we get a number of context substitutions; one of them is

$$\sigma = \left\{ \begin{array}{llll} X_1 \mapsto C_2, & Y_1 \mapsto C_3, & Z_1 \mapsto C_2, & T_1 \mapsto C_3, \\ X_2 \mapsto C_1, & Y_2 \mapsto C_3, & Z_2 \mapsto C_3, & T_2 \mapsto C_1, \\ X_3 \mapsto C_2, & Y_3 \mapsto C_1, & Z_3 \mapsto C_1, & T_3 \mapsto C_2 \end{array} \right\}.$$

This substitution yields the MCS system in Example 34.

Bounded Enumeration. When the pool size gets large, enumerating all bindings for each bridge rule and all bridge substitutions becomes infeasible. A practical approach would be to compute only a small number of bindings for each rule, and also just a few substitutions at each context. For the remainder of this section, let us use b and n to denote corresponding limits.

We have presented the basic algorithm for enumerating all possible context substitutions of a dynamic MCS M w.r.t. a context C_k in M with which M can be bound to original MCS. To keep it simple, in steps (c), (d), (f), (g), (i), and (j), we nondeterministically pick either a rule, a context substitution, or a context as no supporting information is provided. This leaves a lot of room for optimization. Furthermore, we did not mention how to deal with irregular cases such

as when the matchmaker returns no potential neighbor, or the size of the partial MCS has passed some boundary; furthermore, no caching has been foreseen.

In the following subsection, we discuss different heuristics to enhance the search process when more support information is available, so that the context substitutions will be returned in some quality driven order. After that, we briefly describe a strategy for cutting off when reaching a size boundary, hence a possibility to tolerate partial bindings.

Quality-driven local configuration

Quality for the topology (Q_T). Our experimental results reveal that evaluating equilibria of MCS in general does not scale up to very large systems, and [31] and [8] showed that limitations on some specific topologies such as the diamond topology exist. Hence, one of the purposes for configuration is to constrain the size/topology of the resulting system to some boundary, e.g., by trying to reuse as many contexts from the local configuration of the parent as possible; or by trying to avoid troublesome topological properties, such as ones having *join contexts*, i.e., contexts C_i which are accessed from different contexts C_j and $C_{j'}$ which in turn are accessed (possibly by intermediate contexts) from a single context C_k, or cycles.

For this purpose, the selection of neighbors (Step (j)) is crucial. To support a context C_k with more information for this task, we do a *one-step look-ahead* at all of its potential neighbors. In general, looking ahead into a potential neighbor C_i can give back any information that C_i is able to infer from its own knowledge and information provided by the matchmaker. In this work, our setting allows the look-ahead to return the number of s-bridge atoms in C_i, denoted by nba_i.

We define in the following different heuristic possibilities of the topological quality function to reflect attempts to have the resulting MCS in some restricted shape (the smaller the value of the function, the better is the quality).

Assume that having started the configuration from context C_{root}, we are now doing local configuration at context C_k, choosing a binding for a schematic bridge belief $[p]$, considering the possible match (p, q) to a context C_i. As the context substitution σ is carried along, one can easily extract the set of chosen contexts so far, which is denoted here by $\mathcal{C}$. Consider the following topological quality functions:

(H1) $quality_{i,k} = nba_i$: with this function, we prefer potential neighbors with fewer s-bridge atoms.

(H2) $$quality_{i,k} = \begin{cases} 0 & \text{if } C_i \in \mathcal{C} \\ 1 & \text{otherwise.} \end{cases}$$

This function gives priority to contexts which are already chosen, hence to keep the size of the resulting system small. On the other hand, this tends to introduce cycles.

One can imagine more complicated quality functions; for instance, to take the topology of the system built up so far into account in order to avoid cycles or join contexts. To this end, one must transfer not only the substitution σ between contexts, but also the system topology (i.e., the respective graph).

Along the same lines, for Step (c) (resp., (i)) one can define quality functions, for instance based on some syntactic criteria combined with some history information, to provide an heuristic ranking for choosing the next rule (resp., the next non-ordinary s-bridge atom).

Quality for bindings of a schematic rule (Q_S). This type of quality measures the closeness between the bindings and the intended meaning of the schematic rule based on the matching quality of each single s-bridge atom. Notice that after getting the schematic substitution η from the matchmaker, θ is determined by the context substitution σ. Each realization of σ gives us a possibility to bind the schematic rules. What we need is a means to compare these possibilities. To make it generic, we define the quality function ρ of a bound rule $r' \in r\sigma$ of a schematic rule r of form (6.1) as follows:

$$\rho(r') = \mathrm{op} \left\{ \begin{array}{l} \alpha \mid sb = (X : P) \in B(r) \wedge \\ \quad (C_j : b) \in \theta(sb) \wedge (bel(P), b) \in \eta(C_i, C_j) \wedge \\ \quad (C_j : b) \in B(r') \wedge f_M(bel(P), b) = \alpha \end{array} \right\}.$$

Basically, we take the measure of similarity of all bindings used to construct r' and apply an operator op on top. Here, op is generic and can be instantiated to any operator for a specific use. For example, two plausible options are (i) $\mathrm{op} = \min$, and (ii) $\mathrm{op} = \mathrm{avg}$.

Roughly, in case (i), following an overly cautious approach, the quality of the whole binding is determined as the minimal quality of all matches of the s-bridge atoms in its body. In a different approach, case (ii) takes all matches into account and respects a contribution of each match in the overall quality of the rule. Depending on different philosophies to establish the overall quality of a binding that is based on bindings of each single schematic bridge atom, one can provide more complicated operators and plug them into this scheme.

To benefit from this quality, one can sort the output buffer $obuf_r$ (resp., $obuf_{C_j}$) according to Q_S in Step (d) (resp., (g)), and then pick the best context substitution up to this point to continue with.

Combined quality (Q_C). The two types of quality functions above look into two aspects of the resulting system, namely the (Q_T) topology and the (Q_S) similarity in meaning of the bindings. To exploit the latter, one needs to enumerate all possible bindings for a rule.

When we have a limit on the number of solutions (see discussion above), it is very important to approximate Q_T when choosing a binding, since then one cannot compute all bindings of a rule, and then sort them.

To this end, we modify Q_T in a way that it also takes care of the quality of the match. Intuitively, when two potential contexts are equally ranked by Q_T, one looks at the quality of the match, i.e., approximating Q_S, to rank them. More specifically, the heuristic functions H1 and H2 are changed to:

(H3) $quality_{i,k} = nba_i - \alpha$, and

(H4) $quality_{i,k} = \begin{cases} -\alpha & \text{if } C_i \in \mathcal{C} \\ 2 - \alpha & \text{otherwise,} \end{cases}$

where α is the quality of the match being considered to bind the current s-bridge atom.

Dealing with irregular cases

In practice, it is convenient for the user to have the possibility to specify an upper bound for the size of the resulting system. However, a full substitution respecting the given limit may not always exist. A flexible approach to deal with such a situation, rather than to increase the bound, is to *cut off* when reaching the boundary and to tolerate partial answers.

By cutting off, more precisely we mean to remove all unbound negative s-bridge atoms from s-bridge rules, and remove all s-bridge rules with unbound positive bridge atoms. Intuitively, this amounts to considering a system where any further contexts that would exist for binding are considered to return empty belief sets (and thus the respective bridge atoms are pre-evaluated accordingly). Note that one also needs to undo the on-going substitutions for such s-bridge rules, and this might trigger the cancellation of substitutions in a backward manner: since once a context is not used anymore for instantiating other contexts, it is not needed and the part of the substitution w.r.t. this context should be removed from the final result.

Another case in which cutting off might be used is when substitutions do not exist due to non-existent matchings, i.e., when the matchmaker does not return any match for a schematic constant. We can apply the same strategy as above, i.e., remove the corresponding s-bridge atom if it appears in the negative body of an s-bridge rule, or remove the whole s-bridge rule if the s-bridge atom is in its positive body.

The cutoff is in fact easy to implement. One can create a dummy context C_0 which has only a single belief $dumb$ with the unique acceptable belief set $\emptyset$ (i.e., intuitively $dumb$ is $false$, and all beliefs in every other context are matchable to $dumb$. Then cutting off as discussed above is simply achieved by matching unbound s-bridge atoms to $(C_0 : dumb)$.

Prototype implementation

A prototype implemetation of the configuration algorithm was reported in [32] with experimental results on different dynamic topologies. As this implementation was independently developed from the DMCS system, it will not be covered in Chapters 7 and 8. For more details on the implementation and experimental results, we refer the reader to [32].

Part III

Implementation and Evaluation of Multi-Context Systems

CHAPTER 7

The DMCS System

A preliminary system description of DMCS was given in [9]. This Chapter provides more technical hints and updates on the newest features of the system. Following the content here, one can get an insightful on the implementation of the system in order to extend or modify it for further improvement. We start with describing the system architecture at both levels: the global distributed setting (Section 7.1) and the local level at each node (Section 7.2). The wrapper to communicate with local solvers is explained in Section 7.3. Finally, instructions to use the system are given in Section .1.

7.1 Global Level Architecture

Figure 7.1 illustrates the architecture of the global MCS distributed setting, which has the following main components:

(i) a front-end `dmcsc` for the user to query the system;

(ii) daemons `dmcsd`, where each of them represents a node which contains a set of contexts; the daemons interact with each other. A node is identified by a hostname and a port which are shared by its contexts to communicate to other contexts. Details on the architecture of these daemons are in Section 7.2;

(iii) a manager `dmcsm` holding meta information about the system that has been collected from the contexts, such as system topology, the interface of exchanging beliefs between pairs of `dmcsd`. Right now, `dmcsm` is implemented in a simplified version, that is, it takes care of system initialization, while more sophisticated functionalities such as computing the optimal interface, removing connection to create optimal topologies are simulated by a configuration file generated by a generator called `dmcsgen` (see Section .1 for instructions on how to generate the test cases).

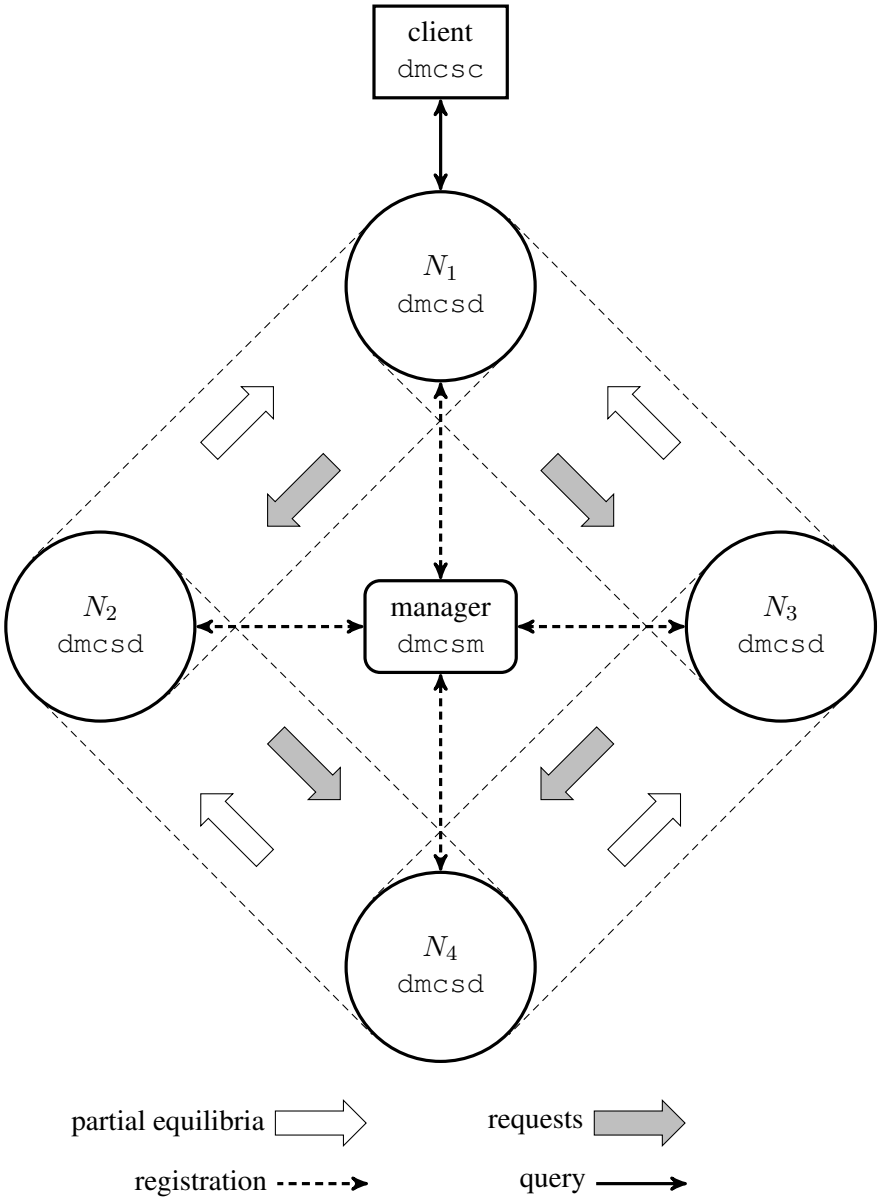


Figure 7.1: Distributed MCS Setting

The regions between pairs of nodes which have big arrows covered by dashed line represent the connection links between them. Each link between two nodes N_i, N_j is established once when a context of N_i first requests to another context of N_j and is kept while the system is running. Multiple requests and returns of results between two contexts belonging to N_i and N_j are transferred via just one connection. This approach saves the implementation from exponential blowup of connections to be created, thus enable scalability in terms of system size. The running

of the system includes the following stages:

1. **System start-up.** Each `dmcsd` starts up in two phases. First, it initializes the contexts and let them read the local knowledge bases as well as the bridge rules. Then, all contexts register at `dmcsm` and provide their information such as hostname, port, signatures, and bridge rules. After this, `dmcsm` computes the system topology, optimizes the communicating interface, and returns corresponding information (neighbors and interface) to each context. Then, the second initialization phase is carried out at each node as contexts establish network connections with their neighbors. This stage ends with every `dmcsd` starting to listen to its own port for incoming requests from other daemons or from `dmcsc` as described next.

   However, as `dmcsm` is now implemented at its simplified version, its main purpose is to keep all contexts not to immediately connect to each other but wait until all finishing their local reading, parsing, and setting up the server component for listening to requests. Other purposes are simulated by configuration files, therefore, this stage is accomplished by letting each context read the configuration file to get its neighbor list (parents and children) and the interface of exchanging beliefs to those.

2. **Querying the system.** When the user wants to know partial equilibria of the system wrt a starting context C_k (in Figure 7.1, this is context C_1), she uses `dmcsc` to pose the query. The daemon `dmcsc` inquiries `dmcsm` to get the hostname and port of C_k, which is done now by reading the system configuration. Then, `dmcsc` establishes a connection to C_k, which is in fact the connection to the daemon holding C_k, and sends it the request of the form (k_1, k_2), where $[k_1, k_2]$ is the range of equilibria to be returned when $1 \leq k_1 \leq k_2$; while $[0, 0]$ is the special request for all equilibria.

3. **Evaluating the system.** After `dmcsc` has sent a request to C_k, the context checks whether it needs beliefs from neighboring contexts. If not, then C_k is a leaf context and it can simply compute the local models, convert them into partial belief states and return to the user. Otherwise, C_k is an intermediate context and needs to send further requests to the neighbors. Essentially, those requests look just as the one sent from `dmcsc`, and every neighboring context will process them in a uniform manner. As such, each neighbor will be asked for models in a range $[1, s]$ only, where s is a pre-specified package size, except for the special setting in which all request is $[0, 0]$ and each context returns all partial equilibria at once.

   Back to the "streaming" setting, after all neighbors return their partial equilibria to C_k, it combines these in a consistent way, which might result in 0 or more partial equilibria. In case the combination gives no partial equilibria, C_k has to issue further requests to the neighbors, starting from $[s + 1, 2s]$ at the first neighbor, and so on, i.e., $[2s + 1, 3s]$ at the first neighbor, until reaching the total number of equilibria at this one, then back to $[1, s]$ at the first neighbor and $[s + 1, 2s]$ at the second one; this way, all possible combinations of results from the neighbors will be considered.

   When there are consistent combinations from the neighbors, C_k picks each of them in some order to evaluate the bridge rules, updates the local knowledge base, computes par-

tial equilibria wrt each update, and selects those in range $[k_1, k_2]$ to return to the user. If this range has not yet been covered, C_k has to issue further requests to its neighbors, following the same manner as above. The process continues until partial equilibria in the expected range are returned, or all combinations of partial equilibria from the neighbors are considered. The latter case means that the total number of partial equilibria is smaller than k_2.

7.2 Architecture At Local Nodes

This section describes the internal architecture at local nodes, which is presented in Figure 7.2. The figure illustrates an intermediate nodes with two parents and two children nodes. When removing the lower part of the figure, we have the architecture of leaf nodes.

Each squared rectangle represents a thread, and the rounded-corners rectangle represents a group of threads which together form the core of a context. This part is further detailed in Figure 7.3. The threads communicate with each other via a special shared-memory means, the *Concurrent Message Queues* (CMQ). Each CMQ allows for atomic read and write operations from threads knowing the pointer to it. When a queue is empty, any thread reading from it must wait for something to be written in. The content from the queue that has just been read is also flushed from the queue. Furthermore, each CMQ has a *pre-fixed size* k; when the queue is full, any thread trying to write to it must wait until some other thread reads something from the queue to make some free space. This way, we automatically obtain *synchronization* and have an *upper polynomial bound* on the memory consumption for storing intermediate partial equilibria wrt the number of contexts in the system. The latter is very important when local contexts are ASP programs or SAT theories that usually produce exponentially many local models.

Looking into more details on the threads, there is a pair of Handler and Output threads for each parent node. The Handler is created when a parent node sends the first request to the current node, and then this thread is kept alive to listen to further requests from the same node. Each Handler is accompanied with an Output thread, which is responsible to sending results back to the same node. In fact, the Output thread is created from the corresponding Handler. On the other direction, for intermediate nodes, there is a pair of NOut and NIn threads to communicate with a child node. NOut takes care of sending requests and NIn waits for the results. As a consequence, NOut talks to Handler and NIn listens to Output of the child node, respectively.

Since each node can have more than one context which share one communication channel (Handler and Output, NOut and NIn), there must be some means to deliver the requests and results to the right recipient. It can be done inside Handler and NIn, but this approach will make these components complicated as they have to handle tasks beyond their functionalities. Thus, we introduced *dispatchers* for this purpose; they are:

- RequestDispatcher: dispatches requests from Handler to the right context. This requires the receiver's identifier to be added to the request.

- JoinerDispatcher: conceptually, it helps dispatching partial equilibria from NIn to the right context. This requires adding the identifier of the context to the results, which in turn

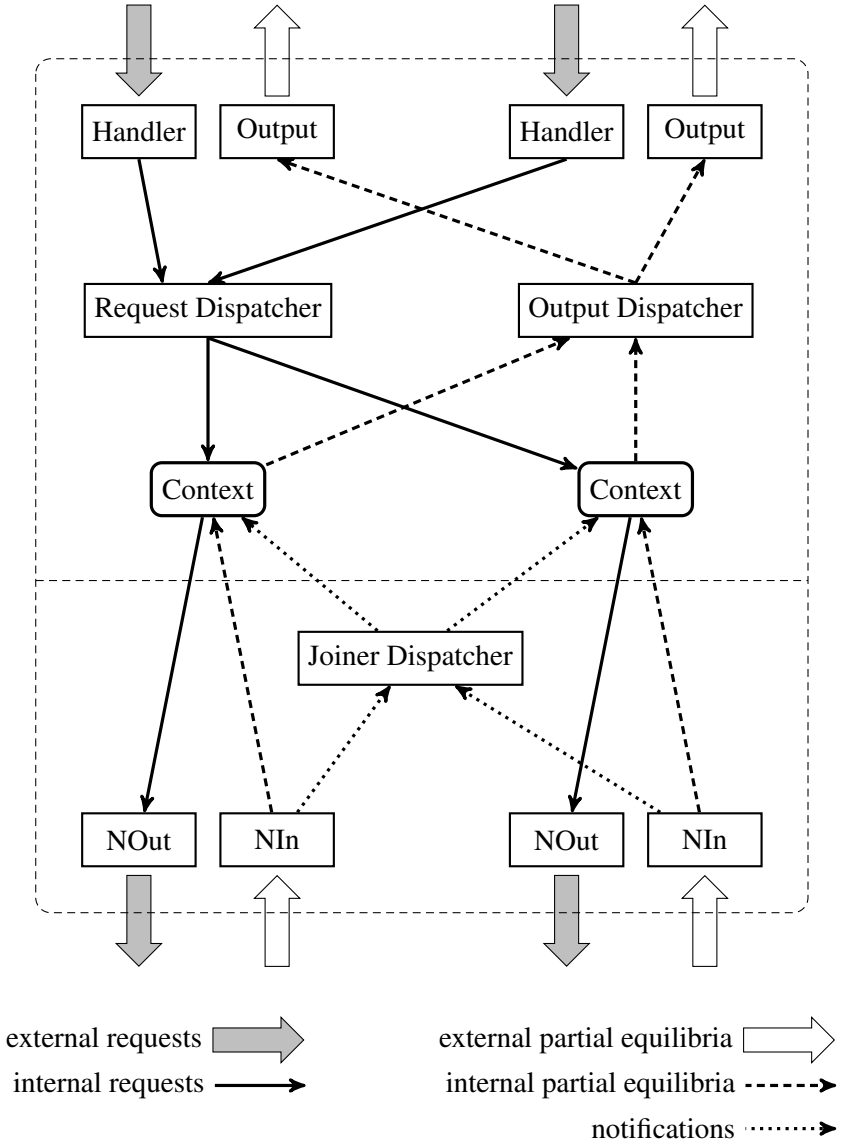


Figure 7.2: Local Node's Architecture

forces this identifier to be attached to the requests, thus the neighboring context can keep track of where to return the results.

Practically, to save communication cost, partial equilibria are not passed from NIn to JoinerDispatcher and then to Context. Instead, NIn only informs JoinerDispatcher with a notification of the form (c, n) where c is the identifier of the receiving context and n is the number of available partial equilibria for Context. Then, JoinerDispatcher notifies the corresponding context to directly read n partial equilibria from the CMQ of that NIn.

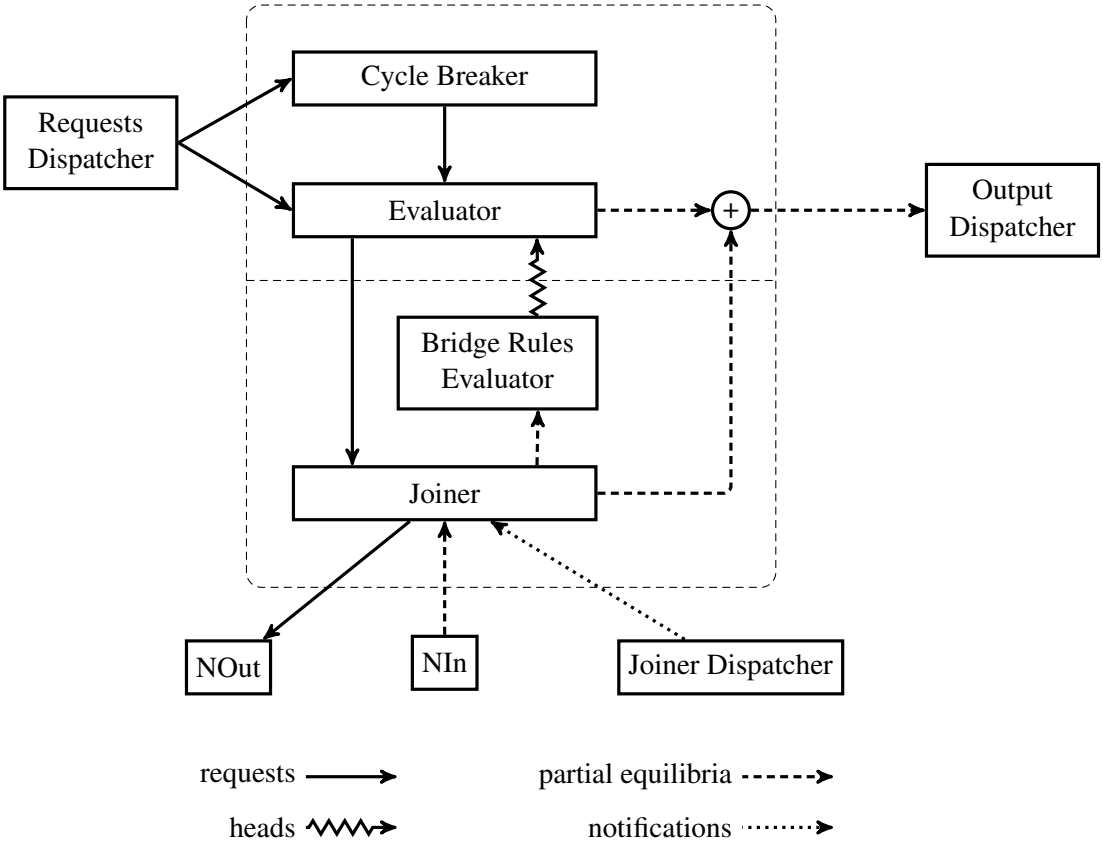


Figure 7.3: Contexts Components

Hence, to support messages dispatching, the requests are in fact of the form $(sender, receiver, k_1, k_2)$, where k_1, k_2 determine the answer range as described in Section 7.1, and $sender$, $receiver$ are identifiers of the contexts involving in the communication, respectively. Another thread responsible to dispatch partial equilibria to the right communication channels, not the contexts, is OutputDispatcher.

We next zoom in the core of Context in Figure 7.3. The figure depicts component threads of an intermediate context, which consists of the following threads: Evaluator, CycleBreaker, Joiner, and BridgeRulesEvaluator; leaf contexts just contain of Evaluator.

When receiving a request from RequestDispatcher, a leaf context simply uses Evaluator to compute its local models, then turns them into partial equilibria form and feeds them to the thread OutputDispatcher.

For intermediate contexts, the process is more involved. Based on a history H of context identifiers sent along with the request, RequestDispatcher can detect whether a cycle started by the current context C exists, by checking the presence of C's identifier in H.

If no cycle was detected, the request is sent to Evaluator, which then forwards it to Joiner, where further requests are initiated to neighboring contexts via NOut. The Joiner then waits for

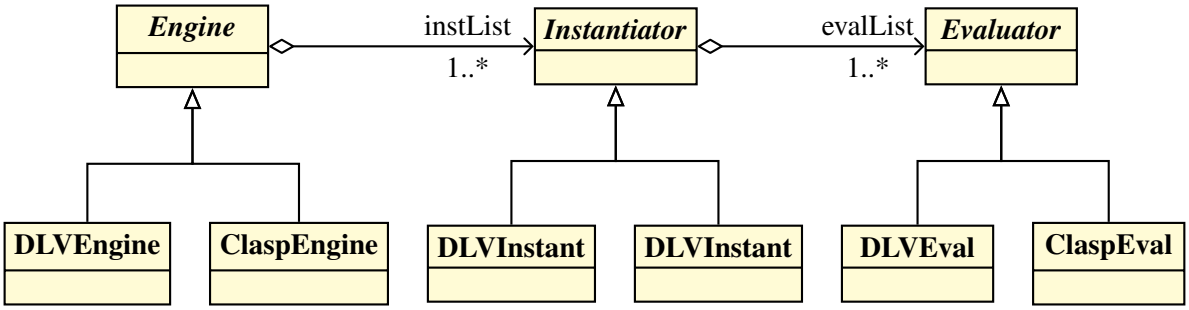


Figure 7.4: Class Diagram for Wrapping Local Solvers

notifications from JoinerDispatcher of the form (c, n), where c is the identifier of this context. Once this notification is received, Joiner goes to the respective NIn (coupled with the thread NOut that sent out the request) and reads out n partial equilibria.

After all neighbors have returned some partial equilibria, Joiner combines them and rules out inconsistent combinations, i.e., it processes the operator $\bowtie$ defined in Section 3.1. Outcomes of the joining process are passed to BridgeRulesEvaluator, which evaluate bridge rules of the current context to produce bridge heads of the applicable rules. These heads are added to the local theory, and each updated theory is then evaluated by Evaluator. The local models from this process are then combined with the corresponding partial equilibria (marked by having the same head) to form new resulting partial equilibria of the current context, for returning to OutputDispatcher.

If a cycle was detected, the request is sent to CycleBreaker, which makes guesses on the bridge atoms of the bridge rules, and then feeds each of these guesses into Evaluator for local solving. Therefore, in this case, no invocation to neighbors, no joining, nor bridge rules evaluation are needed. The local models computed from the guessed input will be sent to OutputDispatcher without being combined with input from the neighbors. Eventually, these models based on guessing will be fed into the same context later following the chain of calls. At that point, the combining of the output from Evaluator and the joint input from Joiner will have to check whether any inconsistency appears, and only consistent combinations can go through.

The only technical issue left for clarification is how one organizes the interface with external solvers, to execute the function Isolve inside Evaluator. We discuss this design next.

7.3 Wrapping the Local Solvers

Figure 7.3 shows that the local solving process is done in the Evaluator thread, which calls a particular engine instance associated with a local knowledge base of a context. The design presented in this section however pursues a more generic purpose, i.e., to provide a uniform interface to call different kinds of external solvers from a context in DMCS. To do this, three layers are needed as depicted by three abstract classes in in Figure 7.4:

- ***Engine***: subclasses of this class represent different kinds of reasoning engines, e.g.,

DVL,[1] clasp[2] for evaluating ASP programs, RacerPro [3] for ontological reasoning, relsat[4] for SAT solving, etc.

- ***Instantiator***: each engine can have multiple instantiators. Each instantiator takes care of a local knowledge base which is identified by a string (either a filename or a URL to the storage of the local knowledge base). In other words, an instantiator is corresponding to a local context.

- ***Evaluator***: this layer is present to allow future possibilities of parallelizing the local solving process. Evaluators created by an instantiator share a local knowledge base; more sophisticated algorithms are needed to control the parallelization of multiple evaluators in an efficient way.

Note that by exploiting polymorphism, the developer just needs to operate on pointers, say enp, inp, and evp to the three respective abstract classes ***Engine***, ***Instantiator***, ***Evaluator***. Choosing the type of engines for enp is done via calling a static method `create()` of the respective subclass of ***Engine***. Then, creating an instantiator for inp wrt a local knowledge base whose path is a string $kbspec$ is done via calling `createInstantiator(kbspec)` from enp. And finally, creating an evaluator for evp is done by calling `createEvaluator()` from inp. Once an evaluator is available via evp, local solving can be triggered through a method named `solve`.

7.4 DMCS System Usage

The DMCS implementation has two main command-line tools for generating test data and realizing the algorithms proposed in this thesis. This section present full instructions on invoking these tools.

Generating test cases with `dmcsgen`

The main purpose of `dmcsgen` is to aid experiments on the DMCS system with automatically generated test cases that reflecting different aspects of MCSs, including system topologies, system size, local theory size, interface size. `dmcsgen` also generates query plans and returns plans that define the interface between contexts at run time. The former defines the importing interface while the latter defines the exporting one. This replaces the current missing functionality of `dmcsm`. Moreover, command lines to run the whole system are also generated for the purpose of automatic testing. To invoke this tool, we use:

```
dmcsGen [OPTIONS]
```

where `OPTIONS` can be:

[1] `http://www.dlvsystem.com`
[2] `http://potassco.sourceforge.net/`
[3] `http://www.racer-systems.com/`
[4] `http://code.google.com/p/relsat/`

- `--help`: print help message
- `--gen-data=0/1`: `0` to disable generating data, i.e., only generate command lines, and `1` to enable this option
- `--contexts=N1`: set the number of contexts (system size)
- `--atoms=N2`: set the number of ground atoms per context (local theory size)
- `--interface=N3`: set the number of atoms used for creating the interface between contexts
- `--bridge-rules=N4`: set the maximum number of bridge rules between pairs of contexts. When generating, to vary the number of bridge rules, we iterate `N4` rounds, and for each round, have a probability of 50% to create a bridge rule.
- `--topology=N5`: set the topology type. There are in total 9 different types of topologies. Use option `--help` for more details.
- `--prefix=STR1`: set a string as prefix for all files generated in a single test case.
- `--dmcspath=STR2`: set the path to the DMCS binaries.
- `--startup-time=N6`: set start up time (in seconds) to call `dmcsc` after initializing all contexts.
- `--packsize=N7`: set the package size as the number of partial equilibria in each return message. `N7>0` triggers streaming mode while `N7=0` triggers the original DMCS algorithms (with or without the topological optimization).
- `--timeout=N8`: set a time out (in seconds).

As the result, `dmcsgen` generates

- `N1` text files with `.lp` extension, containing `N1` contexts' local theories
- `N1` text files with `.br` extension, containing sets of bridge rules of the contexts
- `N1` text files with `.qp` (resp., `.oqp`) extension, containing the query plans of the contexts in case of original (resp., optimal) topology.
- `N1` text files with `.rp` (resp., `.orp`) extension, containing the return plans of the contexts in case of original (resp., optimal) topology.
- a file called `client.qp` containing the signatures of all contexts in the system, for the purpose of returning results with atom names in stead of internal encoding to the user.
- several `.txt` files containing command lines that can be used to manually test the system.
- several `.sh` files containing shell scripts used to automatically test the system in different modes: original or optimal topologies, non-streaming or streaming algorithms, and in case of streaming, whether to finish after the first package of results.

Running the system with `dmcsm`, `dmcsc` and `dmcsd`

The actual DMCS system is activated via three binaries: a `dmcsm` simulates a simple manager, N `dmcsd` running as daemons to represent N contexts, and a `dmcsc` used as a client to trigger the querying to a `dmcsd`. We need to start these binaries in the following order:

First, start `dmcsm` as:

```
dmcsm [--help] --port=PORT --system-size=N
```

where `PORT` is the port where the manager listens to, and `N` is the number of contexts in the system.

Second, start `N` `dmcsd` as follows:

```
dmcsd [--help] OPTIONS
```

where `OPTIONS` are:

- `--context=N1`: set the context identifier (ranging from `0` to `N-1`).
- `--port=N2`: set the port where the current context listens to.
- `--manager=HOST:PORT`: set hostname and port of the manager (must be in correspondence to the options set by `dmcsm`).
- `--system-size=N3`: set system size, which is `N`.
- `--queue-size=N4`: (optional) set interal concurrent message queues' size.
- `--belief-state-size=N5`: set belief state size of every context. Note that to simplify the experiment, we let all contexts have the same local theory size, which is generated from the option `--atoms` of `dmcsgen`.
- `--packsize=N6`: (optional) set size of package, i.e., the number of partial belief states, to be transferred back in each return message.
- `--kb=STR1`: set the filename containing the local theory.
- `--br=STR2`: set the filename containing the bridge rules.
- `--queryplan=STR3`: set the filename containing the query plan wrt. the original topology
- `--optqueryplan=STR4`: set the filename containing the query plan wrt. the optimal topology
- `--returnplan=STR5`: set the filename containing the return plan. Depending on the mode (original or optimal) one would like to run, the return plan will be set to either generated files with extensions `.rp` or `.orp`.

Finally, start `dmcsc` as follows:

```
dmcsc [--help] OPTIONS
```

where `OPTIONS` are:

- `--hostname=STR1`: set hostname of the context to be queried.
- `--port=N1`: set port of the context to be queried.
- `--root=N2`: set indentifier of the context to be queried.
- `--signature=STR2`: set the filename to the file that contains signatures of all contexts in the system, which is `client.qp` in case of automatically generated.
- `--belief-state-size=N3`: set the (uniform) belief state size of all contexts in the system.
- `--loop=0/1`: set a flag to indicate whether we would like to immediately finish after the first round of answers, in streaming mode.
- `--k1=N4 --k2=N5`: set the range of partial belief states one would like to query. Setting `N4=N5=0` requests for all answers; otherwise, it must hold that `0 < N4 <= N5`.

Availability

Further information regarding our DMCS system can be found at `http://www.kr.tuwien.ac.at/research/systems/dmcs/index.html`. The source code is publicly hosted at `http://sourceforge.net/projects/dmcs/`.

CHAPTER 8

Experimental Evaluation

This Chapter provides a thorough experimental evaluation of different aspects of the prototype DMCS implementation describe in the Chapters above, to compare different algorithms (basic, topology-based optimization, and streaming) for evaluating partial equilibria (PE) in an MCS.

We carried out 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 use DLV [build BEN/Sep 28 2011 gcc 4.3.3] as a back-end ASP solver. Next, we explain how the benchmark is set up before going into the experimental and interpreting its results.

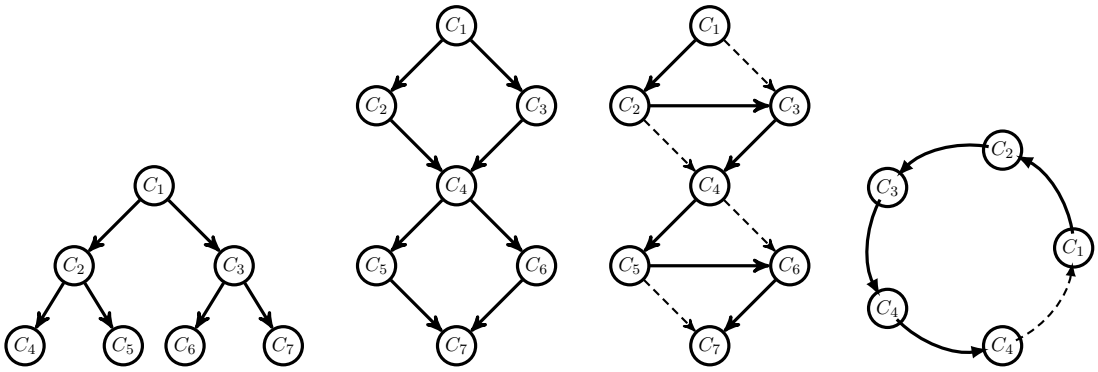
8.1 Benchmark Setup

The idea is to analyze the strong and weak points of each algorithm with respect to different parameters, namely system topology, system size, local theory size, and interface size. Specifically, we considered MCSs with topologies as in Figure 8.1, including:

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

- (Stack of) *Diamond* (D): a diamond consists of 4 nodes connecting as C_1 to C_4 in Figure 8.1b. 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 decomposition to get the query plan. Wrt 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, which helps reducing a significant amount of repetition of PEs in evaluation.

- (Stack of) *Zig-Zag* Diamonds (Z): a zig-zag diamond is an ordinary diamond with a connection between the two middle contexts, as depicted by contexts C_1 to C_4 in Figure 8.1c.



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

Figure 8.1: Topologies for testing DMCS algorithms

A stack of zig-zag diamonds is built in the same way as above. Moreover, this topology is interesting as after removing two edges per block, the query plan turns into a linear topology.

- *Ring* (R): unlike the above topologies that are acyclic, ring is a cyclic topology consisting of a cycle (Figure 8.1d). The query plan removes the connection from context C_n to C_1 and then carries the interface between these two all the way back to C_1. This topology requires guessing and checking in any DMCS algorithm, and this arbitrary factor makes it very unpredictable to say which algorithm performs better in general.

The other quantitative parameters are represented as a quadruple of the form $P = (n, s, b, r)$, where[1]

- 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. Note that when generating bridge rules, the generator uses a probability of 50% to generate a bridge rule while iterating from 1 to r, hence the average number of generated bridge rules is $r/2$. Furthermore, we allow bridge bodies of size 1 or 2.

Under this setting, a test configuration can be formulated as $X/(n, s, b, r)$ where $X \in \{T, D, Z, R\}$ representing the topology and n, s, b, r are integers representing the quantitative, or in other words, 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.

[1] See Appendix .1 for how to generate the test data wrt these parameters.

Figure 8.2: Local Theories' Structure

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:

- if j is odd then $k = j + 1$;
- 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 8.2. With this setting, a local context has 2^m answer sets, where $m \in [0, s/2]$.

8.2 Experiments

We ran an exhaustive set of benchmarks under the setup described in Section 8.1. Based on some initial testing while varying all the experiments, we decided for the parameter tuple $P = (n, s, b, r)$, we vary the following variables accordingly:

- n was chosen based on the topology:
  - T: $n \in \{7, 10, 15, 31, 70, 100\}$
  - D: $n \in \{4, 7, 10, 13, 25, 31\}$
  - Z: $n \in \{4, 7, 10, 13, 25, 31, 70\}$
  - R: $n \in \{4, 7, 10, 13, 70\}$
- s, b, r are fixed to either $10, 5, 5$ or $20, 10, 10$, respectively.

Each parameter setting is tested on 5 instances. For each instance, we measure the total running time and the total number of returned PEs on DMCS, DMCSOPT in non-streaming and streaming mode. In particular for the latter mode, DMCS-STREAMING, we asked for different number k of answers, namely 1, 10, and 100. This input parameter also influences the size of the packages transferred between contexts, i.e., at most k PEs are transferred in one message from a context to an invoker. As in streaming mode, asking for more than one PE may require more than one round to complete the total number of answers, it is interesting to see how fast the first set of answers arrives compared to the time consumed to get the whole answer. Therefore, with $k = 10$ and $k = 100$, we compare the running time of these two cases.

8.3 Observations and Interpretations

Tables 8.1—8.8 display the test results of the experiments as explained in Section 8.2. For the runs where both running time and total number of answers are reported, or in streaming mode with $k = 1$, we used "—" to denote timeout. Otherwise, the syntax (m) is used to denote that the test run was timed out and m PEs were received up to that time. From these data, several interesting properties can be observed and the interpretation opens up a number of interesting issues for further deep investigation.

In the sequel, we present our analysis of the data on the following aspects:

- comparing DMCS and DMCSOPT
- comparing streaming and non-streaming modes
- effect of the package size
- role of the topologies

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
T_1 / $(7,10,5,5)$	0.15	50	0.03	7	0.01	0.01	0.17	10	0.16	0.41	7	1.29	0.57	50	1.83	0.52	7	1.70
T_2 / $(7,10,5,5)$	0.22	192	0.04	12	0.01	0.01	0.16	10	0.16	0.15	10	0.14	0.27	100	0.26	0.55	12	1.78
T_3 / $(7,10,5,5)$	0.41	192	0.04	7	0.01	0.01	0.08	10	0.08	0.17	7	0.68	0.41	100	0.41	0.55	7	1.80
T_4 / $(7,10,5,5)$	2.61	1536	0.08	36	0.01	0.01	0.10	10	0.10	0.05	10	0.05	0.84	100	0.84	0.59	36	1.89
T_5 / $(7,10,5,5)$	4.90	2656	0.08	28	0.01	0.01	0.10	10	0.10	0.04	10	0.05	0.94	56	4.95	0.59	28	1.84
T_1 / $(10,10,5,5)$	0.23	160	0.04	12	0.01	0.02	0.17	10	0.17	0.73	10	0.73	0.58	100	0.61	1.18	12	3.68
T_2 / $(10,10,5,5)$	1.71	1104	0.05	12	0.02	0.02	0.15	10	0.15	0.57	6	2.74	0.97	100	0.98	1.26	12	3.90
T_3 / $(10,10,5,5)$	12.78	5760	0.06	18	0.02	0.01	0.07	10	0.07	0.18	10	0.18	1.30	100	1.30	1.29	18	4.01
T_4 / $(10,10,5,5)$	18.90	9600	0.09	32	0.02	0.02	0.08	10	0.08	0.05	10	0.05	1.51	100	1.51	1.25	32	3.85
T_5 / $(10,10,5,5)$	176.33	5632	0.09	4	0.02	0.02	0.06	6	0.43	0.23	4	2.25	0.80	52	4.32	1.72	4	5.25
T_1 / $(25,10,5,5)$	—	0	0.12	12	0.03	0.03	0.32	10	0.31	0.56	10	0.55	10.07	100	10.09	4.70	12	14.47
T_2 / $(25,10,5,5)$	—	0	0.16	28	0.03	0.03	0.31	10	0.30	0.47	10	0.47	10.29	100	10.28	5.17	28	15.83
T_3 / $(25,10,5,5)$	—	0	0.16	3	0.03	0.03	0.29	10	0.28	2.16	3	52.42	7.87	100	7.86	9.27	3	27.81
T_4 / $(25,10,5,5)$	—	0	0.23	24	0.03	0.03	0.12	10	0.12	0.09	10	0.09	2.53	100	2.53	12.04	24	36.55
T_5 / $(25,10,5,5)$	—	0	0.25	88	0.03	0.03	0.19	10	0.18	0.32	10	0.33	1.78	100	1.79	1.89	88	5.70
T_1 / $(31,10,5,5)$	—	0	0.16	48	0.04	0.04	0.28	10	0.28	0.21	10	0.22	3.40	100	3.32	5.64	48	16.98
T_2 / $(31,10,5,5)$	—	0	0.16	16	0.04	0.07	0.37	10	0.33	1.60	10	1.54	7.60	100	7.67	5.44	16	16.51
T_3 / $(31,10,5,5)$	—	0	0.18	24	0.04	0.04	0.32	10	0.36	0.89	8	2.83	6.83	100	6.84	5.51	24	16.77
T_4 / $(31,10,5,5)$	—	0	0.24	18	0.03	0.04	0.24	10	0.23	0.21	10	0.20	5.89	100	5.94	20.13	18	61.39
T_5 / $(31,10,5,5)$	—	0	0.38	24	0.04	0.04	0.20	10	0.20	0.10	6	0.38	8.08	100	8.11	6.02	24	206.09
T_1 / $(70,10,5,5)$	—	0	0.29	8	0.08	0.08	0.79	10	0.82	4.24	8	12.99	23.61	100	22.87	84.82	8	105.88
T_2 / $(70,10,5,5)$	—	0	0.31	16	0.08	0.08	0.39	10	0.40	1.13	10	1.12	11.12	100	11.18	29.70	16	94.31
T_3 / $(70,10,5,5)$	—	0	0.31	8	0.08	0.08	0.44	10	0.43	1.21	8	3.79	15.24	100	15.03	28.14	8	85.29
T_4 / $(70,10,5,5)$	—	0	0.35	8	0.08	0.07	0.43	10	0.41	43.78	8	133.24	13.29	100	13.31	43.94	8	155.90
T_5 / $(70,10,5,5)$	—	0	0.58	28	0.08	0.08	0.54	10	0.59	2.96	10	2.96	27.71	100	27.72	95.95	28	333.49
T_1 / $(100,10,5,5)$	—	0	0.47	32	0.11	0.11	0.54	10	0.56	1.85	10	1.77	19.13	100	19.58	46.17	32	142.32
T_2 / $(100,10,5,5)$	—	0	0.50	10	0.11	0.11	0.52	10	0.50	1.01	6	3.27	19.93	100	20.10	55.50	10	168.88
T_3 / $(100,10,5,5)$	—	0	0.53	84	0.13	0.11	0.59	10	0.58	2.05	10	1.99	19.78	100	20.06	89.52	84	276.10
T_4 / $(100,10,5,5)$	—	0	0.59	72	0.11	0.11	0.57	10	0.58	7.19	10	7.25	17.29	100	17.30	118.75	72	378.13
T_5 / $(100,10,5,5)$	—	0	0.64	36	0.11	0.11	0.49	10	0.48	0.34	10	0.34	15.59	100	15.01	217.98	36	(36)

Table 8.1: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
T_1 / (7, 20, 10, 10)	337.58	25344	0.72	736	0.01	0.01	0.06	10	0.06	0.05	10	0.05	2.42	68	7.44	0.30	100	0.31
T_2 / (7, 20, 10, 10)	—	0	0.58	208	0.01	0.01	0.05	10	0.05	0.05	10	0.05	0.37	100	0.37	0.19	100	0.18
T_3 / (7, 20, 10, 10)	—	0	1.19	840	0.01	0.01	0.05	10	0.05	0.05	10	0.05	0.67	100	0.67	0.23	100	0.23
T_4 / (7, 20, 10, 10)	—	0	1.74	2016	0.01	0.01	0.05	10	0.05	0.04	10	0.04	2.56	100	2.56	0.24	100	0.25
T_5 / (7, 20, 10, 10)	—	0	30.51	17088	0.01	0.01	0.08	10	0.08	0.08	10	0.08	3.71	100	3.72	1.10	100	1.10
T_1 / (10, 20, 10, 10)	—	0	0.48	224	0.02	0.02	0.06	10	0.06	0.05	10	0.05	2.11	60	8.69	0.46	100	0.46
T_2 / (10, 20, 10, 10)	—	0	0.85	256	0.02	0.02	0.06	10	0.05	0.05	10	0.05	0.39	100	0.39	0.46	64	1.95
T_3 / (10, 20, 10, 10)	—	0	1.94	1792	0.02	0.02	0.04	10	0.05	0.04	10	0.04	0.32	100	0.33	0.20	100	0.20
T_4 / (10, 20, 10, 10)	—	0	2.64	176	0.02	0.02	0.11	10	0.11	0.10	10	0.10	1.13	100	1.14	0.36	100	0.36
T_5 / (10, 20, 10, 10)	—	0	3.25	3200	0.02	0.02	0.06	10	0.06	0.05	10	0.05	0.99	100	1.01	0.42	100	0.41
T_1 / (25, 20, 10, 10)	—	0	0.57	224	0.03	0.03	0.47	10	0.46	0.42	10	0.42	8.30	100	8.35	10.56	100	10.55
T_2 / (25, 20, 10, 10)	—	0	1.07	576	0.03	0.03	0.08	10	0.09	0.08	10	0.08	1.09	100	1.09	0.35	100	0.35
T_3 / (25, 20, 10, 10)	—	0	1.56	768	0.03	0.03	0.11	10	0.10	0.10	10	0.10	2.79	100	2.75	0.48	100	0.47
T_4 / (25, 20, 10, 10)	—	0	3.29	40	0.03	0.03	0.34	10	0.36	0.31	10	0.31	3.15	100	3.14	1.30	40	10.09
T_5 / (25, 20, 10, 10)	—	0	5.69	336	0.03	0.03	0.14	10	0.12	0.13	10	0.13	1.98	100	1.95	0.44	100	0.46
T_1 / (31, 20, 10, 10)	—	0	1.43	72	0.04	0.04	0.12	10	0.11	0.10	10	0.10	2.30	100	2.26	0.56	72	1.75
T_2 / (31, 20, 10, 10)	—	0	2.25	672	0.04	0.04	0.12	10	0.14	0.12	10	0.12	2.22	100	2.27	0.72	100	0.68
T_3 / (31, 20, 10, 10)	—	0	2.96	32	0.04	0.04	0.14	10	0.13	0.20	10	0.20	2.98	100	2.99	2.93	32	72.68
T_4 / (31, 20, 10, 10)	—	0	3.36	896	0.04	0.04	0.16	10	0.17	0.12	10	0.12	3.72	100	3.72	0.81	100	0.80
T_5 / (31, 20, 10, 10)	—	0	22.53	384	0.04	0.04	0.17	10	0.18	0.13	10	0.15	7.71	100	7.74	0.77	100	0.81
T_1 / (70, 20, 10, 10)	—	0	3.52	144	0.08	0.08	0.31	10	0.32	0.24	10	0.23	11.44	100	11.38	1.56	80	4.63
T_2 / (70, 20, 10, 10)	—	0	5.35	192	0.08	0.08	0.50	10	0.47	0.30	10	0.30	7.83	100	7.82	7.56	64	23.91
T_3 / (70, 20, 10, 10)	—	0	5.95	576	0.08	0.08	0.24	10	0.23	0.20	10	0.22	5.16	100	5.20	0.99	100	0.94
T_4 / (70, 20, 10, 10)	—	0	7.02	648	0.09	0.08	0.35	10	0.35	0.21	10	0.28	5.82	100	5.85	1.21	100	1.24
T_5 / (70, 20, 10, 10)	—	0	20.31	224	0.08	0.08	0.29	10	0.29	0.25	10	0.24	17.89	100	24.81	2.75	100	3.85
T_1 / (100, 20, 10, 10)	—	0	6.34	1056	0.11	0.11	0.32	10	0.33	0.55	10	0.58	6.57	100	6.51	11.74	100	11.82
T_2 / (100, 20, 10, 10)	—	0	6.40	704	0.12	0.11	0.33	10	0.33	0.28	10	0.30	6.06	100	6.11	1.59	100	1.57
T_3 / (100, 20, 10, 10)	—	0	8.31	1280	0.11	0.11	0.54	10	0.55	0.89	10	0.86	7.19	100	7.46	2.87	100	2.90
T_4 / (100, 20, 10, 10)	—	0	25.45	2688	0.11	0.11	0.34	10	0.35	0.31	10	0.32	11.61	100	11.82	1.58	100	1.57
T_5 / (100, 20, 10, 10)	—	0	32.63	160	0.11	0.11	0.40	10	0.38	0.35	10	0.36	8.21	100	7.91	3.70	96	11.26

Table 8.2: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
D_1 / $(4,10,5,5)$	0.02	8	0.02	8	0.01	0.01	0.26	8	1.83	0.26	8	1.82	0.42	8	1.40	0.42	8	1.40
D_2 / $(4,10,5,5)$	0.25	196	0.09	86	0.01	0.00	0.16	10	0.16	0.11	10	0.11	0.25	100	0.25	0.50	86	1.51
D_3 / $(4,10,5,5)$	0.40	192	0.07	24	0.01	0.01	0.03	10	0.03	0.02	6	0.30	0.27	76	1.20	0.68	24	2.18
D_4 / $(4,10,5,5)$	0.52	168	0.06	24	0.01	0.01	0.04	6	0.22	0.03	6	0.21	0.45	52	3.04	0.48	24	1.56
D_5 / $(4,10,5,5)$	1.90	1120	0.13	84	0.01	0.01	0.04	10	0.04	0.03	10	0.03	0.66	100	0.67	0.31	84	0.89
D_1 / $(7,10,5,5)$	2.67	1568	0.28	112	0.14	0.14	0.37	10	0.37	0.36	10	0.36	1.45	100	1.45	1.99	100	1.99
D_2 / $(7,10,5,5)$	3.74	320	0.23	16	0.14	0.14	0.21	10	0.20	0.42	10	0.41	2.43	100	2.43	3.91	16	11.96
D_3 / $(7,10,5,5)$	10.44	3264	0.36	72	0.14	0.14	0.22	10	0.21	0.21	10	0.21	1.04	100	1.04	5.23	72	15.92
D_4 / $(7,10,5,5)$	12.93	1536	0.30	24	0.14	0.14	0.39	10	0.39	0.37	6	1.26	2.75	100	2.77	4.07	24	12.44
D_5 / $(7,10,5,5)$	13.68	1536	0.43	48	0.14	0.14	0.43	10	0.43	1.58	10	1.58	3.37	100	3.35	4.70	48	14.52
D_1 / $(10,10,5,5)$	24.15	1728	0.56	16	0.32	0.32	0.84	10	0.84	1.30	10	1.32	4.70	100	4.66	32.60	16	101.94
D_2 / $(10,10,5,5)$	50.15	1920	0.54	16	0.32	0.32	1.23	10	1.22	2.27	10	2.27	5.70	100	5.69	32.69	16	98.45
D_3 / $(10,10,5,5)$	—	0	1.03	3	0.32	0.32	26.77	10	26.64	3.37	3	81.80	—	0	(0)	457.01	3	(3)
D_4 / $(10,10,5,5)$	—	0	1.15	72	0.32	0.32	0.63	10	0.54	1.14	10	1.21	7.75	100	7.75	52.37	72	177.31
D_5 / $(10,10,5,5)$	—	0	2.00	8	0.32	0.32	0.55	10	0.55	2.09	8	6.40	9.10	56	74.24	278.94	8	(8)
D_1 / $(13,10,5,5)$	—	0	1.65	72	0.72	0.83	1.62	10	1.65	1.61	8	5.20	12.16	100	12.17	162.78	52	(76)
D_2 / $(13,10,5,5)$	—	0	2.44	8	0.79	0.76	1.16	6	3.58	1.17	6	3.60	34.45	28	393.00	—	0	(0)
D_3 / $(13,10,5,5)$	—	0	4.04	6	0.75	0.71	1.21	6	2.57	4.96	6	21.89	19.09	60	117.02	—	0	(0)
D_4 / $(13,10,5,5)$	—	0	4.38	24	0.83	0.80	1.81	10	1.80	1.82	10	1.81	16.16	52	117.91	—	0	(0)
D_5 / $(13,10,5,5)$	—	0	9.03	64	0.76	0.72	1.27	10	1.17	1.19	10	1.16	12.51	100	12.53	—	0	(0)
D_1 / $(25,10,5,5)$	—	0	37.50	9	13.81	13.74	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
D_2 / $(25,10,5,5)$	—	0	78.68	168	13.66	13.46	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
D_3 / $(25,10,5,5)$	—	0	96.37	16	13.70	13.67	31.66	10	30.97	361.19	10	361.28	499.09	100	501.42	—	0	(0)
D_4 / $(25,10,5,5)$	—	0	113.33	12	13.90	13.94	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
D_5 / $(25,10,5,5)$	—	0	124.88	19	13.86	14.07	123.47	10	122.82	211.46	10	211.30	—	0	(0)	—	0	(0)
D_1 / $(31,10,5,5)$	—	0	142.19	6	55.22	55.46	177.68	10	176.17	—	0	(0)	—	0	(0)	—	0	(0)
D_2 / $(31,10,5,5)$	—	0	185.71	112	55.43	55.37	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
D_3 / $(31,10,5,5)$	—	0	312.69	56	56.39	56.04	211.35	10	211.47	—	0	(0)	—	0	(0)	—	0	(0)
D_4 / $(31,10,5,5)$	—	0	—	0	55.78	55.12	87.42	10	88.47	209.15	10	209.50	—	0	(0)	—	0	(0)
D_5 / $(31,10,5,5)$	—	0	—	0	56.97	56.15	237.93	10	239.03	—	0	(0)	—	0	(0)	—	0	(0)

Table 8.3: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
D_1 / (4, 20, 10, 10)	0.92	1280	0.72	1024	0.01	0.01	0.03	10	0.03	0.02	10	0.02	0.18	100	0.18	0.16	100	0.16
D_2 / (4, 20, 10, 10)	1.74	960	0.34	288	0.01	0.01	0.02	10	0.02	0.02	10	0.02	0.19	60	1.39	0.14	100	0.14
D_3 / (4, 20, 10, 10)	6.96	1776	0.65	192	0.01	0.01	0.02	10	0.02	0.02	10	0.02	0.38	100	0.43	0.26	36	3.31
D_4 / (4, 20, 10, 10)	7.96	896	0.47	224	0.01	0.01	0.04	10	0.04	0.04	10	0.04	0.89	100	0.88	0.25	100	0.25
D_5 / (4, 20, 10, 10)	—	0	5.00	2016	0.01	0.01	0.03	10	0.03	0.02	10	0.03	0.40	100	0.41	0.16	100	0.17
D_1 / (7, 20, 10, 10)	165.29	4096	0.84	36	0.14	0.14	0.19	10	0.19	0.18	10	0.19	1.18	100	1.19	3.98	36	74.03
D_2 / (7, 20, 10, 10)	—	0	1.92	256	0.14	0.14	0.21	10	0.21	0.21	10	0.21	1.69	100	1.66	4.53	100	4.47
D_3 / (7, 20, 10, 10)	—	0	7.19	336	0.14	0.14	0.19	10	0.19	0.19	10	0.19	1.31	100	1.29	3.16	100	3.11
D_4 / (7, 20, 10, 10)	—	0	45.32	1248	0.14	0.14	0.17	10	0.17	0.17	10	0.17	0.50	100	0.50	0.32	100	0.33
D_5 / (7, 20, 10, 10)	—	0	485.04	1600	0.14	0.14	0.19	10	0.19	0.19	10	0.19	1.70	100	1.69	1.28	100	1.27
D_1 / (10, 20, 10, 10)	—	0	7.90	128	0.32	0.32	0.41	10	0.41	0.41	10	0.41	—	0	(0)	8.16	100	8.18
D_2 / (10, 20, 10, 10)	—	0	41.37	114	0.32	0.32	0.45	10	0.45	0.56	10	0.56	14.82	100	14.82	34.93	100	34.94
D_3 / (10, 20, 10, 10)	—	0	85.45	126	0.32	0.32	0.45	10	0.46	0.46	10	0.53	16.97	100	16.99	65.43	100	65.44
D_4 / (10, 20, 10, 10)	—	0	256.82	13600	0.32	0.32	0.57	10	0.57	0.56	10	0.56	5.75	100	5.75	8.94	100	8.94
D_5 / (10, 20, 10, 10)	—	0	—	0	0.32	0.32	0.39	10	0.39	0.40	10	0.39	2.21	100	2.22	2.11	100	2.11
D_1 / (13, 20, 10, 10)	—	0	38.97	576	0.76	0.80	0.99	10	0.99	0.99	10	0.99	5.44	100	5.45	5.11	100	5.12
D_2 / (13, 20, 10, 10)	—	0	46.27	480	0.72	0.72	1.14	10	1.11	1.05	10	1.06	9.00	100	9.07	23.61	100	23.64
D_3 / (13, 20, 10, 10)	—	0	288.35	90	0.76	0.72	1.54	10	1.47	1.32	8	4.82	9.28	94	18.52	5.28	90	15.81
D_4 / (13, 20, 10, 10)	—	0	295.47	240	0.83	0.76	1.20	10	1.05	1.12	10	1.19	6.80	100	6.84	22.24	100	22.24
D_5 / (13, 20, 10, 10)	—	0	—	0	0.72	0.76	1.19	10	1.17	1.15	10	1.16	11.77	100	11.65	12.20	100	12.16
D_1 / (25, 20, 10, 10)	—	0	586.73	912	14.08	13.94	19.09	10	19.30	19.09	10	19.15	54.95	100	55.12	53.23	100	53.22
D_2 / (25, 20, 10, 10)	—	0	—	0	13.76	13.83	28.29	10	28.40	28.47	10	28.67	—	0	(0)	—	0	(0)
D_3 / (25, 20, 10, 10)	—	0	—	0	13.83	13.87	20.62	10	20.66	21.14	10	21.02	—	0	(0)	—	0	(0)
D_4 / (25, 20, 10, 10)	—	0	—	0	13.90	13.53	19.13	10	18.99	19.24	10	19.20	—	0	(0)	—	0	(0)
D_5 / (25, 20, 10, 10)	—	0	—	0	14.01	13.99	—	0	(0)	73.07	10	73.51	—	0	(0)	—	0	(0)
D_1 / (31, 20, 10, 10)	—	0	—	0	56.51	56.05	234.31	10	236.10	183.22	10	182.01	—	0	(0)	—	0	(0)
D_2 / (31, 20, 10, 10)	—	0	—	0	55.84	55.76	77.41	10	76.85	77.29	10	77.53	—	0	(0)	—	0	(0)
D_3 / (31, 20, 10, 10)	—	0	—	0	55.95	56.05	71.48	10	72.84	112.35	10	112.14	—	0	(0)	—	0	(0)
D_4 / (31, 20, 10, 10)	—	0	—	0	55.99	56.31	76.28	10	76.23	76.07	10	76.28	—	0	(0)	—	0	(0)
D_5 / (31, 20, 10, 10)	—	0	—	0	56.36	54.91	70.65	10	69.50	68.75	10	69.23	—	0	(0)	—	0	(0)

Table 8.4: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
Z_1 / $(4,10,5,5)$	0.17	40	0.04	16	0.09	0.01	0.34	10	0.34	0.23	8	0.88	0.99	40	3.20	0.69	16	2.27
Z_2 / $(4,10,5,5)$	0.20	36	0.05	12	0.09	0.01	0.65	8	2.84	0.19	4	0.51	1.13	36	3.56	0.69	12	2.26
Z_3 / $(4,10,5,5)$	0.43	160	0.11	48	0.09	0.01	0.52	10	0.52	0.19	6	0.57	0.72	100	0.72	0.38	36	4.00
Z_4 / $(4,10,5,5)$	0.63	224	0.12	32	0.09	0.01	0.24	10	0.24	0.04	8	0.37	0.86	100	0.86	0.37	16	5.03
Z_5 / $(4,10,5,5)$	1.09	88	0.15	16	0.09	0.01	0.19	10	0.19	0.03	8	0.26	1.97	88	5.95	0.66	16	2.33
Z_1 / $(7,10,5,5)$	1.97	192	0.06	8	0.51	0.02	1.74	10	1.74	5.20	6	15.83	10.31	100	10.30	6.25	8	19.05
Z_2 / $(7,10,5,5)$	4.50	220	0.11	14	0.51	0.02	2.17	10	2.16	0.21	10	0.21	8.56	100	8.56	6.45	14	19.66
Z_3 / $(7,10,5,5)$	17.80	286	0.12	8	0.51	0.02	0.65	10	0.65	0.05	8	1.88	6.62	100	6.62	6.51	8	19.78
Z_4 / $(7,10,5,5)$	143.21	3328	0.23	24	0.51	0.02	0.63	10	0.62	0.05	8	0.52	13.09	100	13.11	7.60	24	70.39
Z_5 / $(7,10,5,5)$	—	0	0.42	48	0.51	0.02	0.63	10	0.63	0.35	6	1.02	6.69	100	6.63	4.31	24	147.65
Z_1 / $(10,10,5,5)$	437.45	1344	0.10	4	1.83	0.02	6.32	8	12.73	1.72	4	37.20	47.15	96	94.76	49.73	4	152.54
Z_2 / $(10,10,5,5)$	—	0	0.15	16	1.83	0.02	2.18	10	2.23	1.07	8	3.45	26.10	100	26.22	26.41	16	254.15
Z_3 / $(10,10,5,5)$	—	0	0.19	36	1.83	0.02	5.94	10	5.93	0.47	10	0.47	28.47	100	28.53	89.41	36	338.14
Z_4 / $(10,10,5,5)$	—	0	0.21	8	1.84	0.02	2.26	10	2.20	1.61	4	19.45	30.19	70	(70)	55.95	8	279.51
Z_5 / $(10,10,5,5)$	—	0	0.29	16	1.83	0.02	144.31	10	144.17	18.80	4	324.72	269.74	100	270.24	103.39	16	(16)
Z_1 / $(13,10,5,5)$	—	0	0.36	40	5.81	0.03	8.62	10	8.66	1.51	6	8.39	125.21	100	125.77	467.77	22	(22)
Z_2 / $(13,10,5,5)$	—	0	0.42	48	5.82	0.03	11.46	10	11.66	0.75	10	0.75	—	0	(0)	—	0	(0)
Z_3 / $(13,10,5,5)$	—	0	0.58	12	5.80	0.03	28.82	6	93.61	0.93	6	1.90	—	0	(0)	—	0	(0)
Z_4 / $(13,10,5,5)$	—	0	0.88	4	5.80	0.03	37.81	6	197.21	10.54	4	121.94	—	0	(0)	—	0	(0)
Z_5 / $(13,10,5,5)$	—	0	0.88	60	5.82	0.03	11.50	10	11.47	0.48	10	0.48	136.36	52	(100)	248.20	24	(24)
Z_1 / $(25,10,5,5)$	—	0	0.57	16	516.79	0.06	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_2 / $(25,10,5,5)$	—	0	0.66	8	520.64	0.06	—	0	(0)	32.84	6	103.05	—	0	(0)	—	0	(0)
Z_3 / $(25,10,5,5)$	—	0	0.77	12	521.11	0.06	—	0	(0)	476.09	8	(8)	—	0	(0)	—	0	(0)
Z_4 / $(25,10,5,5)$	—	0	0.96	24	510.86	0.06	—	0	(0)	19.62	10	19.50	—	0	(0)	—	0	(0)
Z_5 / $(25,10,5,5)$	—	0	1.99	32	518.69	0.06	—	0	(0)	4.63	8	18.84	—	0	(0)	—	0	(0)
Z_1 / $(31,10,5,5)$	—	0	0.60	8	—	0.07	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_2 / $(31,10,5,5)$	—	0	0.90	20	—	0.07	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_3 / $(31,10,5,5)$	—	0	1.54	36	—	0.07	—	0	(0)	28.75	8	57.67	—	0	(0)	—	0	(0)
Z_4 / $(31,10,5,5)$	—	0	1.73	16	—	0.07	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_5 / $(31,10,5,5)$	—	0	2.05	14	—	0.07	—	0	(0)	23.34	10	20.05	—	0	(0)	—	0	(0)
Z_1 / $(70,10,5,5)$	—	0	1.51	6	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_2 / $(70,10,5,5)$	—	0	2.31	21	—	0.17	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_3 / $(70,10,5,5)$	—	0	2.31	40	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_4 / $(70,10,5,5)$	—	0	3.37	33	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
Z_5 / $(70,10,5,5)$	—	0	3.57	72	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)

Table 8.5: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
Z_1 / $(4,20,10,10)$	6.90	4288	2.37	2176	0.09	0.01	0.18	10	0.17	0.03	10	0.03	1.17	100	1.17	0.18	48	0.90
Z_2 / $(4,20,10,10)$	23.24	7168	4.33	1024	0.09	0.01	0.12	10	0.12	0.02	10	0.02	0.39	100	0.39	0.10	68	0.34
Z_3 / $(4,20,10,10)$	25.97	4608	2.66	640	0.09	0.01	0.12	10	0.12	0.02	10	0.02	0.59	100	0.58	0.11	84	0.45
Z_4 / $(4,20,10,10)$	46.89	17920	14.57	1536	0.09	0.01	0.13	10	0.13	0.02	10	0.02	0.74	100	0.75	0.10	64	0.49
Z_5 / $(4,20,10,10)$	—	0	—	0	0.09	0.01	0.18	10	0.17	0.02	10	0.02	4.44	64	13.45	0.16	64	0.59
Z_1 / $(7,20,10,10)$	6.22	1060	0.19	84	0.51	0.02	1.03	10	1.02	0.17	10	0.17	4.41	100	4.39	3.28	44	23.70
Z_2 / $(7,20,10,10)$	—	0	3.66	64	0.51	0.02	0.80	6	3.29	0.34	6	1.22	—	0	(0)	1.95	52	14.86
Z_3 / $(7,20,10,10)$	—	0	4.73	448	0.51	0.02	0.64	10	0.62	0.04	10	0.04	9.30	100	9.34	0.46	32	2.53
Z_4 / $(7,20,10,10)$	—	0	32.64	768	0.51	0.02	0.66	10	0.66	0.04	10	0.04	8.69	100	8.72	0.26	80	1.42
Z_5 / $(7,20,10,10)$	—	0	403.90	3984	0.51	0.02	0.60	10	0.60	0.04	10	0.04	5.12	100	5.05	0.22	40	2.19
Z_1 / $(10,20,10,10)$	—	0	11.61	48	1.85	0.02	32.71	4	244.10	0.72	6	6.18	—	0	(0)	3.97	12	(88)
Z_2 / $(10,20,10,10)$	—	0	16.84	720	1.84	0.02	2.24	10	2.26	0.05	10	0.05	—	0	(0)	0.85	48	4.60
Z_3 / $(10,20,10,10)$	—	0	37.40	192	1.85	0.02	2.30	10	2.28	0.06	10	0.06	18.75	100	18.84	0.36	24	3.55
Z_4 / $(10,20,10,10)$	—	0	123.55	256	1.84	0.02	2.20	10	2.28	0.06	10	0.06	10.99	84	22.02	1.14	16	26.35
Z_5 / $(10,20,10,10)$	—	0	269.31	168	1.81	0.02	2.27	10	2.28	0.06	10	0.06	43.20	100	43.23	2.85	20	24.22
Z_1 / $(13,20,10,10)$	—	0	46.74	168	6.02	0.04	12.04	10	11.76	0.42	10	0.41	—	0	(0)	0.65	68	2.62
Z_2 / $(13,20,10,10)$	—	0	63.69	536	5.80	0.03	7.80	10	7.83	0.07	10	0.07	119.56	100	121.83	0.49	64	1.55
Z_3 / $(13,20,10,10)$	—	0	86.68	288	8.47	0.36	11.70	10	11.25	0.58	10	0.62	95.33	100	173.33	1.59	36	7.24
Z_4 / $(13,20,10,10)$	—	0	114.47	48	5.84	0.03	7.05	10	7.08	0.07	10	0.07	65.44	100	65.38	1.35	32	10.32
Z_5 / $(13,20,10,10)$	—	0	—	0	5.85	0.03	11.77	10	11.79	0.16	10	0.16	97.47	52	404.02	1.08	100	1.08
Z_1 / $(25,20,10,10)$	—	0	88.29	1280	527.30	0.06	—	0	(0)	0.14	10	0.14	—	0	(0)	7.81	32	32.97
Z_2 / $(25,20,10,10)$	—	0	109.93	224	525.83	0.06	—	0	(0)	0.14	10	0.14	—	0	(0)	2.78	16	23.35
Z_3 / $(25,20,10,10)$	—	0	200.73	300	526.13	0.06	—	0	(0)	0.13	10	0.13	—	0	(0)	0.86	36	8.17
Z_4 / $(25,20,10,10)$	—	0	267.01	576	513.87	0.06	—	0	(0)	0.82	10	0.78	—	0	(0)	—	0	(0)
Z_5 / $(25,20,10,10)$	—	0	—	0	526.88	0.06	598.36	10	599.26	0.13	10	0.13	—	0	(0)	5.28	40	54.81
Z_1 / $(31,20,10,10)$	—	0	50.29	768	—	0.08	—	0	(0)	27.55	10	27.53	—	0	(0)	—	0	(0)
Z_2 / $(31,20,10,10)$	—	0	134.79	1024	—	0.08	—	0	(0)	0.18	10	0.18	—	0	(0)	41.91	64	188.89
Z_3 / $(31,20,10,10)$	—	0	142.42	96	—	0.08	—	0	(0)	0.60	10	0.60	—	0	(0)	29.81	16	412.42
Z_4 / $(31,20,10,10)$	—	0	326.03	44	—	0.08	—	0	(0)	1.76	10	1.77	—	0	(0)	—	0	(0)
Z_5 / $(31,20,10,10)$	—	0	—	0	—	0.08	—	0	(0)	1.57	10	1.60	—	0	(0)	22.90	100	22.87
Z_1 / $(70,20,10,10)$	—	0	371.55	1120	—	0.18	—	0	(0)	3.83	10	3.83	—	0	(0)	—	0	(0)
Z_2 / $(70,20,10,10)$	—	0	373.68	144	—	0.18	—	0	(0)	6.74	10	6.78	—	0	(0)	—	0	(0)
Z_3 / $(70,20,10,10)$	—	0	477.78	512	—	0.18	—	0	(0)	0.99	10	0.99	—	0	(0)	—	0	(0)
Z_4 / $(70,20,10,10)$	—	0	—	0	—	0.18	—	0	(0)	9.85	10	9.82	—	0	(0)	—	0	(0)
Z_5 / $(70,20,10,10)$	—	0	—	0	—	0.18	—	0	(0)	1.04	10	1.08	—	0	(0)	—	0	(0)

Table 8.6: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
R_1 / $(4,10,5,5)$	0.38	40	0.33	12	0.17	0.01	1.84	10	1.84	1.54	6	59.99	1.93	40	6.40	1.63	12	5.67
R_2 / $(4,10,5,5)$	0.71	60	0.24	18	—	299.68	7.92	10	7.92	0.47	10	0.47	4.35	60	16.55	1.07	18	3.65
R_3 / $(4,10,5,5)$	0.79	96	0.10	8	0.01	—	0.24	10	0.24	3.51	4	24.49	4.08	96	13.31	0.86	8	2.78
R_4 / $(4,10,5,5)$	1.57	28	0.14	4	—	154.89	11.35	10	11.32	2.82	4	31.24	9.77	28	55.65	1.19	4	4.01
R_5 / $(4,10,5,5)$	1.85	144	0.16	24	—	—	14.85	10	14.84	2.18	6	9.07	2.65	100	2.67	0.92	24	3.18
R_1 / $(7,10,5,5)$	8.44	624	0.27	14	0.64	—	2.11	9	4.27	16.64	7	52.55	4.91	99	10.38	11.38	14	37.35
R_2 / $(7,10,5,5)$	13.47	120	0.18	24	—	0.02	—	0	(0)	4.50	6	16.38	—	0	(0)	10.01	24	32.23
R_3 / $(7,10,5,5)$	62.09	1104	0.14	4	0.31	0.02	6.66	9	20.29	106.75	2	(2)	9.63	100	9.58	8.99	4	28.96
R_4 / $(7,10,5,5)$	130.29	2268	0.37	28	0.02	—	1.82	10	1.83	62.29	4	258.51	5.18	100	5.23	18.54	28	60.03
R_5 / $(7,10,5,5)$	—	0	1.14	80	—	—	—	0	(0)	—	0	(0)	19.57	100	19.58	—	0	(0)
R_1 / $(10,10,5,5)$	199.38	1994	0.59	14	0.03	—	2.41	10	0.66	—	0	(0)	7.45	100	42.74	519.89	14	(0)
R_2 / $(10,10,5,5)$	—	0	0.59	24	—	0.02	5.24	10	5.25	148.61	4	(10)	6.38	100	6.35	—	0	(0)
R_3 / $(10,10,5,5)$	—	0	0.69	16	0.44	—	2.33	10	2.32	—	0	(0)	9.77	100	10.52	—	0	(0)
R_4 / $(10,10,5,5)$	—	0	1.48	3	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(10,10,5,5)$	—	0	1.81	32	0.63	—	20.25	8	41.45	—	0	(0)	19.69	100	19.67	—	0	(0)
R_1 / $(13,10,5,5)$	—	0	0.26	24	0.03	0.03	0.59	10	0.59	339.92	10	339.65	2.35	100	2.33	—	0	(0)
R_2 / $(13,10,5,5)$	—	0	0.88	12	—	0.87	—	0	(0)	7.60	1	(4)	—	0	(0)	—	0	(0)
R_3 / $(13,10,5,5)$	—	0	1.12	32	1.77	—	1.32	10	1.32	—	0	(0)	2.17	100	2.16	—	0	(0)
R_4 / $(13,10,5,5)$	—	0	1.37	10	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(13,10,5,5)$	—	0	1.91	28	0.67	—	6.61	9	23.82	—	0	(0)	211.88	98	(98)	—	0	(0)
R_1 / $(70,10,5,5)$	—	0	2.16	4	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_2 / $(70,10,5,5)$	—	0	4.48	16	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_3 / $(70,10,5,5)$	—	0	4.76	12	—	0.16	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_4 / $(70,10,5,5)$	—	0	5.20	24	0.17	0.16	101.42	9	231.02	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(70,10,5,5)$	—	0	9.35	6	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)

Table 8.7: Runtime in secs, timeout 600 secs (—)

Topo / Parameter	DMCS		DMCSOPT		DMCS-STREAMING													
	time	#	time	#	$k=1$		$k=10$						$k=100$					
					ORIG	OPT	ORIG			OPT			ORIG			OPT		
							1st	#	All	1st	#	All	1st	#	All	1st	#	All
R_1 / $(4,20,10,10)$	244.44	1920	0.62	128	—	0.01	—	0	(0)	0.20	10	0.21	—	0	(0)	1.13	32	12.23
R_2 / $(4,20,10,10)$	401.64	4096	1.38	112	0.01	0.01	0.60	7	3.17	0.20	7	0.94	3.91	96	8.16	3.73	28	532.66
R_3 / $(4,20,10,10)$	—	0	22.85	1536	0.77	—	2.44	7	7.80	—	0	(0)	18.96	100	19.00	6.79	48	40.10
R_4 / $(4,20,10,10)$	—	0	202.16	384	—	—	—	0	(0)	—	0	(0)	312.33	98	(98)	—	0	(0)
R_5 / $(4,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	12.49	94	49.08	110.06	4	(4)
R_1 / $(7,20,10,10)$	—	0	7.44	800	—	0.02	—	0	(0)	0.06	10	0.06	1.74	100	1.75	0.79	36	4.24
R_2 / $(7,20,10,10)$	—	0	23.09	14592	0.02	—	4.48	6	21.25	—	0	(0)	17.19	99	59.71	—	0	(0)
R_3 / $(7,20,10,10)$	—	0	66.36	256	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_4 / $(7,20,10,10)$	—	0	213.82	144	0.02	—	1.55	9	5.03	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(7,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_1 / $(10,20,10,10)$	—	0	4.95	256	—	305.31	—	0	(0)	0.06	10	0.06	—	0	(0)	24.18	32	(96)
R_2 / $(10,20,10,10)$	—	0	5.40	48	—	0.02	—	0	(0)	1.34	4	18.36	—	0	(0)	91.01	8	(8)
R_3 / $(10,20,10,10)$	—	0	48.51	1440	—	0.02	—	0	(0)	0.06	10	0.06	—	0	(0)	0.79	100	0.79
R_4 / $(10,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(10,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_1 / $(13,20,10,10)$	—	0	47.17	384	0.03	0.03	54.30	3	(8)	0.48	10	0.47	—	0	(0)	—	0	(0)
R_2 / $(13,20,10,10)$	—	0	76.21	64	0.03	—	2.24	4	19.35	—	0	(0)	65.96	57	(84)	—	0	(0)
R_3 / $(13,20,10,10)$	—	0	84.15	24	0.69	19.47	1.98	7	11.01	1.74	4	24.97	14.49	96	36.95	—	0	(0)
R_4 / $(13,20,10,10)$	—	0	187.01	2544	—	0.03	—	0	(0)	0.33	10	0.33	—	0	(0)	20.88	100	20.75
R_5 / $(13,20,10,10)$	—	0	212.75	72	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_1 / $(70,20,10,10)$	—	0	417.71	112	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_2 / $(70,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_3 / $(70,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_4 / $(70,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)
R_5 / $(70,20,10,10)$	—	0	—	0	—	—	—	0	(0)	—	0	(0)	—	0	(0)	—	0	(0)

Table 8.8: Runtime in secs, timeout 600 secs (—)

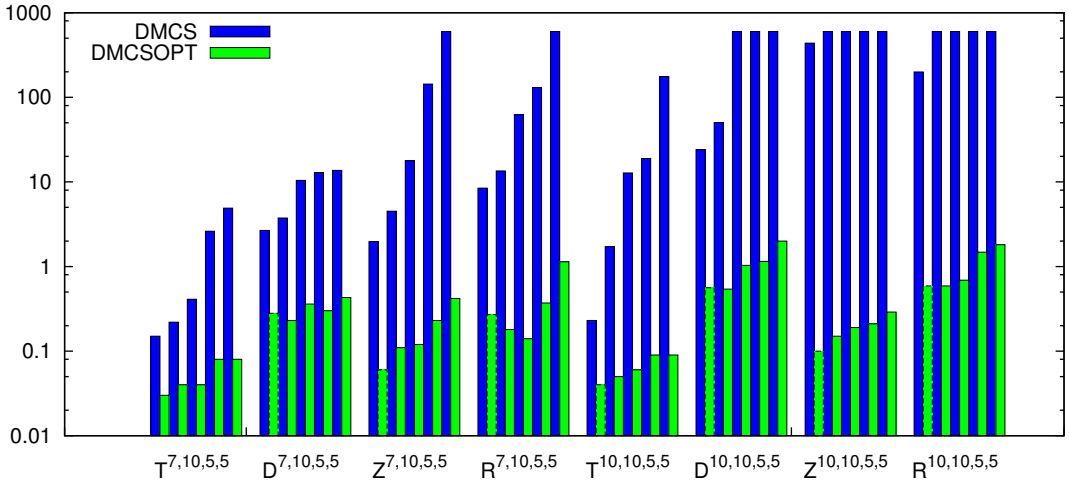


Figure 8.3: DMCS v.s. DMCSOPT in *non-streaming* mode

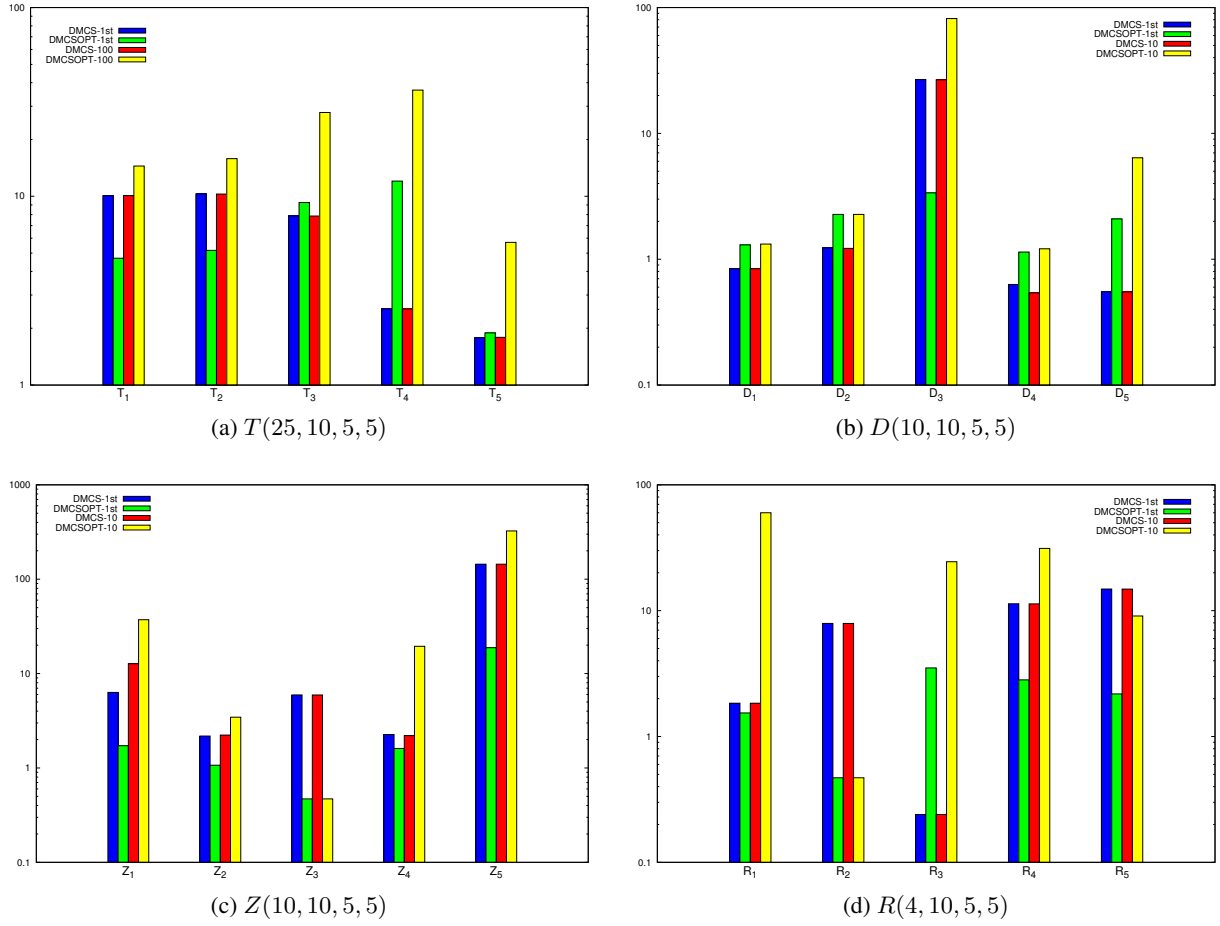
DMCS v.s. DMCSOPT

We first compare DMCS and DMCSOPT. Figure 8.3 shows the running time of these two algorithms in computing all partial equilibria, i.e., *non-streaming mode*, of 5 instances of the respective parameter settings (here denoted as $X^{n,s,b,r}$ due to space reason). In this mode, it is clearly the case that DMCSOPT outperforms DMCS. This can be explained by the fact that when one aims at computing all answers, DMCS always produces many more PEs than DMCSOPT, since one PE returned by DMCSOPT can be obtained from projecting many PEs 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 two algorithms. As such, DMCS must spend more time on processing possibly exponentially many more input and it is no surprise that it is consistently slower than DMCSOPT.

However, the observation is not the same in *streaming* mode. Figure 8.4 shows the running time of DMCS and DMCSOPT in streaming mode to compute the first 100 *unique* PEs for $T(25, 10, 5, 5)$ and first 10 of those for $D/Z(10, 10, 5, 5)$ and $R(4, 10, 5, 5)$, on 5 instances of each setting. On a first view, we noticed that DMCSOPT is consistently slower than DMCS. This might raise questions for the correctness of the result; however, it is not a surprise. Note again that one PE returned by DMCSOPT should be corresponding to several PEs returned by DMCS. Hence, to complete the first k unique answers in DMCS corresponds to only a few number of unique answers in DMCSOPT.

Therefore, comparing the performance of DMCS and DMCSOPT by measuring the time they need to compute the first k answers in streaming mode seems not fair. For a proper comparison, we took the time when both algorithms finishing the first round of answers into account (denoted by DMCS-1st and DMCSOPT-1st in Figure 8.4). With this setting, we observed that:

- There are many instances that DMCSOPT finishes the first round faster than DMCS. This



(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 8.4: DMCS v.s. DMCSOPT in *streaming* mode

confirms the effect of using the query plan.

- However, there are cases that DMCS wins. This can be explained as follows. First of all, in streaming mode, we transfer only a package of k PEs at a time; therefore, the effect of reducing the amount of total work to be done between DMCS and DMCSOPT does not always apply as in the non-streaming mode. Furthermore, at every local context, we compute k PEs, and then project them to the output interface before returning the results to the invoker. According to this strategy, when a context C_i returns k_1 PEs in non-streaming mode and k_2 PEs in streaming to another context C_j, it might happen that k_2 is much smaller than k_1, hence does not provide enough input for C_j to compute k PEs to return to its parent. Therefore, C_j will need to 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 more time for DMCSOPT to even compute the first set of PEs at the root context to the end user. Another approach is to always compute k unique PEs before returning to a parent context.

But this strategy risks to compute even all local models before k unique PEs can be found.

Overall, there is not much difference in running time when DMCSOPT is slower than DMCS, except for instance R_3 (Figure 8.4d). This however comes from a different reason: the cyclic topology with guess-and-check's effects, which play a much more important role than choosing between DMCS and DMCSOPT. We will analyze this case in Section 8.3.

Streaming v.s. non-streaming DMCS

In the previous section, we compared DMCS and DMCSOPT in both streaming and non-streaming mode. We now look into another orthogonal aspect: comparing these two modes when running the same algorithm.

Figure 8.5 compares the running time in non-streaming with streaming mode of DMCS (8.5a), DMCSOPT to compute the first 10, 100 PEs with small systems/local theories (8.5b), and with large systems/local theories (8.5c). For each parameter setting, we show the running time of five instances, each with three above runnning modes. Except for Ring that behaves abnormally due to the involvement of 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 8.3), while in streaming mode we still could find some answers after a reasonable time.
- For DMCSOPT, the situation is a bit different as the streaming mode loses against the non-streaming one on small instances. This is due to the recomputation that the streaming mode pays for transferring just a portion of PEs between contexts; furthermore, there are duplications between answers which wastes even more recomputation. But when one moves to larger systems and local theories, the streaming mode starts gaining back. But it does not always win, since recomputation still significantly takes time in some cases.

To sum up, when the system is small enough, one should try the non-streaming mode as it does not suffer the problem of recomputation and duplication of PEs between different rounds of computation. But for large systems, the streaming mode can rescue us from timing out. Even though we have to pay for recomputation, it still helps in cases when one only needs some but not all answers, for example, in answering queries bravely.

Effects of the package size in streaming mode

We have just concluded that the streaming mode helps with large systems/local theories in (many practical) cases, when only a portion of the results is needed, e.g., in brave query answering. The open question is: "What is the optimal number of PEs should be transferred in each return message between pairs of contexts?" We will analyze the experimental results on the streaming mode with package sizes 1, 10, and 100 to give some hints for this question.

Figure 8.6 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 with the purpose of getting the first answer is acceptable in most of the case, in particular the situation where

no guessing is needed. However, a small package size is sometime better, as one can save communication time (sending once with a package of, for example, 10 PEs instead of sending ten times, each time a package with a single PE). This setting (small package sizes like 10) will be more effective when communication plays a big factor in the total running time of the system, which happens in real application where contexts are located at physically distributed nodes. In such cases, computing 10 PEs 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., instance $R_3(4, 20, 10, 10)$. 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 ends up with timeout. The reason is that there are many duplications and once DMCSOPT is stuck with a local search branch that promises fewer than 100 PEs, the algorithm will lose time here without finding new unique answers and will eventually time out.

As a suggestion to find a good package size with a specific setting (topology, system size, local theory size) is to run the system on a set of training tests and apply binary search on the package sizes.

Roles of topologies

Taking a quick glance through all plots from Figure 8.3 to 8.6, one can see a pattern that the algorithms, especially the optimizations, perform better on tree than on zigzag and diamond, depending on whether it is DMCS or DMCSOPT, and worst on ring.

System topology does play 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. As such, tree has fewer connections than diamond and zigzag, which reduces not only communication time 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 be the topology that offer the best performance. However, this is actually not the case due to aspect (iii) that we will shortly analyze.

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 beliefs in PEs returned from child contexts to a parent context. In other words, only local beliefs from a context C_i are needed for transferring back to its parents. This drastically decreases the amount of information to be communicated, and more importantly, the number of calls to lsolve as there is no duplication with respect to the local beliefs in a result package (when beliefs from a child context C_j are carried along in the returning message, there can be two different PEs identical on the interpretation on the beliefs of C_i, which might cause recomputation at its parents). Due to this special property, DMCSOPT performs extremely well on the tree topology, and it can scale to hundreds of contexts compare to smaller system sizes that can be handled on the other topologies.

Comparing diamond and zigzag, they have the same number of cut vertexes. However, zigzag is converted to a linear topology with the optimal query plan (with 2 edges removed

per zigzag diamond as shown in Figure 8.1c), and therefore can be operated much faster than diamond. In Figure 8.6, DMCSOPT scales to 70 contexts and still has a better average time to compute one answer on zigzag than on diamond with 25 contexts.

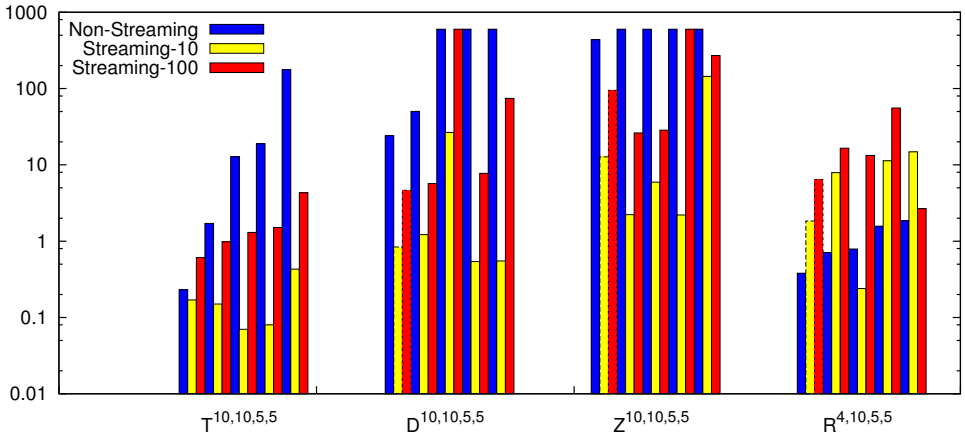
With respect to (iii), ring is a cyclic topology while the other topologies are acyclic. The consequence is that any algorithm evaluating this topology has to do guess-and-check at one context in the topology, while for the other the process can be carried out in a bottom-up manner. When guess-and-check is involved, making the right guess is the most important thing to do, even more important than reducing communication and calls to local solving by other optimization techniques. Since our current implementation does not deploy conflict learning for constraining the search space, guessing is done in a totally trivial way, i.e., gradually proceed from guessing for all beliefs to be false up to all of them to be true, in an binary increasing manner. Therefore, it is not surprising that the result of running DMCS and DMCSOPT on this topology does not follow any pattern (Figure 8.7, Table 8.7 and 8.8). It absolutely depends on a specific instance in which 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 the best strategy to evaluate different types of topologies; and more importantly, when dealing with MCSs having arbitrary topologies, it would be beneficial to decompose the system into parts with familiar topologies on which efficient strategies are known, and then study how to combine these different strategies to evaluate the whole system. This task is beyond the scope of this thesis and can be regarded as an interesting issue for future research.

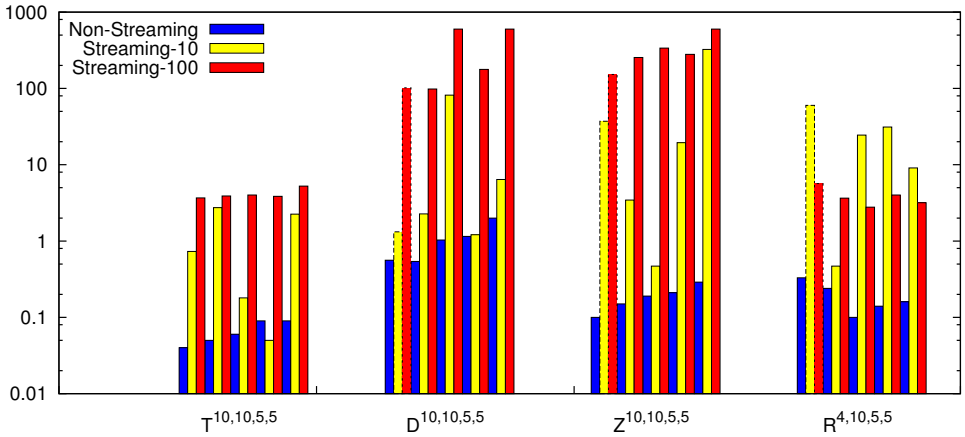
Summary

To sum up, we have analyzed the experimental results of our DMCS prototype implementation on different aspects. The analysis shows that there is no absolute winner between different algorithms (DMCS vs. DMCSOPT) in different running modes (streaming vs. non-streaming, with different package size), on different topologies. What comes out from these observations is a guideline to choose the setup that fits specific instances in practice, some issues are still open for further investigation, which can be briefly recapitulate as follows:

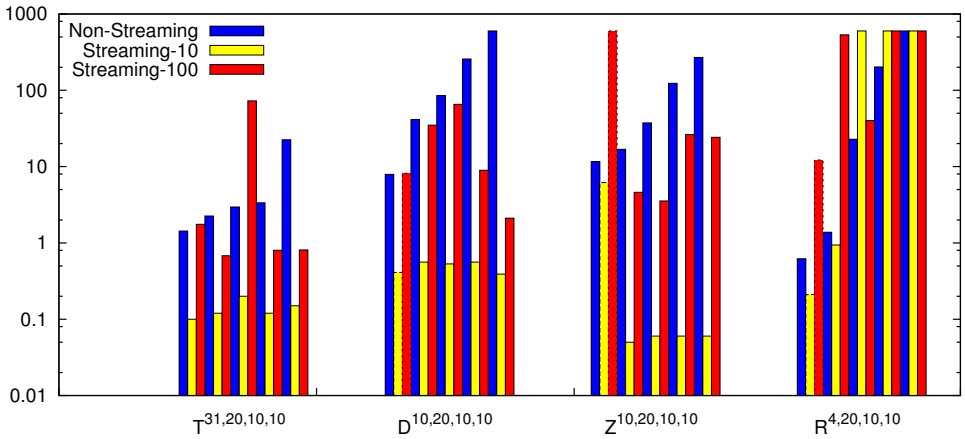
- choose DMCSOPT over DMCS in non-streaming mode, except for cyclic topologies;
- in streaming mode, intelligently choose an appropriate package size, by for example, doing binary search on this criteria on some training instances;
- decompose random topologies into parts whose topologies have effective strategies to evaluate, and study how to combine the strategies for the over all system.



(a) DMCS



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



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

Figure 8.5: Non-streaming v.s. streaming under DMCS and DMCSOPT

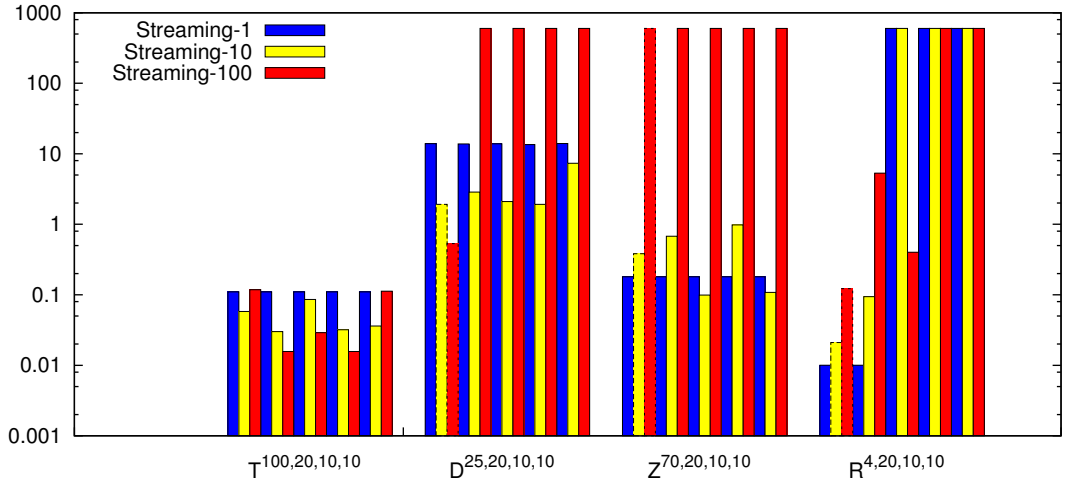


Figure 8.6: DMCSOPT's avg. time to find 1 PE in streaming mode with different package sizes

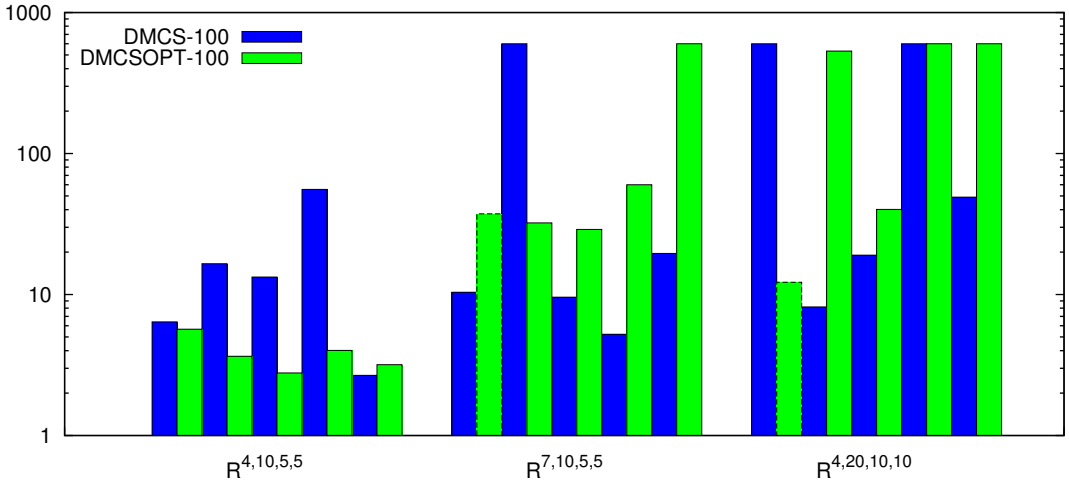


Figure 8.7: DMCS v.s. DMCSOPT in streaming with package size 100 on ring

CHAPTER 9

Related Work, Conclusions and Future Work

9.1 Related Work

Distributed reasoning algorithms

In [86], the authors 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; also, no caching was used.

Distributed tableaux algorithms for reasoning in distributed ontologies are defined in [88, 90]. They can be used to decide consistency of distributed description logic knowledge bases, provided that the distributed TBox is acyclic. The DRAGO system is an implementation of this approach.

The authors of [2] presented a framework of peer-to-peer inference systems. Local theories of propositional clause sets share atoms, and a special algorithm can be used for consequence finding. As we pursue the dual problem of model building, the application for our needs is not straightforward. Similarly, [14] developed a distributed algorithm for query evaluation in a MCS framework based on defeasible logic. Here, contexts are built using defeasible rules, and the algorithm can determine for a given literal l three values: whether l is (not) a logical conclusion of the MCS, or whether it cannot be proved that l is a logical conclusion. Again, applying this approach to model building is not easy.

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 et al. [96] 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 [50] but can be generalized [96]. The predominant solution method are backtracking algorithms. In [62], a suite of algorithms was presented for solving distributed SAT (DisSAT), based on a random assignment and improvement flips to reduce conflicts. However, the 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. A closer comparison, in which the nature of bridge rules and local solvers as in our setting is considered, remains to be done.

In relation to the topology-based optimization techniques described in Chapter 4, biconnected components are used in [6] to decompose constraint satisfaction problems. The decomposition is used to localize the computation of a single solution in the components of undirected constraint graphs. Like our approach, we are based on directed dependencies, which allows us to use a query plan for MCS evaluation.

Regarding the streaming algorithm mentioned in Chapter 5, very recently, a model streaming algorithm for HEX-programs (which generalize answer set programs by external information access) has been proposed [38]. It bares some similarities to the one in this work, but is rather different. There, monolithic programs are syntactically decomposed into modules (akin to contexts in MCS) and models are computed in a modular fashion. However, the algorithm is not fully distributed and allows exponential space use in components. Furthermore, it has a straightforward strategy to combine partial models from lower components to produce input for the upper component.

Distributed configuration

The problem that we considered in dynamic MCS shares some similarities with configuration in Multi-Agent Systems using matchmaking. In this setting [91], provider agents advertise their capabilities to middle agents; requester agents do not directly go to a provider but first ask some middle agent whether it knows of providers with desired capabilities; the middle agent matches the request against the stored advertisements and returns the information about appropriate providers to the requester. In our setting, the matchmaker plays the role of the middle agent. A context has both roles, it is seen as a requester when being instantiated, and as a provider when being used to instantiate s-bridge atoms from other contexts.

A configuration problem for multi-agent system that is in a sense orthogonal respectively complementary to matchmaking is coalition formation. Here, the problem is the assembly of a group of agents for cooperation in order to get some task down (assuming that the agents already know that cooperation is possible) [87]. However, this problem is only remotely related to our configuration problem. Agents have goals and intentions, and decide their participation in a coalition based on utility and reward in a rational manner. This leads in interaction with other agents to complex behaviors, which may be studied using game-theoretic methods and tools. Contexts instead lack such goal and reward orientation, and offer in an altruistic manner information exchange in order to enable the assembly of an MCS. Thus, from a coalition formation point of view, the MCS configuration problem is trivial. The problems gets more complicated if constraints are imposed (e.g., on the solution size or quality), but there is still a difference: at no point, some context may decide not to participate in an MCS as it concludes its payoff is insufficient, or it is being cheated.

Naturally related to dynamic MCS are peer-to-peer (P2P) systems. However, in typical models such as the Peer-Grid [1], a global system semantics does not play a role: peers are strictly localized and can join/leave the system at anytime. Our approach, on the other hand, aims at global model building for an ordinary MCS that is dynamically constructed, where the first step is instantiation, and then the (distributed) evaluation kicks in; this tacitly assumes that no relevant contexts disappear during configuration and evaluation of the configured system. We note that also [26] proposed a global model semantics for P2P systems, which is based on epistemic logic, and presented a distributed algorithm for query answering. This algorithm evaluates P2P mappings dynamically, but no system configuration like in our approach is performed. Similarly, [13] considered distributed query answering in a given P2P system of contexts, but under preferences using an argumentation based approach; however, no dynamic configuration in a potentially open environment is performed.

9.2 Conclusions

In this thesis, we have explored an area that has not been considered before: *design, implement, and analyze truly distributed algorithms to evaluate partial equilibria of Heterogeneous Nonmonotonic Multi-Context Systems*. As the results, we have come up with a basic algorithm DMCS to compute partial equilibria in a meta level, under the availability of the local solvers for local contexts. Then two main improvements were investigated, namely a topology-based optimization algorithm DMCSOPT and a streaming mode to compute partial equilibria in a gradual way by the algorithm DMCS-STREAMING. Eventually, both DMCS and DMCSOPT can be deployed in DMCS-STREAMING that allows to switch on/off the streaming mode. As an additional explorative branch, we looked into the problem of configuring dynamic MCS.

All proposed algorithms were realized in a prototype implementation which is publically available as open source. The implementation allows to switch between different algorithms and modes by simply changing the commandline arguments. On top of this implementation, we did exhaustive experiments to compare the performance of DMCS, DMCSOPT in streaming and non-streaming modes and gave an insight analysis on the experimental results. This shows advantages, disadvantages as well as the time/memory trade off between the algorithms in different situations which are determined by parameters such as system topology, size of local theories, size of the interface, number of answers required by the user. From this result, one can choose the setting (algorithm and mode) that fits her need best in finding partial equilibria in an MCS. Furthermore, experiences gained from these works together with observations on practical needs leads to interesting and promising future work that we present next.

9.3 Future Work

Besides a number of solid results from this thesis, several research problems and implementation issues are still open for further investigations. This section covers main directions for future works on MCSs, including further research problems for dynamic MCS, implementation issues to enhance DMCS performance, investigation of grounding-on-the-fly strategy to evaluate non-

ground MCS, and the potential of establishing a distributed heterogeneous stream reasoning framework based on MCS.

Further research problems for dynamic MCS

On the foundational side, a study of the computational complexity of dynamic MCS, and in particular of the configuration problem, could reveal important insight into computational resources needed to solve this problem, and may help to identify classes of systems for which it is efficiently solvable; here, the distribution and possible parallelism are interesting aspects. Furthermore, an improvement of the configuration algorithm, and in particular a deep investigation into heuristics would be an interesting task. Another aspect related to this is linkage cost. The size of a configuration is a crude measure of such cost, which clearly can be refined, taking, e.g., besides the topology also the cost (or value) of accessing particular beliefs into account.

On the implementation side, an obvious task is the implementation of a full-fledged configuration system that includes rich matchmaking, e.g., as in LARKS [91] instead of just hard-coded matches. Finally, another issue are applications of dynamic MCS. The student example in the Introduction suggests to consider possible applications in social group formation, complementing e.g., recent work in social Answer Set Programming [22, 23]. Another, less mundane area is configuration of small heterogeneous information systems, in which generic components (e.g., some domain ontologies, some decision component, and some fact base) must be suitably instantiated, given various possibilities. Here matchmaking may play an important role, e.g., if aspects like different levels of abstraction in the context knowledge bases should be handled. The usage of logic-based matchmaking approaches (cf. the work of [81]), in combination with other techniques, might here be worthwhile to consider. In particular, configuration of small systems in mobile environments, where openness is a natural requirement, to further the use of multi-context systems in ambient intelligence [5, 13] would be interesting.

Implementation issues for DMCS

As already shown in Chapter 5, parallelization is the next step to improve DMCS performance. The current architecture is well-prepared to allow this extension as each Context can spawn multiple Evaluator threads to be executed in parallel. However, naive parallelism will yield redundant recomputation. An important issue here is how to share information between threads. One possibility is that each Evaluator returns not only just the heads to be added to the local knowledge base but also its internal state when computing up to k local belief sets, then this information can be reused in other Evaluator to start its computation to search for the $(k+1)$-th belief set for the next request. Recently, multi-threaded ASP has been proposed in [53]. This work can serve as a starting investigation point to bring parallelism to DMCS.

In the current implementation of DMCS, polynomial boundary of memory for storing the (partial) belief states is guaranteed by restricting the size of the Concurrent Message Queues to a number k. This however does not give us a strong upper bound of the memory usage, as it will depend on the number of threads created during computation. Furthermore, memory allocation and deallocation are called in every execution of the contexts: allocate memory to store partial belief states from neighboring contexts, to hold results of local solving; deallocate

memory when reading new partial belief states from neighbors, or after sending results back to parents. This requires in general exponentially many allocations and deallocations, which is expensive for the performance. To overcome these limitations, we can shift the memory management to DMCS instead of counting on the operating system as now, by implementing memory pools that have fixed sizes, hence strict bounds. Moreover, the pools request memory from the operating system only once at initialization, and provide allocation, deallocation-like functions for DMCS. Internally, the pools just need simple marking techniques to keep track of memory parts that have been requested and occupied by different components of DMCS. This way, time consuming interactions with the operating system regarding memory manipulation will be reduced significantly.

Last but not least regarding the implementation of DMCS, currently there are only a few interfaces to other external solver, namely an ASP solver DLV, a SAT solver named relsat,[1], and clasp which can serve both as an ASP and a SAT solver. This restricts the current reasoning power to just ASP and SAT solving. To involve more diverse reasoning capabilities into the system, a natural task is to integrate more solvers into it, such as RacerPro,[2] Pellet[3] for ontological reasoning, PostgreSQL,[4] MySQL[5] for database querying, etc. This task is well-prepared in the current DMCS infrastructure with a general purpose design for wrapping the local solvers (c.f. Section 7.3).

Grounding-on-the-fly for non-ground ASP-based MCS

Concerning the implementation of DMCS only for ASP, it follows the conventional approach of ASP solving by first grounding the local knowledge bases and bridge rules (under the assumption that all exchanged ground beliefs are known in advanced), then the evaluation is carried out on the ground version of the MCS. This approach requires to store all ground rules and to expand local, neighbor signatures prior to evaluation, where not all these rules are necessary.

Recently, a new approach to evaluate ASP programs without pre-grounding has been proposed. According to this approach, the grounding process is done on-the-fly during computation. There are two basic actions: to propagate and to make a choice. The former is first executed to derive new facts from available ground facts given in the program. Then, at each later step, making a choice is followed by a propagation. A choice is done by grounding a rule and guessing either this rule instance is applicable or not. The process finds an answer set when there is no choice left and no conflict is derived during computation. Examples of solvers implementing this method are ASPeRix [72, 73], GASP [83], and OMiGA [34].

Investigating grounding-on-the-fly DMCS is certainly an interesting direction to pursue in the future. One can predict that this approach can be benefitial to compute the first equilibria when conflicts are rare, but more precise classification of cases when grounding-on-the-fly outperforms pre-grounding needs a thorough comparison between the two approaches on a wide

[1] `http://code.google.com/p/relsat/`
[2] `http://www.racer-systems.com/`
[3] `http://clarkparsia.com/pellet/`
[4] `http://www.postgresql.org/`
[5] `http://www.mysql.com/`

range of test cases. From the results, one can give useful suggestions to the users which method they should use in which application scenarios.

Conflict learning in DMCS

Until now, the DMCS algorithms simply combine partial belief states from neighboring contexts to rule out inconsistent combinations, and then call external solvers for local solving. In ASP and SAT solving, conflicts-driven evaluation is well-known to improve the performance drastically [12,52]. Given the current optimizations and experimental results on DMCS, conflict learning is the key to gain further performance and scalability for the system. As DMCS considers local solvers as black boxes, the conflicts to learn are those between contexts. For example, knowing in advance that pushing a belief set containing $p(a)$ and $q(b)$ into a local theory will always make this theory inconsistent, one can ignore input with this property to avoid redundant calls to the local solvers and speed up the computation. There has been only one proposal following this idea [7], but finding constraints requires computing at the beginning *all* models at each local context, and a context can push a constraint such as "do not send me answers containing $p(a)$ and $q(b)$" to its neighbors. This approach is not applicable in our stream setting (Chapter 5) and is incompatible with our effort of restricting memory consumption to a fixed, non-exponential boundary.

The idea of bringing conflict-learning techniques from ASP and SAT solving to DMCS implicitly assumes a pre-grounding step for the bridge rules. Another novel approach is to explore non-ground conflict learning, inspired by the grounding-on-the-fly idea in the previous Section. This calls for a new branch of research on non-ground conflict learning in ASP.

Query answering in multi-context systems

All algorithms proposed and implemented in this thesis aim at evaluating partial equilibria in MCSs, in other words, model building. Another interesting aspect of knowledge representation and reasoning that will make MCSs more practical in real life application is query answering. In such applications, the user poses a query asking for the truth of a belief at a context and the system searches for possibly remote evidences to conclude either the belief is true of false, the whole model building process is not required and therefore, one can get a better performance.

Note that in the most general MCS setting, we do not have the notion of non-ground queries, hence only yes/no queries can be posed in this setting. When one considers a narrower setting of MCSs with relational information exchanging and beliefs represented in terms of predicates such as ASP, sophisticated techniques such as magic sets [3] can help in dealing with intelligent grounding to reduce the search space.

Distributed heterogeneous stream reasoning potential

In this work, we approach evaluating MCSs from a *top-down* and *static* angle; that is, at evaluation, the system consists of contexts with fixed local theories (rules and ground facts) and

bridge rules.[6] The user posts a query and wait for the results, a *Human-Active System-Passive* interaction model. However, this model cannot meet application scenarios, especially in sensor networks, social networks, or smart cities, in which fixed queries are placed at a context and live answers are expected automatically under the continuous arrival of streams of input data at leaf contexts, a similar view when moving from traditional database management systems (DBMSs) to data stream management systems (DSMSs). This new *bottom-up* and *dynamic* approach on MCSs raises a whole new range of interesting and challenging research issues. Once these issues are successfully resolved, MCSs will have a new power of providing *distributed heterogeneous stream reasoning* frameworks that allow for novel applications in dynamic environments, which are valuable for recent efforts in realizing smart-cities applications.

[6]Chapter 6 tackles the problem of configuring dynamic MCSs to static ones before evaluation. Furthermore, the dynamic aspect there concerns about the topology of the system rather than contexts' data.

.1 DMCS System Usage

The DMCS implementation has two main command-line tools for generating test data and realizing the algorithms proposed in this thesis. This section present full instructions on invoking these tools.

Generating test cases with `dmcsgen`

The main purpose of `dmcsgen` is to aid experiments on the DMCS system with automatically generated test cases that reflecting different aspects of MCSs, including system topologies, system size, local theories size, interface size. `dmcsgen` also generate query plans and return plans that define the interface between contexts at run time. The former defines the importing interface while the latter defines the exporting ont. This is in replace to the current missing functionality of `dmcsm`. Moreover, command lines to run the whole system are also generated for the purpose of automatic testing. To invoke this tool, we use:

```
dmcsGen [OPTIONS]
```

where `OPTIONS` can be:

- `--help`: print help message
- `--gen-data=0/1`: `0` to disable generating data, i.e., only generate command lines, and `1` to enable this option
- `--contexts=N1`: set number of contexts (system size)
- `--atoms=N2`: set number of ground atoms per context (local theory size)
- `--interface=N3`: set number of atoms used for creating the interface between contexts
- `--bridge-rules=N4`: set maximum number of bridge rules between pairs of contexts. When generating, to vary the number of bridge rules, we iterate `N4` rounds, and for each round, have a probability of 50% to create a bridge rule.
- `--topology=N5`: set topology type. There are in total 9 different types of topologies. Use option `--help` for more details.
- `--prefix=STR1`: set a string as prefix for all files generated in a single test case.
- `--dmcspath=STR2`: set path to the DMCS binaries.
- `--startup-time=N6`: set start up time (in seconds) to call `dmcsc` after initializing all contexts.
- `--packsize=N7`: set package size as the number of partial equilibria in each return message. `N7>0` triggers streaming mode while `N7=0` triggers the original DMCS algorithms (with or without the topological optimization).

- `--timeout=N8`: set a time out (in seconds).

As the result, `dmcsgen` generates

- `N1` text files with `.lp` extension, containing `N1` contexts' local theories
- `N1` text files with `.br` extension, containing sets of bridge rules of the contexts
- `N1` text files with `.qp` (resp., `.oqp`) extension, containing the query plans of the contexts in case of original (resp., optimal) topology.
- `N1` text files with `.rp` (resp., `.orp`) extension, containing the return plans of the contexts in case of original (resp., optimal) topology.
- a file called `client.qp` containing the signatures of all contexts in the system, for the purpose of returning results with atom names in stead of internal encoding to the user.
- several `.txt` files containing command lines that can be used to manually test the system.
- several `.sh` files containing shell scripts used to automatically test the system in different modes: original or optimal topologies, non-streaming or streaming algorithms, and in case of streaming, whether to finish after the first package of results.

Running the system with `dmcsm`, `dmcsc` and `dmcsd`

The actual DMCS system is activated via three binaries: a `dmcsm` simulates a simple manager, N `dmcsd` running as daemonds to represent N contexts, and a `dmcsc` used as a client to trigger the querying to a `dmcsd`. We need to start these binaries in the following order:

Firstly, start `dmcsm` as:

```
dmcsm [--help] --port=PORT --system-size=N
```

where `PORT` is the port where the manager listens to, and `N` is the number of contexts in the system.

Secondly, start `N` `dmcsd` as follows:

```
dmcsd [--help] OPTIONS
```

where `OPTIONS` are:

- `--context=N1`: set the context identifier (ranging from `0` to `N-1`).
- `--port=N2`: set the port where the current context listens to.
- `--manager=HOST:PORT`: set hostname and port of the manager (must be in correspondence to the options set by `dmcsm`).
- `--system-size=N3`: set system size, which is `N`.

- `--queue-size=N4`: (optional) set interal concurrent message queues' size.
- `--belief-state-size=N5`: set belief state size of every context. Note that to simplify the experiment, we let all contexts have the same local theory size, which is generated from the option `--atoms` of `dmcsgen`.
- `--packsize=N6`: (optional) set size of package, i.e., the number of partial belief states, to be transferred back in each return message.
- `--kb=STR1`: set filename containing the local theory.
- `--br=STR2`: set filename containing the bridge rules.
- `--queryplan=STR3`: set filename containing the query plan wrt. the original topology
- `--optqueryplan=STR4`: set filename containing the query plan wrt. the optimal topology
- `--returnplan=STR5`: set filename containing the return plan. Depending on the mode (original or optimal) one would like to run, the return plan will be set to either generated files with extensions `.rp` or `.orp`.

Finally, start `dmcsc` as follows:

```
dmcsc [--help] OPTIONS
```

where `OPTIONS` are:

- `--hostname=STR1`: set hostname of the context to be queried.
- `--port=N1`: set port of the context to be queried.
- `--root=N2`: set indentifier of the context to be queried.
- `--signature=STR2`: set filename to the file that contains signatures of all contexts in the system, which is `client.qp` in case of automatically generated.
- `--belief-state-size=N3`: set the (uniform) belief state size of all contexts in the system.
- `--loop=0/1`: set a flag to indicate whether we would like to immediately finish after the first round of answers, in streaming mode.
- `--k1=N4 --k2=N5`: set the range of partial belief states one would like to query. Setting `N4=N5=0` requests for all answers; otherwise, it must hold that `0 < N4 <= N5`.

Bibliography

[1] Karl Aberer, Magdalena Punceva, Manfred Hauswirth, and Roman Schmidt. Improving data access in p2p systems. *IEEE Internet Computing*, 6(1):57–67, 2002.

[2] Philippe Adjiman, Philippe Chatalic, François Goasdoué, Marie-Christine Rousset, and Laurent Simon. Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *J. Artif. Intell. Res.*, 25:269–314, 2006.

[3] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artif. Intell.*, 187:156–192, 2012.

[4] Anastasia Analyti, Grigoris Antoniou, and Carlos Viegas Damásio. Mweb: A principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Log.*, 12(2):17, 2011.

[5] Grigoris Antoniou, Constantinos Papatheodorou, and Antonis Bikakis. Reasoning about context in ambient intelligence environments: A report from the field. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *rinciples of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*, pages 557–559. AAAI Press, May 2010.

[6] Jean-François Baget and Yannic S. Tognetti. Backtracking through biconnected components of a constraint graph. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 291–296. Morgan Kaufmann, August 2001.

[7] Seif El-Din Bairakdar. Local optimization for multi-context systems with constraint pushing. Master's thesis, Vienna University of Technology, April 2011.

[8] Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Decomposition of distributed nonmonotonic multi-context systems. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 24–37. Springer, September 2010.

[9] Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. The dmcs solver for distributed nonmonotonic multi-context systems. In Tomi

Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 352–355. Springer, September 2010.

[10] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. In Verónica Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, Lecture Notes in Computer Science, pages 89–103. Springer, 2007.

[11] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Handbook of satisfiability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[13] Antonis Bikakis and Grigoris Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506, 2010.

[14] Antonis Bikakis, Grigoris Antoniou, and Panayiotis Hassapis. Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowledge and Information Systems*, April 2010.

[15] Markus Bögl, Thomas Eiter, Michael Fink, and Peter Schüller. 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*, volume 6341 of *Lecture Notes in Computer Science*, pages 356–359. Springer, 2010.

[16] Piero A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.

[17] Adrian Bondy and U. S. R. Murty. *Graph Theory*, volume 244 of *Graduate Textx in Mathematics*. Springer, 2008.

[18] Genevieve Bossu and Pierre Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artif. Intell.*, 25(1):13–63, 1985.

[19] Gerhard Brewka and Thomas Eiter. 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*, pages 385–390. AAAI Press, 2007.

[20] Gerhard Brewka, Thomas Eiter, and Michael Fink. Nonmonotonic Multi-Context Systems: A Flexible Approach for Integrating Heterogeneous Knowledge Sources. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of*

His 65th Birthday, volume 6565 of *Lecture Notes in Computer Science*, pages 233–258. Springer, 2011.

[21] Gerhard Brewka, Floris Roelofsen, and Luciano Serafini. Contextual default reasoning. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 268–273, 2007.

[22] Francesco Buccafurri and Gianluca Caminiti. Logic programming with social features. *Theory and Practice of Logic Programming*, 8(5-6):643–690, 2008.

[23] Francesco Buccafurri, Gianluca Caminiti, and Rosario Laurendi. A logic language with stable model semantics for social reasoning. In Maria Garcia de la Banda and Enrico Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08), Udine, Italy, December 9-13 2008*, volume 5366 of *Lecture Notes in Computer Science*, pages 718–723. Springer, 2008.

[24] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive logic programs with inheritance. *TPLP*, 2(3):293–321, 2002.

[25] Sasa Buvac, Vanja Buvac, and Ian A. Mason. Metamathematics of contexts. *Fundam. Inform.*, 23(2/3/4):263–301, 1995.

[26] Diego Calvanese, Guiseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *23rd ACM Symposium on Principles of Database Systems (PODS'04)*. ACM, 2004.

[27] Jan Chomicki. Depth-bounded bottom-up evaluation of logic program. *J. Log. Program.*, 25(1):1–31, 1995.

[28] Jan Chomicki and Tomasz Imielinski. Finite representation of infinite query answers. *ACM Trans. Database Syst.*, 18(2):181–223, 1993.

[29] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[30] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2009.

[31] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Distributed nonmonotonic multi-context systems. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010.

[32] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Dynamic distributed nonmontonic multi-context systems. In Gerhard Brewka, Victor Marek, and Miroslaw Truszczynski, editors, *Nonmonotonic Reasoning, Essays Celebrating its 30th Anniversary, Lexington, Kentucky, U.S.A., October 22–25, 2010*, volume 31, pages 63–88. College Publications, 2011.

[33] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Model streaming for distributed multi-context systems. In Alessandra Mileo and Michael Fink, editors, *2nd International Workshop on Logic-based Interpretation of Context: Modeling and Applications*, volume 738 of *CEUR Workshop Proceedings*, pages 11–22, May 2011.

[34] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. OMiGA: An Open Minded Grounding On-The-Fly Answer Set Solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, September 2012.

[35] Thomas Eiter, Gerhard Brewka, Minh Dao-Tran, Michael Fink, Giovambattista Ianni, and Thomas Krennwallner. Combining nonmonotonic knowledge bases with external sources. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 18–42. Springer, 2009.

[36] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the dlv system. *AI Commun.*, 12(1-2):99–111, 1999.

[37] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. System description: The dlv$^{\mathrm{k}}$ planning system. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, pages 429–433. Springer, 2001.

[38] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, and Peter Schüller. Pushing efficient evaluation of hex programs by modular decomposition. In James P. Delgrande and Wolfgang Faber, editors, *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, volume 6645 of *Lecture Notes in Computer Science*, pages 93–106. Springer, May 2011.

[39] Thomas Eiter, Michael Fink, and João Moura. Paracoherent answer set programming. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010.

[40] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. An update front-end for extended logic programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, pages 397–401. Springer, 2001.

[41] Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 1997.

[42] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[43] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pages 90–96, 2005.

[44] Thomas Eiter and Mantas Simkus. Bidirectional answer set programs with function symbols. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 765–771, 2009.

[45] Thomas Eiter and Mantas Simkus. FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Trans. Comput. Log.*, 11(2), 2010.

[46] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2004.

[47] Boi Faltings and Makoto Yokoo. Introduction: Special issue on distributed constraint satisfaction. *Artif. Intell.*, 161(1-2):1–5, 2005.

[48] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A generalization of the lin-zhao theorem. *Ann. Math. Artif. Intell.*, 47(1-2):79–101, 2006.

[49] Michael Fink, Lucantonio Ghionna, and Antonius Weinzierl. Relational information exchange and aggregation in multi-context systems. In James P. Delgrande and Wolfgang Faber, editors, *11th International Conference on Logic Programming and Nonmonotonic*

Reasoning (LPNMR 2011), Vancouver, BC, Canada, 16-19 May, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 120–133. Springer, May 2011.

[50] Jian Gao, Jigui Sun, and Yonggang Zhang. An improved concurrent search algorithm for distributed csps. In *Australian Conference on Artificial Intelligence*, pages 181–190, 2007.

[51] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2011.

[52] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 386–, 2007.

[53] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded asp solving with clasp. In *ICLP*, September 2012.

[54] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[55] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[56] Chiara Ghidini and Fausto Giunchiglia. Local models semantics, or contextual reasoning=locality+compatibility. *Artif. Intell.*, 127(2):221–259, 2001.

[57] Fausto Giunchiglia. Contextual Reasoning. *Epistemologia, Special Issue on I Linguaggi e le Macchine*, 345:345–364, 1992.

[58] Fausto Giunchiglia and Luciano Serafini. Multilanguage hierarchical logics or: How we can do without modal logics. *Artif. Intell.*, 65(1):29–70, 1994.

[59] Ramanathan V. Guha. *Contexts: a formalization and some applications*. PhD thesis, Stanford University, 1991.

[60] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Open answer set programming for the semantic web. *J. Applied Logic*, 5(1):144–169, 2007.

[61] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Open answer set programming with guarded programs. *ACM Trans. Comput. Log.*, 9(4), 2008.

[62] Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. *Artif. Intell.*, 161(1–2):89–115, 2005.

[63] Martin Homola. *Semantic Investigations in Distributed Ontologies*. PhD thesis, Comenius University, Bratislava, Slovakia, 2010.

[64] Guan-Shieng Huang, Xiumei Jia, Churn-Jung Liau, and Jia-Huai You. Two-literal logic programs and satisfiability representation of stable models: A comparison. In Robin Cohen and Bruce Spencer, editors, *Advances in Artificial Intelligence, 15th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2002, Calgary, Canada, May 27-29, 2002, Proceedings*, volume 2338 of *Springer*, pages 119–131. Lecture Notes in Computer Science, 2002.

[65] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Log.*, 7(1):1–37, 2006.

[66] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res. (JAIR)*, 35:813–857, 2009.

[67] Joohyung Lee. A model-theoretic counterpart of loop formulas. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 503–508. Professional Book Center, 2005.

[68] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, Lecture Notes in Computer Science, pages 451–465. Springer, 2003.

[69] Joohyung Lee and Fangzhen Lin. Loop formulas for circumscription. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 281–286. AAAI Press / The MIT Press, 2004.

[70] Joohyung Lee and Yunsong Meng. On loop formulas with variables. In Gerhard Brewka and Jérôme Lang, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 444–453. AAAI Press, 2008.

[71] Joohyung Lee and Yunsong Meng. First-order stable model semantics and first-order loop formulas. *J. Artif. Intell. Res. (JAIR)*, 42:125–180, 2011.

[72] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2009.

[73] Claire Lefèvre and Pascal Nicolas. The first version of a new asp solver : Asperix. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany,*

September 14-18, 2009. Proceedings, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer, 2009.

[74] Nicola Leone and Wolfgang Faber. The dlv project: A tour from theory and research to applications and market. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2008.

[75] Vladimir Lifschitz and Alexander A. Razborov. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2):261–268, 2006.

[76] Fangzhen Lin and Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

[77] John McCarthy. Generality in artificial intelligence. *Commun. ACM*, 30(12):1029–1035, 1987.

[78] John McCarthy. Notes on formalizing context. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 555–562. Morgan Kaufmann, 1993.

[79] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[80] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[81] Tommaso Di Noia, Eugenio Di Sciascio, and Francesco M. Donini. Semantic matchmaking as non-monotonic reasoning: A description logic approach. *J. Artif. Intell. Res.*, 29:269–307, 2007.

[82] Elth Ogston and Stamatis Vassiliadis. Local distributed agent matchmaking. In Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella, editors, *9th International Conference on Cooperative Information Systems (CoopIS'01)*, volume 2172 of *Lecture Notes in Computer Science*, pages 67–79. Springer, 2001.

[83] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[84] Raymond Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[85] Floris Roelofsen and Luciano Serafini. Minimal and absent information in contexts. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 558–563. Professional Book Center, 2005.

[86] Floris Roelofsen, Luciano Serafini, and Alessandro Cimatti. Many hands make light work: Localized satisfiability for multi-context systems. In Ramon López de Mántaras and Lorenza Saitta, editors, *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*, pages 58–62. IOS Press, August 2004.

[87] Tuomas Sandholm. Distributed rational decision making. In *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, chapter 5, pages 201–258.

[88] Luciano Serafini, Alexander Borgida, and Andrei Tamilin. Aspects of distributed and modular ontology reasoning. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 570–575. AAAI Press, 2005.

[89] Luciano Serafini and Paolo Bouquet. Comparing formal theories of context in ai. *Artif. Intell.*, 155(1-2):41–67, 2004.

[90] Luciano Serafini and Andrei Tamilin. Drago: Distributed reasoning architecture for the semantic web. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *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, pages 361–376. Springer, 2005.

[91] Katia P. Sycara, Seth Widoff, Matthias Klusch, and Jianguo Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5(2):173–203, 2002.

[92] Tommi Syrjänen. Omega-restricted logic programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, Lecture Notes in Computer Science, pages 267–279. Springer, 2001.

[93] Jacobo Valdes, Robert Endre Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, 1982.

[94] Divyanshu Vats and José M. F. Moura. Graphical models as block-tree graphs. *CoRR*, abs/1007.0563, 2010.

[95] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898. ACM, 2012.

[96] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[97] Jia-Huai You, Robert Cartwright, and Ming Li. Iterative belief revision in extended logic programming. *Theor. Comput. Sci.*, 170(1-2):383–406, 1996.