FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Distributed Computation of Diagnoses for Inconsistent Multi-Context Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Fabian Salcher
Matrikelnummer 0227235

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 11.11.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Distributed Computation of Diagnoses for Inconsistent Multi-Context Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Fabian Salcher
Registration Number 0227235

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 11.11.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Fabian Salcher
Wilhelminenstraße 72/35, 1160 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 Verfasser)

Acknowledgments

I would like to thank *Michael Fink* who brought this interesting topic to my mind and who had the patience to attend this thesis through its long journey till its finish. He taught me a lot about writing a good thesis, was a great help while getting familiar with the topic, and placed all those direction signs which guided me well to the end of the thesis. It is also my deep concern to thank *Peter Schüller* for the uncountable meetings which have been a great help in accomplishing the noble technique of theoretical proofs and who also gave me a lot of tips through the whole thesis and not to forget his knowledge about good English writing style. I would also like to thank *Thomas Eiter* who gave me the opportunity to write this thesis under his supervision.

Now it is also time to apologize to all my friends and family who couldn't hear the "But I have to write my thesis" thing any more. I really meant it most of the time and I want to apologize especially for those times where I did not!

Last but not least I want to thank my family in Galtür for their continuous and heart warming support during my studies and also my girlfriend *Juli* who pushed me hard during the last mile of the thesis - hugs to all of you!

Abstract

Multi-Context Systems (MCS) are systems of distributed knowledge bases which interact via so called bridge rules. The $MCSs$ we are interested in are nonmonotonic and therefore bridge rules can cause inconsistencies in the MCS while the knowledge bases for themselves are consistent. We will develop an algorithm which identifies those inconsistencies and proposes bridge rule modifications to the user which will make the system consistent. This algorithm will be effective with respect to requesting only as much information from the distributed knowledge bases as necessary. Therefore, for a user only interested in a part of the system, it is not necessary to know the whole system. We will show that this algorithm is sound and complete and present data demonstrating the performance of a reference implementation. To increase the performance of the algorithm we also propose further optimizations like edge and subset pruning and show the effectiveness of those modifications on the reference implementation.

Kurzfassung

Multi-Context Systeme (MCS) beinhalten verteilte Wissensbasen welche untereinander Informationen mittels sogenannter Bridge Rules austauschen. Die Multi-Context Systeme in welchen wir interessiert sind, sind nicht monoton, was dazu führt, dass die Bridge Rules für Inkonsistenzen verantwortlich sein können, während die Wissensbasen für sich konsistent sind. Wir werden einen Algorithmus entwickeln, welcher diese Inkonsistenzen identifiziert und dem Nutzer vorschlägt, wie die Bridge Rules zu modifizieren sind um das System wieder konsistent zu machen. Dieser Algorithmus ist in der Hinsicht effizient, dass nur so viele Informationen wie nötig aus dem verteilten System abgefragt werden. Das macht es überflüssig, das gesamte System zu kennen. Wir werden zeigen, dass der Algorithmus korrekt und vollständig ist, sowie Daten präsentieren, welche die Performance einer Referenzimplementierung unterstreichen. Um die Performance weiter zu steigern, werden wir Optimierungen, wie Edge und Subset Pruning, vorgeschlagen und demonstrieren die Effizienz dieser Optimierungen an der Referenzimplementation.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

In an increasingly connected world with a consequential specialization of companies and institutions, increased technological possibilities, and a growing amount of knowledge it is more and more necessary to interconnect all involved entities in such a way that they can communicate and exchange knowledge and information efficiently and as much automated as possible. Since there is already a lot of progress with respect to communication between people and the automated exchange of information between computers, the next step is to automate and enable the computer based exchange of knowledge between independent, heterogeneous entities.

Examples are systems for maritime situational awareness for the coast of the Netherlands where around 2000 ships are daily active, from normally (legally) behaving ships to smugglers, illegal fishing ships, and ships dumping garbage. Bringing together the information about ongoing actions and plans of the protagonists (either cooperating or not) and processing it, e.g., for the coast guard, is the challenge of such a system [26]. Another example is health and life sciences where huge amounts of data has to be handled. As an example Erdem et al. [17] and Barilaro et al. [5] show in their papers how to answer complex queries about the interdependencies between drugs and genes over distributed knowledge bases of drug-drug interaction chains, cliques of genes based on gene-gene relations, and similarities/diversities of genes/drugs.

In this thesis we use Multi-Context Systems to interconnect distributed knowledge bases, such as described above, and solve specific problems which arise due to the combination of these independent knowledge bases.

1.1 Problem Description

A *Multi-Context System (MCS)* is a formalism that allows to combine different types of monotonic and nonmonotonic knowledge sources and describes the information exchange between them [10]. Therefore we have contexts which represent a knowledge base in a specific knowledge representation formalism, e.g., a relational database, a logic program or an answer set program. The knowledge held at a context may depend on the beliefs of other contexts. This information flow is modeled via so called *bridge rules*, where bridge rules support negation as failure and are therefore nonmonotonic.

Now contexts, considered individually, may have one or more models. If another context now requests information from other contexts, which for themselves have valid models, the combination of the models from different contexts can lead to contradicting knowledge. Let's take the example from the biomedical field described above: We request information about the effectiveness of a specific drug on patients with specific genes A and B from two knowledge bases from different scientific institutes. One says the effectiveness of the drug is high since the patient has a gene A and the second backups this information. Now one of the institutes publishes a new study which says that if the patient has also gene B then the effect of the drug vanishes. So now we have contradicting information and the system won't state that the drug has no effect on this patient but instead only state nothing without any explanation about the cause.

The task of finding such explanations is not trivial since such a system is distributed and may not scale well on large instances. The context which is requesting the information is not aware of all the information on all contexts and therefore has to send requests to the other contexts to draw its conclusions. In addition for a specific system it is possible that there is not only one explanation but there are a lot of different and complementary explanations which could all be of interest for the user or system which requests them to make a comprehensive decision about fixing them. Those circumstances makes also network bandwidth and size of the exchanged data structures an important aspect to consider especially due to the combinatorial growth of explanations with an increased number of contexts.

1.2 State-of-the art

The work on formalization of contextual information goes back to the papers of McCarthy [27] and Giunchiglia [19]. Giunchiglia states that most cognitive processes are contextual but reasoning is usually performed on a small subset of the global knowledge base and therefore introduces a theory of reasoning with contexts which encapsulate such subsets. These formalisms and many of the successional ones, e.g., the propositional logic of context by McCarthy [28], are based on classical, monotonic reasoning. With the papers of Roelofsen and Serafini [32] and Brewka et al. [11] default negation was introduced and allowed therefore the modeling of absence of information on contexts. But in contrast to former papers those two approaches only allowed homogeneous contexts in the sense that all contexts have the same inference methods. Brewka and Eiter [10] finally defined a formalism which allows default negation and heterogeneous contexts. This formalism is also the basis of this thesis.

Furthermore Dao-Tran et al. [14] have done some work to define partial views on $MCSs$. Therefore not all the contexts have to be consistent anymore but only this part which is of interest (and relevant) for the requested information. An algorithm to calculate queries based on such views in a distributed environment is described there as well. This algorithm runs on every context and calculates partial results, which are communicated to instances at other contexts.

The following papers approach the problem of inconsistencies in an MCS. They describe ways to fix inconsistencies not automatically but show possible ways to solve them and let the final decision of how to fix it to the user. This is mainly due to the fact that the inconsistency itself contains valuable information about the system and the knowledge in it. This knowledge is not easily recognizable by the machine so if the inconsistencies would be fixed by automatic systems this knowledge would be lost. Two ways to explain inconsistencies in MCS have been developed by Eiter et al. [16]. One is to identify those bridge rules which, if removed or applied unconditionally[1], would result in an MCS with at least one consistent state. Such a set of bridge rules is called *diagnosis*. The second way defines two sets of bridge rules E_1 and E_2, E_1 is a set of bridge rules which are relevant for the inconsistency of the MCS. I.e., if the bridge rules of the MCS are replaced with the ones from E_1 the MCS is inconsistent and adding any other bridge rules of the original MCS will not make it consistent. Furthermore E_2 is a set of bridge rules that, if at least on of them is applied unconditionally to the MCS, can make the MCS, replaced with the bridge rules E_1, consistent. This notion has been called *explanation*. Both notions, diagnosis and explanation are defined over the whole MCS and do not support a partial view.

Finally there has been some work on optimizing MCS calculations: In [4] ways to optimize the distributed algorithm as described in [14] has been presented. An analysis of the MCS topology and a resulting smaller representation of the context dependencies is one part of the optimization; another is the characterization of minimal interfaces for the information flow between the contexts to minimize the amount of data send through the network.

1.3 Goals

Our goal is to have a system which provides the requesting user with explanations about inconsistencies in those parts of the MCS where the user is interested in. Therefore the system considers only a minimal subset of contexts of the whole MCS which are necessary to provide those explanations to the user. To accomplish this we first have to define a formalism which allows to express such explanations for the whole MCS or parts of it and then define an algorithm which calculates them.

The system must also handle the distributed nature of an MCS which implies that costs w.r.t time when sending requests and w.r.t. to bandwidth when sending data over the network are an issue. Such a distributed system also has to tackle the challenge of a combinatorial growth of the number of explanations with an increased number of contexts.

We want to give the user as much control as possible over how to handle or fix the inconsistency of (parts of) an MCS. This means the system has to find all inconsistencies and assist the user by giving him as much information as possible about the found solutions to overcome the inconsistencies.

The system must find suitable explanations for the users which are interested in fixing the inconsistent state of their system but also pass theoretical (soundness and completeness) as well as empirical (prototype implementation) tests.

[1]A bridge rule adds information to the context depending on the knowledge of other contexts. If applied unconditionally, the information is added without considering the knowledge from those other contexts.

1.4 Research Methods and Tests

This work can be divided in a theoretical and a practical part. In the theoretical part we will provide all the definitions necessary to define an algorithm which will calculate the explanations as stated in the previous sections. Based on these definitions and the algorithm we will implement a prototype and run tests with it.

The definitions will be stated in formal mathematical notations. Consistent notation with previous work in this field will be ensured. The properties which can be derived from these notations will be stated as propositions and lemmas and they will be justified by formal proofs. We will start by restating the definitions from previous work and extend them until they have the properties we expect and we will again validate this by formal proofs.

The subsequent algorithm will be defined in pseudocode using previously defined data structures and definitions, and we will provide explanations about each parts of the algorithm. Then we will prove that for each possible MCS of arbitrary size this algorithm returns all theoretically possible explanations and the corresponding models and that all results of the algorithm are according to our former definitions.

An actual implementation of the algorithm, based on the theoretical definitions of the data structures and the description of the algorithm in pseudocode will be written in C++ as a proof of concept. This implementation will then be used to carry out empirical experiments on randomly generated problem instances within some defined constraints. These constraints ensure that the results are comparable to test results from previous works like [14]. The implementation will be used to validate the proposed hypothesis on appropriate problem instances, demonstrate the effects of the optimizations and judge their relevance, and to observe the performance of the algorithm respectively its implementation by measuring its runtime on a state-of-the-art machine.

1.5 Contribution and Results

The following results have been accomplished in this thesis:

- We defined a structure which is based on the idea of partial equilibria from [14] and allows us to express diagnoses which cover only those parts of an MCS which are of interest and additionally those on which they are depending on. Such diagnoses will show us how to modify an MCS such that it has at least one resulting partial equilibria.
- An algorithm, based on the idea of calculating partial equilibria described in [14], with the following properties has been developed:
  - It is distributed which means it runs on every context and communicates with other contexts.
  - Every instance can request information (partial diagnosis) from other contexts, do the inference calculations for the local context, combine the results, and send them to requesting contexts.
  - The algorithm can handle MCS topographies which lead to circular requests.

  - The view on the overall MCS is no more a global one but always depending on the context from which we call the algorithm. This reflects the fact that the contexts of the MCS are distributed and a user is often interested only in knowledge relevant for the queried context.

- We were able to show that the algorithm is correct and complete in the sense that it will calculate all partial diagnoses and the supporting partial equilibria w.r.t. to a context and all results calculated by the algorithm are partial diagnoses and the corresponding partial equilibria under those partial diagnoses.

- We accomplished to build a prototype implementation and were able to perform tests on selected problem instances. This gave us an impression on how problem instances of different sizes and structures are performing on state-of-the-art machines. We could also compare our prototype with other implementations in this field which cover a subset of the functionality of ours. The main result of our experimental studies showed that the proposed optimizations had an measurable effect on the tested problem instances.

1.6 Thesis Organization

The thesis will be organized as follows: In Chapter 2 we will give an overview of the existing definitions in the field of Multi-Context Systems which are of interest for our work. Based on this in Chapter 3 we will introduce the formal definitions, lemmas, and theorems which are necessary for the algorithm. The algorithm itself will then be introduced in a formal way in Chapter 4 while the implementation of this algorithm is described in Chapter 5. Then a report about the experimental evaluation will be given in Chapter 6 which is followed by the concluding part in Chapter 7.

CHAPTER 2

Preliminaries

In this chapter we give formal definitions of those concepts the work in this thesis is based on. These are the concept of a Multi-Context System including the definition of an equilibrium [10], the extension of equilibria and its corresponding definitions with a partial view [14], and the concept of diagnoses [16].

2.1 Multi-Context Systems

Multi-Context Systems can deal with different monotonic and nonmonotonic knowledge representation formalisms like propositional logic, answer set programs or description logic. The following notation abstracts from those formalisms:

A *logic* is, viewed abstractly, a tuple $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$, where

- $\mathbf{KB}_L$ is a set of well-formed knowledge bases, each being a set (of formulas),
- $\mathbf{BS}_L$ is a set of possible belief sets, each being a set (of formulas), and
- $\mathbf{ACC}_L \colon \mathbf{KB}_L \to 2^{\mathbf{BS}_L}$ assigns each $kb \in \mathbf{KB}_L$ a set of acceptable belief sets.

Definition 2.1.1 (Multi-Context System)
A Multi-Context System *(MCS) $M = (C_1, \ldots, C_n)$ consists of* contexts $C_i = (L_i, kb_i, br_i)$*, $1 \leq i \leq n$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ is a knowledge base, and br_i is a set of L_i-*bridge rules *of the form*

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathrm{not}\,(c_{j+1} : p_{j+1}), \ldots, \mathrm{not}\,(c_m : p_m) \qquad (2.1.1)$$

where $1 \leq c_k \leq n$, p_k is an element of some belief set of L_{c_k}, $1 \leq k \leq m$, and $kb \cup \{s\} \in \mathbf{KB}_i$ for each $kb \in \mathbf{KB}_i$.

Let r be such a bridge rule of form (2.1.1) *then s is denoted as $head(r)$ and $(c_1 : p_1), \ldots, (c_j : p_j), \mathrm{not}\,(c_{j+1} : p_{j+1}), \ldots, \mathrm{not}\,(c_m : p_m)$ as $body(r)$.*

With such bridge rules, information s can be added to a context, depending on the knowledge in other contexts.

In an MCS contexts depend on the beliefs of other contexts through bridge rules; therefore we define a state which indicates the beliefs of every context as a defined state of the Multi-Context System. Such a state is called *belief state*.

Definition 2.1.2 (Belief State)
Let $M = (C_1, \ldots, C_n)$ be an MCS. A belief state is a sequence $S = (S_1, \ldots, S_n)$ such that each S_i, with $1 \leq i \leq n$, is an element of $\mathbf{BS}_i$.

A bridge rule adds information to a context if it is *applicable* in the MCS. The applicability depends on the belief sets of other contexts and therefore on the belief state of the MCS.

Definition 2.1.3 (Bridge Rule Applicability)
Let S be a belief state and r a bridge rule of form (2.1.1). *Then r is* applicable in S*, if $p_i \in S_{c_i}$, for $1 \leq i \leq j$, and $p_k \notin S_{c_k}$, for $j + 1 \leq k \leq m$. Let $app(R, S)$ denote the set of all bridge rules $r \in R$ that are applicable in S, and $head(r)$ the part s of any r of form* (2.1.1).

If every belief set of a belief state is accepted in its context under consideration of the bridge rules, a belief state is called equilibria and is an accepted state of the whole Multi-Context System.

Definition 2.1.4 (Equilibrium)
A belief state $S = (S_1, \ldots, S_n)$ of a multi-context system M is an equilibrium *iff for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$.*

To illustrate an MCS and the above defined concepts, a simple example with two contexts, each based on Answer Set Programming [18] as knowledge representation formalism, will be given.

Example 2.1.1
First we define an MCS with context C_1 and a knowledge base $kb_1 = \{a \leftarrow b\}$ and a bridge rule $br_1 = \{b \leftarrow (2 : d)\}$ and context C_2 with $kb_2 = \{c \vee d\}$ and no bridge rules. From this the following belief sets follow: $\mathbf{BS}_1 = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ and $\mathbf{BS}_2 = \{\emptyset, \{c\}, \{d\}, \{c, d\}\}$.

From these belief sets we can construct sixteen different belief states: $(\emptyset, \emptyset), (\{a\}, \emptyset), \ldots,$ $(\{a, b\}, \{d\}), (\{a, b\}, \{c, d\})$. To find those belief sets which are equilibria we first have a look at C_2 because no head of a bridge rule is from C_2 and therefore it is easy to find $\mathbf{ACC}_2(kb_2) = \{\{c\}, \{d\}\}$. This means we can reduce our sets of possible equilibria to $S_c = \{(\cdots, \{c\})\}$ and $S_d = \{(\cdots, \{d\})\}$. In the first case it's easy to continue since $app(br_1, S_c) = \emptyset$ therefore $\mathbf{ACC}_1(kb_1) = \emptyset$ and the only equilibria which is left is $(\emptyset, \{c\})$. In the second case $\{head(r) \mid r \in app(br_1, S_d)\} = \{b\}$ and therefore we have to check $\mathbf{ACC}_1(kb_1 \cup \{b\})$ which is $\{a, b\}$ and therefore the only resulting equilibrium is $(\{a, b\}, \{d\})$.

□

2.2 Partial Equilibria

To introduce the concept of *partial* equilibria and later *partial* diagnoses, the *import closure* is an important notion. Every context has a set of bridge rules whose bodies contain elements from other contexts. For a specific context and its bridge rules the contexts of those elements is the *import neighborhood* of the context. The import closure of a specific context is the recursive union of its import neighborhood and the import neighborhoods of every context in the import closure. In other words the import closure indicates the part of the MCS on which a context depends.

The import closure with respect to a single context has already been defined by Dao-Tran et al. [14].

Definition 2.2.1 (Import Closure)
Let $M = (C_1, \dots, C_n)$ 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 a context C_k is the smallest set S such that*

- *$k \in S$ and*
- *for all $i \in S$, $In(i) \subseteq S$.*

According to Dao-Tran et al. [14] an equal definition is: $IC(k) = \{k\} \cup \bigcup_{j \geq 0} IC^j(k))$, where $IC^0(k) = In(k)$, and $IC^{j+1}(k) = \bigcup_{i \in IC^j(k)} In(i)$.

A *partial belief state* is a belief state where some belief sets are left undefined, this indicates that information about those contexts is not available.

Definition 2.2.2 (Partial Belief State)
Let $M = (C_1, \dots, C_n)$ be an MCS, and ϵ a new symbol, where $\epsilon \notin \bigcup_{i=1}^{n} \mathbf{BS}_i$. A partial belief state *of M is a sequence $S = (S_1, \dots, S_n)$, such that $S_i \in \mathbf{BS}_i \cup \{\epsilon\}$, for $1 \leq i \leq n$.*

A *partial equilibrium* is then defined with respect to a specific context, let's say C_k. It is a partial belief state with defined belief sets for every context in the import closure of C_k and those belief sets are accepted on their contexts based on the knowledge base of the context and the applicable bridge rules.

The definition of partial equilibria w.r.t. a single context is from Dao-Tran et al. [14]; for the purpose of this thesis the definition will later be extended to a set of contexts.

Definition 2.2.3 (Partial Equilibrium)
Let $M = (C_1, \dots, C_n)$ be an MCS and $S = (S_1, \dots, S_n)$ a partial belief state of M as defined above. Then S is also a partial equilibrium *of M w.r.t. a context $C_k \in \{C_1, \dots, C_n\}$ iff for all $1 \leq i \leq n$:*

- *$i \in IC(k)$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and*
- *$S_i = \epsilon$ otherwise.*

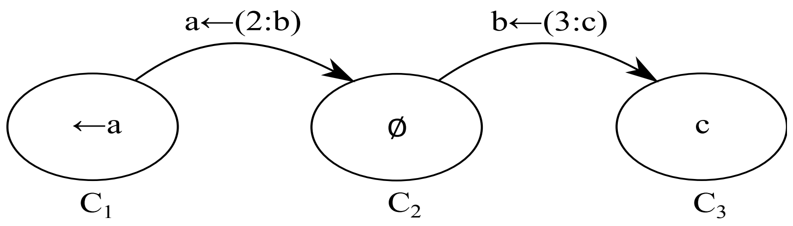


Figure 2.2.1: Example MCS to demonstrate partial equilibria

We next give an example to demonstrate import closures, partial belief states and partial equilibria.

Example 2.2.1
As pictured in Figure 2.2.1 we have a context C_1 with $kb_1 = \{\leftarrow a\}$ and $br_1 = \{a \leftarrow (2 : b)\}$, a context C_2 with $kb_2 = \emptyset$ and $br_2 = \{b \leftarrow (3 : c)\}$ and a context C_3 with $kb_3 = \{c\}$ and no bridge rules.

The import closure of C_3 is $\{3\}$ since there is no bridge rule in the MCS which effects C_3. The import closure of C_2 is $\{2, 3\}$ since the knowledge base of C_2 depends on C_3 due to the bridge rule of br_2 and the import closure of C_1 is $\{1, 2, 3\}$ since the knowledge base if C_1 is effected by C_2 and therefore also by C_3.

Now we can construct partial belief states which cover the above mentioned import closures. Viz. $(\epsilon, \epsilon, \{c\})$ and $(\epsilon, \epsilon, \emptyset)$ for the import closure of C_3, e.g., $(\epsilon, \emptyset, \{c\})$ or $(\epsilon, \{b\}, \{c\})$ for C_2 and $(\{a\}, \{b\}, \{c\})$ for C_1. Note that the belief sets do not have to be accepted on the context nor is the applicability of the bridge rules of relevance. Furthermore we can construct partial belief states which are not connected to any import closure like $(\{a\}, \epsilon, \epsilon)$.

To demonstrate the partial equilibria we can construct one w.r.t. C_2. The import closure of C_2 is $\{2, 3\}$ so the partial equilibria E_{C_2} are of the form (ϵ, S_2, S_3). In this example there is only one partial equilibrium viz. $(\epsilon, \{b\}, \{c\})$. Note that there is no equilibrium for the whole MCS since there is no belief set in C_1 which is accepted, but this is not relevant for the partial view on the MCS.

□

Partial equilibria can be built by combining partial equilibria from different parts of an MCS. An operation called *join* has been defined, again by Dao-Tran et al. [14], to accomplish this.

Definition 2.2.4 (Join of Partial Equilibria)
Let $E' = (E'_1, \dots, E'_n)$ and $E'' = (E''_1, \dots, E''_n)$ be partial equilibria, then the result of a join $E' \bowtie E''$ is $E = (E_1, \dots, E_n)$, where

- $E_i = E'_i = E''_i$, *if* $E'_i = E''_i$,
- $E_i = E'_i$, *if* $E'_i \neq \epsilon \wedge E''_i = \epsilon$,
- $E_i = E''_i$, *if* $E''_i \neq \epsilon \wedge E'_i = \epsilon$

for all $1 \leq i \leq n$.
Note that $E' \bowtie E''$ *is void iff* $E'_i \neq \epsilon$, $E''_i \neq \epsilon$ *and* $E'_i \neq E''_i$ *for some* $1 \leq i \leq n$.

The next example illustrates a join operation between two partial equilibria.

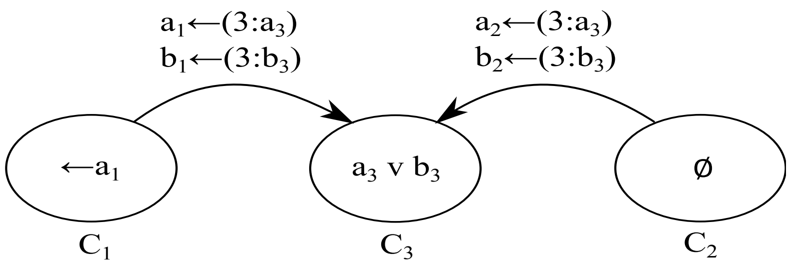


Figure 2.2.2: Example to demonstrate a join between two partial equilibria

Example 2.2.2
We have a context C_1 with $kb_1 = \{\leftarrow a_1\}$ and $br_1 = \{a_1 \leftarrow (3 : a_3), b_1 \leftarrow (3 : b_3)\}$, a context C_2 with $kb_2 = \emptyset$ and $br_2 = \{a_2 \leftarrow (3 : a_3), b_2 \leftarrow (3 : b_3)\}$ and a context C_3 with $kb_3 = \{a_3 \vee b_3\}$ and no bridge rules. Moreover we have one partial equilibrium w.r.t. C_1, $E_{C_1} = (\{b_1\}, \epsilon, \{b_3\})$ and two w.r.t. C_2, $E'_{C_2} = (\epsilon, \{a_2\}, \{a_3\})$ and $E''_{C_2} = (\epsilon, \{b_2\}, \{b_3\})$. A join between E_{C_1} and E'_{C_2} returns no result since the belief sets in C_3 are different ($\{b_3\} \neq \{a_3\}$). The join between E_{C_1} and E''_{C_2} results in the the partial equilibrium $E_{\{C_1,C_2\}} = (\{b_1\}, \{b_2\}, \{b_3\})$ which is the only one w.r.t. $\{C_1, C_2\}$[1]. This example shows also that the resulting partial equilibrium is a total equilibrium since it covers the whole *MCS*.

□

2.3 Total Diagnoses

Diagnoses have already been defined by Eiter et al. [16]: "As well-known, in nonmonotonic reasoning, adding knowledge can both cause and prevent inconsistency; the same is true for removing knowledge. For our consistency-based explanation of inconsistency, we therefore consider pairs of sets of bridge rules, s.t. if we deactivate the rules in the first set, and add the rules in the second set in unconditional form, the MCS becomes consistent (i.e., admits an equilibrium)."

To clearly differentiate between diagnoses of the whole system and partial diagnoses, which we will define later, in the following we call the diagnoses of the whole *MCS total diagnoses*.

Notation. Let M be an *MCS* and R a set of bridge rules compatible with M. Then $M[R]$ is an *MCS* derived from M by replacing all bridge rules with the bridge rules R. E.g., $M[\emptyset]$ is the MCS M without any bridge rules. Furthermore the following notations are used: $M \models \perp$

[1]A formal definition of equilibria w.r.t. to a set of contexts will be given in Chapter 3

denotes that M has no equilibrium and is therefore inconsistent and $M \not\models \bot$ denotes that M has at least one equilibrium.

Definition 2.3.1 (Total Diagnosis)
Given an MCS $M = (C_1, \dots, C_n)$, a diagnosis of M is a pair (D_1, D_2), $D_1, D_2 \subseteq br_M$, s.t. $M[br_M \setminus D_1 \cup heads(D_2) \not\models \bot$

Example 2.3.1
To demonstrate total diagnoses we recall the example from Figure 2.2.1 where we have a context C_1 with $kb_1 = \{\leftarrow a\}$ and $br_1 = \{a \leftarrow (2 : b)\}$, a context C_2 with $kb_2 = \emptyset$ and $br_2 = \{b \leftarrow (3 : c)\}$ and a context C_3 with $kb_3 = \{c\}$ and no bridge rules. To reference the two bridge rules later we denote the bridge rule from br_1 as r_1 and the bridge rule from kb_2 as r_2.

As we have already seen there is no equilibrium in this MCS because there is no accepted belief set for context C_1. It is also easy to see that if we remove one (or both) of the bridge rules we can find an equilibrium. So the total diagnoses $(r_1, \emptyset)$ and $(r_2, \emptyset)$ lead both to the equilibrium $(\emptyset, \emptyset, \{c\})$. In this case we don't need the part of the diagnoses where the head of a bridge rule is applied unconditionally but if we construct such a diagnosis only $(\{r_1\}, \{r_2\})$ leads to an equilibrium. If we would apply r_1 unconditionally C_1 can not be satisfied and in case we add r_2 unconditionally r_1 has to be removed otherwise r_1 would add a to C_1s knowledge base and therefore make it impossible to find an applicable belief set.

□

CHAPTER 3

A Partial View on Diagnosis

In this chapter we combine the idea of a partial view on a Multi-Context System, introduced in Dao-Tran et al. [14], and the concept of diagnoses. We denote the resulting concept of this approach *partial diagnosis* which is the core idea on which the work in this thesis is based on. For a properly functioning algorithm, handling partial diagnoses, some auxiliary concepts strongly related to partial diagnoses are necessary and they also are stated in this chapter. These are *witnesses* which prove the correctness of partial diagnoses and *candidates* which are potential partial belief states based on not yet proved assumptions. Moreover this chapter shows also properties of partial diagnoses w.r.t. generalization and modularity and how and whether partial diagnoses of the same MCS can be combined.

3.1 Preparations

In order to properly define the following concepts of partial diagnoses, witnesses and candidates we have to slightly modify the definitions from the preliminaries.

First we extend the definition of a import closure such that it also covers the import closure of not only a single context but of a set of contexts.

Definition 3.1.1 (Import Closure)
Let $M = (C_1, \dots, C_n)$ be an MCS. The import neighborhood of a set of contexts $\mathcal{C} \subseteq \{C_1, \dots, C_n\}$ *from M is*

$$IC(\mathcal{C}) = \bigcup_{C_k \in \mathcal{C}} IC(C_k)$$

where the import neighborhood of C_k is the set

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

and the import closure *$IC(C_k)$ is the smallest set S such that*

- *$k \in S$ and*
- *for all $i \in S, In(i) \subseteq S$.*

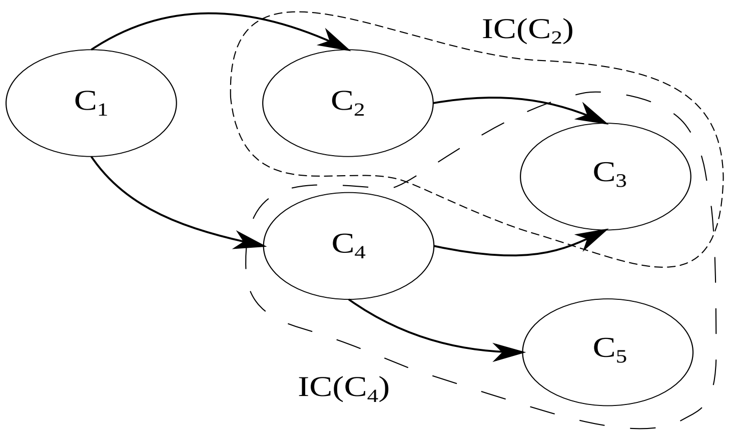


Figure 3.1.1: MCS with two import closures

Example 3.1.1
In Figure 3.1.1 we see an MCS and the import closure $IC(C_2) = \{2, 3\}$ and $IC(C_4) = \{4, 3, 5\}$. In this case the import closure of the set of contexts $\{C_2, C_4\}$, $IC(\{C_2, C_4\})$, is $\{2, 3, 4, 5\}$ which is the union of the import closure $IC(C_2)$ and $IC(C_4)$.

$\square$

The following propositions about the import closure concept are needed in later lemmas respectively their proofs: The import closure of the union of two sets of contexts is the same as the union of the import closures of each set.

Proposition 3.1.1
Let $M = (C_1, \ldots, C_n)$ be an MCS and $\mathcal{C}'$, $\mathcal{C}'' \subseteq \{C_1, \ldots, C_n\}$ two subsets of the contexts from M. Then

$$IC(\mathcal{C}' \cup \mathcal{C}'') = IC(\mathcal{C}') \cup IC(\mathcal{C}'').$$

Proof
By Definition 3.1.1 $IC(\mathcal{C}' \cup \mathcal{C}'') = \bigcup_{C_j \in (\mathcal{C}' \cup \mathcal{C}'')} IC(C_j)$. Then $\bigcup_{C_j \in (\mathcal{C}' \cup \mathcal{C}'')} IC(C_j)$ can be split into $\bigcup_{C_j \in \mathcal{C}'} IC(C_j) \cup \bigcup_{C_j \in \mathcal{C}''} IC(C_j)$ which is $IC(\mathcal{C}') \cup IC(\mathcal{C}'')$.

$\square$

If a set of contexts is a subset of another set of contexts then the import closures of both sets have the same subset relation.

Proposition 3.1.2
Let $M = (C_1, \ldots, C_n)$ be an MCS and $\mathcal{C}$, $\mathcal{C}' \subseteq \{C_1, \ldots, C_n\}$ two subsets of the contexts from M. Then

$$\mathcal{C}' \subseteq \mathcal{C} \textit{ implies } IC(\mathcal{C}') \subseteq IC(\mathcal{C}).$$

Proof
If $\mathcal{C}' \subseteq \mathcal{C}$ then for $\mathcal{C}'' = \mathcal{C} \setminus \mathcal{C}'$ it holds that $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$.

Since $IC(\mathcal{C}') = \bigcup_{C \in \mathcal{C}'} IC(C)$ we can obtain from Proposition 3.1.2 that $IC(\mathcal{C}) = IC(\mathcal{C}' \cup \mathcal{C}'') = IC(\mathcal{C}') \cup IC(\mathcal{C}'') = \bigcup_{C \in \mathcal{C}'} IC(C) \cup \bigcup_{C \in \mathcal{C}''} IC(C)$.

Because of $\bigcup_{C \in \mathcal{C}'} IC(C) \subseteq \bigcup_{C \in \mathcal{C}'} IC(C) \cup \bigcup_{C \in \mathcal{C}''} IC(C)$ we can further obtain $IC(\mathcal{C}') \subseteq IC(\mathcal{C})$.

□

Example 3.1.2
Proposition 3.1.1 and 3.1.2 can easily be demonstrated with the *MCS* from Figure 3.1.1. It is easy to see that the first proposition holds for $IC(C_2) \cup IC(C_4) = IC(\{C_2\} \cup \{C_4\})$ as well as the second proposition holds for $\{C_4\} \subseteq \{C_2, C_4\}$ and therefore $IC(\{C_4\}) \subseteq IC(\{C_2, C_4\})$.

□

The applicability of a bridge rule depends on the part of the *MCS* which is defined by the import neighborhood of the bridge rules context. Given a bridge rule from context C which is applicable w.r.t. a belief state S. If another belief state S' has equal belief sets in the contexts of the import neighborhood of C, then the bridge rule is also applicable w.r.t. S'.

Proposition 3.1.3
Let $S' = (S'_1, \ldots, S'_n)$ and $S'' = (S''_1, \ldots, S''_n)$ be two belief states of an MCS M and C_i a context of M. If for all $k \in In(i)$, $S'_k = S''_k$ then $r \in app(br_i, S')$ iff $r \in app(br_i, S'')$ holds.

Proof
Recall that by Definition 2.1.3 a bridge rule of form (2.1.1) is applicable in S' if $p_l \in S'_{c_l}$, for $1 \leq l \leq j$ and $p_k \notin S'_{c_k}$, for $j+1 \leq k \leq m$. Further recall Definition 3.1.1, $In(i) = \{c_n \mid (c_n : p_n) \in B(r), r \in br_i\}$.

Since $In(i)$ represents exactly those contexts with indexes c_l and c_k from Definition 2.1.3 and $S'_m = S''_m$ for all $m \in In(i)$, it holds that $r \in app(br_i, S') \implies r \in app(br_i, S'')$. The proof works the same way for the other direction and therefore it holds that $r \in app(br_i, S')$ *iff* $r \in app(br_i, S'')$

□

Example 3.1.3
Let's again show this proposition on the *MCS* from Figure 3.1.1. Given C_4 has a bridge rule r_1 which has elements from C_3 and C_5 in its body and is applicable on the belief state $S = (\epsilon, \epsilon, S_3, S_4, S_5)$. Then r_1 is also applicable in any belief state where the belief sets of the contexts C_3, C_4 and C_5 are the same, e.g., $S' = (\epsilon, S'_2, S_3, S_4, S_5)$, because the applicability of r_1 only depends on the contexts of the import closure of C_4 which are C_3, C_4 and C_5.

□

Since we now have extended the definition of a import closure to a set of contexts we can extend the definition of the partial equilibrium accordingly:

Definition 3.1.2 (Partial Equilibrium)
Let $M = (C_1, \ldots, C_n)$ be an MCS and $S = (S_1, \ldots, S_n)$ a partial belief state of M as defined in 2.2.2. Then S is a partial equilibrium of M w.r.t. a set of contexts $\mathcal{C}$ *iff for all $1 \leq i \leq n$:*

- *$i \in IC(\mathcal{C})$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and*
- *$S_i = \epsilon$ otherwise.*

Notation. A partial equilibrium E with respect to a set of contexts $\mathcal{C}$ is also written as $E_{\mathcal{C}}$. Furthermore the following notations are used: $M \models \perp$ denotes that M has no equilibrium and is therefore inconsistent and $M \not\models \perp$ denotes that M has at least one equilibrium. To distinguish equilibria according to Definition 2.1.4 from partial equilibria we call the former *total equilibria*.

Example 3.1.4
Let's recall the MCS from Figure 3.1.1 where a possible partial equilibria could have been E_{C_2} if all the belief sets of the contexts from $IC(C_2)$ are accepted and those from all other contexts are set to ϵ or E_{C_4} if all the belief sets of the contexts from $IC(C_4)$ are accepted and again those from all other contexts are ϵ. Now, with the new definition of partial equilibria we can also express a partial equilibrium of form $E_{\{C_2,C_4\}} = (\epsilon, S_2, S_3, S_4, S_5)$ where all belief sets from the import closure $IC(\{C_2, C_4\}) = \{C_2, C_3, C_4, C_5\}$ are accepted and the belief set of context C_1 is ϵ.

If we have a partial equilibrium $E_{\mathcal{C}}$, select a subset $\mathcal{C}' \subseteq \{C_i \mid i \in IC(\mathcal{C})\}$ from the import closure of $\mathcal{C}$ and set all belief sets from $E_{\mathcal{C}}$ to ϵ if they are not in the import closure of C', then the resulting belief state is again a partial equilibrium $E'_{\mathcal{C}'}$ w.r.t. $\mathcal{C}'$.

Proposition 3.1.4
Let $M = (C_1, \ldots, C_n)$ be an MCS, $E_{\mathcal{C}}$ a partial equilibrium in M, and $\mathcal{C}' \subseteq \{C_i \mid i \in IC(\mathcal{C})\}$. Then $E'_{\mathcal{C}'} = (E'_1, \ldots, E'_n)$, where $E'_i = E_i$ if $i \in IC(\mathcal{C}')$ and $E'_i = \epsilon$ otherwise, is also a partial equilibrium in M.

Proof
The proposition itself states that $E'_i = \epsilon$ if $i \notin IC(\mathcal{C}')$ so it is left to show that $E'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E')\})$ if $i \in IC(\mathcal{C}')$.

$\mathcal{C}'$ is a subset of $\{C_i \mid i \in IC(\mathcal{C})\}$ and therefore we infer from Proposition 3.1.2 that $IC(\mathcal{C}') \subseteq IC(\{C_i \mid i \in IC(\mathcal{C})\}) = IC(\mathcal{C})$.[1] Since E is a partial equilibrium w.r.t. $\mathcal{C}$ we show that $E_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E)\}) \implies E'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E')\})$ for $i \in IC(\mathcal{C}')$.

Because $E_i = E'_i$ if $i \in IC(\mathcal{C}')$ we can reduce it to $r \in app(br_i, E) \implies r \in app(br_i, E')$.

Since $E_i = E'_i$ for all $i \in IC(\mathcal{C}')$ and $\bigcup_{j \in \mathcal{C}'} In(j) \subseteq IC(\mathcal{C}')$, Proposition 3.1.3 states that $r \in app(br_i, E) \implies r \in app(br_i, E')$ holds and therefore $E'_{\mathcal{C}'}$ is a partial equilibrium w.r.t. $\mathcal{C}'$.

$\square$

Example 3.1.5
Let's say we have a partial equilibrium $E_{\{C_2,C_4\}} = (\epsilon, S_2, S_3, S_4, S_5)$ in the MCS from Figure 3.1.1. Since $\{C_4\} \subseteq IC(\{C_4, C_2\})$, according to Proposition 3.1.4, $E_{\{C_4\}} = (\epsilon, \epsilon, S_3, S_4, S_5)$ is as well a partial equilibrium in this MCS because all belief sets which are in $IC(\{C_4\}) = \{3, 4, 5\}$ are the same as in $E_{\{C_2,C_4\}}$ and all other belief sets are ϵ.

[1] $IC(\{C_i \mid i \in IC(\mathcal{C})\})$ is trivially $IC(\mathcal{C})$.

The next proposition states that a partial equilibrium $E_{\mathcal{C}}$ in an $MCS\ M$ is also a partial equilibrium in a modified $MCS\ M'$ if the contexts defined by the import closure of $\mathcal{C}$ are the same.

Proposition 3.1.5
Let $M = (C_1, \ldots, C_n)$ be an MCS and $E_{\mathcal{C}}$ a partial equilibrium in M. Let $M' = (C'_1, \ldots, C'_o)$ be another MCS with $C'_i = C_i$ for all $i \in IC(\mathcal{C})$. Then the partial equilibrium $E_{\mathcal{C}}$ is also a partial equilibrium in the $MCS\ M'$.

Proof
A partial belief state $E_{\mathcal{C}}$ is a partial equilibrium if $E_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E)\})$ for all $i \in IC(\mathcal{C})$.

Since all contexts C'_i in M' are the same as in M if $i \in IC(\mathcal{C})$, it's left to show that the result of $app(br_i, E)$ is not depending on contexts C'_i for $i \notin IC(\mathcal{C})$.

Recall that a bridge rule of the form $s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \text{not}\ (c_{j+1} : p_{j+1}), \ldots, \text{not}\ (c_m : p_m)$ is applicable in E if $p_l \in E_{c_l}$, for $1 \leq l \leq j$, and $p_k \notin E_{c_k}$, for $j+1 \leq k \leq m$. Since $In(i)$ represents exactly those contexts on which the applicability of a bridge rule from C_i depends on and it holds for $i \in IC(\mathcal{C})$ that $In(i) \subseteq IC(\mathcal{C})$, $app(br_i, E)$ does not depend on contexts C_k with $k \notin IC(\mathcal{C})$. Therefore $E_{\mathcal{C}}$ is also a partial equilibrium in M'.

□

Example 3.1.6
We use the MCS from the previous examples illustrated in Figure 3.1.1 to show this proposition and call it M. Let's assume we have a partial belief state $(\epsilon, S_2, S_3, S_4, S_5)$ which is a partial equilibrium $E_{\{C_2, C_4\}}$ in M. Now we take M and modify C_1 (the knowledge base and/or the bridge rules) and/or add new contexts (without changing the indexes from the already existing contexts) and call this new $MCS\ M'$. Since we didn't change the contexts C_2, C_3, C_4 and C_5 which are the import closure $IC(\{C_2, C_4\})$ in M, $IC(\{C_2, C_4\})$ is still the same in M' and therefore also the partial belief state $(\epsilon, S_2, S_3, S_4, S_5)$ still a partial equilibria in M'. If we would add a context to M, e.g., C_6, then $(\epsilon, S_2, S_3, S_4, S_5, \epsilon)$ would still be a partial equilibria in M'.

□

The definition of a join between the adapted partial equilibria remains the same as between the original partial equilibria:

Definition 3.1.3 (Join of Partial Equilibria)
Let $E'_{\mathcal{C}'} = (E'_1, \ldots, E'_n)$ and $E''_{\mathcal{C}''} = (E''_1, \ldots, E''_n)$ be partial equilibria, then the result of a join $E' \bowtie E''$ is $E = (E_1, \ldots, E_n)$, where

- *$E_i = E'_i = E''_i$, if $E'_i = E''_i$,*
- *$E_i = E'_i$, if $E'_i \neq \epsilon \wedge E''_i = \epsilon$,*
- *$E_i = E''_i$, if $E''_i \neq \epsilon \wedge E'_i = \epsilon$*

for all $1 \leq i \leq n$.
Note that $E' \bowtie E''$ is void iff $E'_i \neq \epsilon$, $E''_i \neq \epsilon$ and $E'_i \neq E''_i$ for some $1 \leq i \leq n$.

The result of a join between two partial equilibria is, if it exists, again a partial equilibrium. Later the Multi-Context Systems, respectively the bridge rules, will be altered by diagnoses and it will be necessary to join partial equilibria under different diagnoses. This necessitates additional assumptions about the structure of the $MCSs$ involved in a join operation. Roughly speaking the assumptions state that those contexts where the partial equilibria of a join operation are defined must be equal in both $MCSs$. All other contexts do not influence the join and can therefore be different.

Lemma 3.1.6
Let $E'_{\mathcal{C}'}$ be a partial equilibrium in the MCS $M' = (C'_1, \ldots, C'_n)$ and $E''_{\mathcal{C}''}$ be a partial equilibrium in $M'' = (C''_1, \ldots, C''_n)$. Further lets assume that $\mathcal{C}' \subseteq \{C'_1, \ldots, C'_n\}$, $\mathcal{C}'' \subseteq \{C''_1, \ldots, C''_n\}$ and $M = (C_1, \ldots, C_n)$ an MCS such that $C_i = C'_i$ if $i \in IC(\mathcal{C}')$ and $C_i = C''_i$ if $i \in IC(\mathcal{C}'')$[2]. Then the join $E'_{\mathcal{C}'} \bowtie E''_{\mathcal{C}''}$, if not void, returns $E_{\mathcal{C}' \cup \mathcal{C}''}$ where E is a partial equilibrium in M with respect to the contexts $\mathcal{C}' \cup \mathcal{C}''$.

Note that $E_{\mathcal{C}' \cup \mathcal{C}''}$ is a partial equilibrium in M' respectively M'' if $M' = M''$.

Proof
Using Proposition 3.1.5 we know that E' and E'' are also partial equilibria in M.
$E_{\mathcal{C}' \cup \mathcal{C}''} = (E_1, \ldots, E_n)$ is a partial equilibrium in M if it fulfills the following two statements:

a) $i \in IC(\mathcal{C}' \cup \mathcal{C}'') \implies E_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E)\})$ and

b) $i \notin IC(\mathcal{C}' \cup \mathcal{C}'') \implies E_i = \epsilon$

for all $1 \leq i \leq n$.

Statement a):

Proposition 3.1.1 states that $IC(\mathcal{C}' \cup \mathcal{C}'') = IC(\mathcal{C}') \cup IC(\mathcal{C}'')$ and therefore if $i \in IC(\mathcal{C}' \cup \mathcal{C}'')$ it is either true that $i \in IC(\mathcal{C}')$ or $i \in IC(\mathcal{C}'')$.

For every $i \in IC(\mathcal{C}')$ it holds that $E'_i \neq \epsilon$ and therefore $E_i = E'_i$. We also know that for every $i \in IC(\mathcal{C}')$, $E'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E')\})$. According to Proposition 3.1.3 $app(br_i, E')$ depends only on contexts $C_i \mid i \in IC(\mathcal{C}')$ and because for those contexts $E_i = E'_i$ it also holds that $E_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E)\})$.

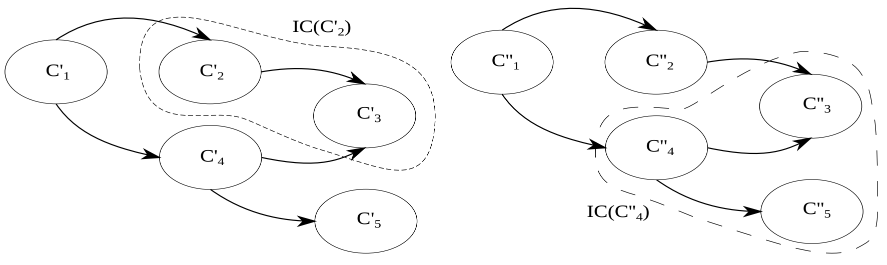
For every $i \in IC(\mathcal{C}'')$ it holds that $E''_i \neq \epsilon$ and therefore $E_i = E''_i$. We also know that for every $i \in IC(\mathcal{C}'')$, $E''_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E'')\})$. According to Proposition 3.1.3 $app(br_i, E'')$ depends only on contexts $C_i \mid i \in IC(\mathcal{C}'')$ and because for those contexts $E_i = E''_i$ it also holds that $E_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, E)\})$.

Statement b):

If $i \notin IC(\mathcal{C}' \cup \mathcal{C}'')$ then $i \notin IC(\mathcal{C}')$ and $i \notin IC(\mathcal{C}'')$ and therefore $E'_i = E''_i = \epsilon$. Because E_i is either E'_i or E''_i in every case $E_i = \epsilon$.

$\square$

[2]This implies that if $i \in IC(\mathcal{C}')$ and $i \in IC(\mathcal{C}'')$ then $C'_i = C''_i$. Otherwise no such M exists. This is consistent with the definition of the join between partial diagnoses, if $C'_i \neq C''_i$ then the partial diagnosis applied to M' is different in context i from the one applied to M'' and therefore no result of the join between those two partial diagnoses exists which would produce M.



(a) $MCS\ M'$ (b) $MCS\ M''$

Figure 3.1.2: Two $MCSs$ with partial equilibria to join

Example 3.1.7
Let's say we have an $MCS\ M$ where we apply some modifications which result in a new MCS M' (Figure 3.1.2a) and with some other modifications applied in M'' (Figure 3.1.2b). We further assume that we now have a partial equilibrium E'_{C_2} in M' and a partial equilibrium E''_{C_4} in M''. Lemma 3.1.6 now states that the join between E' and E'' is, if not void, a partial equilibrium in any MCS which has the same contexts as in the import closure of C'_2 and the import closure of C''_4. Note that this is only possible if $C'_3 = C''_3$ and that it does not matter how the contexts C'_1, C'_4 and C'_5 in M' and C''_1 and C''_2 in M'' do look like. The new partial equilibrium is defined in the new MCS's corresponding contexts of $IC(C'_2)$ and $IC(C''_4)$.

□

Before defining partial diagnoses we will state a modified syntax for total diagnoses which emphasizes the contexts where the bridge rules of the diagnosis belongs to. The semantic remains the same.

Definition 3.1.4 (Total Diagnosis)
Given an $MCS\ M = (C_1, \ldots, C_n)$, $DG = (DG_1, \ldots, DG_n)$ is a total diagnosis *of M iff $DG_i = \{(D_i, A_i), D_i, A_i \subseteq br_i\}$, s.t. $M[br_M \setminus (D_1 \cup \ldots \cup D_n) \cup heads(A_1 \cup \ldots \cup A_n)] \not\models \bot$*

The notation $DG = (D, A)$ is an abbreviation for $DG = (D_1 \cup \ldots \cup D_n, A_1 \cup \ldots \cup A_n)$.

3.2 Partial Diagnoses

We have already seen definitions of belief states and equilibria which are only defined over parts of the MCS. Now we also introduce such a definition for (total) diagnoses and call them *partial diagnoses*.

The definition of partial diagnoses is based on total diagnoses combined with the idea of a partial view on the Multi-Context System as done with partial equilibria. Like a partial equilibrium, a partial diagnosis is always defined with respect to a set of contexts. To be more specific, it is always defined for the contexts of the import closure of a specific set of contexts. To be a consistent partial diagnosis, there has to be at least one partial equilibrium with respect to the same set of contexts if the partial diagnosis is applied to the MCS.

Definition 3.2.1 (Partial Diagnosis)
Let $M = (C_1, \dots, C_n)$ be an MCS, let be ϵ a new symbol with $\epsilon \notin \{(D_i, A_i) | D_i, A_i \subseteq br_i)\}$, and let $\mathcal{C}$ be a set of contexts $\mathcal{C} \subseteq \{C_1, \dots, C_n\}$. The n-tuple $(DG_1, \dots, DG_i, \dots, DG_n)$, for short $DG_{\mathcal{C}}$, is a partial diagnosis *of M w.r.t. the set of contexts $\mathcal{C}$ iff:*

- *$DG_i \in \{(D_i, A_i) | D_i, A_i \subseteq br_i)\}$ if $i \in IC(\mathcal{C})$ and $DG_i = \epsilon$ otherwise, for $1 \leq i \leq n$, and*
- *there is a partial equilibrium $E_{\mathcal{C}}$ in $M[br_M \setminus \bigcup_{j \in IC(\mathcal{C})} D_j \cup heads(\bigcup_{j \in IC(\mathcal{C})} A_j)]$.*

Notation. $M[DG]$ is used in the following as a short notation for $M[br_M \setminus \bigcup_{j \in IC(\mathcal{C})} D_j \cup heads(\bigcup_{j \in IC(\mathcal{C})} A_j)]$. Furthermore $remove(DG_i)$ denotes those bridge rules which are removed from context C_i in $M[DG]$ compared to M and $ucond(DG_i)$ those bridge rules which are applied unconditionally. Formally, for a diagnosis $DG = (DG_1, \dots, DG_i, \dots, DG_n)$ and $DG_i = (D_i, A_i)$, $remove(DG_i) = D_i$ and $ucond(DG_i) = A_i$.

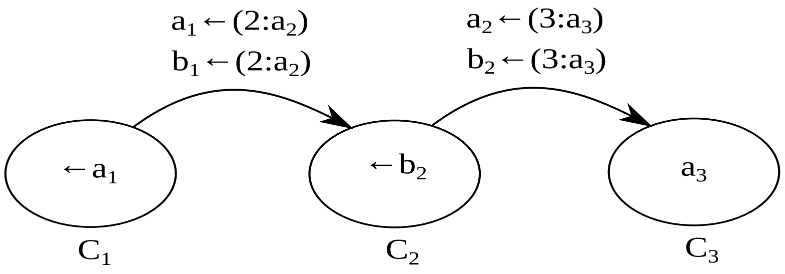


Figure 3.2.1: Example to demonstrate partial diagnoses and witnesses

Example 3.2.1
We have a context C_1 with $kb_1 = \{\leftarrow a_1\}$ and $br_1 = \{a_1 \leftarrow (2 : a_2), b_1 \leftarrow (2 : a_2)\}$, a context C_2 with $kb_2 = \{\leftarrow b_2\}$ and $br_2 = \{a_2 \leftarrow (3 : a_3), b_2 \leftarrow (3 : a_3)\}$, and a context C_3 with $kb_3 = \{a_3\}$ and no bridge rules. Context C_3 has a partial equilibrium $(\epsilon, \epsilon, \{a_3\})$ but then both bridge rules from C_2 are applicable, thus kb_2 extended with a_2 and b_2 and therefore we have no partial equilibrium w.r.t. C_2 because of b_2. If we apply a partial diagnosis $(\epsilon, (\{b_2 \leftarrow (3 : a_3)\}, \emptyset), (\emptyset, \emptyset))$ to the given MCS only bridge rule $a_2 \leftarrow (3 : a_3)$ is left at context C_2 and applicable and moreover now a partial equilibrium $E_{C_2} = (\epsilon, \{b_2\}, \{a_3\})$ exists. Note that for the partial diagnosis we constructed it does not matter that the bridge rules from C_1 are making the whole MCS inconsistent again because we are only interested in the part of the MCS which is covered by the import closure of C_2.

3.3 Witnesses

A witness is a partial equilibrium which proves a partial diagnosis to be valid. This means, if a partial diagnosis is applied to a Multi-Context System a witness is a partial equilibrium in the resulting MCS defined over the same set of contexts as the partial diagnosis. For every partial diagnosis there exists a set of witnesses which is not empty.

Definition 3.3.1 (Witness)
Let $M = (C_1, \ldots, C_n)$ be an MCS, let $DG_{\mathcal{C}}$ be a partial diagnosis w.r.t. the set of contexts $\mathcal{C}$, and let W be a partial belief state. Then W is a witness of $DG_{\mathcal{C}}$ iff W is a partial equilibrium w.r.t. $\mathcal{C}$ in $M[DG_{\mathcal{C}}]$.

Notation. We will use $wit(DG_{\mathcal{C}})$ to express the set of all witnesses of a partial diagnosis $DG_{\mathcal{C}}$. Note that $wit((\epsilon, \ldots, \epsilon))$ is the empty partial belief state $(\epsilon, \ldots, \epsilon)$.

The following example illustrates partial diagnoses and their witnesses.

Example 3.3.1
Let's recall Example 3.2.1 where we showed that after applying the partial diagnosis $(\epsilon, (\{b_2 \leftarrow (3 : a_3)\}, \emptyset), (\emptyset, \emptyset))$ the MCS had a partial equilibrium $E_{C_2} = (\epsilon, \{b_2\}, \{a_3\})$. Here E_{C_2} proves that the partial diagnosis is valid and therefore, according to Definition 3.3.1, we call it a witness for the above mentioned partial diagnosis. In this case there is only one witness for this partial diagnoses but we can also construct a MCS where we have many witnesses. E.g., let's add $c_2 \vee d_2$ to kb_2, then we would have two witnesses, viz., $(\epsilon, \{b_2, c_2\}, \{a_3\})$ and $(\epsilon, \{b_2, d_2\}, \{a_3\})$.

$\square$

3.4 Candidates

Partial diagnoses are an important intermediate result in every step of the distributed algorithm. In case the MCS has no cycles[3] every call to another instance returns a set of partial diagnoses with its corresponding witnesses. But in case of a cycle the context answering the request will not return complete partial diagnoses and witnesses w.r.t. to its context, instead it guesses possible local belief sets without checking the applicability of the bridge rules. Those "partial diagnoses" and "witnesses" with undefined contexts in the import closure are called candidates which are send back in order to prevent an infinite calling loop. They will be completed when the results are send back to a context where the guesses can be validated. From this point on the candidates are again valid partial diagnoses with corresponding witnesses which are well defined in all contexts of their import closure.

Those partial diagnoses, which are based on guesses and therefore only proper partial diagnoses if the underlying guesses turn out to be true, are called partial diagnosis candidates to distinguish them from proper ones. Note that partial diagnosis candidates, applied to an MCS, does not necessarily have a partial equilibrium and that a partial diagnosis candidate is not defined in every context of the given import closure.

Definition 3.4.1 (Partial Diagnosis Candidate)
Let $M = (C_1, \ldots, C_n)$ be an MCS, $\epsilon \notin \{(D_i, A_i) | D_i, A_i \subseteq br_i)\}$, $\mathcal{C} \subseteq \{C_1, \ldots, C_n\}$ and $\mathcal{D} \subseteq \{C_1, \ldots, C_n\}$ two sets of contexts with $\mathcal{C} \cup \mathcal{D} = \emptyset$. Then the n-tuple $(DG_1, \ldots, DG_i, \ldots, DG_n)$

[3]Contexts can request information from other contexts via bridge rules. If such a request chain leads to a context which is already in the chain it is called *cycle*.

or for short $DG_{\mathcal{C},\mathcal{D}}$, is a partial diagnosis candidate *of M w.r.t. to the set of contexts $\mathcal{C}$ and the set of guessed contexts $\mathcal{D}$ iff:*

a) $DG_i \in \{(D_i, A_i) | D_i, A_i \subseteq br_i\}$ *iff* $C_i \in \mathcal{C}$,

b) $DG_i = \epsilon$ *iff* $C_i \notin \mathcal{C}$ *and*

c) $M[DG_{\mathcal{C},\mathcal{D}}]$ *has at least one witness candidate* $W_{\mathcal{C},\mathcal{D}}$.

Since a partial diagnosis candidate does not have a partial equilibrium as a witness, the function $wit(DG_{\mathcal{C},\mathcal{D}})$ of a partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ returns a set of partial belief states called *witness candidates*. These belief states are defined in all contexts of $\mathcal{C}$ and $\mathcal{D}$. The belief states in the contexts of $\mathcal{C}$ are checked for acceptance under the corresponding partial diagnosis candidate and the belief states from the import neighborhood. The contexts from $\mathcal{D}$ are the guessed belief states and therefore are not checked for acceptance.

Definition 3.4.2 (Witness Candidate)
Let $M = (C_1, \ldots, C_n)$ be an MCS and $\mathcal{C} \subseteq \{C_1, \ldots, C_n\}$ as well as $\mathcal{D} \subseteq \{C_1, \ldots, C_n\}$ two sets of contexts with $\mathcal{C} \cup \mathcal{D} = \emptyset$. Then a partial belief state $S = (S_1, \ldots, S_n)$ is a witness candidate in M w.r.t. $\mathcal{C}$ and $\mathcal{D}$, denoted as $W_{\mathcal{C},\mathcal{D}}$ iff:

- $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$ *if* $C_i \in \mathcal{C}$,
- $S_i \in \mathbf{BS}_i$ *if* $C_i \in \mathcal{D}$ *and*
- $S_i = \epsilon$ *otherwise.*

$wit(DG_{\mathcal{C},\mathcal{D}})$ denotes all witness candidates of $DG_{\mathcal{C},\mathcal{D}}$. Note that a witness $W_{\mathcal{C}}$ is a witness candidate $W_{\mathcal{C}',\mathcal{D}}$ with $\mathcal{C}' = IC(\mathcal{C})$ and $\mathcal{D} = \emptyset$.

In Example 3.4.1 we show a partial diagnosis candidate with its corresponding witness candidates.

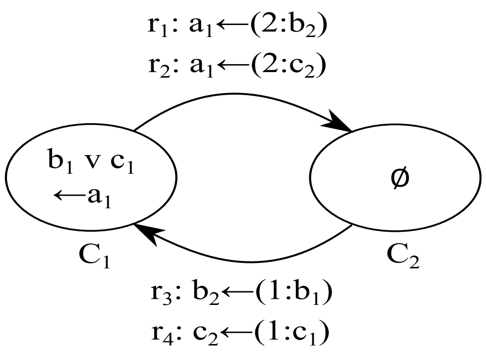


Figure 3.4.1: Example to show a partial diagnosis and witness candidate

Example 3.4.1
We have a context C_1 with $kb_1 = \{b_1 \vee c_1, \leftarrow a_1\}$ and $br_1 = \{r_1, r_2\}$ with $r_1 : a_1 \leftarrow (2 : b_2), r_2 : a_1 \leftarrow (2 : c_2)$ and a context C_2 with $kb_2 = \emptyset$ and $br_2 = \{r_3, r_4\}$ with $r_3 : b_2 \leftarrow (1 : b_1), r_4 : c_2 \leftarrow (1 : c_1)$. This MCS has no equilibrium therefore a diagnosis is necessary to make this system consistent. Since the contexts of this MCS build a cycle a partial diagnosis candidate can be shown. $((\epsilon, \epsilon), (\epsilon, \epsilon))$ is a partial diagnosis candidate $DG_{\emptyset,\{C_1\}}$ with, e.g., the witness candidates $W'_{\emptyset,\{C_1\}} = (\{c_1\}, \epsilon)$ and $W''_{\emptyset,\{C_1\}} = (\{b_1\}, \epsilon))$ where the belief sets $\{c_1\}$ and $\{b_1\}$ are guesses. Now we can extend $DG_{\emptyset,\{C_1\}}$ to $DG_{\{C_2\},\{C_1\}} = ((\epsilon, \epsilon), (\{r_4\}, \emptyset))$ and the witness candidates $W'_{\emptyset,\{C_1\}}$ and $W''_{\emptyset,\{C_1\}}$ to $W'_{\{C_2\},\{C_1\}} = (\{c_1\}, \emptyset)$ respectively $W''_{\{C_2\},\{C_1\}} = (\{b_1\}, \{b_2\})$. Finally, we extend the partial diagnosis candidate $DG_{\{C_2\},\{C_1\}}$ to a partial diagnosis $DG_{\{C_1\}} = ((\emptyset, \emptyset), (\{r_4\}, \emptyset))$ with the witness $W'_{\{C_1\}} = (\{c_1\}, \emptyset)$ because $(\{c_1\}, \emptyset)$ is a partial equilibrium under $DG_{\{C_1\}}$. $W''_{\{C_2\},\{C_1\}}$ will be dropped because the guess $\{b_1\}$ is not applicable under $DG_{\{C_1\}}$.

$\square$

3.5 Generalization and Modularity

Now we state some properties of the relation between total and partial equilibria. It is assumed that partial equilibria are a generalization of total equilibria. The next lemma points this out and shows which partial equilibria are also total equilibria. This lemma is needed later to show the generalization and modularization properties of partial diagnoses.

Lemma 3.5.1
As a trivial consequence of the definition of total equilibria (Definition 2.1.4) it follows that a total equilibria E in an MCS $M = (C_1, \ldots, C_n)$ is always a partial equilibria $E_{\mathcal{C}}$ with $IC(\mathcal{C}) = \{1, \ldots, n\}$. And a partial equilibria $E_{\mathcal{C}}$ in $M = (C_1, \ldots, C_n)$ is a total equilibria iff $IC(\mathcal{C}) = \{1, \ldots, n\}$.

Generalization. The definition of partial diagnoses is a generalization of total diagnoses, e.g., the set of all partial diagnoses is a superset of all total diagnoses. Moreover it will be shown that, as a result of Lemma 3.5.2 and Lemma 3.5.3, partial diagnoses are a *conservative extension* of total diagnoses.

Lemma 3.5.2
In an MCS $M = (C_1, \ldots, C_n)$ total diagnoses as defined in Definition 2.3.1, are also partial diagnoses.

The next lemma identifies those partial diagnoses that are also total diagnoses. It states that the subset of total diagnoses from the whole set of partial diagnoses are those partial diagnoses which are defined in all contexts (i.e., $\neq \epsilon$).

Lemma 3.5.3
In an MCS $M = (C_1, \ldots, C_n)$ a partial diagnoses $DG_{\mathcal{C}} = (DG_1, \ldots, DG_n)$ is a total diagnosis iff $IC(\mathcal{C}) = \{1, \ldots, n\}$.

If a theory T_1 is extended to a theory T_2 and every theorem from T_1 is also a theorem of T_2, moreover every theorem from T_2 is already a theorem of T_1, it is called a *conservative extension*. Lemma 3.5.2 shows us that all total diagnoses are too partial diagnoses and Lemma 3.5.3 that all partial diagnoses without an ϵ are syntactically and semantically the same as total diagnoses. Therefore partial diagnoses are a conservative extension of total diagnoses. Results on total diagnoses remain valid and the algorithm and methods for partial diagnoses described in this thesis can also be applied on total diagnoses and will produce the same results as those methods solely based on total diagnoses.

Modularity. Results of the previous lemmas are now used to show the relationship between partial and total diagnoses regarding modularity. As we have already seen an MCS can be split up into partial $MCSs$. Such a partial MCS contains a subset of all contexts where the import closure of the subset is the subset itself. The relation between such partial MCS is either they are distinct, i.e., they have no contexts in common, or they have a common part which itself is again a partial MCS. Let's say we have two MCS M_1 and M_2 with a common part M_{12}, then M_{12} is a subset of M_1 and M_2 and according to Proposition 3.1.2 the import closure of M_{12} is also a subset of the import closure of M_1 and M_2. The other way around M_1 and M_2 are both extensions of M_{12}. Remind that for each of those partial MCS partial diagnoses and witnesses can be defined. Therefore a total diagnosis (which is also partial diagnosis) is iteratively built up by partial diagnoses.

The following lemma states formally the subset relation between partial diagnoses. In Section 3.6 operators will be introduces which are based on the modular design of partial diagnoses.

Lemma 3.5.4 states that every partial diagnosis DG is an extension of those partial diagnoses which are defined over a subset of the import closure of DG. In other words, for every subset of the import closure of a partial diagnosis there is at least one partial diagnosis w.r.t. to this subset.

Lemma 3.5.4
Let $DG_{\mathcal{C}} = (DG_1, \ldots, DG_n)$ be a partial diagnosis of $M = (C_1, \ldots, C_n)$. Then for every $\mathcal{C}' \subseteq \mathcal{C}$, $DG'_{\mathcal{C}'} = (DG'_1, \ldots, DG'_n)$ where $DG'_i = DG_i$ for all $i \in IC(\mathcal{C}')$ and $DG'_i = \epsilon$ otherwise, is a partial diagnosis of M.

Proof
Let $DG_{\mathcal{C}} = (DG_1, \ldots, DG_n)$ be a partial diagnosis in the MCS $M = (C_1, \ldots, C_n)$ and $\mathcal{C}' \subseteq \mathcal{C}$.

Construct a partial diagnosis $DG'_{\mathcal{C}'} = ({DG'}_1, \ldots, {DG'}_n)$ by copying $DG_{\mathcal{C}}$ and setting every $DG'_i = \epsilon$ for all $i \notin IC(\mathcal{C}')$. Now we have to show that $DG'_{\mathcal{C}'}$ is indeed a partial diagnosis w.r.t. $\mathcal{C}'$.

It is trivial that $DG'_{\mathcal{C}'}$ is defined (e.g., $DG'_i \neq \epsilon$) for all $i \in IC(\mathcal{C}')$ since $IC(\mathcal{C}') \subseteq IC(\mathcal{C})$ (Proposition 3.1.2).

It remains to show that $M[DG']$ has a partial equilibrium w.r.t. $\mathcal{C}'$. Take a partial equilibrium $E_{\mathcal{C}} = (E_1, \ldots, E_n)$ from $M[DG]$. From Proposition 3.1.4 it follows that there is also a partial equilibrium $E'_{\mathcal{C}'}$ in $M[DG]$.

Since for all $i \in IC(\mathcal{C}')$, $DG_i = DG'_i$ and therefore C_i the same in $M[DG]$ and $M[DG']$ for $i \in IC(\mathcal{C}')$, Proposition 3.1.5 shows us that the partial equilibrium $E'_{\mathcal{C}'}$ from $M[DG]$ is also a partial equilibrium in $M[DG']$ and therefore $DG'_{\mathcal{C}'}$ a partial diagnosis in $M[DG']$.

□

If the import closure of the subset $\mathcal{C}'$ is equivalent with the import closure of $\mathcal{C}$, then $DG_{\mathcal{C}} = DG'_{\mathcal{C}'}$. Recall that a total diagnosis is also a partial diagnosis. So for every total diagnosis we have a set of partial diagnoses which represents parts of the total diagnosis. With the join operator, defined below, these parts can be merged together and the result is again a part of the same total diagnosis or the total diagnosis itself.

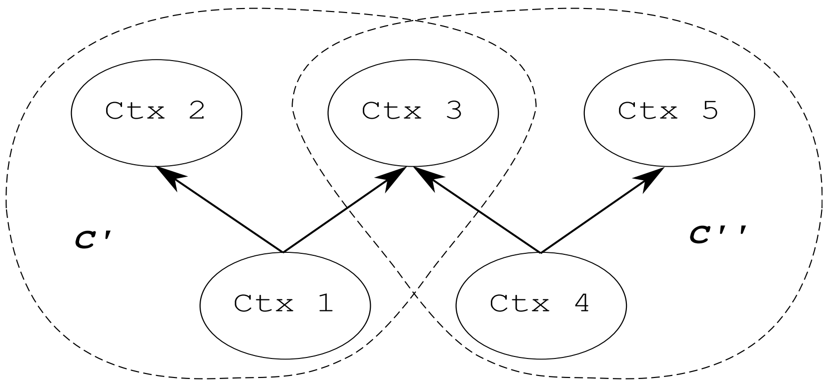


Figure 3.5.1: Example of partial $MCSs$

Example 3.5.1
We have a context C_1 which requests information from C_2 and C_3 (i.e., has bridge rules whose bodies refer to those contexts) and a context C_4 which requests information from context C_3 and C_5. Contexts C_2, C_3 and C_5 have an import closure which contains only the contexts itself. C_1 has an import closure $\mathcal{C}' = \{1, 2, 3\}$ and C_4 has an import closure $\mathcal{C}'' = \{3, 4, 5\}$. So if there is a total diagnosis we have for all those import closures listed before at least one partial diagnosis which is part of the total diagnosis. Those parts can be combined by the join operation. E.g., the union of $\mathcal{C}'$ and $\mathcal{C}''$ is the whole MCS and therefore the join of the partial diagnoses $DG_{\mathcal{C}'}$ and $DG_{\mathcal{C}''}$ a total diagnosis.

□

3.6 Combining Partial Diagnoses

In this section we define operators which combine partial diagnoses and partial diagnosis candidates. Every partial diagnosis (candidate) is defined over a subset of contexts from an MCS. If two partial diagnoses (candidates) DG', defined in $\mathcal{C}'$, and DG'', defined in $\mathcal{C}''$, are equal in the contexts of the intersection $\mathcal{C}' \cap \mathcal{C}''$ then they can be joined together. Otherwise a join between DG' and DG'' returns an empty result. If the result is not empty it is again a partial diagnosis (candidate) defined in the contexts of the union of $\mathcal{C}'$ and $\mathcal{C}''$ containing both partial diagnoses (candidates) DG' and DG''.

Definition 3.6.1 (Join of Partial Diagnoses and Partial Diagnosis Candidates)
Let $DG' = (DG'_1, \ldots, DG'_n)$ and $DG'' = (DG''_1, \ldots, DG''_n)$ be partial diagnoses or partial diagnosis candidates of an MCS $M = (C_1, \ldots, C_n)$. The result of a join $DG' \bowtie DG''$ is $DG = (DG_1, \ldots, DG_n)$, where

- *$DG_i = DG'_i = DG''_i$, if $DG'_i = DG''_i$,*
- *$DG_i = DG'_i$, if $DG'_i \neq \epsilon \wedge DG''_i = \epsilon$,*
- *$DG_i = DG''_i$, if $DG''_i \neq \epsilon \wedge DG'_i = \epsilon$*

for all $1 \leq i \leq n$.
Note that $DG' \bowtie DG''$ is void iff $DG'_i \neq \epsilon$, $DG''_i \neq \epsilon$ and $DG'_i \neq DG''_i$ for some $1 \leq i \leq n$.

Join of Witnesses and Witness Candidates. Witnesses and witness candidates are partial belief states like partial equilibria and therefore a join between two witnesses respective witness candidates is already defined in Definition 3.1.3.

Join Semantics. The following two lemmas state that the results of a join between two partial diagnosis candidates $DG'_{\mathcal{C}',\mathcal{D}'}$ and $DG''_{\mathcal{C}'',\mathcal{D}''}$ and their witness candidates $W'_{\mathcal{C}',\mathcal{D}'}$ and $W''_{\mathcal{C}'',\mathcal{D}''}$ is again a partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ and a corresponding witness candidate $W_{\mathcal{C},\mathcal{D}}$ if both results exist. Those relations hold also for partial diagnoses instead of partial diagnosis candidates and their witnesses instead of witness candidates.

Lemma 3.6.1
Let $M = (C_1, \ldots, C_n)$ be an MCS, $DG'_{\mathcal{C}',\mathcal{D}'} = (DG'_1, \ldots, DG'_n)$ a partial diagnosis candidate in M with the witness candidate $W'_{\mathcal{C}',\mathcal{D}'} = (W'_1, \ldots, W'_n)$ and $DG''_{\mathcal{C}'',\mathcal{D}''} = (DG''_1, \ldots, DG''_n)$ a partial diagnosis candidate in M with the witness candidate $W''_{\mathcal{C}'',\mathcal{D}''} = (W''_1, \ldots, W''_n)$. Then $DG_{\mathcal{C},\mathcal{D}} = DG'_{\mathcal{C}',\mathcal{D}'} \bowtie DG''_{\mathcal{C}'',\mathcal{D}''}$ is a partial diagnosis candidate with the witness candidate $W_{\mathcal{C},\mathcal{D}} = W'_{\mathcal{C}',\mathcal{D}'} \bowtie W''_{\mathcal{C}'',\mathcal{D}''}$ with $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$ and $\mathcal{D} = \mathcal{D}' \cup \mathcal{D}''$ if $DG_{\mathcal{C},\mathcal{D}}$ and $W_{\mathcal{C},\mathcal{D}}$ exist.

Proof
According to Definition 3.6.1 the result of a join between two partial diagnosis candidates is defined in those contexts where at least one of the partial diagnosis candidates is defined. DG' is defined in all contexts of $\mathcal{C}'$ with $DG'_i \in \{(D_i, A_i) | D_i, A_i \subseteq br_i\}$ *iff* $C_i \in \mathcal{C}'$ and $DG'_i = \epsilon$ otherwise and DG'' is defined in all contexts of $\mathcal{C}''$ with $DG''_i \in \{(D_i, A_i) | D_i, A_i \subseteq br_i\}$ *iff* $C_i \in \mathcal{C}''$ and $DG''_i = \epsilon$ otherwise. Therefore DG is defined in all contexts of $\mathcal{C}' \cup \mathcal{C}''$ with $DG_i \in \{(D_i, A_i) | D_i, A_i \subseteq br_i\}$ *iff* $C_i \in (\mathcal{C}' \cup \mathcal{C}'')$ and $DG_i = \epsilon$ *iff* $C_i \notin (\mathcal{C}' \cup \mathcal{C}'')$. Now we show that if $W_{\mathcal{C},\mathcal{D}}$ exists as a result of a join between $W'_{\mathcal{C}',\mathcal{D}'}$ and $W''_{\mathcal{C}'',\mathcal{D}''}$ then $W_{\mathcal{C},\mathcal{D}}$ is a witness candidate of DG and therefore DG a partial diagnosis candidate.

Since W' and W'' are belief states it is clear that W is a belief state too. Recalling Definition 3.4.2, $W_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D_i \cup heads(A_i), W)\})$ if $C_i \in \mathcal{C}$, $W_i \in \mathbf{BS}_i$ if $C_i \in \mathcal{D}$ and $W_i = \epsilon$ otherwise. Since $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$ either $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D'_i \cup heads(A'_i), W')\})$, $W''_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D''_i \cup heads(A''_i), W'')\})$ or both of them are. Since $DG_i = DG'_i$ respectively $DG_i = DG''_i$ for $C_i \in \mathcal{C}$ we can substitute to $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus$

$\mathbf{D_i} \cup heads(\mathbf{A_i}), W')\})$ respectively $W''_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus \mathbf{D_i} \cup heads(\mathbf{A_i}), W'')\})$. $app(br_i \setminus D_i \cup heads(A_i), W')$ is only defined if $W'_k \neq \epsilon$ for all $k \in In(C_i)$. If this is the case also $W_k \neq \epsilon$ and moreover $W_k = W'_k$ for all $k \in IC(C_i)$ and due to Proposition 3.1.3, $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D_i \cup heads(A_i), \mathbf{W})\})$ respectively $W''_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D_i \cup heads(A_i), \mathbf{W})\})$. Therefore $W_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D_i \cup heads(A_i), W)\})$ for all $i \in (\mathcal{C}' \cup \mathcal{C}'')$.

Now it is left to show that $W_{\mathcal{C},\mathcal{D}}$ has guessed belief sets for all $C_i \in \mathcal{D}$. C_i is in $\mathcal{D}$ if $C_i \in \mathcal{D}'$ or $C_i \in \mathcal{D}''$. If $C_i \in \mathcal{D}'$ then $W'_i \in \mathbf{BS}_i$ and therefore also $W_i \in \mathbf{BS}_i$. The same holds for $C_i \in \mathcal{D}''$.

It is clear that $W_i = \epsilon$ *iff* $W'_i = \epsilon$ and $W''_i = \epsilon$. This is the case for all $C_i \notin (\mathcal{C}' \cup \mathcal{D}' \cup \mathcal{C}'' \cup \mathcal{D}'')$ which is $C_i \notin (\mathcal{C} \cup \mathcal{D})$. Therefore is $W_{\mathcal{C},\mathcal{D}}$ a witness candidate of $DG_{\mathcal{C},\mathcal{D}}$.

$\square$

Lemma 3.6.2
Let $M = (C_1, \ldots, C_n)$ be an MCS, $DG'_{\mathcal{C}'} = (DG'_1, \ldots, DG'_n)$ a partial diagnosis in M with the witness $W'_{\mathcal{C}'} = (W'_1, \ldots, W'_n)$ and $DG''_{\mathcal{C}''} = (DG''_1, \ldots, DG''_n)$ a partial diagnosis in M with the witness $W''_{\mathcal{C}''} = (W''_1, \ldots, W''_n)$. If $DG_{\mathcal{C}} = DG'_{\mathcal{C}'} \bowtie DG''_{\mathcal{C}''}$ and $W_{\mathcal{C}} = W'_{\mathcal{C}'} \bowtie W''_{\mathcal{C}''}$ exist $DG_{\mathcal{C}}$ is a partial diagnosis with the witness $W_{\mathcal{C}}$ and $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$.

Proof
According to Definition 3.6.1 the result of a join between two partial diagnoses is defined in those contexts where at least one of the partial diagnoses is defined. DG' is defined in $IC(\mathcal{C}')$ and DG'' in $IC(\mathcal{C}'')$. Therefore DG is defined in $IC(\mathcal{C}') \cup IC(\mathcal{C}'')$. Using Proposition 3.1.1, $IC(\mathcal{C}') \cup IC(\mathcal{C}'') = IC(\mathcal{C}' \cup \mathcal{C}'')$.

It is left to show that $W_{\mathcal{C}} = W'_{\mathcal{C}'} \bowtie W''_{\mathcal{C}''}$ is a witness for DG. Since W' and W'' are belief states it is clear that W is a belief state too. Recalling Definition 3.1.2 and 3.3.1, $W_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D_i \cup heads(A_i), W)\})$ if $i \in IC(\mathcal{C})$ and $W_i = \epsilon$ otherwise.

$i \in IC(\mathcal{C}' \cup \mathcal{C}'')$ if $i \in IC(\mathcal{C}')$ or $i \in IC(\mathcal{C}'')$. If $i \in IC(\mathcal{C}')$ then $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus D'_i \cup heads(A'_i), W')\})$. Since $DG_i = DG'_i$, because $DG'_i \neq \epsilon$ if $i \in IC(\mathcal{C}')$, we can substitute to $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus \mathbf{D_i} \cup heads(\mathbf{A_i}), W')\})$. $app(br_i \setminus D_i \cup heads(A_i), W')$ is due to Proposition 3.1.3 the same as $app(br_i \setminus D_i \cup heads(A_i), \mathbf{W})$ if $W_k = W'_k$ for all $k \in In(i)$. Since $i \in IC(\mathcal{C}')$ and $In(i) \subseteq IC(C_i)$ for all $k \in In(i)$, $W'_k \neq \epsilon$ and therefore, since $W' \bowtie W''$ exists, $W_k = W'_k$ for all $k \in In(i)$. Therefore we have $W'_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i \setminus \mathbf{D_i} \cup heads(\mathbf{A_i}), W)\})$. The proof for $i \in IC(\mathcal{C}'')$ works the same way.

$i \notin IC(\mathcal{C}' \cup \mathcal{C}'')$ if $i \notin IC(\mathcal{C}')$ and $i \notin IC(\mathcal{C}'')$. Therefore $W'_i = \epsilon$, $W''_i = \epsilon$ and therefore $W_i = \epsilon$. So $W_{\mathcal{C}}$ is a witness candidate of $DG_{\mathcal{C}}$.

$\square$

Example 3.6.1
Let's take the MCS from Figure 3.6.1 and assume that the context C_1 wants to know the partial diagnosis candidates and its witnesses from C_2 and from C_3 so it can join those results and then extend them with its own context. As we will see later in the algorithm to

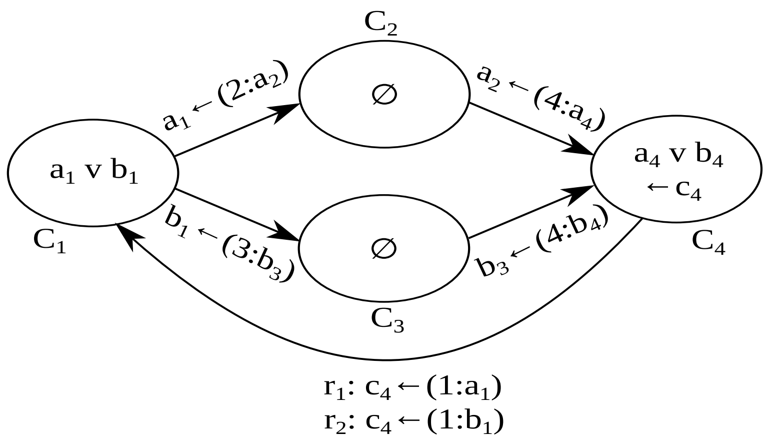


Figure 3.6.1: Join example of partial diagnosis candidates and witness candidates

solve the cycle (the request chain from C_1 to C_2/C_3, C_4 and again to C_1) the belief set in C_1 will be guessed and therefore C_2 and C_3 will return candidates to C_1. From C_2 we get, amongst others, the partial diagnosis candidate $DG'_{\{C_2,C_4\},\{C_1\}} = (\epsilon, (\emptyset, \emptyset), \epsilon, (\{r_1\}, \emptyset))$ with one witness candidate being $W'_{\{C_2,C_4\},\{C_1\}} = (\{a_1\}, \{a_2\}, \epsilon, \{a_4\})$ and from C_3 the partial diagnosis candidate $DG''_{\{C_3,C_4\},\{C_1\}} = (\epsilon, \epsilon, (\emptyset, \emptyset), (\{r_1\}, \emptyset))$ with one witness candidate being $W''_{\{C_3,C_4\},\{C_1\}} = (\{a_1\}, \epsilon, \emptyset, \{a_4\})$.

We can now join the two partial diagnosis candidates because they are the same in those contexts where both are defined so the result is $DG_{\{C_2,C_3,C_4\},\{C_1\}} = (\epsilon, (\emptyset, \emptyset), (\emptyset, \emptyset), (\{r_1\}, \emptyset))$. In this example we can also join the two witness candidates, because the guess in C_1 and the belief set in C_4 is the same on both witness candidates, and get therefore a witness candidate for the diagnosis candidate joined together before. So $W'_{\{C_2,C_4\},\{C_1\}} \bowtie W''_{\{C_3,C_4\},\{C_1\}} = W_{\{C_2,C_3,C_4\},\{C_1\}} = (\{a_1\}, \{a_2\}, \emptyset, \{a_4\})$. Note that there are many more witness candidates for DG' and DG'' which can not be joined, because, e.g., the guess on context C_1 is the same but the belief set of C_4 is different and therefore only the valid witness candidates remain.

$\square$

Every partial diagnosis has a set of witnesses. Now we define a join between two of those witness sets. Such a join is, like a cross product, a join between each witness of one set with each witness of the other set and results in a set of containing all the witness join results.

Definition 3.6.2
Let $DG'_{C'}$ and $DG''_{C''}$ be two partial diagnoses. Further $\mathcal{W}' = wit(DG'_{C'})$ and $\mathcal{W}'' = wit(DG''_{C''})$ be two sets containing the witnesses of $DG'_{C'}$ and $DG'_{C'}$ respectively. Then the join $\mathcal{W}' \bowtie \mathcal{W}''$ is defined as the set $\{\, W'_{C'} \bowtie W''_{C''} \mid W'_{C'} \in \mathcal{W}', W''_{C''} \in \mathcal{W}''\}$.

If two sets of witnesses of partial diagnoses DG' and DG'' respectively are joined together the resulting set contains all witnesses of the partial diagnosis resulting from a join between DG' and DG''.

Lemma 3.6.3
Let $DG'_{\mathcal{C}'}$ and $DG''_{\mathcal{C}''}$ be two partial diagnoses, $\mathcal{W}' = wit(DG'_{\mathcal{C}'})$ as well as $\mathcal{W}'' = wit(DG''_{\mathcal{C}''})$ two sets with all existing witnesses of $DG'_{\mathcal{C}'}$ respectively $DG''_{\mathcal{C}''}$ and $DG_{\mathcal{C}'\cup\mathcal{C}''} = DG'_{\mathcal{C}'} \bowtie DG''_{\mathcal{C}''}$. Then the join $\mathcal{W}' \bowtie \mathcal{W}''$ returns all existing witnesses of the partial diagnosis $DG_{\mathcal{C}'\cup\mathcal{C}''}$.

Proof
From Lemma 3.6.2 we know that all witnesses $W \in \mathcal{W}' \bowtie \mathcal{W}''$ are witnesses of the partial diagnosis $DG_{\mathcal{C}'\cup\mathcal{C}''}$. So it is left to show that this set of witnesses is complete, e.g., it contains all existing witnesses of $DG_{\mathcal{C}'\cup\mathcal{C}''}$.

Let's say $W_{\mathcal{C}'\cup\mathcal{C}''} = (W_1, \dots, W_n)$ is an arbitrary witness from $wit(DG_{\mathcal{C}'\cup\mathcal{C}''})$. Based on this witness we can construct two partial belief states $W'_{\mathcal{C}'} = (W'_1, \dots, W'_n)$ and $W''_{\mathcal{C}''} = (W''_1, \dots, W''_n)$ and show (a) that those partial belief states are witnesses of the partial diagnoses $DG'_{\mathcal{C}'}$ respectively $DG''_{\mathcal{C}''}$ and (b) a join between $W'_{\mathcal{C}'}$ and $W''_{\mathcal{C}''}$ results in $W_{\mathcal{C}'\cup\mathcal{C}''}$.

$W_{\mathcal{C}'\cup\mathcal{C}''}$ is defined over the contexts of the import closure $IC(\mathcal{C}' \cup \mathcal{C}'')$. Proposition 3.1.1 tells us that $IC(\mathcal{C}' \cup \mathcal{C}'') = IC(\mathcal{C}') \cup IC(\mathcal{C}'')$. So we construct $W'_{\mathcal{C}'}$ by setting $W'_i = W_i$ if $i \in IC(\mathcal{C}')$ and $W''_{\mathcal{C}''}$ by setting $W''_i = W_i$ if $i \in IC(\mathcal{C}'')$. All other W'_i and W''_i are set to ϵ.

Recalling Definition 3.1.3, it is trivial that a join between $W'_{\mathcal{C}'}$ and $W''_{\mathcal{C}''}$ results in the witness $W_{\mathcal{C}'\cup\mathcal{C}''}$.

$W_{\mathcal{C}'\cup\mathcal{C}''}$ is a partial equilibrium and according to Proposition 3.1.4 we know that $W'_{\mathcal{C}'}$ and $W''_{\mathcal{C}''}$ are also partial equilibria in $M[DG_{\mathcal{C}'\cup\mathcal{C}''}]$. Since $DG_i = DG'_i$ for $i \in IC(\mathcal{C}')$ and $DG_i = DG''_i$ for $i \in IC(\mathcal{C}'')$ along with Proposition 3.1.5 $W'_{\mathcal{C}'}$ and $W''_{\mathcal{C}''}$ are also partial equilibria in $M[DG'_{\mathcal{C}'}]$ and $M[DG''_{\mathcal{C}''}]$ respectively and moreover witnesses of $DG'_{\mathcal{C}'}$ and $DG''_{\mathcal{C}''}$ respectively.

□

Example 3.6.2
Let's recall Example 3.6.1 where the partial diagnosis candidate $DG'_{\{C_2,C_4\},\{C_1\}}$ has the witness candidates $(\{a_1\}, \{a_2\}, \epsilon, \{a_4\})$ and $(\{a_1\}, \emptyset, \epsilon, \{b_4\})$ and the partial diagnosis candidate $DG''_{\{C_3,C_4\},\{C_1\}}$ has the witness candidates $(\{a_1\}, \epsilon, \emptyset, \{a_4\})$ and $(\{a_1\}, \epsilon, \{b_3\}, \{b_4\})$.

Now to get the witness candidates for the joined partial diagnosis candidate $DG_{\{C_2,C_3,C_4\},\{C_1\}}$ we just have to join the two sets of witness candidates which results in the two witness candidates $(\{a_1\}, \{a_2\}, \emptyset, \{a_4\})$ and $(\{a_1\}, \emptyset, \{b_3\}, \{a_4\})$ which are indeed the only witness candidates for the mentioned partial diagnosis candidate.

□

CHAPTER 4

Algorithm

4.1 Overview

The algorithm, developed in this thesis, is referred to as “Distributed Multi-Context Systems Diagnoses Finder” or DMCS-DF for short.

Multi-Context Systems targeted by the algorithm are realized as a network between nodes. These nodes are hosts for one or more contexts with their knowledge bases and bridge rules. Note that every context is only aware of those bridge rules which add information to his context, viz. which heads refer to the context. Every node in such a network runs an instance of the algorithm for each context. Instances gather information from other contexts on the same node and on other nodes by sending a request over the network to the addressed context. Since it does not matter if the context is on the same or another node the concept of nodes will be omitted in the definition of the algorithm and the implementation.

Every instance is capable of processing the hosted knowledge base in their individual knowledge representation formalism. Therefore every instance must only be specialized for the local context and receives the information from other contexts in a normalized form, defined by the MCS concept and therefore does not need to know or deal with the specific knowledge representation formalisms of other contexts.

The purpose of this algorithm is to find partial diagnoses and their witnesses with respect to *one* specific context. Therefore it is intended that a user sends a request to the context the user is interested in. The algorithm then determines which contexts are necessary for the calculation and requests the needed information from them. Those requests are performed in a depth first search manner.

A problem arising from this method is that the algorithm can get stuck in a cycle of requests between contexts depending on each other. Therefore every instance needs information about the preceding calling contexts and has to check this calling history for such a cycle. If a cycle is detected no more requests are send and instead of requesting needed information from further contexts and calculating the local belief sets, possible local belief sets are guessed and returned to the calling instance. Based on these guesses further calculations are done and those results send back in the calling chain until the instance is reached on which the guesses have been created. Then the guesses are validated and false guesses are discarded. Finally the first called instance returns all possible partial diagnoses and in addition all corresponding witnesses to the user.

4.2 Details and Pseudocode

Main Procedure. The main procedure is the entry point if a request is send to a context. The procedure returns a set of pairs where the first element of the pair is a partial diagnosis candidate and the second a partial belief state. This procedure is called for every context in the import closure of the context which the user is interested in. The returned results of the procedure are proper partial diagnoses (not candidates) if the context is not part of a cycle or it is the first context in a cycle which is called. In all other cases the results are partial diagnosis candidates which will be validated if the results have returned to the first called context in the cycle. All results are complete, meaning that they contain all existing partial diagnoses (candidates) and all of its witnesses.

For the calculations in this part of the algorithm it does not matter if the results returned from the import neighborhood are proper partial diagnoses or partial diagnosis candidates. The join operations are similar on partial diagnoses and partial diagnosis candidates as well as on partial witnesses and witness candidates. The main procedure treats every candidate like a proper partial diagnosis (or witness). The validation of the candidates is done in the guessDiagnosesAndSolve procedure.

The procedure checks first if this context has already been called earlier and is therefore the last context of a cyclic calling chain (Line 2). A list of already called contexts is added by the calling instance to the request so it suffices to check if this list contains the local context. In case a cycle is detected the calling chain is interrupted, i.e., no contexts of the import neighborhood will be called and instead the local belief sets guessed. This is done by the procedure guessBeliefSets. Those guesses are not further processed and send back to the caller.

If no cycle is detected, from every context of the import neighborhood a complete set of partial diagnoses (candidates) w.r.t. those contexts and the corresponding partial belief states are requested (Line 8). Those requests are send iteratively and not in parallel for the following reason: the local context needs the partial results from all contexts in his import neighborhood. If the import closure of one of these contexts contains another context from the import neighborhood a call to this context is redundant and therefore skipped.

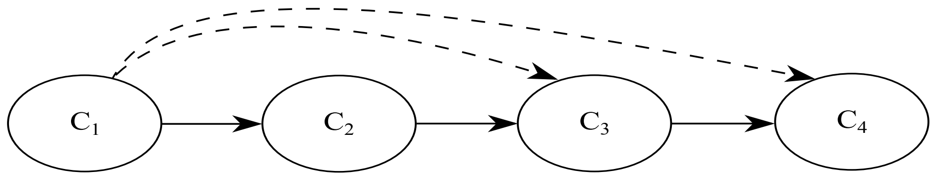

$$\begin{tikzcd} C_1 \arrow[r] \arrow[rr, dashed, bend left] \arrow[rrr, dashed, bend left] & C_2 \arrow[r] & C_3 \arrow[r] & C_4 \end{tikzcd}$$

Figure 4.2.1: Example of skipped requests

Example 4.2.1
Let's take the MCS from Figure 4.2.1 where C_1 is a context with $\{2, 3, 4\}$ being its import neighborhood and the import closure of C_2 being $\{3, 4\}$. If C_1 requests the results from C_2, partial results defined in the contexts C_2, C_3 and C_4 are returned. They are already complete and it is not necessary for C_1 to request results from C_3 and C_4 although they are in the import neighborhood of C_1.

□

```
Algorithm 4.2.1: DMCS-DF($hist$) at $C_t = (L_t, kb_t, br_t)$
  Input: $hist$: visited contexts.
  Output: $\mathcal{S} = \{(DG'_{\{C_t\}}, W'_{\{C_t\}}), (DG''_{\{C_t\}}, W''_{\{C_t\}}), \ldots\}$: set of pairs with the first
          element a partial diagnosis candidate w.r.t. $C_t$ and the second a corresponding
          witness candidate.
1  $\mathcal{S} := \emptyset$;
2  if $t \in hist$ then
3      $\mathcal{S}$ = guessBeliefSets($C_t$)
4  else
5      $\mathcal{T} := \{((\epsilon, \ldots, \epsilon), (\epsilon, \ldots, \epsilon))\}$;
6      foreach $i \in In(t)$ do
7          if for some $(DG, W) \in \mathcal{T}$, $W_i = \epsilon$ then
8              $\mathcal{T}' := C_i$.DMCS-DF($hist \cup \{t\}$);
9              $\mathcal{T} := \bigcup_{(DG',W')\in\mathcal{T},(DG'',W'')\in\mathcal{T}'} (DG' \bowtie DG'', W' \bowtie W'')$;
10         end
11     end
12     foreach $(DG, W) \in \mathcal{T}$ do
13         $\mathcal{S} := \mathcal{S} \cup$ guessDiagnosesAndSolve($C_t, (DG, W)$);
14     end
15 end
16 return $\mathcal{S}$;
```

The results of these requests, the partial diagnoses (candidates), and their witnesses (candidates) are then joined together (Line 9). The resulting partial diagnoses are at least defined in the import neighborhood of the local context. Based on this, the procedure guessBeliefSets calculates for every result so far the partial diagnoses and their witnesses with respect to the local context (Line 13). Those results are then sent back to the calling instance.

Guessing Local Belief Sets. This part of the algorithm takes place when a cycle is detected. Therefore no information can be requested from the contexts of the import neighborhood otherwise the algorithm would get stuck in an infinite loop. Instead all possible local belief sets will be guessed (Line 2) and returned. Note that this instance makes no guess about the local partial diagnoses since this has already been done by the instance (running at this context) which handles the first request coming to this context.

The guessed belief sets are returned as witness candidates for a partial diagnosis candidate which is ϵ in all the contexts of the MCS (Line 3). This partial diagnosis candidate will then be extended by the instances from the calling chain.

Example 4.2.2
Given an MCS with three contexts C_1, C_2 and C_3 and let's assume a request is sent from C_1 to C_2, then to C_3 and from there again to C_1. C_1 gets the request and asserts that C_1 is already in the request chain and therefore makes a guess about the local belief set and returns it with an undefined partial diagnosis. E.g., if the belief sets are all combinations of the elements a and b the request to C_1 would return $((\epsilon,\epsilon,\epsilon),(\emptyset,\epsilon,\epsilon))$, $((\epsilon,\epsilon,\epsilon),(\{a\},\epsilon,\epsilon))$, $((\epsilon,\epsilon,\epsilon),(\{b\},\epsilon,\epsilon))$, and $((\epsilon,\epsilon,\epsilon),(\{a,b\},\epsilon,\epsilon))$.

□

Algorithm 4.2.2: guessBeliefSets(C_t)

```
Input: $C_t = (L_t, kb_t, br_t)$: context whose partial belief sets are to be guessed.
Output: $\mathcal{T} = \{((\epsilon,\dots,\epsilon), W'), ((\epsilon,\dots,\epsilon), W''), \dots\}$: set of pairs with the first element
        a partial diagnosis candidate and the second a witness candidate.
1 $\mathcal{T} = \emptyset$;
2 foreach $B_t \in \mathbf{BS}_t$ do
3     $\mathcal{T} = \mathcal{T} \cup \{((\epsilon,\dots,\epsilon),(\epsilon,\dots,B_t,\dots,\epsilon))\}$;
4 end
5 return $\mathcal{T}$;
```

Guessing and Solving Partial Diagnoses. This procedure extends the given partial diagnosis candidates with the part corresponding to the local context and further calculates the local belief sets based on the extended partial diagnosis candidates. Moreover it checks if those calculated belief sets equal to existing guessed belief sets.

All possible local partial diagnoses are guessed by taking the cross product over the power set of the local bridge rules (Line 2). This yields all possible combinations of the bridge rules from the local context which are to be removed or applied unconditionally. Then the given partial diagnosis candidate is combined with each guess. Based on the newly created partial diagnosis candidates the local belief sets are calculated (Line 4). Now it has to be distinguished between the case that the given witness candidate is defined at the local context (Lines 8-10) or it is not defined there (Line 6). The first case means that this defined belief set is a guess and if the guess is also in the calculated belief sets the guess is validated. Therefore the partial diagnosis candidate with this witness candidate is returned. If the guess is not part of the calculated belief sets, the partial diagnosis candidate is dropped. In the second case the given witness candidate is combined with all calculated local belief sets and they are returned with the newly created partial diagnosis candidate.

Example 4.2.3
Let's take the MCS from Example 4.2.2 and assume that C_1 has a knowledge base $kb_1 = \{a \vee b\}$, a bridge rule r_1 and, for the sake of simplicity, that $\mathbf{ACC}_1(kb_1 \cup \{head(r) \mid r \in app(br_1 \setminus D_1 \cup heads(A_1), W)\})$ under all witness candidates and under all partial diagnosis candidates returns $\{a, b\}$.

Then all witness candidates which contain $\{a, b\}$ or $\emptyset$ as a belief set for C_1 are dropped and all the other ones with $\{a\}$ and $\{b\}$ are kept because they are accepted locally. This is the point where the guesses are checked. All those belief states which are now left are witnesses of the partial diagnoses which have been returned to C_1 combined with the four possible local diagnoses $(\emptyset, \emptyset)$, $(\{r_1\}, \emptyset)$, $(\emptyset, \{r_1\})$ and $(\{r_1\}, \{r_1\})$. Note that this is only true under our assumption that all those partial diagnoses lead to the same set of accepted local belief sets.

□

Algorithm 4.2.3: guessDiagnosesAndSolve$(C_t, T = (DG, W))$

```
Input: $C_t = (L_t, kb_t, br_t)$: context whose partial diagnoses will be guessed and whose
       belief sets will be calculated; $T$: a partial diagnosis candidate $DG$ with a
       corresponding witness candidate $W$.
Output: $\mathcal{S} = \{(DG'_{\{C_k\}}, W'_{\{C_t\}}), (DG''_{\{C_k\}}, W''_{\{C_t\}}), \dots\}$: set of pairs with the first
        element a partial diagnosis candidate w.r.t. $C_t$ and the second a corresponding
        witness candidate.
1  $\mathcal{S} = \emptyset$;
2  $\mathcal{G}_t = \{(D_t, A_t) \mid D_t, A_t \subseteq br_t\}$;
3  foreach $G_t \in \mathcal{G}_t$ do
4      $\mathcal{B}_t = \mathbf{ACC}_t(kb_t \cup \{head(r) \mid r \in app(br_t \setminus D_t \cup heads(A_t), W)\})$;
5      if $W_t = \epsilon$ then
6          $\mathcal{S} = \mathcal{S} \cup \{(DG_1, \dots, G_t, \dots, DG_n), (W_1, \dots, W'_t, \dots, W_n) \mid W'_t \in \mathcal{B}_t\}$;
7      else
8          if $W_t \in \mathcal{B}_t$ then
9              $\mathcal{S} = \mathcal{S} \cup \{((DG_1, \dots, G_t, \dots, DG_n), W)\}$;
10         end
11     end
12 end
13 return $\mathcal{S}$;
```

4.3 Soundness and Completeness

To show the soundness and completeness of the algorithm we first have to introduce the concepts of a limited import closure and a guessing edge. Each request to a context in an MCS has as a parameter a history which is a set of already visited contexts. So depending on this history, as a result of such an request partial diagnosis candidates and witness candidates are returned which are well defined in some contexts and are guesses in some other contexts. They are only defined in those contexts which are not in the history and can be reached by further request without passing a context from the history. We call a set of those contexts *limited import closure*. All contexts which are in the import neighborhood of a context of the limited import closure and are also in the history return guesses to prevent an infinite request cycle. This set of contexts is called *guessing edge*.

In the following definitions, we generalize the history with an arbitrary set of contexts $\mathcal{C}$. The limited import closure is defined iteratively beginning with the import neighborhood of the context itself without the contexts from $\mathcal{C}$. Then on each step we extend the limited import closure with the import neighborhood of all contexts which are already in the limited import closure again without the contexts from $\mathcal{C}$. Since an MCS is always finite and in no step contexts are removed from the limited import closure it will converge at some point.

Definition 4.3.1 (Limited Import Closure)
Let $M = (C_1, \dots, C_n)$ be an MCS, C_t a context in M and $\mathcal{C} \subseteq \{1, \dots, n\}$. Then a $\mathcal{C}$-limited import closure $LIC(C_t, \mathcal{C})$ is defined as $LIC(C_t, \mathcal{C}) = LIC^{\infty}(C_t, \mathcal{C})$ with:

- $LIC^0(C_t, \mathcal{C}) = \{t\} \setminus \mathcal{C}$ *and*
- $LIC^{j+1}(C_t, \mathcal{C}) = LIC^j(C_t, \mathcal{C}) \cup \bigcup_{i \in LIC^j(C_t, \mathcal{C})} (In(i) \setminus \mathcal{C})$ *for* $j \geq 0$.

Note that

- $t \in \mathcal{C}$ implies $LIC(C_t, \mathcal{C}) = \emptyset$ and
- $\mathcal{C} = \emptyset$ implies $LIC(C_t, \mathcal{C}) = IC(C_t)$.

For a context to be in the guessing edge it must be in $\mathcal{C}$ and the context must be in the import neighborhood of a context from the limited import closure. An exception is a request from a context to itself. In this case the limited import closure is empty because such a request returns just a guess and nothing else. Therefore if the context itself is in $\mathcal{C}$ it is also in the guessing edge.

Definition 4.3.2 (Guessing Edge)
Let $M = (C_1, \dots, C_n)$ be an MCS and let $LIC(C_t, \mathcal{C})$ be a $\mathcal{C}$-limited import closure w.r.t. M. Then the Guessing Edge of $LIC(C_t, \mathcal{C})$ denoted $GE(C_t, \mathcal{C})$ is a set S with $i \in GE(C_t, \mathcal{C})$ iff:

a) $i \in \mathcal{C}$ *and*

b) $i \in \{t\} \cup \bigcup_{s \in LIC(C_t, \mathcal{C})} In(s)$.

Example 4.3.1
Let's take the example from Figure 4.3.1 and assume that a request from C_1 has been sent to C_2 which triggers a further request to C_3 with a history set $\{1, 2\}$. Now C_3 will answer this request with information about its import closure, which is $\{3, 4, 2, 5\}$. But since C_2 is already in the history the request will only return a guess from C_2 and nothing from C_5. This is expressed by the $\{1, 2\}$-limited import closure of C_3 being $LIC(C_3, \{1, 2\}) = \{4\}$ and its guessing edge being $GE(C_3, \{1, 2\}) = \{2\}$.

□

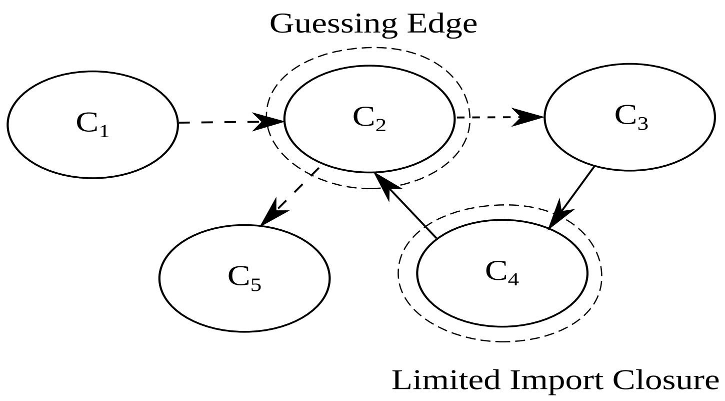


Figure 4.3.1: Example of a limited import closure and a guessing edge

Lemma 4.3.1
Let $M = (C_1, \ldots, C_n)$ be an MCS, C_t a context in M and $\mathcal{C} \subseteq \{1, \ldots, n\}$ with $t \notin \mathcal{C}$. Then the following equation holds:

$$LIC(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, \mathcal{C} \cup \{t\}). \tag{4.3.1}$$

Proof
According to Definition 4.3.1, (4.3.1) is equal to

$$LIC^{\infty}(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC^{\infty}(C_i, \mathcal{C} \cup \{t\}).$$

We now show by induction that (4.3.1) is true for each step in the iterative calculation of the limited import closures. More precisely it is equal if the calculation of the limited import closure on the left side of the equation is one step ahead of the calculation of the limited import closure on the right side. Therefore we show:

$$LIC^{x}(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}). \tag{4.3.2}$$

Induction Base. For $x = 1$ we have[1]

$$LIC^{1}(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC^{0}(C_i, \mathcal{C} \cup \{t\}) \tag{4.3.3}$$

which, due to Definition 4.3.1, is equivalent to

$$LIC^{0}(C_t, \mathcal{C}) \cup \bigcup_{i \in LIC^{0}(C_t, \mathcal{C})} (In(i) \setminus \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC^{0}(C_i, \mathcal{C} \cup \{t\}). \tag{4.3.4}$$

[1]Since $LIC^{-1}(C_i, \mathcal{C} \cup \{t\})$ is not defined also (4.3.2) is not defined for $x = 0$. Therefore our base case is $x = 1$.

Since $t \notin \mathcal{C}$ we know that $LIC^0(C_t, \mathcal{C}) = \{t\} \setminus \mathcal{C} = \{t\}$. Moreover $LIC^0(C_i, \mathcal{C} \cup \{t\}) = \{i\} \setminus (\mathcal{C} \cup \{t\})$ and therefore (4.3.4) is

$$\{t\} \cup \bigcup_{i \in \{t\}} (In(i) \setminus \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} (\{i\} \setminus (\mathcal{C} \cup \{t\}))$$

and further

$$\{t\} \cup (In(t) \setminus \mathcal{C}) = \{t\} \cup (In(t) \setminus (\mathcal{C} \cup \{t\})).$$

Since the set on the right side is always containing t we do not have to subtract t in each step of the union. Therefore we can reduce it to:

$$\{t\} \cup (In(t) \setminus \mathcal{C}) = \{t\} \cup (In(t) \setminus \mathcal{C}). \tag{4.3.5}$$

Induction Step. We now show that under the assumption that (4.3.2) holds, it also holds that

$$\underbrace{LIC^{x+1}(C_t, \mathcal{C})}_{L} = \underbrace{\{t\} \cup \bigcup_{i \in In(t)} LIC^x(C_i, \mathcal{C} \cup \{t\})}_{R}.$$

Now we replace $LIC^{x+1}(C_t, \mathcal{C})$ using Definition 4.3.1 and obtain

$$L = LIC^x(C_t, \mathcal{C}) \cup \bigcup_{i \in LIC^x(C_t, \mathcal{C})} (In(i) \setminus \mathcal{C})$$

As already seen in the induction base $(In(t) \setminus \mathcal{C}) \subseteq LIC^1(C_t, \mathcal{C})$ and $LIC^1(C_t, \mathcal{C}) \subseteq LIC^x(C_t, \mathcal{C})$ so it does not make any difference if $t \in LIC^x(C_t, \mathcal{C})$ or not and we further obtain

$$L = \underbrace{LIC^x(C_t, \mathcal{C})}_{L'} \cup \underbrace{\bigcup_{i \in (LIC^x(C_t, \mathcal{C}) \setminus \{t\})} (In(i) \setminus \mathcal{C})}_{L''}. \tag{4.3.6}$$

For the right side R we can replace $LIC^x(C_i, \mathcal{C} \cup \{t\})$ using Definition 4.3.1 and obtain

$$R = \{t\} \cup \bigcup_{i \in In(t)} \left(LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}) \cup \bigcup_{k \in LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})} (In(k) \setminus (\mathcal{C} \cup \{t\})) \right).$$

Since the union of sets is associative and commutative we obtain

$$R = \underbrace{\{t\} \cup \bigcup_{i \in In(t)} \left(LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}) \right) \cup}_{R'}$$

$$\underbrace{\bigcup_{i \in In(t)} \left(\bigcup_{k \in LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})} (In(k) \setminus (\mathcal{C} \cup \{t\})) \right)}_{R''}.$$

Since (4.3.2) holds we know that $L' = R'$. Now it is left to show that $L'' = R''$. In R'' we use the index i only to define the index set of the second union and it is not used in the sets itself. Therefore we can combine both unions to the following:

$$R'' = \underbrace{\bigcup_{k \in \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})} \left(In(k) \setminus (\mathcal{C} \cup \{t\})\right)}_{R'''}.$$

Since $t \in R$ we do not need to remove t in every element of the union and we can write

$$R'' = \underbrace{\bigcup_{k \in \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})} \left(In(k) \setminus \mathcal{C}\right)}_{R'''}. \tag{4.3.7}$$

Now we can match L'' in (4.3.6), with R'' in (4.3.7) and it is left to show that the index set of the unions coincide, i.e., that

$$LIC^x(C_t, \mathcal{C}) \setminus \{t\} = \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}).$$

Using (4.3.2) we can replace

$$(\{t\} \cup \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})) \setminus \{t\} = \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\})$$

which can be further reduced to

$$\bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}) = \bigcup_{i \in In(t)} LIC^{x-1}(C_i, \mathcal{C} \cup \{t\}).$$

which trivially holds.

Induction Conclusion. Since the $MCSs$ we are working with are finite a limited import closure in such a MCS has to be finite as well. Moreover on each step of building a limited import closure contexts are only added but never removed. Therefore for each LIC^x there is a unique and least fixpoint for $x \to \infty$ and we can say that

$$LIC^\infty(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC^\infty(C_i, \mathcal{C} \cup \{t\})$$

which is

$$LIC(C_t, \mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, \mathcal{C} \cup \{t\}).$$

□

Lemma 4.3.2
Let $M = (C_1, \dots, C_n)$ be an MCS, C_t a context in M, and $\mathcal{C} \subseteq \{1, \dots, n\}$ with $t \notin \mathcal{C}$. Then for each $i \in In(t)$ we have

$$|LIC(C_i, \mathcal{C} \cup \{t\})| < |LIC(C_t, \mathcal{C})| \,. \tag{4.3.8}$$

Proof
Since $t \notin \mathcal{C}$ we can use Lemma 4.3.1 to obtain from (4.3.8)

$$\underbrace{|LIC(C_i, \mathcal{C} \cup \{t\})|}_{L} < \underbrace{|\{t\} \cup \underbrace{\bigcup_{j \in In(t)} LIC(C_j, \mathcal{C} \cup \{t\})|}_{R'}}_{R}.$$

Recalling the definition of limited import closures (Definition 4.3.1) it is easy to see that $t \notin LIC(C_j, \mathcal{C} \cup \{t\})$ and therefore $t \notin R'$. Then it holds that $|R'| < |R|$. Since $i \in In(t)$ by definition and therefore i also an element of the index set of R' we know that $L \subseteq R'$ and therefore $|L| \leq |R'|$. So we can obtain $|L| \leq |R'| < |R|$.

$\square$

Lemma 4.3.3
Let $M = (C_1, \dots, C_n)$ be an MCS, C_t a context in M and $\mathcal{C} \subseteq \{1, \dots, n\}$ with $t \notin \mathcal{C}$. Then

$$GE(C_t, \mathcal{C}) \setminus \{t\} = \left(\bigcup_{i \in In(t)} GE(C_i, \mathcal{C} \cup \{t\}) \right) \setminus \{t\}.$$

Proof
We have to prove that

$$\underbrace{GE(C_t, \mathcal{C}) \setminus \{t\}}_{L} = \underbrace{\left(\bigcup_{i \in In(t)} GE(C_i, \mathcal{C} \cup \{t\}) \right) \setminus \{t\}}_{R} \,. \tag{4.3.9}$$

For an element to be in one of the sets L or R from (4.3.9) all of the following three properties have to be true:

	properties for $x \in L$	properties for $x \in R$
a)	$x \in \mathcal{C}$	$x \in (\mathcal{C} \cup \{t\})$
b)	$x \in \{t\} \cup \bigcup_{s \in LIC(C_t, \mathcal{C})} In(s)$	$x \in \bigcup_{i \in In(t)} \left(\{i\} \cup \bigcup_{s \in LIC(C_i, \mathcal{C} \cup \{t\})} In(s) \right)$
c)	$x \neq t$	$x \neq t$

Properties a) and b) are derived from Definition 4.3.2. Now we show that for $x \in \{1, \cdots, n\}$ properties a), b), and c) on the left side simultaneously hold *iff* they simultaneously hold on the right side.

Due to property c) it is clear that $\mathcal{C} \cup \{t\} = \mathcal{C}$ and therefore property a) is the same for L and R.

Now it is left to show that

$$\{t\} \cup \bigcup_{s \in LIC(C_t,\mathcal{C})} In(s) = \bigcup_{i \in In(t)} \left(\{i\} \cup \bigcup_{s \in LIC(C_i,\mathcal{C}\cup\{t\})} In(s) \right).$$

Now we can rearrange the equation on the right side. Since $x \neq t$ we can remove t from the left side, and since $t \notin \mathcal{C}$ we can use Lemma 4.3.1 to obtain

$$\bigcup_{s \in (\{t\} \cup \bigcup_{i \in In(t)} LIC(C_i,\mathcal{C}\cup\{t\}))} In(s) = In(t) \cup \bigcup_{s \in \bigcup_{i \in In(t)} LIC(C_i,\mathcal{C}\cup\{t\})} In(s).$$

We can further rearrange the left side and obtain

$$In(t) \cup \bigcup_{s \in \bigcup_{i \in In(t)} LIC(C_i,\mathcal{C}\cup\{t\})} In(s) = In(t) \cup \bigcup_{s \in \bigcup_{i \in In(t)} LIC(C_i,\mathcal{C}\cup\{t\})} In(s).$$

which trivially holds.

□

Now we show the soundness and the completeness of the algorithm. Therefore we state helper lemmas which are generalizations of the theorems for the soundness and completeness. The lemmas state that the results returned by each request in an MCS w.r.t. a history (a set of already visited contexts) are partial diagnosis candidates and its witness candidates and that the request returns all possible partial diagnosis candidates and its witness candidates. The theorem is then a special case where the partial diagnosis candidates are partial diagnosis and the witness candidates are witnesses.

For the following paragraphs a notation to define the index set of a set of contexts is introduced for easier reading: $idx(\mathcal{C}) = \{i \mid C_i \in \mathcal{C}\}$.

Soundness.

Lemma 4.3.4

Let $M = (C_1, \ldots, C_n)$ be an MCS, $hist \subseteq \{1, \ldots, n\}$, and C_t.DMCS-DF$(hist)$ a request in M. Then each *request C_t.DMCS-DF$(hist)$ returns a set of pairs $(DG_{\mathcal{C},\mathcal{D}}, W_{\mathcal{C},\mathcal{D}})$ such that $DG_{\mathcal{C},\mathcal{D}}$ is partial diagnosis candidate and $W_{\mathcal{C},\mathcal{D}}$ a witness candidate w.r.t. $DG_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = LIC(C_t, hist)$ and $idx(\mathcal{D}) = GE(C_t, hist)$.*

Proof

The following proof is done by induction over the size of the limited import closure of the requests. So in the induction base we proof that the lemma holds in an arbitrary MCS for all requests whose limited import closures are of size 0. In the induction step then we assume that the lemma holds in an arbitrary MCS for a request with a limited import closure of size x and we show then that the lemma holds also in an arbitrary MCS for a request with a limited import closure of size $x + 1$.

Induction Base. We show that Lemma 4.3.4 holds for all requests $C_t.\mathsf{DMCS\text{-}DF}(hist)$ with $|LIC(C_t, hist)| = 0$.

$|LIC(C_t, hist)| = 0$ implies that $LIC(C_t, hist) = \emptyset$. According to Definition 4.3.1 $t \notin hist$ implies $t \in LIC(C_t, hist)$. Therefore $LIC(C_t, hist) = \emptyset$ implies $t \in hist$. According to Definition 4.3.2 it is clear that property b) holds only for the element t because $LIC(C_t, hist) = \emptyset$. Since $t \in hist$ which means t also fulfills property a) it is clear that $GE(C_t, hist) = \{t\}$.

Now we have to show that each result returned by $C_t.\mathsf{DMCS\text{-}DF}(hist)$ is a pair $(DG_{\mathcal{C},\mathcal{D}}, W_{\mathcal{C},\mathcal{D}})$ with $\mathcal{C} = \emptyset$ and $\mathcal{D} = \{C_t\}$.

Since $t \in hist$ the if condition in Algorithm 4.2.1 Line 2 is true and therefore Algorithm 4.2.2 is called with C_t as argument. The result of this call is a set of pairs $((\epsilon, \ldots, \epsilon), (\epsilon, \ldots, B_t, \ldots, \epsilon))$ with $B_t \in \mathbf{BS}_t$. According to Definition 3.4.1 and 3.4.2 it is easy to see that the first tuple is a partial diagnosis candidate and the second tuple a witness candidate of it with $\mathcal{C} = \emptyset$ and $\mathcal{D} = \{C_t\}$.

Induction Step. Assuming that Lemma 4.3.4 holds for requests $C_{t'}.\mathsf{DMCS\text{-}DF}(hist')$ with $\left|LIC(C_{t'}, hist')\right| \leq x$ and $x \geq 0$ we now show that the Lemma holds also for requests with $|LIC(C_t, hist)| = x + 1$.

Let $M = (C_1, \ldots, C_n)$ be an MCS, $hist \subseteq \{1, \ldots, n\}$, $C_t.\mathsf{DMCS\text{-}DF}(hist)$ a request in M and $|LIC(C_t, hist)| = x + 1$. Then Algorithm 4.2.1 is called at context C_t with $hist$ as argument. Since $x \geq 0$ it follows that $|LIC(C_t, hist)| > 0$. Due to Definition 4.3.1 $t \in hist$ implies that $LIC(C_t, hist) = \emptyset$. Therefore $|LIC(C_t, hist)| > 0$ implies that $t \notin hist$. Therefore the if condition in Line 2 evaluates to false.

The loop in Line 6 is called for each context of the import neighborhood of C_t. Since Line 7 is a performance optimization and has no effect on the results[2] it will be assumed that the if condition is always true in this proof. In Line 8 a request $C_i.\mathsf{DMCS\text{-}DF}(hist \cup \{t\})$ is sent to all contexts for each $i \in In(t)$. Due to Lemma 4.3.2 we know that the limited import closure of each request send in Line 8, $LIC(C_i, hist \cup \{t\})$, has less elements than $LIC(C_t, hist)$ which means that $|LIC(C_i, hist \cup \{t\})| \leq x$. Therefore, according to Lemma 4.3.4, we can assume that the returned results $\mathcal{T}'$ are partial diagnoses candidates $DG_{\mathcal{C},\mathcal{D}}$ and their witnesses candidates $W_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}) = GE(C_i, hist \cup \{t\})$ for the respective $i \in In(t)$.

All those requests are joined together in Line 9. Regardless of the order of requests respectively joins Lemma 3.6.1 states that the result, stored in $\mathcal{T}$, is a set of partial diagnosis candidates with their witness candidates where $idx(\mathcal{C}) = \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}) = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\})$.

In Line 13 for all those tuples Algorithm 4.2.3 is called. Lets say the algorithm is called with the partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ and the witness candidate $W_{\mathcal{C},\mathcal{D}}$. We know that $idx(\mathcal{C}) = \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}) = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\})$ for all partial diagnosis candidates and witness candidates. In each iteration of the foreach loop in Line 3,

[2]It's the following idea: If W_i is already defined in some witness candidate then there must have been a request to C_i through other contexts. The results of this request have already been joined with the existing partial diagnosis candidates and witness candidates and therefore another request to C_i will neither remove nor extend the existing results and can therefore be skipped.

$G_t = (D_t, A_t)$ s.t. $D_t, A_k \subseteq br_t$ and Line 4 stores in $\mathcal{B}_t$ all belief sets accepted by context C_t in $M[G_t]$ under the belief state $W_{\mathcal{C},\mathcal{D}}$.

Now we distinguish between $W_t = \epsilon$ and $W_t \neq \epsilon$.

- If $W_t = \epsilon$ the algorithm proceeds in Line 6. There a new tuple $DG'_{\mathcal{C}',\mathcal{D}'}$ is created by extending the partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ with (D_t, A_t) where $D_t, A_t \subseteq br_t$ from context C_t. Recalling Definition 3.4.1, $DG_t = (D_t, A_t)$ implies that $C_t \in \mathcal{C}'$ and since $\mathcal{C}' \cap \mathcal{D}' = \emptyset$ also that $C_t \notin \mathcal{D}'$. So if $DG'_{\mathcal{C}',\mathcal{D}'}$ has a witness candidate in $M[DG']$, $DG'_{\mathcal{C}',\mathcal{D}'}$ is a partial diagnosis candidate with $idx(\mathcal{C}') = \{t\} \cup idx(\mathcal{C}) = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}') = idx(\mathcal{D}) \backslash \{t\} = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\}) \backslash \{t\}$.

  In this line a witness candidate $W'_{\mathcal{C}',\mathcal{D}'}$ is also created by extending $W_{\mathcal{C},\mathcal{D}}$ with $W'_t \in \mathbf{ACC}_t(kb_t \cup \{head(r) \mid r \in app(br_t \setminus D_t \cup heads(A_t)), W)\})$. Since W is a witness candidate for DG, if we recall Definition 3.4.2, it is clear that $W'_{\mathcal{C}',\mathcal{D}'}$ is a witness candidate for $DG'_{\mathcal{C}',\mathcal{D}'}$ in $M[DG']$ with $idx(\mathcal{C}') = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}') = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\})$.

- If $W_t \neq \epsilon$ the algorithm proceeds in Line 8. There the partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ is extended with (D_t, A_t) where $D_t, A_t \subseteq br_t$ at context C_t to a tuple DG'. If W is still accepted in $M[DG']$ then, after recalling Definition 3.4.1, DG' is clearly a partial diagnosis candidate $DG'_{\mathcal{C}',\mathcal{D}'}$ with $idx(\mathcal{C}') = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}') = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\})$ with the witness candidate W. Since DG and DG' differ only at context C_t and $W_t \in \mathbf{ACC}_k(kb_t \cup \{head(r) \mid r \in app(br_t \setminus D_t \cup heads(A_t)), W)\})$, after recalling Definition 3.4.2, it is clear that W is a witness candidate $W_{\mathcal{C}',\mathcal{D}'}$ for DG' with $idx(\mathcal{C}') = \{t\} \cup \bigcup_{i \in In(t)} LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}') = \bigcup_{i \in In(t)} GE(C_i, hist \cup \{t\})$.

In both cases we can use Lemma 4.3.1 to write $idx(\mathcal{C}') = LIC(C_t, hist)$ and Lemma 4.3.3 to write $idx(\mathcal{D}') = GE(C_t, hist)$ which matches with the specification from Lemma 4.3.4. Therefore the result holds for all sizes of a requests limited import closure and therefore for all requests.

□

Theorem 4.3.5
Let $M = (C_1, \ldots, C_n)$ be an MCS and C_t a context in M. Then each *result returned by a request C_t.DMCS-DF($\emptyset$) is a pair (DG_{C_t}, W_{C_t}) with DG_{C_t} a partial diagnosis w.r.t. C_t and W_{C_t} a witness of DG_{C_t}.*

Proof
We know from Lemma 4.3.4 that a request C_t.DMCS-DF($\emptyset$) returns pairs $(DG_{\mathcal{C},\mathcal{D}}, W_{\mathcal{C},\mathcal{D}})$ where $DG_{\mathcal{C},\mathcal{D}}$ is a partial diagnosis candidate and $W_{\mathcal{C},\mathcal{D}}$ a witness candidate with $idx(\mathcal{C}) = LIC(C_t, hist)$ and $idx(\mathcal{D}) = GE(C_t, hist)$. Recalling Definitions 4.3.1 and 4.3.2 of a limited import closure and a guessing edge it is clear that $hist = \emptyset$ implies $LIC(C_t, hist) = IC(C_t)$ and $GE(C_t, hist) = \emptyset$. Under the condition that $idx(\mathcal{C}) = IC(C_t)$ and $idx(\mathcal{D}) = \emptyset$ for $DG_{\mathcal{C},\mathcal{D}}$

and $W_{\mathcal{C},\mathcal{D}}$ the definitions of partial diagnosis candidates and partial diagnoses (Definition 3.4.1 and 3.2.1) and the definitions of witness candidates and witnesses (Definition 3.4.2 and 3.3.1) are the same. Therefore a partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ is a partial diagnosis DG_{C_t} and a witness candidate $W_{\mathcal{C},\mathcal{D}}$ is a witness W_{C_t}.

□

Completeness.

Lemma 4.3.6
Let $M = (C_1, \ldots, C_n)$ be an MCS, $hist \subseteq \{1, \ldots, n\}$, and C_t.DMCS-DF($hist$) a request in M. If $DG_{\mathcal{C},\mathcal{D}}$ is a partial diagnosis candidate and $W_{\mathcal{C},\mathcal{D}}$ the corresponding witness candidate with $idx(\mathcal{C}) = LIC(C_t, hist)$ and $idx(\mathcal{D}) = GE(C_t, hist)$ then $(DG_{\mathcal{C},\mathcal{D}}, W_{\mathcal{C},\mathcal{D}}) \in C_t$.DMCS-DF($hist$).

Note that each partial diagnoses DG_{C_t} with its witness W_{C_t} is also a partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ with its witness candidate $W_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = IC(C_t)$ and $idx(\mathcal{D}) = \emptyset$.

Proof
Like in the proof for the previous lemma this proof is done by induction over the size of the limited import closure of the requests. So in the induction base we proof that the lemma holds in an arbitrary MCS for all requests whose limited import closures are of size 0. In the induction step then we assume that the lemma holds in an arbitrary MCS for a request with a limited import closure of size x and we show then that the lemma holds also in an arbitrary MCS for a request with a limited import closure of size $x + 1$.

Induction Base. We show that Lemma 4.3.6 holds for all requests C_t.DMCS-DF($hist$) with $|LIC(C_t, hist)| = 0$.

$|LIC(C_t, hist)| = 0$ implies that $LIC(C_t, hist) = \emptyset$. According to Definition 4.3.1 $t \notin hist$ implies $t \in LIC(C_t, hist)$. Therefore $LIC(C_t, hist) = \emptyset$ implies $t \in hist$. According to Definition 4.3.2 it is clear that property b) holds only for the element t because $LIC(C_t, hist) = \emptyset$. Since $t \in hist$ which means t also fulfills property a) it is clear that $GE(C_t, hist) = \{t\}$.

After recalling Definition 3.4.1 it is clear that for $LIC(C_t, hist) = \emptyset$ there is only one partial diagnosis candidate $(\epsilon, \cdots, \epsilon)$. Moreover after recalling Definition 3.4.2 and the fact that $GE(C_t, hist) = \{t\}$ it is clear that all witness candidates are of form $(\epsilon, \cdots, W_t, \cdots, \epsilon)$ with $W_t \in \mathbf{BS}_t$.

Now we have to show that the mentioned partial diagnosis candidates and the witness candidates from above are returned by a call C_t.DMCS-DF($hist$) with $idx(\mathcal{C}) = \emptyset$ and $idx(\mathcal{D}) = \{t\}$.

Since $t \in hist$ the if condition in Algorithm 4.2.1 Line 2 is true and therefore Algorithm 4.2.2 is called with C_t as argument. The result of this call is a set of pairs $((\epsilon, \ldots, \epsilon), (\epsilon, \ldots, B_t, \ldots, \epsilon))$ with $B_t \in \mathbf{BS}_t$ which is exactly all partial diagnosis candidates and all witness candidates for $LIC(C_t, hist) = \emptyset$.

Induction Step. Assuming that Lemma 4.3.6 holds for requests $C_{t'}.\textsf{DMCS-DF}(hist')$ with $\left|LIC(C_{t'}, hist')\right| \leq x$ and $x \geq 0$ we now show that the Lemma holds also for requests with $|LIC(C_t, hist)| = x + 1$.

Let $M = (C_1, \ldots, C_n)$ be an MCS, $hist \subseteq \{1, \ldots, n\}$, $C_t.\textsf{DMCS-DF}(hist)$ a request in M and $|LIC(C_t, hist)| = x + 1$. Then Algorithm 4.2.1 is called at context C_t with $hist$ as argument. Since $x \geq 0$ it follows that $|LIC(C_t, hist)| > 0$. Due to Definition 4.3.1 $t \in hist$ implies that $LIC(C_t, hist) = \emptyset$. Therefore $|LIC(C_t, hist)| > 0$ implies that $t \notin hist$. Therefore the if condition in Line 2 evaluates to false.

Lets assume there is a partial diagnosis candidate $DG_{\mathcal{C},\mathcal{D}}$ and its witness candidate $W_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = LIC(C_t, hist)$ and $idx(\mathcal{D}) = GE(C_t, hist)$. We denote $DG^{-t}_{\mathcal{C},\mathcal{D}}$ as $DG_{\mathcal{C},\mathcal{D}}$ with an arbitrary element at C_t and $W^{-t}_{\mathcal{C},\mathcal{D}}$ as $W_{\mathcal{C},\mathcal{D}}$ with an arbitrary element at C_t. We will show for an arbitrary $i \in In(t)$, which therefore shows for each i, that there is a partial diagnosis candidates $DG'_{\mathcal{C}',\mathcal{D}'}$ and a witness candidate $W'_{\mathcal{C}',\mathcal{D}'}$ with $idx(\mathcal{C}') = LIC(C_i, hist \cup \{t\})$ and $idx(\mathcal{D}') = GE(C_i, hist \cup \{t\})$ which, if joined together, result in $DG^{-t}_{\mathcal{C},\mathcal{D}}$ and $W^{-t}_{\mathcal{C},\mathcal{D}}$. To accomplish this we set each DG'_j with $j \in LIC(C_i, hist \cup \{t\})$ to DG_j and all other to ϵ. From Lemma 4.3.1 we know that the limited import closure $LIC(C_i, hist \cup \{t\})$ for each neighboring context C_i is a subset of $LIC(C_t, hist)$ and therefore DG' a partial diagnosis candidate according to Definition 3.4.1 if there is a witness candidate in $M[DG']$. Now we show that there is such a witness candidate $W'_{\mathcal{C}',\mathcal{D}'}$ by constructing one. Therefore we are setting $W'_j = W_j$ for all $j \in LIC(C_i, hist \cup \{t\})$ and $j \in GE(C_i, hist \cup \{t\})$. We now show that $W_{\mathcal{C},\mathcal{D}}$ is defined in $LIC(C_i, hist \cup \{t\})$ and $GE(C_i, hist \cup \{t\})$. From Lemma 4.3.1 we know that the limited import closure $LIC(C_i, hist \cup \{t\})$ is a subset of $LIC(C_t, hist)$. We know from Lemma 4.3.3 that $GE(C_i, hist \cup \{t\}) \setminus \{t\} \subseteq GE(C_t, hist) \setminus \{t\}$ and we know that $t \in LIC(C_t, hist)$ and therefore that $W_{\mathcal{C},\mathcal{D}}$ is defined in $LIC(C_i, hist \cup \{t\})$ and $GE(C_i, hist \cup \{t\})$. Therefore W' is defined in all contexts of its limited import closure and its guessing edge. If a belief set W'_j from W' is accepted depends on the belief sets of the contexts from $In(j)$. For all $j \in LIC(C_i, hist \cup \{t\}$ we know that $In(j) \subseteq (LIC(C_i, hist \cup \{t\}) \cup GE(C_i, hist \cup \{t\}))$ because of Definition 4.3.1 and 4.3.2. Since all belief sets in $LIC(C_i, hist \cup \{t\}) \cup GE(C_i, hist \cup \{t\})$ are the same in W and W' and all the belief sets in W are accepted by definition also the belief sets in W' are accepted and therefore W' is a witness candidate for DG'.

According to Lemma 4.3.2 we know that $|LIC(C_i, hist \cup \{t\})| < |LIC(C_t, hist)|$ for each i and therefore we know that each request in the loop in Line 6 returns the constructed partial diagnosis candidate and its witness candidate from above. Since Line 7 is a performance optimization and has no effect on the results[3] it will be assumed that the if condition is always true in this proof. All those requests are joined together in Line 9 and it is clear that one result of the accumulated joins is DG^{-t} and W^{-t}. In Line 13 Algorithm 4.2.3 is called with DG^{-t} and W^{-t}. In each iteration of the foreach loop in Line 3, $G_t = (D_t, A_t)$ s.t. $D_t, A_k \subseteq br_t$ and Line 4 stores in $\mathcal{B}_t$ all belief sets accepted by context C_t in $M[G_t]$ under the belief state $W_{\mathcal{C},\mathcal{D}}$.

Now we distinguish between $W_t = \epsilon$ and $W_t \neq \epsilon$.

[3]It's the following idea: If W_i is already defined in some witness candidate then there must have been a request to C_i through other contexts. The results of this request have already been joined with the existing partial diagnosis candidates and witness candidates and therefore another request to C_i will neither remove nor extend the existing results and can therefore be skipped.

- If $W_t = \epsilon$ the Algorithm proceeds in Line 6. There the partial diagnosis candidate is extended with (D_t, A_t) where $D_t, A_t \subseteq br_t$ from context C_t obviously resulting in DG. In this line the witness candidate is also extended with $W'_t \in \mathbf{ACC}_t(kb_t \cup \{head(r) \mid r \in app(br_t \setminus D_t \cup heads(A_t)), W)\})$ which means there must be some case where W^{-t} is extended to the witness candidate W.

- If $W_t \neq \epsilon$ the Algorithm proceeds in Line 8. There the partial diagnosis candidate is extended with (D_t, A_t) where $D_t, A_t \subseteq br_t$ at context C_t which again results in one case to DG and obviously we have already W and since W_t must be in $\mathcal{B}_t$ this result is returned.

Therefore the result holds for all sizes of a requests limited import closure and therefore for all requests.

□

Theorem 4.3.7
Let $M = (C_1, \ldots, C_n)$ be an MCS and C_t a context in M. Then if (DG_{C_t}, W_{C_t}) is a partial diagnosis and a corresponding witness w.r.t. C_t, the request C_t.DMCS-DF$(\emptyset)$ returns (DG_{C_t}, E_{C_t}).

Proof
Recalling the definitions of partial diagnosis candidates and partial diagnoses (Definition 3.4.1 and 3.2.1) and the definitions of witness candidates and witnesses (Definition 3.4.2 and 3.3.1) it is clear that a partial diagnosis with its witness DG_{C_t} and W_{C_t} are a partial diagnosis candidate and its witness candidate $DG_{\mathcal{C},\mathcal{D}}$ and $W_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = IC(C_t)$ and $idx(\mathcal{D}) = \emptyset$. From the Definitions 4.3.1 and 4.3.2 of the limited import closure and the guessing edge we also know that the request C_t.DMCS-DF$(\emptyset)$ has $LIC(C_t, hist) = IC(C_t)$ and $GE(C_t, hist) = \emptyset$. So from Lemma 4.3.6 we know that all partial diagnosis candidates and witness candidates $DG_{\mathcal{C},\mathcal{D}}$ and $W_{\mathcal{C},\mathcal{D}}$ with $idx(\mathcal{C}) = LIC(C_t, hist) = IC(C_t)$ and $idx(\mathcal{D}) = GE(C_t, hist) = \emptyset$ are returned by the request C_t.DMCS-DF$(\emptyset)$. Since we have shown before that all partial diagnoses and its witnesses DG_{C_t} and W_{C_t} are a subset of those partial diagnosis candidates and witness candidates we have shown that Theorem 4.3.7 holds.

□

CHAPTER 5

Implementation

The implementation of the described algorithm is based on the DMCS system [3], developed at the Vienna University of Technology. To distinguish them the DMCS system is referred to as DMCS and the implementation of the diagnoses finder as DMCS-DF (DMCS Diagnoses Finder). The DMCS system is designed to calculate equilibria of a distributed Multi-Context System. To accomplish this the calculation itself is split up and done at distributed agents which are realized as daemons. Every such daemon is responsible for a context and an additional client queries those daemons initially to get the equilibria. The system is capable of calculating partial and full equilibria in an MCS. The implementation presented here extends this system with the capability to calculate partial and full diagnoses. Therefore parts of the DMCS system are reused, such as the network implementation, the belief set calculation, and the file parsing architecture.

In the following we explain the existing system, especially those systems reused by our implementation. Then we describe the extension in-depth.

5.1 The DMCS System

The DMCS program is a modular software which supports different modes to evaluate MCS semantics. The following explanations are based on the default mode of the program. The extensions described in this theses are all based on this mode and therefore the description will be limited to this one.

An extension of the DMCS program to optimize the calculation is called DMCS-OPT [4]. Ideas of this extension are used to optimize the DMCS-DF implementation. An introduction of the DMCS-OPT system and explanation why it can not be used as a basis of the DMCS-DF system can be found in Section 5.4.1.

5.1.1 Program Architecture

The basic system consists of two programs: One is the dmcsd daemon, which runs as an distinct instance for every context in the system. This daemon reads the knowledge base and the bridge rules for its context from the corresponding files. Moreover it reads information about its context neighbors and communication details to correspond with those neighbors, which are also provided by the input files. Then the daemon waits for a request to deliver partial diagnoses w.r.t. his context. If it is necessary to request more information from other contexts to complete the calculations the daemon requests this information from the corresponding contexts. There another instance of the daemon is waiting and then calculates the requested information. When the daemon has completed his calculations it returns the results back to the requesting daemon.

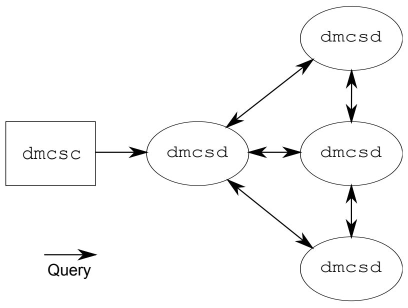


Figure 5.1.1: Basic System Architecture

The functions of a dmcsd daemon are described in the following: When a daemon is started it parses the input files, creates the loop formulas based on all possible input variables combinations and waits for a request from the network. If such an request arrives the loop formulas are solved by the SAT solver. Now the partial belief states from the neighbors are requested if the requesting context is not a leaf context and if no cycle has been detected. The results of the requests are iteratively joined with the local results from the SAT solver and then finally send back to the invoker.

5.1.2 Input Files

There are three different types of files which describe the MCS and provide additional information about the system which are necessary for the system to function properly. There is the *topology file* which contains information about the topology of the MCS and network informations such that separate parts of the system can communicate among each other. Each context gets the same topology file. The *knowledge base file* contains the knowledge base for each context and the *bridge rule file* lists the bridge rule for each context. Each context has its own knowledge base and bridge rule file. In the following those files will be described in more detail:

Topology File. The topology file contains a list of contexts including the following information: the network address of every context, a list of atoms for each context, including the atom identifiers, and a list of context dependencies, i.e, those context which are referred to in the

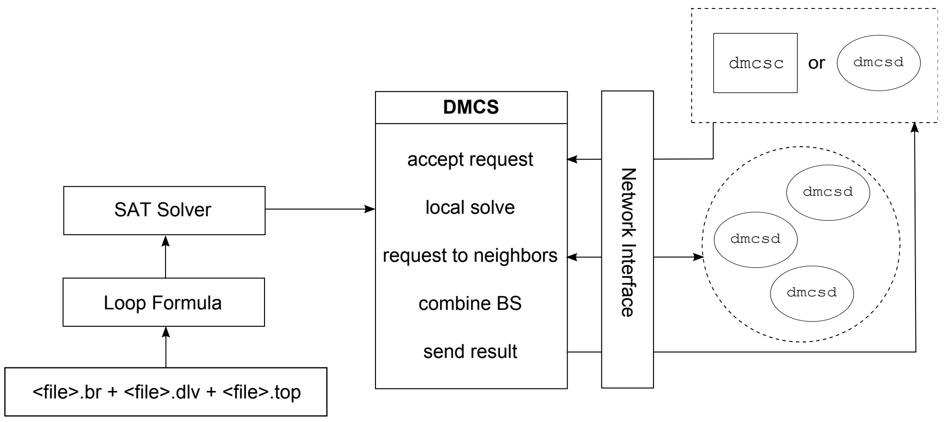


Figure 5.1.2: Basic System Functions

bridge rules. The information is arranged in a graph like style written in the DOT Language[1] syntax. Each node of the graph represents a context with the following properties:

- hostname: DNS name or IP address of the context,
- port: port where the corresponding daemon is listening,
- sigma: list of 4-tuples each describing an atom of the context or an atom from a contexts bridge rule body.

The tuples describing the atoms contain four values: The first, named `sym`, is the string representation of the atom. The second, named `ctxId` is the ID of the context the atom belongs to. The third, named `localId`, is an ID which is unique among all atoms in the realm where the atom is used and the fourth, named `origId`, an ID unique among all atoms of the same context. The difference between the `localId` and the `origId` is that an atom has always the same `origId` but the `localId` can be different depending on which context the atom is used. I.e., each daemon assigns its own `localId`s for atoms from other contexts. The denoted `localId` in the tuple from the topology file is only valid in the context where the tuple is defined.

Example 5.1.1
We have a context C_1 with an atom `(a1 1 1 1)` meaning `sym` is $a1$ and the `ctxId`, `localId` and `origId` are 1. Moreover we have a context C_2 with `(a2 2 1 1)` and a bridge rule at C_1 with $a2$ in its body. If the daemon running at C_1 refers to atom $a2$ it can not use the `origId` since this would interfere with the `origId` of $a1$ (both are 1). Therefore, e.g., the daemon gives $a2$ a `localId` of 2 and is now able to distinguish between $a1$ and $a2$ by their `localId`.

□

[1]`http://graphviz.org/content/dot-language` (Last checked on November 11, 2015)

It is important that the nodes are ordered by their context ID. This means the first node represents the context with ID 1, the second the context with ID 2 and so on.

Example 5.1.2
The following is a section of the topology file covering the graph part:

```
digraph G {
0 [hostname="10.0.0.10", port="5000", \
    sigma="(a1 1 1 1),(b1 1 2 2),(c1 1 3 3),(d1 1 4 4)"];
1 [hostname="10.0.0.10", port="5001",  \
    sigma="(a2 2 1 1),(b2 2 2 2),(c2 2 3 3),(d2 2 4 4)"];
2 [hostname="10.0.0.11", port="5000", \
    sigma="(a3 3 1 1),(b3 3 2 2),(c3 3 3 3),(d3 3 4 4)"];
3 [hostname="10.0.0.12", port="5000", \
    sigma="(a4 4 1 1),(b4 4 2 2),(c4 4 3 3),(d4 4 4 4)"];
```

The MCS described in this example consists of four contexts. Context 1 and 2 are hosted on a machine with the IP address 10.0.0.10 and are distinguished by the port on which the daemons are listening (5000 and 5001). Context 3 and 4 are on a machine with the IP address 10.0.0.11 and 10.0.0.12 respectively and both are listening on port 5000. The alphabet of context 1 consists of the atoms $a1$, $b1$, $c1$ and $d1$. `localId` and `origId` are consecutively numbered from 1 to 4 on all contexts.

□

Knowledge Base File. The knowledge bases for the contexts are defined in a separate file for each context. Each line represents one rule and is written with the following syntax:

```
a [v b] :- c, d, not e, not f.
```

The head of the rule is mandatory and can not be omitted. Therefore constraints of form "`:- {body}.`" are not possible. Instead the substitution "`kill :- not kill, {body}.`" with a newly introduced atom, in this case `kill`, is recommended.

Bridge Rule File. The bridge rules for the contexts are defined in a separate bridge rule file for each context. Each line represents a bridge rule. A bridge rule is of form:

```
{head} :- {body}. |
{head}.
```

where

```
{head} = h1 [v h2 [v h3 ... ]]
```

and

```
{body} = (c1:b1) [, (c2:b2)] [, (c3:b3)] [...]
    [, not (c4:b4)] [, not (c5:b5)] [...]
```

$\mathtt{h}_x$ are atoms of the bridge rules context. $\mathtt{b}_x$ are atoms of the context $\mathtt{c}_x$ where $\mathtt{c}_x$ is the index of the context.

Example 5.1.3
The following is a bridge rule file at a context with two bridge rules. a_1 and c_1 are atoms of the context, a_3 and c_3 of a context C_3, and a_2 of a context C_2.

```
c1 :- (3:a3), not (2:a2).
a1 :- (3:c3), not (2:a2).
```

□

5.1.3 System Usage

DMCS Daemons (dmcsd).

```
./dmcsd --context=1 --port=5001 \
    --kb=example_c1.dlv --br=example_c1.br --topology=example.top
```

The `--context` argument tells the daemon his own context ID, the `--port` argument the network interface port the daemon listens for incoming requests, the `--kb` and `--br` arguments provide the path to the contexts knowledge base and bridge rule file, and the `--topology` argument the path to the global topology file.

DMCS Caller (dmcsc).

```
 ./dmcsc --hostname=localhost --port=5001 \
    --system-size=2 --manager=example.top \
    --query-variables="15 45"
```

The `--hostname` and `--port` argument tells the program the network address of the dmcsd daemon where the user is interested in and the request for equilibria is sent. `--system-size` is the number of contexts in the MCS and `--manager` is the path to the topology file of the MCS. Finally the `--query-variables` argument defines the atoms which are requested from the contexts. The argument must contain an integer number for each context in the system. The first number defines the atoms for the first context, the second number for the second context an so on. The number itself is a decimal representation of a bitmap where the (n+1)th bit stands for the nth atom in the context (according to the encoding of belief sets, see 5.1.4). In the example above this means: $15 \equiv 1111$, therefore the atoms with id 1, 2 and 3 are requested in context 1, further $45 \equiv 101101$, which means atoms 2, 3 and 5 are requested from context 2.

5.1.4 Data Types

In this section we give a short introduction to the encoding and representation of basic data structures of an MCS in the DMCS system. The most important ones, and also the ones we will extend later, are belief states and bridge rules.

Belief States. A belief state is a tuple consisting of a belief set for each context in the MCS. Every belief state is implemented as a vector of belief sets and the index of the vector corresponds to the ID of the belief set's context. The belief set itself is represented as an `integer` data type interpreted as a bitmap. The first bit is set if the belief set is defined and not set if the belief set is ϵ. The subsequent bits represent the atoms of the context and are set if the atom is contained in the belief set and otherwise not set. E.g., the second bit represents the atom with the `origId` 0, the third bit the one with the `origId` 1, and so on. DMCS stores the belief sets as 64 bit integer values which restricts each belief set to 63 atoms or beliefs.

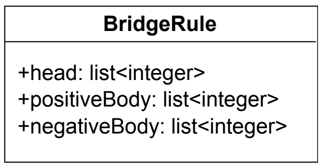


Figure 5.1.3: Simplified BridgeRule Diagram

Bridge Rules. A bridge rule consists of three lists, one containing the head atoms and two listing the positive resp. negative body atoms. The head atoms are encoded as `integers` which represent the `origId` of the atom in the context. The body atoms are represented by a pair of `integers`, one identifying the context by the `context ID` and one representing the `localId` of the atom[2]. Note that the `localId` can be different for the same atom in different contexts. But this does not pose a problem since bridge rules are only used at the local context for the belief set calculations and only a string representation is communicated through the network.

5.1.5 Algorithm and Belief Set Calculation

The main program logic of the DMCS system is outlined in Algorithm 5.1.1.

Algorithm 5.1.1 shows the code sequence which is executed if a request arrives. Before the request arrives, when the daemon has been started, the contexts knowledge base and its bridge rules have been converted into a SAT theory. Then, when the request arrives, the SAT theory is solved w.r.t. a projection of the atoms given by the caller (see Dao-Tran et al. [14] for details). This projection restricts the results to those atoms which are of interest for the caller. The result is a list of belief states which are defined at the local context and over the atoms of the contexts import interface. If the local context has been requested before (its ID is in the $hist$ variable) or it has no neighbors then nothing else is done and the results are send back to the invoker. In case the context has not been requested before and there are neighbors, the neighbors are requested and the results are joined with the already calculated belief states. These results are then send back to the invoker.

[2]See Paragraph "Topology File" in Section 5.1.2 for an explanation of the different IDs.

Algorithm 5.1.1: Program Logic of DMCS at $C_k = (L_k, kb_k, br_k)$

```
Input: list<integer> history, BeliefState projection
Output: list<BeliefState> bsList
1 bsList = calculateLocalBeliefSets;
2 bsList = projectBS(bsList, projection);
3 if $contextID \notin hist$ then
      // no cycle detected
4     foreach $neighbor \in neighborList$ do
5         nResult = request(neighbor);
6         bsList = combine(bsList, neighbor.bsList);
7     end
8 end
9 return (bsList);
```

5.2 Problem Instance Generator

The problem instance generator is a tool for creating random $MCSs$ constrained by some parameters. The main parameters are the number of contexts, the topology, the number of atoms per context, the number of interface atoms per context, and the number of bridge rules per context. Interface atoms are those atoms of a context which are used in the bodies of bridge rules. Moreover the generator creates shell scripts to start all parts of the DMCS or DMCS-DF (introduced in 5.3) systems. This feature is very convenient for automated testing.

The generator has been developed with the DMCS system and been extended in line with this thesis to make it usable for the DMCS-DF system.

5.2.1 Adaptations

Inconsistent Instances. Since the DMCS-DF systems main purpose is to find diagnoses for inconsistent $MCSs$, in order to create more instances which are inconsistent, a command line option has been added which affects the probability of consistent or inconsistent instances. The option changes the probability of adding positive body atoms in the knowledge base rules. Experiments have shown that a higher probability of adding positive body atoms correlates with more inconsistent instances.

Shell Scripts. As already mentioned the generator produces shell scripts for automated program starts. The generator has been extended with a command line option `mode` to control the output of the tool. Now you can tell the tool to generate only the MCS files (knowledge bases and bridge rules for each context), the start script for the DMCS system, the start script for the DMCS-DF system, or a start script for the DMCS-DF system which uses the optimized MCS files (see Section 5.4.1 for the mentioned optimization).

Tool Support. The RunLim[3] tool can limit time and memory consumption of Linux programs. Moreover it produces log files about the resources consumed by the program. Those features are very convenient for testing purposes. On the one hand to give an upper limit for time and memory consumption such that unexpected long running instances do not hold back the whole automated experiment and on the other hand to determine the memory consumption of the programs.

The time usage of the programs/daemons and parts of the system are measured with build in features. To compare the DMCS-DF with the DMCS system a tool to measure the overall runtime of a program is needed. For this purpose the Linux command `time`[4] has been used.

The tool support has been implemented by integrating the RunLim tool into the produced shell scripts which are responsible for starting the DMCS resp. DMCS-DF system with the generated problem instances. A call to those shell scripts produces logging output with time and memory consumption.

5.2.2 Usage of the Problem Instance Generator

The generator can be used with

```
./dmcsGen --<option>=<value>
```

with the following command line options:

- `--contexts=<integer>`: number of contexts in the MCS. Depending on the topology not all numbers are possible.
- `--atoms=<integer>`: number of atoms per context.
- `--interface=<integer>`: number of atoms (as a subset of all atoms) per context used in bodies of bridge rules.
- `--bridge_rules=<integer>`: number of bridge rules per context.
- `--topology=<integer>`: topology type with: 0 = random, 1 = diamond, 2 = diamond arbitrary, 3 = diamond zigzag, 4 = pure ring, 5 = ring edge, 6 = binary tree, 7 = house, and 8 = multiple ring topology (see Section 6.1 for details to the topologies used in this thesis).
- `--prefix=<string>`: prefix used for the MCS file names.
- `--dmcspath=<file path>`: path to the dmcsc and dmcsd binaries.
- `--positiveBodyP=<integer>`: probability for a positive atom in the body of each knowledge base rule. An integer value i means $(i * 10)\%$. E.g. $5 = 50\%$.
- `--sleep=<float>`: This value is the time in seconds the shell script waits to start the dmcsc program after the dmcsd daemons have been started.

[3]`http://fmv.jku.at/runlim/` (Last checked on November 11, 2015)

[4]`https://www.gnu.org/software/time/` (Last checked on November 11, 2015)

- `--use_run=<file path>`: file path to the RunLim executable. If this option is empty no RunLim will be used.
- `--run_time=<integer>`: time used by the program in seconds after that the RunLim tool will end the DMCS(-DF) system.
- `--run_realtime=<integer>`: real time in seconds after that the RunLim tool will end the DMCS(-DF) system.
- `--run_space=<integer>`: the program will be terminated by the RunLim tool if it uses more memory than stated here in MB.
- `--mode=default|mcs|dfStart|dfOptStart|dmcsStart`: In `default` mode the generator produces the files defining the MCS and a script to start the DMCS system requesting this MCS. The `mcs` mode produces only the files for the MCS, which are the knowledge base, bridge rule, and topology files. The `dfStart` mode creates a shell script to start the DMCS-DF system using the standard (non optimized) topology file. The `dfOptStart` mode creates a shell script to start the DMCS-DF system using the optimized (see Section 5.4.1) topology file. The `dmcsStart` mode creates a shell script to start the DMCS system using the standard topology file.
- `--logPrefix=<string>`: file name prefix for the log files produced by the DMCS/DMCS-DF system and the performance measurement tools.
- `--shPrefix=<string>`: file name prefix for the shell start scripts.
- `--timeTool=<boolean>`: if set to true the start scripts use the Linux time command to measure the total time usage.

5.3 DMCS Diagnoses Finder (DMCS-DF)

The DMCS-DF system is based on the DMCS system revision 2256[5]. It has been extended in following ways:

- New data structures representing the newly introduced diagnosis (candidate) and witness (candidate) concepts have been defined,
- the algorithms from Section 4.2 have been implemented in order to calculate diagnoses and witnesses,
- in order to inspect the calculated results and evaluate the program, detailed result output is provided, debugging options to inspect the algorithm have been implemented, and the program gathers statistical data about the calculations.

[5]`http://sourceforge.net/p/dmcs/code/2256/tree/dmcs/` (Last checked on November 11, 2015)

5.3.1 Data Structures

The DMCS system defined already data structures for belief sets, belief states, and bridge rules. The newly introduced data structures are diagnoses and the corresponding witnesses whereupon diagnoses are completely new and witnesses are nothing else than belief states connected to a diagnosis. Therefore a container class which connects diagnoses and belief states has also been introduced.

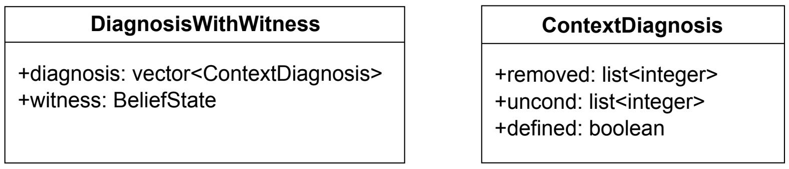


Figure 5.3.1: Class Diagram of DiagnosisWithWitness and its Components

Diagnoses. As seen in Figure 5.3.1, a diagnosis is stored in a class which has two vectors one holding the diagnosis and one the witness. Each item of those two vectors represent a context ordered by `contextID`. I.e. the first item refers to context 1, the second to context 2, and so on.

The witness vectors is of type BeliefState as already described in Section 5.1.4. The items of the diagnosis vector are classes of type `ContextDiagnosis` which has two lists each containing bridge rule IDs. One list contains the IDs of the bridge rules which are removed by this diagnosis and on this context and the other list holds the IDs of the bridge rules which are applied unconditionally by this diagnosis and on this context. The bridge rule ID has been introduced by this extension and is explained later in this section. The `ContextDiagnosis` class also has a property named *defined* which is true if the diagnosis is ϵ with respect to this context.

Bridge Rules. Bridge rules have already been defined in the DMCS system (see Section 5.1.4). A bridge rules is an object, which stores its atoms via the `localID` which is only unique within a specific context. In the DMCS system bridge rules are only used at one context and therefore the `localID` suffices. The DMCS-DF system on the other hand needs to send information about bridge rules through the MCS which makes it necessary to add a new identification symbol for bridge rules which is unique in the whole MCS. For this purpose the `BridgeRuleInformation` class (see Figure 5.3.2) has been introduced.

The `contextID` is the ID of the context where the bridge rule belongs to. `bridgeRuleID` is a hash value[6] of the string representation of the bridge rule and therefore unique among the context. The pair `contextID`-`bridgeRuleID` is unique among the MCS. The `bridgeRuleString` is the string representation of the bridge rule. This is necessary because the string representation of the atoms used in the bridge rules are only available at the context where the atoms

[6]The calculation of hash values is done with the boost library found at `http://www.boost.org/doc/libs/1_46_1/doc/html/hash.html` (Last checked on November 11, 2015).

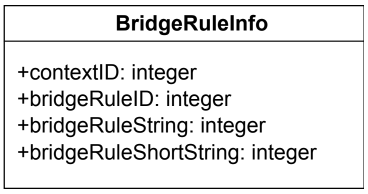


Figure 5.3.2: Class Diagram of BridgeRuleInfo

belong to. Therefore the string representation is send along with the results such that a human readable output can be generated. The output also includes a list of bridge rules of the part of the MCS which has been processed and an abbreviation of each rule. Those abbreviations are used in the representation of the diagnoses and are of the form `r<ctxID>_<consecutive number>` (e.g. `r1_4`). The consecutive number is a sequential number unique for each context. In the `BridgeRuleInfo` class the abbreviation of the bridge rules is stored in the `bridgeRuleShortString` attribute.

Network Messages. We have two types of network messages, one which is send when an invocation is done, and one when the result is send back (see Figure 5.3.3).

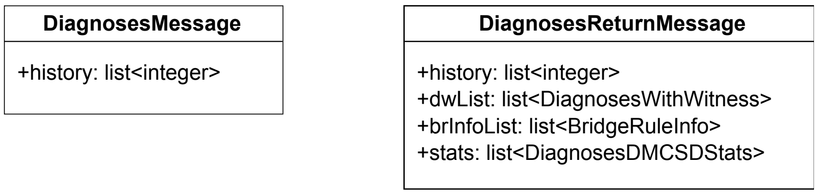

DiagnosesMessage
+history: list<integer>
DiagnosesReturnMessage
+history: list<integer>
+dwList: list<DiagnosesWithWitness>
+brInfoList: list<BridgeRuleInfo>
+stats: list<DiagnosesDMCSDStats>

Figure 5.3.3: Class Diagram of the Message Classes

If a request for diagnoses is send to a context the `DiagnosesMessage` is send to the corresponding daemon. The only information which is included with such a request is the history of already requested contexts. This history is a property of the `DiagnosesMessage` class implemented as a list of integer values where the integer values are the context IDs. Since the calculation of the partial diagnoses and the witnesses does not depend on information from contexts prior in the calling chain[7] (see Section 4.2) no more information needs to be transmitted to calculate the requested diagnosis candidates.

For returning the resulting diagnosis candidates as an answer to the `DiagnosesMessage` the `DiagnosesReturnMessage` class is used. The most important thing to send back is the list of partial diagnoses and witnesses which is stored in the `dwList` member of the class. To display the bridge rules in a proper format information about them must also be send back (see Section 5.3.1) which is done with the `brInfoList` member. In certain cases two invocations

[7]If the context is part of a cycle, the calculation indeed depends on information of previous contexts, but then they will be invoked again.

from one context can bring back bridge rule information about the same context. E.g. a context A and a context B is invoked and both of these contexts invoke context C so both return messages contain information about context C. To ease the handling of such redundant information every return message contains a list with contexts, which have been invoked. This is the member `history` in the return message class. Finally every context gathers performance information about its calculations. Those informations are also send back with the return message to enable an aggregated display of the performance information as dmcsc output. Those information are stored in the `stats` member and are required for empirical evaluation in Section 6. There you find also details about the `DiagnosesDMCSDStats` class.

5.3.2 Program Logic

Overview. The main program logic is shown in Algorithm 5.3.1.

A major difference between the theoretical approach (Algorithm 4.2.1) and the actual implementation is the calculation of local belief sets. The theoretical approach is: get all the results from the neighboring contexts, take each of this results, extend it to all possible local diagnoses, and then calculate the local belief sets based on these extended results.

In contrast to that, the actual implementation calculates the local belief sets independently from the results of the neighboring contexts: As in the theoretical approach first all results from the neighboring contexts are requested and joined together. Then for each possible local diagnosis the local belief sets are calculated. The resulting local belief sets are based on guesses about the interface atoms and not on the actual results from the neighbors. Now the results from the neighbors and the local belief sets are joined together and the results are the requested partial diagnosis candidates with its witness candidates.

The benefit of the implemented approach is that it reduces the upper limit of the number of local solve operations. In the theoretical approach the number of local solve operations is one calculation for each partial diagnosis candidate from the joined result of the neighbors times the number of local diagnoses. With the implemented approach the number of calculations is the number of possible local diagnoses which is in every case less or equal then in the theoretical approach. On the other hand the theoretical approach reduces the complexity of the local solve calculation because the results from the neighboring contexts determine the state of the interface atoms and therefore the local belief states do not have to be calculated for all possible combinations of the interface atoms.

So the performance of these approaches depend on the actual problem instances. A deeper investigation on the relation between the calculation method and the problem instances is subject of future work. It may be efficient to design a hybrid solution which is also subject of future work.

The implementation of the DMCS system is designed to calculate the local belief sets based on guesses about the interface atoms. Since the extension DMCS-DF is based on the DMCS system, this approach has been chosen for the DMCS-DF system.

The actual implementation of the program logic has been done by defining a new subclass inherited from the `BaseDMCS` class.

Algorithm 5.3.1: Program Logic of DMCS-DF at $C_k = (L_k, kb_k, br_k)$

```
   Input: list<integer> history
   Output: DiagnosesReturnMessage returnMessage
1  DiagnosisWithWitnessList dwList;
2  dwList.add($(\epsilon, \dots, \epsilon), (\epsilon, \dots, \epsilon)$);
3  if contextID $\in$ hist then
       // cycle detected
4      foreach bs $\in \mathrm{BS}_k$ do
5          $dwList.add((\epsilon, \dots, \epsilon), bs)$;
6      end
7  else
       // no cycle detected
8      foreach neighbor $\in$ neighborList do
9          foreach dw $\in$ dwList do
10             if $dw.witness_{contextID} = \epsilon$ then
11                 nResult = request(neighbor);
12                 dwList = combine(dwList, neighbor.dwList);
13             end
14         end
15     end
16     foreach diagnosis $\in \{(D_k, A_k) \mid D_k, A_k \subseteq br_k\}$ do
17         localBSList = localSolve(diagnosis);
18         foreach beliefset $\in localBSList$ do
19             localDwList.add(diagnosis, beliefset);
20         end
21     end
22     dwList = combine(localDwList, dwList);
23     dwList = calculateSubsetMinimalOverSameWitness(dwList);
24 end
25 return (dwList, stats, history, BridgeRuleInfo, mask);
```

Belief State Combination Masks. In the DMCS system the data structures for belief sets can only be tagged as "undefined" as a whole. The belief set is implemented as a bitmap and therefore each atom has only two states, which is true or false. Optimizations (see Section 5.4.1) make it necessary to define single atoms as undefined. Therefore in the DMCS-DF system a second bitmap of the same size as the belief set has been introduced to resolve this issue. This bitmap is called the belief state combination mask. Each bit of the mask corresponds with the atom at the same position in the belief set. A bit set to 1 means the atom is defined, a 0 means the atom is undefined. Like a belief state, a mask consists of a bitmap for each context.

Now the function of the belief state combination mask within the algorithm will be explained: In case the algorithm detects a cycle all possible belief sets of this context are guessed including all atoms. In this case we create a mask which states all atoms at this context as defined and all atoms at other contexts as undefined.

In case no cycle is detected, according to the algorithm, the results from the neighboring contexts are requested. All those results come with a corresponding mask. Those results are combined together with a join operation. A join is defined like in Definition 3.1.3. Since atoms are now also declared as defined or undefined the atoms of the result of a join are: Lets say a' is an atom from the first operand, a'' the corresponding atom from the second operand, and a the corresponding atom from the result. Then

- $a = a'$, if $a' = a''$,
- $a = a'$, if a' is defined and a'' is undefined,
- $a = a'$, if a'' is defined and a' is undefined, and
- the join is void, if a' and a'' are defined and $a' \neq a''$.

This holds for all atoms in the belief state.

This is implemented in the following way: Let s' and s'' be the belief sets which are the operands of the join, s the resulting belief set, and m the belief state combination mask.

```
if (s′ & m) == (s″ & m) then
  s = s′ | s″
end if
```

Since it is required that the atoms are only compared if both atoms are defined, the mask m is derived from the masks of the operands by a boolean "and" operation. If one or both atoms are not defined m is 0 and the if condition always true. Since the undefined atom is 0 the assignment of s results in the defined atom due to the boolean "or" operation. The mask for the result of the join is then derived from the masks of the operands by a boolean "or" operation because now, if the result exists, every atom is defined where at least one atom was defined in the operands.

Bridge Rule Guessing. To construct a list of all possible bridge rule guesses, first a list of all subsets of the bridge rules from the context is made. Then the list of guessed bridge rules is made by combining those subsets as pairs. The first item of those pairs is the list of bridge rules which are removed and the second item the list of bridge rules which are applied unconditionally.

```
for all $subset_1 \in bridgeRuleSuperSets$ do
  for all $subset_2 \in bridgeRuleSuperSets$ do
    if $subset_1 \cap subset_2 = \emptyset$ then
      $guessList.add(subset_1, subset_2)$
    end if
  end for
end for
```

If a diagnosis is applied to an MCS first the bridge rules are removed and then the heads of the bridge rules added. Therefore a diagnosis which has a bridge rule in the remove part and the same bridge rule in the unconditional part has the same effect as a diagnosis which is the same except the before mentioned bridge rule is only in the unconditional part. These redundant diagnoses are removed in line 3.

Minimality Check. In the definition of diagnoses by Eiter et al. [16] they also defined *point-wise subset-minimal* diagnoses which are an interesting notion of diagnoses. They represent a subset of diagnoses where those diagnoses are excluded which modify the same bridge rules as other diagnoses and additional ones. So the intention is to only get those bridge rules modifying a minimal set of bridge rules. The definition is:

Definition 5.3.1
Given two diagnoses $DG_1 = (D_1, A_1)$ and $DG_2 = (D_2, A_2)$, the diagnosis DG_1 is subset-minimal in relation to DG_2 iff $D_1 \subseteq D_2$ and $A_1 \subseteq A_2$.

Example 5.3.1
E.g., let's say we have the diagnoses $(\{r_1, r_2\}, \emptyset)$ and $(\{r_1\}, \emptyset)$ then the second one is the subset minimal one but if we have two diagnoses $(\{r_1, r_2\}, \emptyset)$ and $(\{r_1\}, \{r_3\})$ then none of them is subset minimal.

□

The DMCS-DF system provides the functionality to calculate subset-minimal diagnoses. This is basically done by a pairwise comparison between every diagnoses in the candidate list. The comparison itself is a subset-minimal check on both sets of the diagnoses, namely D (the set with the removed bridge rules) and A (the set with bridge rules applied unconditionally). A diagnosis is subset-minimal if it is subset-minimal in D and A. The check between two sets of diagnosis is done on the set itself by going through one set, let's name it set A, and count the number of same bridge rules in the other set, let's name it set B. If the result is less than the number of bridge rules in A, this set can not be a subset-minimal diagnosis. On the other hand if the result is less than the number of bridge rules in set B, this set can also not be a subset-minimal diagnosis. After this check in both sets, A and B, it can be determined if one of the diagnosis is a subset of another one. This is the case if it has not turned out that the diagnosis is not subset-minimal but the other diagnoses is indeed not subset-minimal. If the second diagnoses had not turned out to be not subset-minimal both diagnoses are the same and therefore none of them is subset-minimal.

The calculation of a subset-minimal diagnoses is independent of the witnesses of the diagnoses and therefore the witnesses are not considered in the calculation of subset-minimal diagnoses.

If it is desired to get the subset-minimal diagnoses of the Multi-Context System this check is done in the querying client program when all diagnoses have been calculated.

5.3.3 Input Files

The DMCS-DF system uses the same syntax for its input files as the DMCS system (see Section 5.1.2). But the topology file needs to contain additional information for the DMCS-DF system to work properly.

The topology file of the DMCS system contains only the nodes of the topology graph (see Section 5.1.2) and gets the neighbor list directly from the bridge rule file. To determine the neighbor list without modifying the bridge rule file the DMCS-DF system reads the neighbor list from the topology file. This is needed later for optimizations. Therefore we add an additional part with the edges of the topology graph. They are defined in the following way: `x->y;`. x is the ID of the context which requests information from the context with the ID y.

Example 5.3.2

```
0->1;
1->2;
2->3;
3->0;
```

This example shows a ring topology (see Section 6.1) with four contexts.

□

5.3.4 System Usage

Like DMCS, the DMCS-DF system is split into daemon programs (dmcsd) for each context which do the processing of the diagnoses and the witnesses and a querying client program (dmcsc) which sends a request to the daemons.

DMCS-DF Daemons (dmcsd). The dmcsd program must be called for each context with the following options:

- `--context=<integer>`: the context ID of the context the daemon is assigned.
- `--port=<integer>`: the port number where the daemon is listening for incoming requests.
- `--kb=<file path>`: filename of the knowledge base file of the context.
- `--br=<file path>`: filename of the bridge rule file of the context.
- `--topology=<file path>`: filename of the topology file of the MCS.
- `--print_timing=<boolean>`: prints time statistics of the assigned context and those contexts from whom results have been requested to the standard output (see Section 5.3 [Performance Measurement]).
- `--outputlevel={<none|low|medium|high}`: see Section 5.3 [Output] for details. Default value is `low`.
- `--debuglevel=<string>`: see Section 5.3 [Debugging] for details.

Example 5.3.3

```
./dmcsd --context=1 --kb=./mcs/context_1.lp \
        --br=./mcs/context_1.br --topology=./mcs/mcs.top \
        --outputlevel=none
```

This example starts a daemon for the context with ID 1, the knowledge base is stored in a file with the path `./mcs/context_1.lp` and the bridge rules in a file with the path `./mcs/context_1.br`. The topology file is `./mcs/mcs.top` and there is no output printed to the standard output by this daemon.

□

DMCS-DF Querying Client Program (dmcsc). With the dmcsc program a request to the *MCS* for partial diagnoses and its witnesses is started. It is configured by the following command line options:

- `--host=<string>`: IP address or DNS name of the host where the dmcsd daemon is running which the dmcsc program will request.
- `--port=<integer>`: port number where the dmcsd daemon is running which will be requested.
- `--manager=<file path>`: filename of the topology file of the *MCS*.
- `--diagnoses=<boolean>`: this option must be set to true to start the DMCS-DF mode.
- `--print_timing=<boolean>`: prints time statistics of the assigned context and those contexts from whom results have been requested to the standard output (see Section 5.3 [Performance Measurement]).
- `--outputlevel={none|low|medium|high}`: see Section 5.3 [Output] for details). Default value is `low`.
- `--debuglevel=<string>`: see Section 5.3 [Debugging] for details. Default is no debugging.

Example 5.3.4

```
./dmcsc --host=10.0.0.10 --port=5000 --manager=mcs.top \
        --diagnoses=true --print\_timing=true \
        --outputlevel=high
```

This example starts the dmcsc program which sends a request for the partial diagnoses and its witnesses to the dmcsd daemon with the IP address 10.0.0.10 and port 5000. It reads the topology file `./mcs/mcs.top` to get the string representations of the atoms from the contexts. The output printed to the standard output has the highest level of detail and contains time statistics for all involved dmcsd daemons.

□

5.3.5 Output

Diagnoses and witnesses are printed in the following format:

```
<bridgeRuleShortString>: <bridgeRuleString>
<bridgeRuleShortString>: <bridgeRuleString>
...

<diagnosis>:<witness>
<diagnosis>:<witness>
...
```

First a list of bridge rules is printed which contains the whole bridge rule string and an abbreviation which is used in the diagnoses listed below.

A diagnosis has the following format:

```
({<bridgeRuleShortString>, <bridgeRuleShortString>, ...}, \
    {<bridgeRuleShortString>, <bridgeRuleShortString>, ...})
```

The first set of abbreviations of bridge rules are those bridge rules which are removed from the MCS and the second set are those who are applied unconditionally.

The witnesses have the following format:

```
([-]{<atom>, <atom>, ...}, [-]{<atom>, <atom>, ...}, ...)
```

Each set represents a contexts belief set, ordered by the context ID. The - in front of the sets is printed if the belief set is ϵ. A belief set is printed as a list with those atoms which are true in this belief set.

Example 5.3.5

```
r5_1: (5:c5) :- (1:e1), not (1:d1).
r5_2: (5:e5) :- not (1:a1).
r4_1: (4:f4) :- not (5:d5).
r4_2: (4:e4) :- not (5:c5), not (5:e5).

({}, {r4_2, r5_2, r5_1}):({a1}, -{}, -{}, {a4, c4}, {c5, e5, f5})
({r4_1, r4_2}, {r5_2, r5_1}):({a1}, -{}, -{}, {b4, e4}, {f5})
({}, {}):({a1, d1}, -{}, -{}, {a4, c4, e4, f4}, {b5, f5})
({r4_1, r4_2}, {}):({d1, e1}, -{}, -{}, {b4, e4}, {b5, f5})
```

This is a fragment of a partial result of an MCS with five contexts. First four bridge rules are listed from context 4 (`r4_1` and `r4_2`) and 5 (`r5_1` and `r5_2`). Then four partial diagnoses with their witnesses are listed. The witnesses are defined in the contexts 1, 4, and 5. The first diagnosis applies the bridge rules `r4_2, r5_2`, and `r5_1` unconditionally. The third row is an empty diagnosis and the fourth removes the bridge rules `r4_1` and `r4_2`.

□

The detail level of the output can be configured by the `--outputlevel` command line option on both, the dmcsd and the dmcsc program. It supports the four levels `none`, `low`, `medium`, and `high`.

dmcsd. If the the output level is `none` no output is printed at all. The output level `low` prints basic information about the state of processing but no results. The output level `medium` prints the same information as the `low` level and in addition the calculated results which are send back. The output level `high` prints the same as the `medium` level and the results of the calculation before they are reduced to the subset-minimal diagnoses with respect to the same witness (see Section 5.4.2).

dmcsc. If the the output level is `none` no output at all is printed. With output level set to `low` the network address of the requested context, the history, the list of bridge rules, and the subset-minimal results are printed to the standard output. With the output level set to `medium` or `high` additional results received from the requested context (before the subset-minimal ones are calculated) are printed.

5.3.6 Built-In Performance Measurement

The DMCS-DF system has been build with some time measurement classes to measure and compare the wall-clock time consumption of different parts of the program. Functions to get accurate clock values are used from the Boost library[8].

The parts of the program which are measured are the bridge rule guessing, the local solving, the local belief set guessing, the neighbor combination, the calculation of the subset-minimal diagnoses, the serializing, and the local combination. The *local solving* includes the calculation of the loop formulas based on the knowledge base and a set of bridge rules and the solving of the loop formulas by the clasp solver. The *local belief set guessing* is the part of the algorithm, where a cycle is detected and all possible belief sets of the context are guessed. The *neighbor combination* is the part where results from neighboring contexts are received and joined with the already received and joined results from other neighbors. The *calculation of subset-minimal diagnoses* includes the the calculation of the subset-minimal diagnoses at the dmcsc program in the end as well as the calculation of the subset-minimal diagnose with the same witness on each context. *Serializing* is the process of converting the objects to a serialized data stream which can then be send through the network. Serialization has to be done when data is send to another context and when data is received from other contexts. *Local combination* is the join of the results from the local solving and the results received form the neighboring contexts.

The time statistics of each context are send back to the invoking instance. Therefore the dmcsc program can print a list of time statistics of each context involved in the calculation process. Moreover it can sum up the the measured times for each context and get therefore statistics about the mentioned parts of the calculation process over the whole system.

[8] `http://www.boost.org/doc/libs/1_45_0/doc/html/date_time/posix_time.html` (Last checked on November 11, 2015)

Example 5.3.6

```
Overall Sum (0):
Total: 46.851
  BridgeRule Guessing: 0.716
    Local Solve: 0.715
    Rest: 0.001
  Combination of Neighbor Results: 0.827
  Local BeliefSet Guessing: 0.000
  Calculation of Subset-Minimal DG: 2.256
  Serializing: 28.350
  Local Combination: 11.682
  Rest: 3.018

Context 5 (105):
Total: 0.051
  BridgeRule Guessing: 0.047
    Local Solve: 0.046
    Rest: 0.000
  Combination of Neighbor Results: 0.000
  Local BeliefSet Guessing: 0.000
  Calculation of Subset-Minimal DG: 0.000
  Serializing: 0.002
  Local Combination: 0.001
  Rest: 0.000

...

dmcsc (0):
Total: 17.638
  BridgeRule Guessing: 0.000
    Local Solve: 0.000
    Rest: 0.000
  Combination of Neighbor Results: 0.000
  Local BeliefSet Guessing: 0.000
  Calculation of Subset-Minimal DG: 2.256
  Serializing: 13.135
  Local Combination: 0.000
  Rest: 2.246


|T: 46.851 |LS: 0.715 |LC: 11.682 |NC: 0.827 \
  |SM: 2.256 |SE: 28.350 |RC: 488592 |SC: 14|
```

Each block begins with an identifier which tells us the instance the statistic is about. This can be *Overall Sum* for the whole system, *Context* i for the dmcsd daemon on the context with ID i, and *dmcsc* for the dmcsc program. The number in brackets are the number of results which are send back by this instance. Then each row lists a part of the program which has been measured. The time is given in seconds. If a line is indented it means that this time interval is a subset of the interval given by the next row above which is less intended. E.g., here:

```
BridgeRule Guessing: 0.716
  Local Solve: 0.715
  Rest: 0.001
```

0.716 is the sum of 0.715 and 0.001.

The last line is a short form of the Overall Sum block for easier automated processing. The columns are: total time, local solve time, local combination time, neighbor combination time, subset-minimal calculation time, serialization time, the number of results returned by the first requested context, and the number of subset-minimal results.

5.3.7 Debugging

The DMCS-DF system can print fine-grained information about the calculation process for debugging purposes. This output is written to the standard error stream. The level of information can be controlled by the `--debuglevel=<string>` command line option. The various options are:

- `network`: prints information about network communication processes.
- `dmcsc`: prints status information from the dmcsc program.
- `calculatePartialDiagnoses`: gives information about the state of processing in the calculation process on the dmcsd daemons and prints intermediate results.
- `createBRGuessAndBSList`: prints information about the process of guessing local diagnoses and the results of each local solve process.
- `diagnosesCombination`: gives detailed information about operands, masks, calculation process, and the result when diagnoses are joined together.
- `createMask`: prints information about which masks are created from which belief states.
- `diagnosisPrint`: prints information about all processes involved into converting data from the program into printable strings.
- `DMCSdInit`: prints information about initialization procedures at each context.

- `minimality`: prints status information about the calculation of subset-minimal diagnoses.
- `beliefStateCombination`: gives detailed information about operands, masks, calculation process and the result when belief states are joined together.
- `parserAndLSolver`: prints information about the parsing process of the input files and the communication with the clasp SAT solver.

With the option `debugAll` the above listed options can be enabled all at once.

In addition to those debugging options which print debug information there are options which interfere with the calculation process and some of them produce incorrect results but give the opportunity to investigate different aspects of the program. They can for example be used to limit the number of results per context and therefore efficiently debug program functionality which is independent of the correctness of the result.

- `dontCalculateSubsetMinimalDG`: no subset-minimal diagnoses with the same witnesses are calculated at each context. The result of the calculations are still correct with this option enabled. This makes it possible to compare the system with and without the SP optimization (see Section 5.4.2).
- `resetReceivedResult`: drops the received results from the neighboring contexts.
- `setResultCount=<int>`: instead of sending the regular results back one result is taken and duplicated x times where x is the number given by this option. Therefore x results are send back.

5.4 Optimizations

First we describe already defined optimizations from Bairakdar et al. [4] for the DMCS system. We then show which parts of this optimization can be used for the DMCS-DF system and call this optimization OPT. Then an optimization called subset pruning (SP) is introduced.

5.4.1 Edge Pruning (OPT)

There has been some work done in order to speed up the DMCS algorithm by Bairakdar et al. [4]. They describe two types of optimizations: *refined recursive import* and *pruning*.

The first optimization is focused on reducing the information transmitted between contexts. A context c is a cut vertex in the graph representation of the MCS topology $\mathit{iff}\, G$ is a weakly connected graph and $G \setminus c$ a disconnected graph. The resulting belief states w.r.t. c send back to the parent context can then be reduced to the belief set of c and all other belief sets can be discarded without compromising further calculations in the parent context. In this work it is intended to find diagnoses and its witnesses of the MCS. The diagnoses and witnesses are defined in the whole import closure of the requested context. Therefore this optimization is not applicable for DMCS-DF and will not be considered.

The second optimization is based on a reduction of the topology to avoid unnecessary calls to the neighboring contexts as explained in the following example:

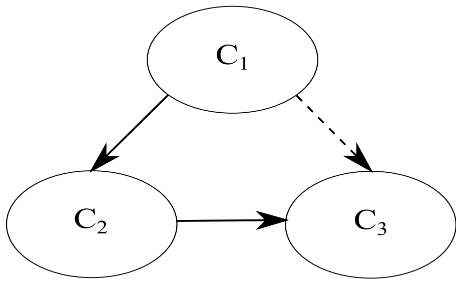


Figure 5.4.1: Edge pruning optimization

Example 5.4.1
We have an MCS with three contexts C_1, C_2, and C_3. Context C_1 requests information from context C_2 and C_3, and context C_2 from C_3. If we calculate partial diagnoses for C_1, this context needs partial diagnoses from C_2 and C_3. If the results from C_2 are requested, C_2 itself first requests the partial diagnoses from C_3, combines them with the local results, and sends them back to C_1. Then C_1 does not need to request the partial diagnoses directly from C_3 since all possible partial diagnoses are already in the returned results from C_2. So the call from C_1 to C_3 can be skipped which reduces the amount of data send between contexts and all the processing a request initiates on a context.

□

A limited version of this optimization is already integrated in the DMCS-DF implementation. The DMCS-DF algorithm makes a check before every request to a context if partial diagnoses from this context are already present in the returned results from other neighbors. But a context daemon has not enough information about the topology of the MCS to determine an optimized order of neighbor calls. The algorithm only benefits from an optimized order if such an optimized order of calls has been chosen by chance. Therefore an optimized plan of neighbor calls is stored in the topology file available to all contexts. The DMCS and DMCS-DF utilize the optimization by reading the neighbor list from this topology file. The optimization itself is a transitive reduction [22] of the topology graph.

Another part of the optimization strategy is to break cycles. Since every context guesses the atoms from its bridge rule bodies before calculating the local belief sets the request to the cycle context (the one where the cycle is detected) can be omitted. This also applies to the diagnoses which are not guessed when the cycle context is requested for the first time. The detection of cycles in the MCS requires information about the topology of the MCS and is therefore implemented in the query plan specified in the topology file. The cycle breaking itself is done by making an *ear decomposition* of the topology graph which results in a single cycle and additional paths beginning and ending at the cycle. This cycle and the last edge of every path is removed to get a cycle free topology.

The DMCS-DF system has been adapted to integrate the described optimizations above. Suppose we are at a context which has a neighboring context which marks the end of a cycle. In the non-optimized case a request to this context is send and the context returns fully defined belief sets from the context itself. In the optimized case no request is send and the interface atoms from this context are guessed. So in the optimized case we get only belief sets which are defined at those atoms which are interface atoms. This information, which atoms are defined and which are not, has to be stored and send with the belief sets such that further join operations operate correctly (see Section 5.3 for Belief State Combination Masks).

5.4.2 Subset Pruning (SP)

Earlier in this work (Definition 5.3.1) the concept of pointwise subset-minimal diagnoses has been introduced. They can be calculated by a comparing each (partial) diagnoses with each other. Subset-minimal diagnoses are always a subset of all diagnoses and therefore reduce the number of results. An obvious question to ask here is, whether we can we calculate subset-minimal diagnoses already at each context. This would reduce combination costs and network traffic. But as shown in the example below there are cases where a partial diagnosis is not subset-minimal at some context, but is part of a subset-minimal result later in the calculation process.

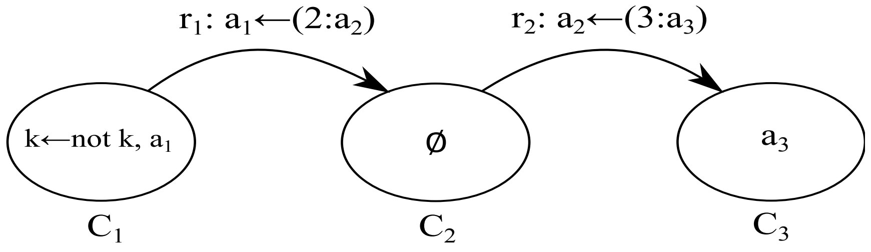


Figure 5.4.2: Example to demonstrate subset pruning

Example 5.4.2
We have three contexts C_1, C_2, and C_3. C_3 has the fact a_3 in its knowledge base and no bridge rules. C_2 has an empty knowledge base and the bridge rule $r_2 : a_2 \leftarrow (3 : a_3)$ and C_1 the bridge rule $r_1 : a_1 \leftarrow (2 : a_2)$ and rule $k \leftarrow not\ k, a_1$ in its knowledge base. For the empty diagnoses $(\epsilon, (\emptyset, \emptyset), (\emptyset, \emptyset))$ w.r.t. C_2 there is only one witness $(\epsilon, \{a_2\}, \{a_3\})$. Assume we calculate the subset-minimal diagnoses of the results at C_2 and send back only those results. Since an empty diagnosis is always the only subset-minimal diagnosis this diagnosis with the mentioned witness is send back to C_1. With this partial result the only diagnosis w.r.t. C_1 is $((\{r_1\}, \emptyset), (\emptyset, \emptyset), (\emptyset, \emptyset))$ with its witness $(\emptyset, \{a_2\}, \{a_3\})$. But this is not the only subset-minimal diagnosis w.r.t. C_1. E.g., $((\emptyset, \emptyset), (\{r_2\}, \emptyset), (\emptyset, \emptyset))$ with its witness $(\emptyset, \emptyset, \{a_3\})$ is also a subset-minimal diagnoses. This shows that it does not suffice to communicate subset-minimal results between contexts.

□

Subset-minimal diagnoses w.r.t. a partial belief state. We have seen that subset-minimal partial diagnoses, as an intermediate result, do not suffice to calculate all the partial diagnoses from a larger import closure. This is because a context has no information how other contexts depend on his belief sets. But other contexts do not depend directly on the partial diagnoses of the contexts in their import closure. Therefore all partial diagnoses with the same witness, which are the basis of further calculations, result in the same partial diagnoses and witnesses at the parent contexts. Because of this we can calculate the subset-minimal diagnoses for all partial diagnoses with the same witness at each context. This will reduce the number of results send back by every called context and therefore reduce the amount of data send between contexts and the number of joins at the contexts.

We have therefore found a way to calculate specific subset-minimal diagnoses which increases the efficiency of the DMCS-DF system and still produces correct results. Moreover we have seen that a straight forward approach to calculate subset-minimal diagnoses in order to increase the performance of the algorithm is impossible.

CHAPTER 6

Experimental Evaluation

We now have defined an algorithm and optimizations of it, we have shown that the algorithm is sound and complete, and we have an implementation of the algorithm. What is left to show is that the implementation of the algorithm and its optimizations are feasible when they run on state-of-the-art hardware. This means that we want to check whether the optimizations make a difference and result in noticeable faster and less memory consuming runs and to study how the effectiveness is determined by properties of the problem instances. Finally we will compare the capability of our implementation to calculate equilibria in consistent $MCSs$ and compare it with the DMCS system, which was solely designed to calculate equilibria and was the basis for our implementation. During this comparison experiments we take a deeper look into our implementation and examine the performance of the different subsystems of the algorithm in order to explain differences to the DMCS system and to identify a starting point for further optimizations.

In summary we will check the following hypotheses:

- The edge pruning optimization will reduce the runtime and memory consumption on problem instances with one or more cycles.
- The subset pruning optimization will dramatically reduce the runtime and memory consumption on all problem instances and the effect will increase on larger instances.
- Due to different implementations of the joining process in the DMCS and our system, we assume that both systems have a different performance characteristics w.r.t. problem instance topologies, more specific different neighborhood relations, and a different scaling behavior with increasing belief set sizes.

First we introduce the categorization of problem instances which are tested by the experiments. Next the effects of the optimizations described in Section 5.4 are presented. First the DMCS-DF system without optimizations is compared to the DMCS-DF system with the OPT optimization enabled and then this configuration to the system with both optimizations enabled.

Finally the DMCS-DF system is tested against the DMCS system in finding equilibria in consistent MCS. We conclude with a brief summary about the results of the experiments which is a noticeable performance gain due to the optimizations, especially on instances with cycles, and that our implementation is able to compete with the DMCS system in calculating equilibria.

The problem instance files as well as the log output files can be found under http://www.kr.tuwien.ac.at/research/systems/dmcs/dmcs-df/

6.1 Definition of Problem Instances

Problem instances for empirical experiments are categorized by 5 parameters which are described in the following.

The first parameter is the topology of the contexts, more specific, the topology of the graph representation of the Multi-Context System. Table 6.1.1 lists the topologies and their abbreviations.

D	Diamond
Z	Zigzag
H	House
R	Ring

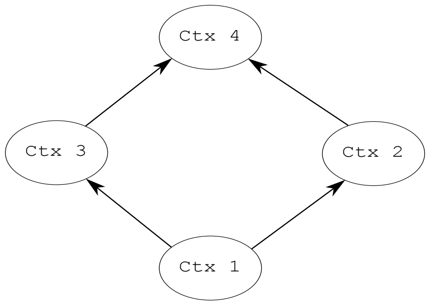
Table 6.1.1: Abbreviations of Topologies

Diamond Topology. A *diamond* is based on $3 * n + 1$ contexts as shown in figure 6.1.1a. If there is more than one diamond, the first context of the second diamond and the fourth context of the first diamond are the same and so on. Diamond topologies contain no cycles.

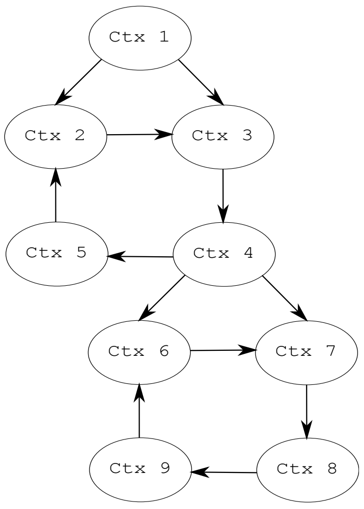
Zigzag Topology. The *zigzag* topology is the same as the diamond topology with additional information flow from context 3 to context 2 in all its diamonds. Compared to the diamond topology, a zigzag has more neighbor relations for one context of each diamond but also no cycles.

House Topology. An example of a *house* topology can be seen in Figure 6.1.1b with two houses. Larger $MCSs$ are composed in the same way which means that the roof is always connected at the bottom of the preceding house. A house MCS has at least one cycle.

Ring Topology. In a *ring* topology every context is connected in a row. E.g., in a ring topology with four contexts, context 1 requests information from context 2, context 2 from context 3, context 3 from 4, and context 4 from context 1. Such an MCS has always exactly one cycle which is the whole MCS itself and each context has exactly two neighbors: one requesting information one it requests information from.



(a) Diamond Topology



(b) House Topology

Figure 6.1.1: Topologies

The other four parameters are the number of contexts in the MCS, the number of atoms per context, how many of them are used in bridge rule bodies, and at the number of bridge rules per context.

An example configuration is `D`-4-4-3-3 which means that the MCS has a diamond topology with 4 contexts, 4 atoms per context where 3 of them are used in bridge rule bodies, and 3 bridge rules per context.

6.2 Optimizations

In this section the results of the empirical experiments will be shown. They compare the performance on state of the art consumer hardware and tell us what the effect of the implemented optimizations in the DMCS-DF system are. The tests have been performed on a single linux machine hosting the daemons for all contexts of the tested $MCSs$. The machine has a dual core 2.53 GHz processor and 4 GB of physical RAM. An Ubuntu 11.04[1] installation serves as operating system.

6.2.1 Edge Pruning (OPT)

We compare the DMCS-DF system without optimizations (*edge pruning* and *subset pruning*) to the system with *edge pruning* enabled (DMCS-DF-OPT) to see the benefits of this optimization in terms of time and memory consumption.

[1]http://www.ubuntu.com (Last checked on November 11, 2015)

The effect of this optimization depends very much on the problem instance topology. Therefore tests with the topologies *diamond*, *house*, *ring*, and *zigzag* have been performed. For each problem configuration (a specific topology, number of contexts, atoms, interface atoms, and bridge rules) ten instances have been created and tested. Since the purpose of the DMCS-DF system is to find diagnoses in inconsistent *MCSs*, inconsistent instances have been created.

The results are presented with graphs which show the median, the minimum, and the maximum values over those ten instances. The runtime and the memory consumption have been evaluated. Runtime is the time from the start of the querying client program till the printing of the returned results. Memory consumption is the sum of the maximum allocated memory from each daemon and the client program.

First Figure 6.2.1 shows the runtime of the system in seconds to give an overview of the system runtime. Since the problem instances are very heterogeneous it is easier to read the results on graphs showing relative values. Therefore Figure 6.2.2 shows the runtime of the DMCS-DF-OPT system in percent of the DMCS-DF runtime. The memory consumption is shown in Figure 6.2.3 as absolute values in MB and also a graph with the memory consumption of the DMCS-DF-OPT system as percentage of the DMCS-DF system in Figure 6.2.4.

Diamond Topology We can see that on instances with a diamond topology the optimization has no effect. This is clear since no cycles can be broken and the transitive reduction of the topology graph remains the same.

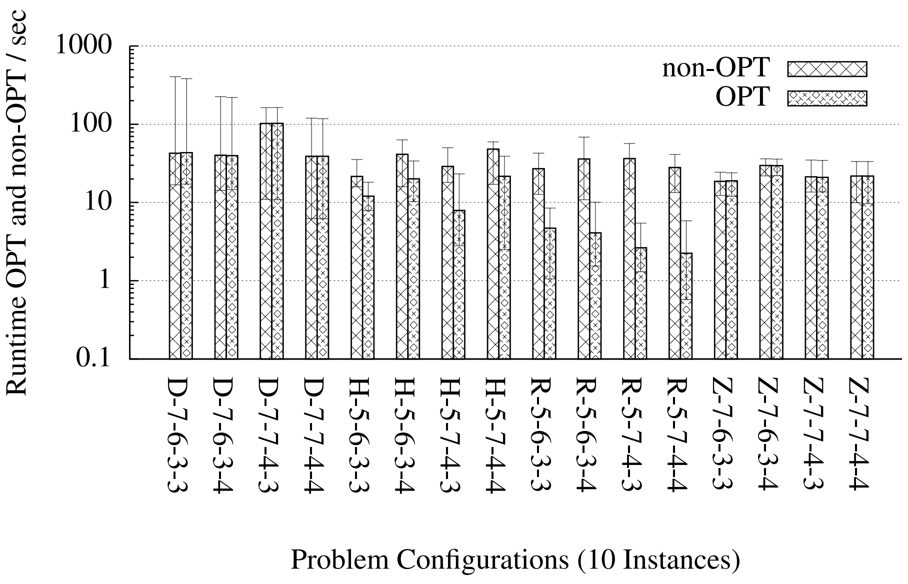


Figure 6.2.1: Runtime of DMCS-DF-OPT and DMCS-DF in Seconds (Minimum, Median, Maximum)

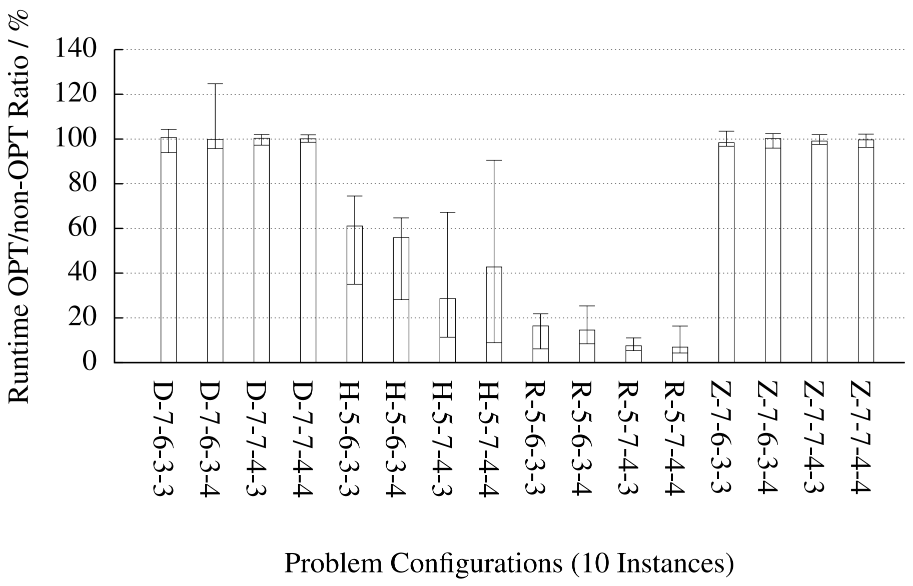


Figure 6.2.2: Runtime of DMCS-DF-OPT in Relation to DMCS-DF in % (Minimum, Median, Maximum)

Zigzag Topology Zigzag topologies have, like diamond topologies, no cycles and therefore no optimization effect by cycle breaking. But in contrary to the diamond topology a zigzag can be optimized with the a transitive reduction as shown in Figure 6.2.5a (dotted requests can be skipped).

Therefore the transitive reduction could have an effect on the runtime and the memory consumption of zigzag topologies. However, transitive reduction works by defining a special order of requests, and as this order is by coincidence the same as without OPT for the Zigzag instances, no effect can be observed. Therefore the DMCS-DF system without the graph pruning optimization acts the same way as with the edge pruning optimization. This is why there is no observable gain in time usage and memory consumption in Figure 6.2.2 and 6.2.4.

We conclude with the observation that this optimal behavior can only be guaranteed by the OPT algorithm.

Ring Topology Ring topologies are cycles and can therefore be optimized by breaking the cycle. This is reflected in the figures with respect to total time and memory consumption. Both are around or less than 20% of the values without the optimization. The effectiveness of the optimization is increasing with more (interface-) atoms per context. This can be seen if the results from R-5-6-3-3 and R-5-6-3-4 instances are compared with the R-5-7-4-3 and R-5-7-4-4 instances. The latter are going down to 10% in memory consumption and less than 10% in total time usage.

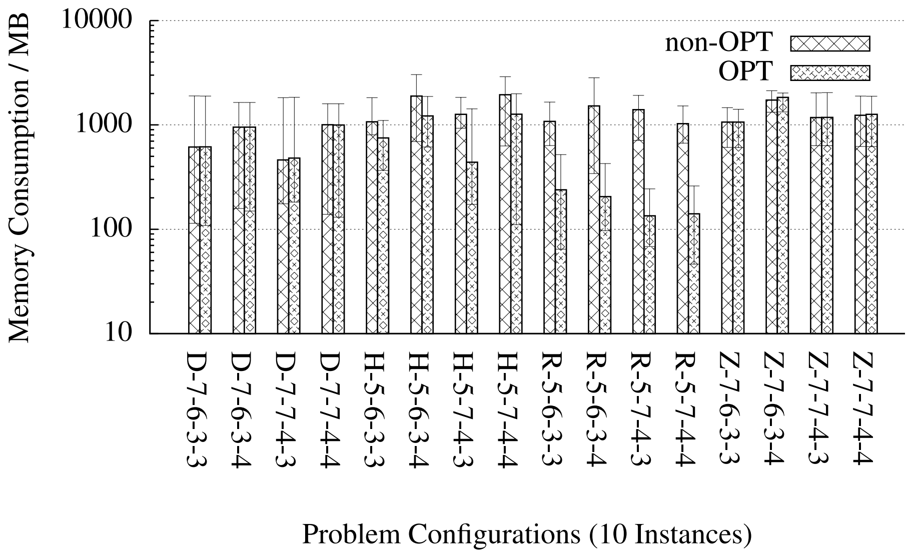


Figure 6.2.3: Memory Consumption of DMCS-DF-OPT and DMCS-DF System in MB (Min, Median, Max)

House Topology A house topology is optimized by (a) skipping the request from Context 1 to Context 3 and (b) breaking the cycle (Figure 6.2.5b).

A more close examination of the optimization (a) shows that the removal of the request from Context 1 has no effect. This is because in case without optimization, no matter which context is requested first from Context 1, the returned result is defined in all contexts of the cycle, namely contexts 2, 3, 4, and 5. Therefore the second request from Context 1 is dropped as well in the non optimized case.

The optimization (b) is effective on the contexts of the cycle. The optimization therefore reduces the resource consumption on all but one contexts of a house topology.

Figure 6.2.2 and 6.2.4 shows that the effectiveness of the OPT algorithm is not as high as on a ring topology. This is due to the mentioned reasons above and the fact that the optimization is, in contradiction to the house topology, effective on all contexts of a ring topology. Nevertheless, the median runtime is below 60% and the median memory consumption below 70% on all four tested house configurations.

6.2.2 Subset Pruning (SP)

Subset pruning is the reduction of a set of partial diagnoses which have the same witness to those partial diagnoses which are subset-minimal within this set (see Section 5.4.2). This can be done on the results of each context. So this optimization reduces the number of results which are send back and therefore reducing:

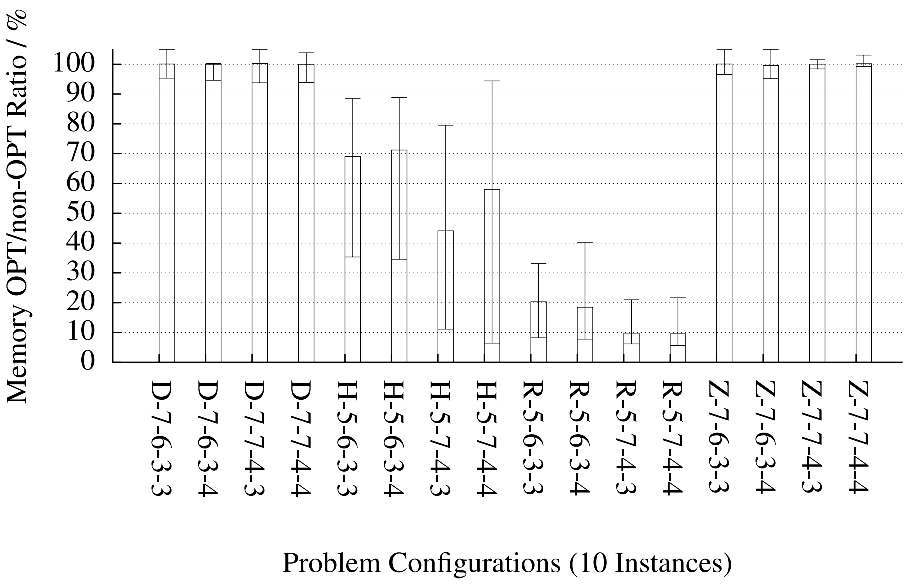
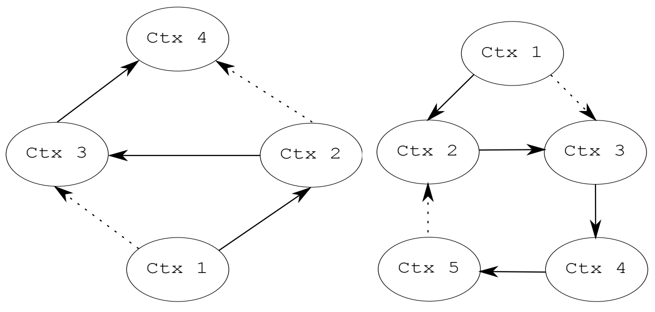


Figure 6.2.4: Memory Consumption of DMCS-DF-OPT System in Relation to DMCS-DF in % (Min, Median, Max)



(a) Zigzag Optimization (b) House Optimization

Figure 6.2.5: Graph Optimizations

- the resources used to send the results over the network,
- the number of joins between the results from different neighbors,
- the number of joins between the received results from the neighbors and the local solutions, and
- the effort of calculating the subset-minimal diagnoses at each context.

The only part of the calculation which is not influenced by this optimization is the solving of the local belief sets. The reason is that they are calculated based on all combinations of input atoms before they are joined with the results from the neighboring contexts (see Section 5.3.2).

To show the effect of the SP optimization, in this section the DMCS-DF system with the OPT and SP optimizations enabled (DMCS-DF-OPT-SP) is compared to the DMCS-DF system with only OPT enabled (DMCS-DF-OPT). The tests have been performed on different problem configurations and for each configuration ten instances have been tested. Only inconsistent instances have been created such that the results consist of non empty diagnoses. The problem instances have been chosen such that the system performs in a representative range regarding runtime and memory consumption. The graphs illustrating the results of this section show the minimum, maximum, and medium values of all ten instances of each problem configuration. The analyzed values are the number of results send back to the querying client program, the runtime of the whole system (from the start of the querying client program to the printing of the results), and summed memory consumption of all context daemons and the client program.

To give an impression on how much the SP optimization can reduce the number of intermediate results, Figure 6.2.6 shows the number of diagnosis/witness pairs which are send back to the querying client program. This is the last stage of processing before the overall subset-minimal results are calculated in the querying program. This is done independently of the SP optimization and therefore the number of results equal in the system with and without the optimization after this step.

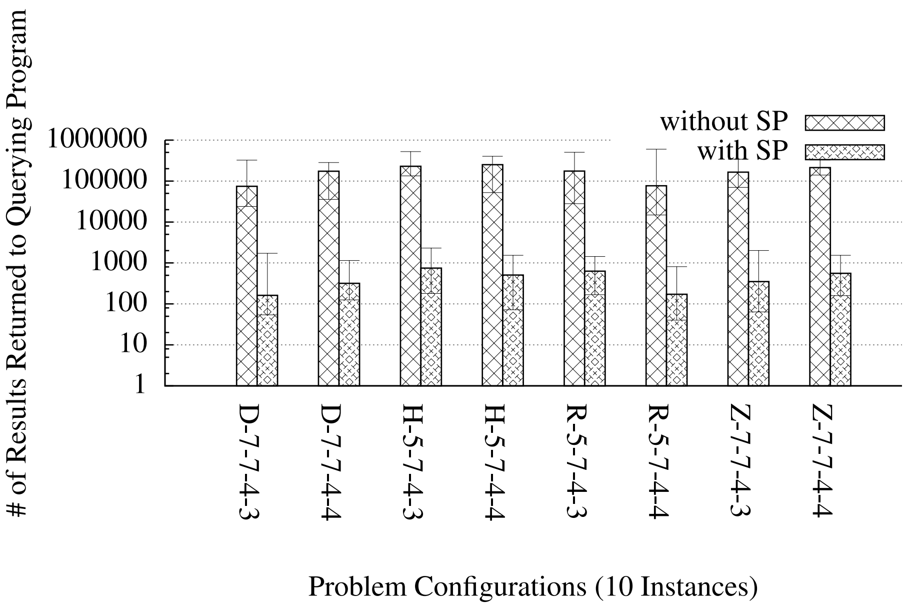


Figure 6.2.6: # of Results Returned to Querying Program with DMCS-DF-OPT vs. DMCS-DF-OPT-SP (Min, Median, Max)

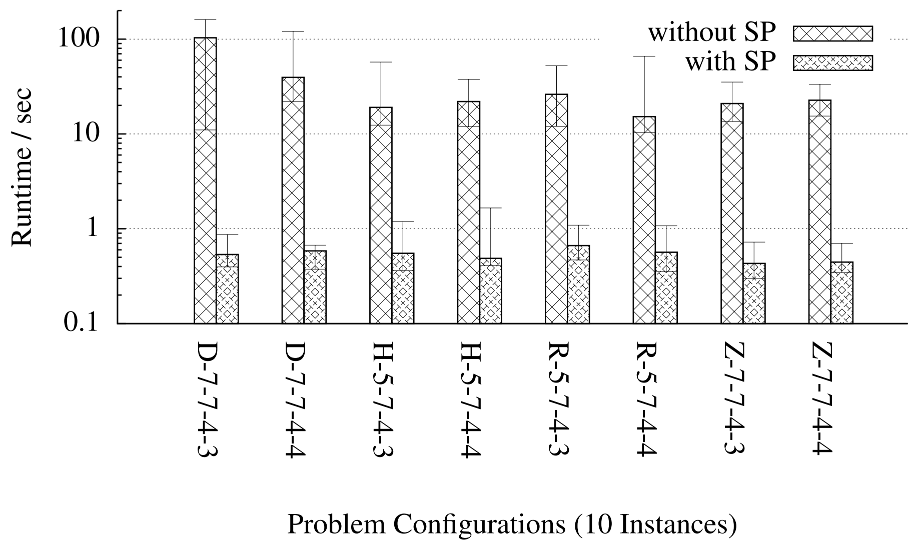


Figure 6.2.7: Runtime of DMCS-DF-OPT and DMCS-DF-OPT-SP in Seconds (Min, Median, Max)

On 95% of all 80 tested instances (10 per configuration) the number of returned diagnosis/witness pairs with SP optimization enabled is below 1% of the number of results without SP. On the other 5% of the instances the number of results drops below 2.5%.

Table 6.2.1 shows the effect of the SP optimization on the runtime of the system. This is the time between sending the query and just before printing the results. The first three columns show the runtime of the DMCD-DF-OPT-SP system in relation to the DMCS-DF-OPT system. The values are the minimum, median, and maximum value over all ten tested instances per configuration. The next two columns show the runtime in seconds of the DMCS-DF-OPT and the DMCS-DF-OPT-SP system. Both are the median over all ten instances of each configuration. Over all tested instances the SP optimization *reduces the runtime* between 90% and 99.66% concluding that the optimization performs well on all tested topologies.

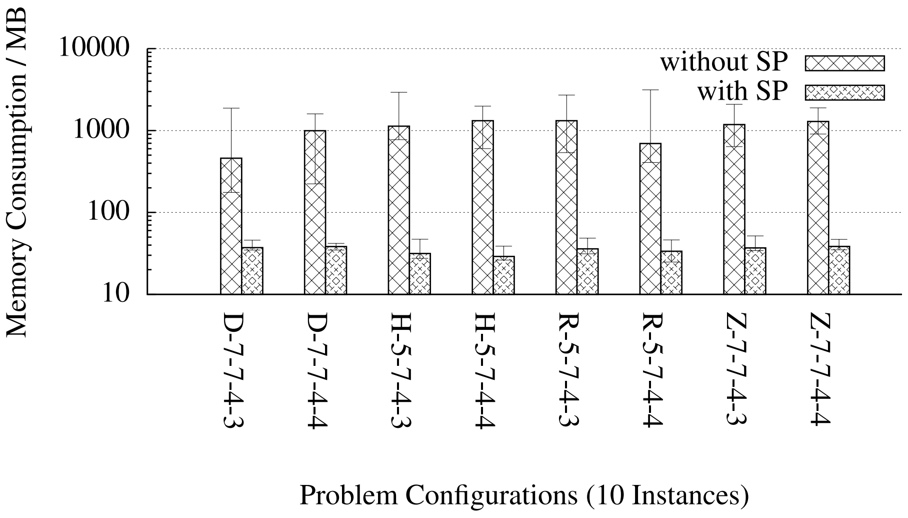


Figure 6.2.8: Memory Consumption of DMCS-DF-OPT and DMCS-DF-OPT-SP in MB (Min, Median, Max)

	SP in Relation to non-SP			Absolute Runtime	
Problem Configuration	**Min**	**Median**	**Max**	**Median non-SP**	**Median SP**
D-7-7-4-3	0.34%	0.64%	3.59%	103.47s	0.53s
D-7-7-4-4	0.50%	1.51%	2.48%	39.45s	0.58s
H-5-7-4-3	1.72%	2.78%	3.77%	19.05s	0.55s
H-5-7-4-4	1.66%	2.55%	4.80%	21.94s	0.49s
R-5-7-4-3	1.42%	2.61%	4.66%	26.21s	0.66s
R-5-7-4-4	1.23%	3.13%	7.55%	15.23s	0.57s
Z-7-7-4-3	1.16%	2.26%	3.25%	20.92s	0.43s
Z-7-7-4-4	1.31%	2.06%	2.63%	22.70s	0.45s

Table 6.2.1: Reduction using Subset Pruning (DMCS-DF-OPT vs. DMCS-DF-OPT-SP System)

The median of the runtime of both tested systems for each configuration is also shown in Figure 6.2.7 to give a visual impression of the runtime reduction in seconds. Note the logarithmic scale of the graph.

The SP optimization not only reduces the runtime of the system but also the memory consumption. Figure 6.2.8 shows the sum of the memory usage of each context and the querying program. As with the runtime the SP optimization reduces the memory consumption of the system significantly. The memory consumption of all tested instances, including the one with a memory usage up to 3GB, can be reduced to about 50MB and below.

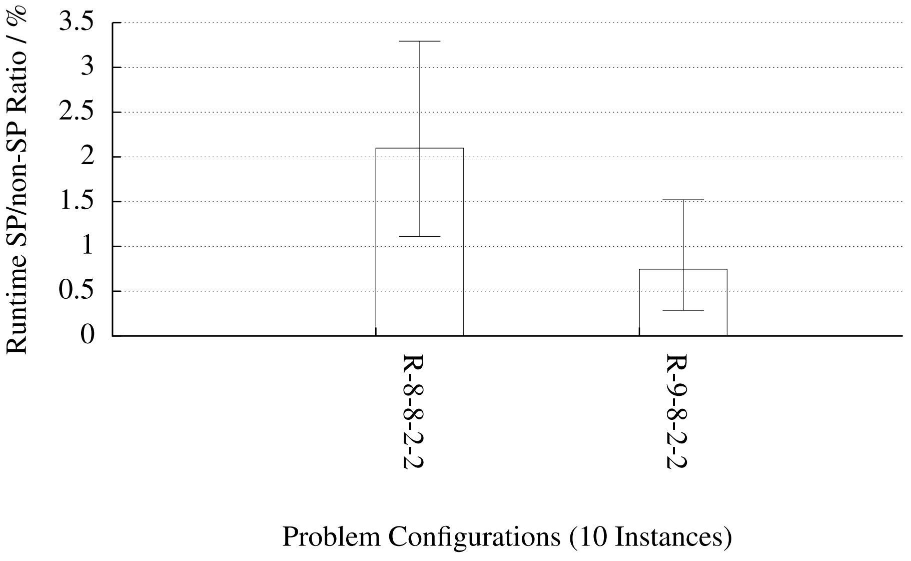


Figure 6.2.9: Upscaling Comparison of DMCS-DF-OPT in Relation to DMCS-DF-OPT-SP Runtime on Ring Topologies in % (Min, Median, Max)

Upscaling in Form of Larger Systems. The SP optimization is the more efficient the more contexts a system has. Assuming that the contexts in an MCS have a fixed number of bridge rules, the number of possible diagnoses increase with the number of contexts. This is because diagnoses are, speaking in simple terms, a subset of the power set of all bridge rules and a power set has trivially more elements if the set itself is larger. So a subset-minimal diagnosis is subset-minimal out of a larger set of diagnoses leading to the fact that the SP optimization drops more diagnoses. The conclusion from this assertion is that the SP optimization is the more efficient the more contexts are in the MCS.

Figure 6.2.9 shows the runtime of the DMCS-DF-OPT-SP system in percentiles of the runtime of the DMCS-DF-OPT system. Since ring topology configurations can have an arbitrary number of contexts two such configurations have been chosen for the experiment. They have both the same parameters except the number of contexts which differs by one context. The figure shows the minimum, median, and maximum value of ten instances which are tested for each configuration. The runtime of the 8 context configuration can be reduced by the SP optimization to 2.10% (median) of the runtime without optimization and the runtime of the 9 context configuration down to 0.75% (median). This indicates that the SP optimization is significantly more effective on configurations with more contexts.

6.3 Comparison DMCS vs. DMCS-DF-OPT-SP

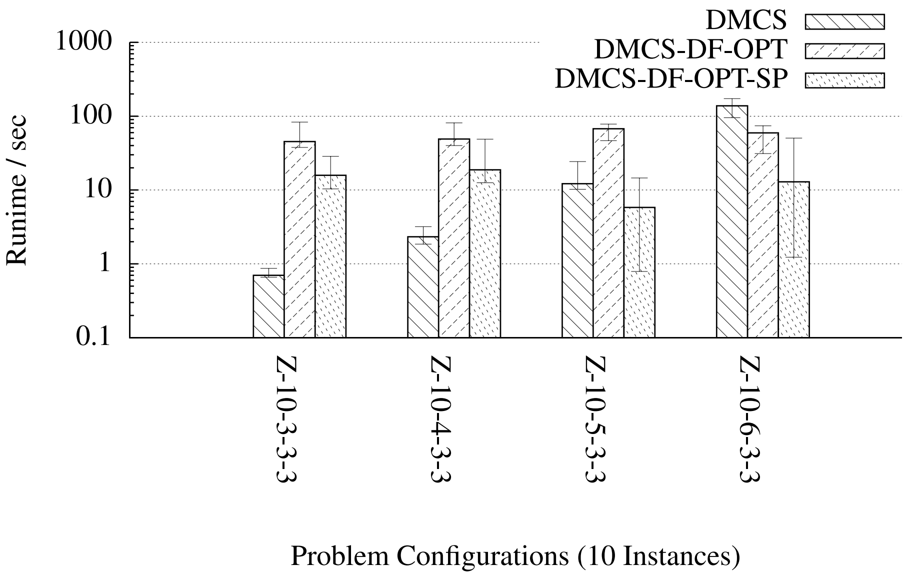


Figure 6.3.1: Runtime of DMCS and DMCS-DF-OPT-SP for Zigzag Configurations (Min, Median, Max)

Now we show the performance of the DMCS-DF system in finding equilibria for consistent $MCSs$, e.g., without diagnoses. The results are compared to the DMCS system running in default mode.

The DMCS system has been enhanced with an optimization described by Bairakdar et al. [4]. But as a consequence of the optimization the results are not comparable with the DMCS-DF system. The optimization, named *refined recursive import*, cuts information from intermediate results which are not necessary for local calculations at the initially requested context. This reduces the amount of information transmitted between the contexts. Therefore the results are not covering the whole import closure of the requested context, in contrast to the DMCS-DF system which returns results defined in all contexts of the import closure. Therefore we compare the DMCS-DF system with the DCMS system in default mode which returns fully defined equilibria like the DMCS-DF system.

Due to our implementation of the mentioned optimizations for the DMCS-DF system we sometimes outperform the DMCS system even though we calculate additional information. To get a clearer picture on the performance differences between the two systems, it would be better to compare the DMCS-DF system to the DMCS system with a version of OPT which returns equilibria defined in the whole import closure. Such an extension of the DMCS system is left for future work.

Since both tested systems have a different order of joining results from neighbors and joining them with local results, they have a different scaling behavior. Therefore experiments with zigzag problem configurations differing in the number of atoms per context are done. Then an explanation for a deeper understanding of this difference is given including a listing of the resource usage of distinct parts of the DMCS-DF algorithm. The performance of the DMCS-DF system is then emphasized by experiments with house topologies. From the given explanation about the different performance of the systems we can conclude that ring topologies are better solved by the DMCS system due to not existing neighbor combinations. We verify this in an additional experiment.

We now compare the DMCS and the DMCS-DF-OPT-SP system on four zigzag configurations. All configurations have the same number of contexts, interface atoms, and bridge rules. They only differ in the number of atoms per context. Figure 6.3.1 shows the minimum, median, and maximum value of the runtime in seconds. It can be seen that both systems scale quite differently with a different number of atoms per context. On the DMCS system the runtime increases with more atoms per context, whereas the runtime of the DMCS-DF-OPT-SP system has no obvious relation to the number of atoms per context. The additional data row from the DMCS-DF-OPT system, e.g., without the SP optimization, shows that the SP optimization has a positive effect on the runtime but does not explain the different scaling behavior between the DMCS and the DMCS-DF-OPT-SP system.

	Relative Runtime / %		
Task	**Minimum**	**Median**	**Maximum**
Local Solve	1.53%	4.75%	49.62%
Joining of Neighbor Results	0.64%	1.70%	3.29%
Calculation of Subset-Minimal DG	7.02%	14.88%	20.09%
Serializing	21.05%	37.27%	57.08%
Joining of Local Results	16.84%	33.74%	46.94%
Rest	2.89%	4.62%	7.17%

Table 6.3.1: Runtime Proportion of Subtasks of the DMCS-DF-OPT-SP System With ZigZag Problem Instances in Relation to the Total Runtime (Minimum, Medium, Maximum).

Next we show which tasks of the DMCS-DF-OPT-SP system are mainly responsible for the system runtime. Then we compare the implementation of these tasks with the implementation of the DMCS system and look for explanations of the different scaling behavior.

Table 6.3.1 shows that the local solve part is about 5% of the total runtime (with spikes to 50%). Therefore a higher number of local solve operations with the DMCS-DF system due to diagnosis guesses has generally a minor effect on the total runtime. The calculation of subset-minimal diagnoses is only necessary with the DMCS-DF system and can therefore not be responsible for longer runtimes of the DMCS system. The number of results sent between the contexts is only reduced by the SP optimization. Since the better scaling performance of the DCMS-DF system with more atoms per context is also visible without the SP optimization, the serialization can also be excluded from the list of possible responsible tasks. So the combination operations are left as a task with a high proportion of the total time. A major difference between the DCMS and the DMCS-DF systems is the order in which results are joined. The DMCS system first calculates the local belief sets and then requests the neighbors and immediately joins the received results with the local belief sets. In contrast the DMCS-DF system first joins the results received from the neighbors and then joins them with the local belief sets. A higher number of atoms per context increases the probability of a higher number of local belief sets. Since local belief sets are earlier involved in the combination process of the DMCS system a higher number of atoms has a higher impact on the processing time with the DMCS system.

The difference in implementation of the DMCS and the DMCS-DF system has not only an effect on the runtime but also on the memory consumption of the systems. Figure 6.3.2 shows the minimum, median, and maximum sum of the daemons and client program memory consumption. The graph shows that the memory consumption of the DMCS-DF-OPT-SP is higher than the one from the DMCS system. However, DMCS-DF-OPT-SP scales better with larger contexts.

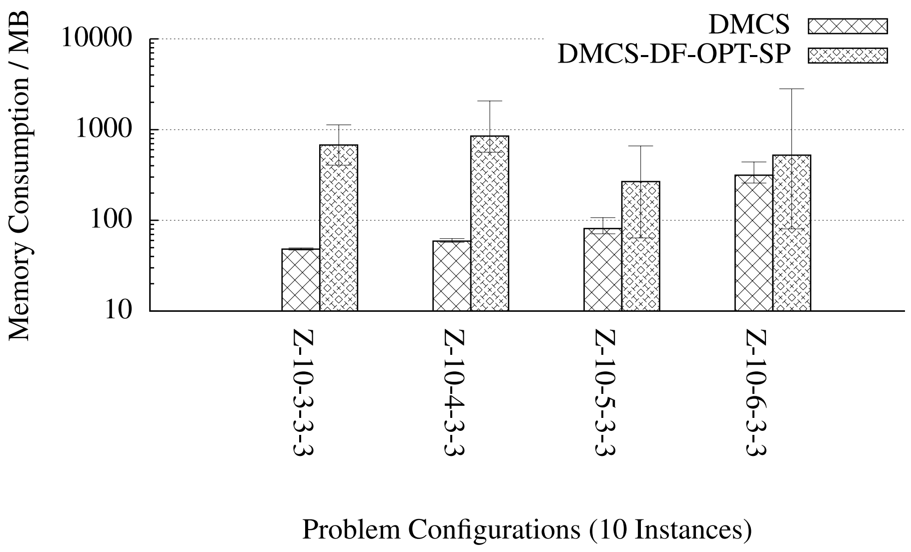


Figure 6.3.2: Memory Consumption of the DMCS and DMCS-DF-OPT-SP system in MB with Zigzag Instances (Minimum, Median, Maximum)

To show that the DMCS-DF-OPT-SP system has also good performance on other topologies, ten house topology instances have been tested. Figure 6.3.3 shows that the DMCS-DF-OPT-SP system has a better time performance on all instances. The graph shows also that the time gain is quite different on different instances but cause less than 1% of DMCS-DF-OPT-SP runtime compared to the DMCS system.

Before we concluded that the good performance of the DMCS-DF-OPT-SP system com-

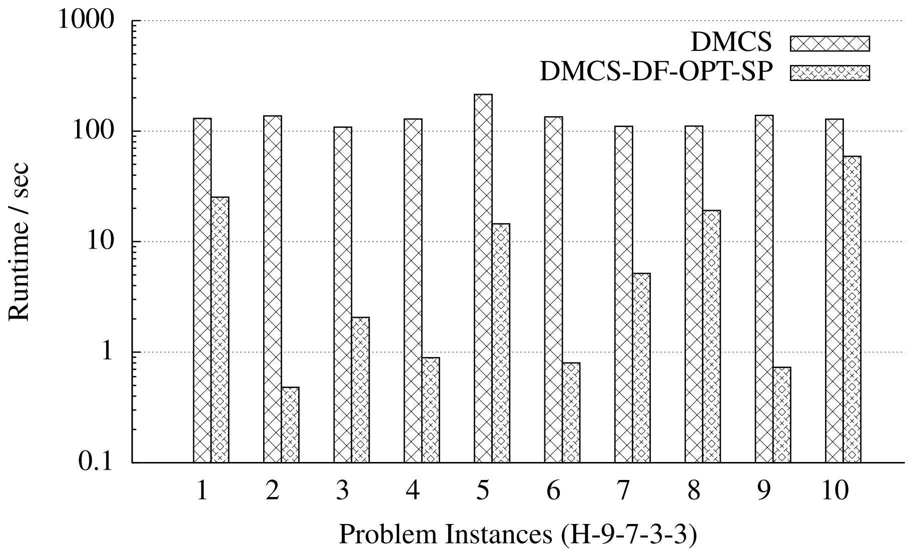


Figure 6.3.3: Runtime of DMCS and DMCS-DF-OPT-SP with Individual House Instances

pared to the DMCS system must be due to the order of join operations. This difference is only of relevance if there are multiple neighboring contexts whose results are joined together. Otherwise there are no join operations between the results of the neighbors and thus the DMCS and DMCS-DF system have the same order of joins. Therefore the DMCS system must have a better performance on ring topologies because on such topologies every context has exactly one neighboring context. Figure 6.3.4 shows the results of tests with ring topologies. Three configurations with a different number of contexts have been tested w.r.t. the runtime of the DMCS and the DMCS-DF system. The results confirm that the DMCS system performs better on ring topologies.

Summary. The experiments have shown that the DMCS-DF-OPT-SP system calculates partial equilibria in consistent MCS with good performance and has a better runtime compared to the DMCS reference program on some zigzag and house topology configurations. Whereas the DMCS system runtime and memory consumption increases with the number of atoms the DMCS-DF-OPT-SP system has no such correlation. Further investigations of the performance of the sub systems of the DMCS-DF-OPT-SP system let us conclude that this scaling behavior has its cause in the different order of joining operations on both systems. This also explains why the DMCS-DF-OPT-SP system performs better in comparisons with zigzag and house topologies which have multiple neighbor relations.

Due to different optimizations implemented in both systems it can not be said that the DMCS-DF-OPT-SP system has a better performance in general. The DMCS system has a better

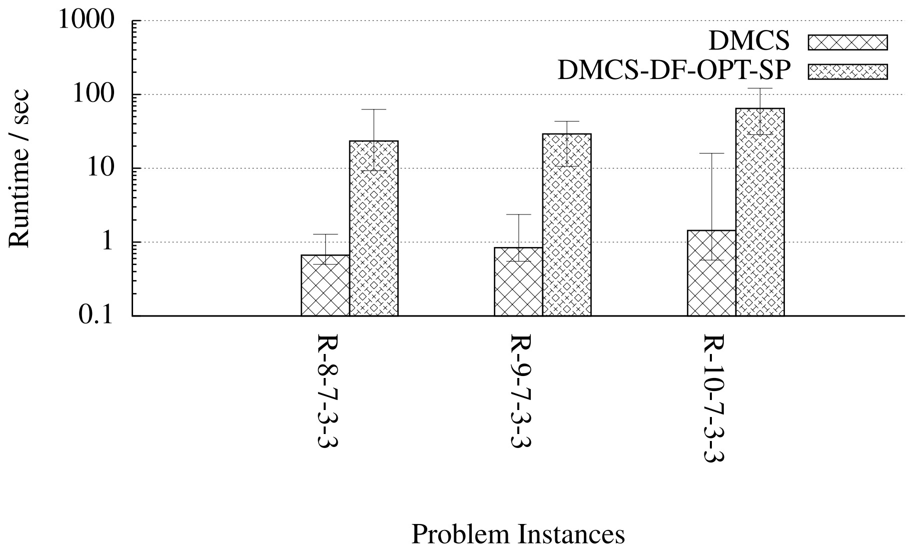


Figure 6.3.4: Runtime of DMCS and DMCS-DF-OPT-SP Ring Configurations (Minimum, Median, Maximum)

runtime, e.g., with ring topologies. Implementing a variety of OPT that still returns full equilibria in DMCS would allow for a better comparison of methods. We conjecture that in this case DMCS would be consistently more efficient as it simply performs a subset of the operations of DMCS-DF-OPT-SP.

6.4 Summary

We can conclude that both optimizations have the effects which we expected. Which is for the edge pruning optimization that there is no observable effect on diamond and zigzag topographies which have no cycles but up to a 90% reduction in computing time and memory consumption for ring topologies and still up to 30% reduction on house topologies. The difference can be explained by the fact that all contexts of a ring topology are part of a cycle (compared to the house topology where only a part of the contexts are in a cycle) and therefore the optimization can effect the whole MCS.

For the subset pruning we could show that the number of results which are returned can be pushed below 1% of the number without the optimization on over 95% of all tested instances and over all tested topologies. Therefore also the runtime and the memory consumption was reduced drastically over all topologies. We can further conclude that the effect of this optimization increases with larger instances.

The comparison between the DMCS and our system showed us that the DMCS system reduces its performance with larger contexts while our system can keep its performance due to the late involvement of the local belief set in the joining process. This finding could be backed up by showing that the joining of the requested results uses much more time than the local solve operation in our system. Generally we can conclude that our system can compete with the DMCS system in solving equilibria and has a slightly better performance on some house and zigzag topologies.

CHAPTER 7

Conclusion

We have approached the problem of inconsistencies in Multi-Context Systems with the notion of diagnoses as used by Eiter et al. [16] and modified them such that they can be computed in a distributed manner. Dao-Tran et al. [14] have done this already for equilibria and as well developed an algorithm calculating them. Inspired by this we introduced partial diagnosis which are diagnosis solving the inconsistencies of a defined part of the MCS. If a user is interested in a specific context the according partial diagnoses of this context span by definition over this part of the MCS which is necessary to fix all inconsistencies such that the user gets a consistent result for the context she is interested in. The algorithm we defined is distributed over the contexts and we reduced the information each instance requests from other instances such that the network utilization remains low and it is not necessary for each instance to reveal all local information.

We have shown that the developed algorithm is able to find all partial diagnoses and that all results which are returned are partial diagnosis. In addition to the partial diagnoses the algorithm returns also all partial equilibria resulting from the applied partial diagnoses.

With an actual implementation of the algorithm in C++ we were able to perform benchmarks on specific problem instances. The implementation was designed to return performance data and we build in useful debugging options such that it is possible to compare different problem instance classes and different algorithm optimizations.

Two optimizations have been included in the implementation. Edge Pruning which is reducing unnecessary request to contexts where all necessary information about this context has already been returned by another context. With this optimization the runtime and memory consumption on some specific problem instance classes has been reduced by 60%. The second optimization is subset pruning which has the effect of only returning subset-minimal diagnoses. On over 95% of all tested instances with this optimization the number of results has been reduced by 99% which goes also hand in hand with an significant reduction of memory usage and computation time.

Finally we can conclude that the implementation can also compete with the DMCS system which only calculates partial equilibria meaning that it has a better scaling characteristics on some problem instances. We therefore belief that this algorithm is a good basis for further developments in approaching inconsistencies in Multi-Context System.

7.1 Related Work

We will give a short overview of approaches to find and solve inconsistencies in a wide range of systems. Therefore not all approaches are easy to compare but nonetheless due to the general structure of the problem it may help to find new solutions by modifying and combining existing approaches.

The problem of inconsistencies in distributed, non-monotonic knowledge bases is not new and has, e.g., been approached by Bikakis and Antoniou who described $MCSs$ with defeasible rules [7], [8] and handle the problem of inconsistencies with a preference ordering. This means that each context has an individual ordered list of its information sources such that any inconsistency arising in any context can be solved locally.

Binas and McIlraith proposed a method for resolving inconsistencies in peer-to-peer systems using a preference relation on peers [9] which could express trust between peers and therefore support or weaken a consistent solution.

An approach called model-based diagnosis was introduced by Reiter [31] and also discussed w.r.t. nonmonotonic systems by Preist et al. [30]. Here a logical model for correct and faulty behavior of system components is introduced and used in combination with observations of a real system to detect the problem.

In order to restore consistency some approaches have been developed which are combining beliefs. One is belief revision [1] [29] where existing knowledge bases are updated with new beliefs and therefore may remove others and another is belief merging [23] where inconsistent beliefs are tried to overcome by combining them from different knowledge bases with the help of expressive mappings.

In other areas like information integration [25] similar approaches are studied. E.g., in the described Informix system different sources are materialized in one schema and by modifying this materialized information instead of the sources inconsistencies are resolved. This is somehow similar to our approach although their mapping formalisms are more expressive.

Another area which has to deal with inconsistencies is ontology mapping [13] where the challenge is to connect identical concepts, roles, or individuals from different, heterogeneous ontologies. This can also result in inconsistencies over all mapped ontologies and is normally avoided by leaving out the responsible mappings.

A different approach to deal with inconsistencies is depicted by Calvanese et al. and their peer-to-peer data integration [12]. Here queries can ignore parts of the system or belief which are only held by a minority of peers in the system. But in contrary to our systems the structure of their systems is changing dynamically by added or removed peers.

Other approaches for automatically solving inconsistencies can as well be found in data integration [20], consistent query answering in data bases [6], [15] and description logics [24] or by the use of abduction [21].

7.2 Future Work

First we want to point out possible improvements of the algorithm which would probably result in better performance:

If two different guesses result in the same belief set it may suffice to only use one guess for further calculations and store the second one attached to it and construct it at the end from the results of the first guess. This should be valid because the calculations on a specific context are only based on the belief sets of the neighboring contexts and not on the partial diagnoses or candidates of those contexts.

There are probably a lot of cases where the user is only interested in subset minimal diagnoses. In such cases it might be worth to investigate an iterative approach where first only the subset minimal diagnoses at all stages of the request chain are calculated without considering that this may lead later on to no result and if no witness results in a partial equilibrium a new request is send where all contexts provide the next larger set of diagnoses w.r.t. subset minimality. Such an approach will probably reduce the total amount of calculation time but as a downside increase network traffic.

As already pointed out in Section 5.3 the theoretical approach requests the partial diagnoses from the import neighborhood and calculates the local belief sets based on them whereas the actual implementation first guesses the atoms from the neighborhood and then joins this information with the results received from the requests. Both approaches have advantages and disadvantages and therefore it would be interesting to develop an algorithm combining both approaches and make use of the advantages of both of them.

There should also be some room for improvement by using more compact data structures. A lot of diagnosis/witness pairs have the same witnesses (of course depending on the problem instances) but are still calculated redundantly for each further context. A more intense use of pointers could help here to reduce the amount of unnecessary calculations.

We can also think about improvements or extensions to the whole theoretical system which would make it more general and therefore suitable for broader use cases. To give the user more information about the diagnoses and to help her to judge which diagnosis would be better to apply we can think about determining the influence of a diagnosis. This could be the number of contexts which are affected or some other measurements which keep track of the changes resulting from a diagnosis.

Bibliography

[1] Carlos E Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic*, 50 (02):510–530, 1985.

[2] Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Decomposition of distributed nonmonotonic multi-context systems. In Marc Denecker and Marina de Vos, editors, *Proceedings 15th International Workshop on Nonmonotonic Reasoning (NMR-2010), Declarative Programming Paradigms and Systems*, May 2010. `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/`.

[3] 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, *Proceedings 12th European Conference on Logics in Artificial Intelligence (JELIA 2010)*, LNCS, pages 352–355. Springer, 2010.

[4] 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, *Proceedings 12th European Conference on Logics in Artificial Intelligence (JELIA 2010)*, LNCS, pages 24–37. Springer, 2010. Preliminary version at NMR 2010 [2].

[5] Rosamaria Barilaro, Michael Fink, Francesco Ricca, and Giorgio Terracina. Towards query answering in relational multi-context systems. In Pedro Cabalar and TranCao Son, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 8148 of *Lecture Notes in Computer Science*, pages 168–173. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40563-1.

[6] Leopoldo Bertossi and Jan Chomicki. Query answering in inconsistent databases. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*, pages 43–83. Springer Berlin Heidelberg, 2004. ISBN 978-3-642-62248-9.

[7] Antonis Bikakis and Grigoris Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *Knowledge and Data Engineering, IEEE Transactions on*, 22(11): 1492–1506, 2010. ISSN 1041-4347.

[8] Antonis Bikakis, Grigoris Antoniou, and Panayiotis Hasapis. Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowledge and Information Systems*, 27 (1):45–84, 2011.

[9] Arnold Binas and Sheila A McIlraith. Peer-to-peer query answering with inconsistent knowledge. In *Proceedings on the 11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 329–339, 2008.

[10] Gerd Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings 22nd Conference on Artificial Intelligence (AAAI '07), July 22-26, 2007, Vancouver*, pages 385–390. AAAI Press, 2007.

[11] Gerhard Brewka, Floris Roelofsen, and Luciano Serafini. Contextual default reasoning. In Manuela M. Veloso, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 268–273, 2007.

[12] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Inconsistency tolerance in p2p data integration: an epistemic logic approach. In *Database Programming Languages*, pages 90–105. Springer, 2005.

[13] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *ACM Sigmod Record*, 35(3):34–41, 2006.

[14] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Distributed nonmonotonic multi-context systems. In *Proceedings 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), May 9-13, 2010, Toronto, Canada*, pages 60–70, 2010.

[15] Thomas Eiter, Michael Fink, Gianluigi Greco, and Domenico Lembo. Repair localization for query answering from inconsistent databases. *ACM Transactions on Database Systems (TODS)*, 33(2):10, 2008.

[16] Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Finding explanations of inconsistency in multi-context systems. In *Proceedings 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), May 9-13, 2010, Toronto, Canada*, pages 329–339, 2010.

[17] Esra Erdem, Yelda Erdem, Halit Erdogan, and Umut Öztok. Finding answers and generating explanations for complex biomedical queries. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[18] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. ISSN 0288-3635. URL `http://dx.doi.org/10.1007/BF03037169`. 10.1007/BF03037169.

[19] Fausto Giunchiglia. Contextual reasoning. *Epistemologia, special issue on I Linguaggi e le Macchine*, 16:345–364, 1993.

[20] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, pages 9–16. VLDB Endowment, 2006.

[21] Katsumi Inoue and Chiaki Sakama. Abductive framework for nonmonotonic theory change. In *IJCAI*, volume 95, pages 204–210. Citeseer, 1995.

[22] U.S.R. Murty J.A. Bondy. *Graph Theory*. Springer, 2008.

[23] Sébastien Konieczny and Ramón Pino Pérez. Propositional belief base merging or how to merge beliefs/goals coming from several sources and some links with social choice theory. *European Journal of Operational Research*, 160(3):785–802, 2005.

[24] Domenico Lembo and Marco Ruzzi. Consistent query answering over description logic ontologies. In Massimo Marchiori, Jeff Z. Pan, and Christian de Sainte Marie, editors, *Web Reasoning and Rule Systems*, volume 4524 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-72981-5.

[25] Nicola Leone, Gianluigi Greco, Giovambattista Ianni, Vincenzino Lio, Giorgio Terracina, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Riccardo Rosati, et al. The infomix system for advanced integration of incomplete and inconsistent data. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 915–917. ACM, 2005.

[26] Bas Huijbrechts Jan Laarhuis Jesper Hoeksma Steffen Michels Marina Velikova, Peter Novák. An integrated reconfigurable system for maritime situational awareness. *Frontiers in Artificial Intelligence and Applications*, 263:1197 – 1202, 2014.

[27] John McCarthy. Generality in artificial intelligence. *Communications of the ACM*, 30(12): 1030–1035, 1987.

[28] John McCarthy. Notes on formalizing context. In Ruzena Bajcsy, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 555–562, 1993.

[29] Pavlos Peppas. Belief revision. *Foundations of Artificial Intelligence*, 3:317–359, 2008.

[30] Chris Preist, Kave Eshghi, and Bruno Bertolino. Consistency-based and abductive diagnoses as generalised stable models. *Annals of Mathematics and Artificial Intelligence*, 11 (1-4):51–74, 1994.

[31] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1): 57–95, 1987.

[32] Floris Roelofsen and Luciano Serafini. Minimal and absent information in contexts. In *IJCAI*, volume 5, pages 558–563, 2005.