FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Inconsistency in Multi-Context Systems: Analysis and Efficient Evaluation

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Peter Schüller
Matrikelnummer 0125596

an der
Fakultät für Informatik der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

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

(Prof. Dr. Giovambattista Ianni)

Wien, 8.8.2012

(Peter Schüller)

Erklärung zur Verfassung der Arbeit

Peter Schüller
Wehlistraße 326/609, 1020 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)

To my parents, Monika and Werner.
To my sisters and brothers,
Michael, Elisabeth, Katharina, and Andreas.

Acknowledgments

First of all I want to thank my dissertation advisors, Thomas Eiter and Michael Fink, for making this thesis project possible through funding and a lot of their individual personal effort. Thomas Eiter was a great and very efficient advisor. He has as much time as every other person has, but much more responsibilities than most other persons; nevertheless he often took the effort to explain the content and form of his contributions in papers to me, which gave me a great possibility to improve my skills. Michael Fink taught me a lot about considering a research problem or a solution to a research problem from many different perspectives and in much detail. Furthermore he did a great job teaching me the fine art of scientific political correctness, both with respect to reviews and with respect to authoring publications. Finally this thesis would not exist without Michael's successful acquisition of the project grant which funded my studies and conference visits.

Next I want to thank Antonius Weinzierl, who was a very likable and supportive colleague during the course of this project. In particular he always listened to my ideas, understood them fast, often rightfully questioned them, and he allowed me to do the same with his ideas.

Major thanks go to the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) who funded the research that is reflected in this thesis under project grant ICT08-020.

It is impossible to name all persons who inspired me during my PhD, so I will thank some of them specifically and in alphabetical order. Esra Erdem did a great job of asking me the right—often tough—questions to challenge me and push my mind forward to look for even tougher questions. I am thankful to Georg Gottlob who is a very helpful and kind personality; he held a lecture on research and career planning, which was one of the most useful lectures I ever attended. Giovambattista Ianni was very supportive of my ideas, which were often immature and unstructured; he contributed a lot to transforming these ideas into a more understandable format. Yuliya Lierler collaborated with me over a great distance and time difference, working with her was a great pleasure and she became more than a collaborator to me, she became a friend, a person I truly appreciate, and I want to thank her very much for that. Thomas Krennwallner was a great person to work with during my PhD studies, he never got tired of creating, discussing, and re-discussing various scientific as well as nonscientific topics. Torsten Schaub and his team in Potsdam provide great open source software and supported me during my thesis by reacting promptly to bug reports and other inquiries. I want to thank the Knowledge Based Systems group at TU Wien, and several people not belonging to that group, for being a bright and very friendly community which made my PhD a very enjoyable experience. Many thanks also go to our secretary Eva Nedoma and to our technician Matthias Schlögel, for smoothly assisting with every administrative issue I encountered.

I thank my friends for distracting me and showing me that life is more than work, particular thanks go to Anna, Brumma, Carmen, Dagmar, Edda, Kathi, Peter, Phlo, Sabine, and Vali.

Last but not least I am deeply grateful to my parents and to my family, who always had an open ear when I talked about my work and my worries, and who always were supportive of my career choice even though it meant that I sacrificed a lot of time and effort for that career, time and effort I could have spent on them.

Abstract

Multi-Context Systems are a formalism for representing systems that consist of multiple knowledge-based systems (contexts). Knowledge exchange between contexts is represented in the form of bridge rules, which are rules that add formulas to knowledge bases if certain beliefs are accepted (or not accepted) at certain contexts. As an example, we consider a hospital information system that links patient and laboratory databases with a decision support system for suggesting therapies. Inconsistency of such a system can easily arise due to unexpected interactions caused by bridge rules and contexts, for example a wrong birth date could make the lab database inconsistent, and a patient allergy might eliminate all treatment options for a patient, making the global system inconsistent. As a consequence, the system can no longer provide meaningful answers to requests. Existing approaches for dealing with inconsistency mostly ignore or remove inconsistency or they regard inconsistency as a special truth value in the system.

In this thesis, we aim at a new approach that allows for analyzing inconsistency and reasoning about it, and eventually allows for repairing inconsistency with or without support from a human operator. To facilitate practical applicability, we put a particular emphasis on efficiency. We use formal methods of computer science, in particular from logic programming, to investigate these problems on a theoretical level. We realize formal results in a software prototype, conduct empirical experiments to validate efficiency, and investigate potential optimizations.

As a main theoretical result we obtain two novel formal notions for analyzing inconsistency in multi-context systems: diagnoses and inconsistency explanation. These notions characterize inconsistency in terms of sets of bridge rules that cause inconsistency. Diagnoses provide possible repairs of the system, while inconsistency explanations separate independent reasons for inconsistency in one system. We establish formal relationships between these notions, and show how they can be approximated if parts of the system are hidden from our observations. Furthermore we describe a novel policy language for managing inconsistency in multi-context systems (IMPL) which allows for analyzing inconsistencies and for specifying in a declarative way which inconsistencies shall be repaired automatically (e.g., by ignoring a slightly differing birth date) and which shall be presented to a human user (e.g., a doctor, in case of the allergy mentioned above) for deciding how to deal with them.

A main empirical result of this thesis is the realization of our inconsistency analysis notions in the MCS-IE prototype software, which uses HEX programs—a formalism that extends logic programming with external computations—as underlying formalism. To improve scalability of MCS-IE and of HEX, we introduce a novel flexible evaluation framework for HEX programs, formally show its correctness, and describe an empirical study on the performance benefits compared to the previous state-of-the-art in HEX evaluation.

Using our inconsistency management methods it is possible to make operation of distributed knowledge-based systems more robust. Our improved HEX evaluation formalism widens applicability of HEX for practical reasoning tasks and prompts additional research on the subject of efficient evaluation of logic programs.

Kurzfassung

Multi-Kontext-Systeme sind ein Formalismus zur Repräsentation von Systemen, die aus mehreren wissensbasierten Systemen (Kontexten) bestehen. Wissensaustausch zwischen Kontexten wird in Form von Regeln (Bridge Rules) modelliert; diese Regeln fügen Formeln zur Wissensbasis eines Kontexts hinzu, wenn bestimmte Aussagen von anderen Wissensbasen akzeptiert werden (oder nicht akzeptiert werden). Als Beispiel betrachten wir ein Spitalsinformationssystem, welches Patienten- und Labordatenbanken sowie ein Expertensystem für Behandlungsvorschläge vernetzt. In einem derartigen wissensverarbeitenden System kommt es durch unerwartete Interaktionen zwischen Kontexten und Regeln leicht zu einer Inkonsistenz des Gesamtsystems. Beispielsweise kann ein PatientInnendaten-Import dem Inhalt der Labordatenbank widersprechen, oder eine PatientInnenallergie kann sämtliche Behandlungsmöglichkeiten ausschließen. Globale Inkonsistenz eines Systems hat zur Folge, dass keine sinnvollen Informationsabfragen an das System mehr möglich sind. Bestehende Verfahren zur Inkonsistenzbehandlung ignorieren oder entfernen Inkonsistenz oder sie betrachten Inkonsistenz als spezifischen Wahrheitswert im System.

In dieser Arbeit erörtern wir einen neuen Ansatz, der Inkonsistenzen analysiert, automatisches Schlussfolgern über dieser Analyse zulässt, und letztendlich die Möglichkeit bietet, automatische oder benutzergestützte Systemreparaturen durchzuführen. Vor dem Hintergrund der praktische Anwendbarkeit unserer Verfahren betrachten und bewerten wir stets deren Effizienz. Wir untersuchen unsere Ansätze auf theoretischer Ebene mithilfe von formalen Methoden der Informatik, im Speziellen aus dem Gebiet der Logikorientierten Programmierung. Wir realisieren unsere Verfahren in Software Prototypen, führen empirische Untersuchungen durch um die Effizienz dieser Prototypen zu messen, und untersuchen mögliche Optimierungen.

Auf theoretischer Ebene ist das Hauptergebnis dieser Arbeit die Einführung von zwei neuen formalen Begriffen für die Analyse von Inkonsistenzen in Multi-Kontext-Systemen: Diagnose und Inkonsistenzerklärung. Diese Begriffe charakterisieren Inkonsistenz über Mengen von Bridge Rules, welche die Inkonsistenz verursachen. Diagnosen zeigen mögliche Reparaturen des Systems auf, während Inkonsistenzerklärungen unabhängige Inkonsistenzen in einem System voneinander abgrenzen. Wir zeigen formale Eigenschaften dieser neu eingeführten Begriffe und stellen ein Verfahren vor, das es ermöglicht, sogar bei unvollständiger Information über ein System dessen Inkonsistenz zu analysieren. Weiters beschreiben wir die neue Sprache IMPL zur Spezifikation von Strategien für Inkonsistenzmanagement von Multi-Kontext-Systemen. Diese Sprache erlaubt zu spezifizieren, welche Inkonsistenzen auf Basis einer Inkonsistenzanalyse automatisch repariert werden sollen (beispielsweise durch Ignorieren eines geringfügig abweichenden Geburtsdatums) und welche Inkonsistenzen einem menschlichen Benutzer (beispielsweise einem Arzt) vorgelegt werden müssen, um über deren Reparatur zu entscheiden.

Ein wichtiges empirisches Ergebnis dieser Arbeit ist die Realisierung der Inkonsistenzanalysebegriffe im MCS-IE Software Prototypen, der Diagnosen und Inkonsistenzerklärungen auf der Basis von HEX-Programmen—ein Formalismus der logische Programmierung mit externen Berechnungen erweitert—berechnet. Um die Skalierbarkeit von MCS-IE und HEX zu verbessern, erstellen wir ein neuartiges Berechnungsframework für HEX-Programme. Wir führen dieses Framework formal ein, beweisen dessen Korrektheit, und beschreiben eine empirische

Untersuchung, die zeigt, dass unser neuen Berechnungsframework Effizienzvorteile gegenüber dem vorher existierenden Verfahren aufweist.

Die Inkonsistenz-Management-Methoden, welche wir in dieser Arbeit vorstellen, ermöglichen es, den Betrieb von verteilten wissensbasierten Systemen robuster zu machen. Unser verbessertes Verfahren zur Evaluierung von HEX-Programmen erweitert die Anwendbarkeit von HEX in der Praxis und wirft neue wissenschaftliche Fragestellungen im Bereich der effizienten Auswertung von logischen Programmen auf.

Contents

1 Introduction

Combining different sources of knowledge is an important ability of human beings. We regularly use this ability to solve our day-to-day problems, and interdisciplinary science is even based on the idea of reusing knowledge from one area of expertise and applying it to another area to solve challenging problems.

In this thesis we consider the combination of different sources of knowledge in computer systems. At least since the common availability of the internet, such combination of digital knowledge has been of great interest and nowadays the interlinking of knowledge is a permanent reality in our highly digital culture.

In 2007, Brewka and Eiter introduced the formalism called heterogeneous nonmonotonic multi-context systems (in short MCSs) [BE07]. The MCS formalism mathematically describes a collection of knowledge based systems, called contexts, and a collection of rules which transmit knowledge between contexts, called bridge rules. This formalism has three very important properties which, put together, distinguish MCSs from other approaches of interlinking knowledge:

1. each context in an MCS may use a different mathematical formalism for representing its internal knowledge, i.e., MCSs represent *heterogeneous* systems and can combine knowledge from heterogeneous systems,

2. gaining additional knowledge may invalidate existing inferences, i.e., MCSs represent *nonmonotonic* systems.[1] Nonmonotonicity is supported both in context and in bridge rules. Furthermore,

3. MCSs support *information hiding*, i.e., not all knowledge in one context is exposed to another context.

These features make MCSs a powerful tool for real-world applications. At the same time, the power of this framework creates difficult mathematical and computational challenges, which will be the recurring theme of this thesis.

As an example, consider the following hospital information MCS which comprises the following contexts: (i) a patient database, (ii) a database of the medical laboratory, (iii) an ontology which represents knowledge about diseases and how they are related to symptoms and laboratory results, and (iv) a decision support system which suggests therapies to a medical practitioner. The MCS links information from the patient and laboratory database to classify the diseases of patients. The disease classifications and information from the patient database is used by the decision support system to suggest treatment options for the patient. The contexts in this system are heterogeneous, as they are typically realized in different knowledge representation formalisms. Furthermore they are hiding information, as not all knowledge stored in one system

[1]The standard example for nonmonotonic reasoning is that, given the information that Tweedy is a bird, we infer that Tweedy can fly. However, as soon as we learn that Tweedy is a penguin, we can no longer infer that, because Penguins do not obey the default assumption that birds can fly.

part should be transmitted to other system parts. Finally it should be nonmonotonic, for example adding new information about a patient's allergy into the patient database might suppress certain therapy suggestions by the decision support system.

The flexibility of the MCS formalism incurs two omnipresent challenges:

1. a high potential for *inconsistency* in the system, and

2. high *computational complexity* of reasoning in the system.

We define *inconsistency* in a MCS based on model-based equilibrium semantics of MCSs: a MCS is inconsistent if it has *no equilibrium model*. This implies that an inconsistent MCS cannot provide useful answers to queries; in our example the system would not provide any output and give no justification why there is no output.

Inconsistencies can appear without an obvious reason. Even if we make only a small change to the system and this is the obvious cause for inconsistency, the inconsistency might manifest in a different place than the change which caused it. In our example, additional knowledge in the patient database (e.g., about an allergy) can make the MCS inconsistent because suddenly the decision support system can no longer identify any possible therapy option. A doctor then would not gain any information from the system, except probably for the fact that it 'does not work' because it does not provide any output.

Computational complexity of reasoning describes the relationship between the size of a MCS plus a query to the MCS, and the resources in terms of computation time or storage space that are required to compute an answer to this query. Systems must be reasonably fast, otherwise they simply will not be used in practice. Therefore every idea in this thesis will be evaluated not only with respect to its usefulness and correctness, but also with respect to its computational complexity. In fact a large part of this work is about improving the efficiency of an algorithm for heterogeneous reasoning.

Most existing approaches for dealing with inconsistency repair a system in order to restore consistency, where repairing means to ignore just as much knowledge in the system as is required to restore consistency.[2] In our example, such an approach can be dangerous in several ways: ignoring the allergy might cause harm to the patient by treatment with a drug that triggers the allergy, ignoring the illness might cause harm by not treating the patient at all, or by treating them in the wrong way. Because they can be dangerous, we consider purely automatic approaches for restoring consistency to be undesirable.

In this thesis we will pursue the idea of *managing inconsistency*. A system shall first analyze inconsistency in order to gain knowledge about the nature of this inconsistency. The system can then automatically classify the inconsistency and if required ask a human user to make a decision that cannot be made by the system. (For example if no treatment is possible, then the potential solutions could be displayed to a doctor.) If desired, the user can then specify how to repair inconsistency in the system.

Most importantly we see *inconsistency as part of the knowledge* in a system, we attribute inconsistency to the *knowledge exchange* in a system (i.e., to bridge rules), we *reject the idea of restoring consistency at any cost*, and we aim at *computationally efficient solutions*. In cases where inconsistency *must* be repaired, the methods we develop in this work allow to analyze inconsistency and to repair the MCS, i.e., make it consistent.

1.1 Goals and Methods

Analyzing Inconsistency The first aim in this thesis is to develop methods for *analyzing inconsistency in MCSs*. This will provide information about the structure of each inconsistency

[2]We can distinguish between approaches related to Consistent Query Answering which in some sense make use of temporary repairs, and approaches related to Belief Revision, which create permanent repairs.

that occurs, it will show which parts of the system are involved in an inconsistency, and it will provide possibilities for restoring consistency.

We will formulate our analysis methods as a mathematical framework, show how much computational effort we need to analyze inconsistency, and prove useful properties of the framework. The outcome is an approach for getting information about the inconsistency in a MCS, even though the classical effect of inconsistency is that reasoning in an inconsistent system no longer provides useful information. A very nice property of our framework will be that it separates independent reasons for inconsistency in a MCS, therefore we can obtain *partial repairs* of the system that deal with a single inconsistency only. This feature will become important next, because it allows for repairing certain inconsistencies automatically and others manually.

Managing Inconsistency Contexts and bridge rules usually cannot cope with inconsistency on their own. Therefore our next goal is to *manage inconsistency* using the information gained from inconsistency analysis. As we have seen, automatic repair of an inconsistency can be dangerous and is not a universal solution. Neither is it feasible to display all analyzed inconsistencies to the user and let the user repair everything manually, because many different inconsistencies might exist in a MCS, and usually multiple possibilities exist for repairing each inconsistency.

Therefore we need a way to find out which inconsistencies shall be repaired automatically, and which manually. This decision cannot be made on a general level, because it depends on the concrete application scenario. As a consequence, we will create a universally applicable method for *specifying a policy for managing inconsistency* in an MCS. Such a policy configures how a concrete MCS deals with inconsistencies, which inconsistencies are repaired automatically and in which ways, and which inconsistencies must be displayed to a human user for a decision whether and how to repair them.

Efficient Realization To achieve practical relevance, an important aspect of this thesis is the efficient realization of inconsistency analysis and management methods in algorithms and actual software prototypes. We will provide *algorithms for realizing inconsistency analysis and management* in computational logic. As foundation we use the Answer Set Programming (ASP) formalism which is a model-based formalism for knowledge representation with logic programming rules. In particular we will use HEX, an extension of ASP, which allows to interleave reasoning over rules with other 'external' reasoning methods. Furthermore we will describe a *software prototype* that implements these algorithms and provides reasoning about inconsistency using the theoretical notions developed before.

The goal of efficiency had an important impact on the progress of this thesis, because it made us investigate the efficiency of evaluating HEX programs, a logic programming formalism which we use to realize inconsistency analysis and management. This investigation led to the insight that the state-of-the-art implementation of HEX can be improved significantly, which furthermore led to a digression of this thesis from inconsistency in MCSs to HEX evaluation. During this digression, which we describe in Chapter 5, we make a general improvement to the HEX evaluation framework, which can be counted as central and significant result of this work. Therefore this thesis not only contributs to inconsistency in MCSs, but to the HEX formalism and to logic programming in general.

1.2 State-of-the-Art

This section gives an overview of the state-of-the-art of the central topics in this work: combining knowledge sources, dealing with inconsistency, and answer set programming with external evaluations. We describe why we decided to develop an approach different from existing approaches, and how our approach differs from those approaches.

Detailed literature reviews are given in the respective chapters. Readers who are not interested in the scientific context of this work, and do not want to know why we went along a certain path of research among multiple other possibilities, can safely jump to Section 1.3.

Combining Knowledge-Based Systems Combining data and knowledge sources has been intensively researched in several areas of computer science, see the following survey articles about data and information integration in databases [HRO06,BN08,FKMP05], and in description logics and ontologies [BS03, CSH06]. Common to most existing approaches is the focus on an underlying logic that does not permit nonmonotonicity, e.g., data exchange operates on monotonic relational databases. Furthermore, most existing approaches combine information from systems which use the same underlying formalism for all contexts, e.g., information integration and ontology merging. In knowledge representation, we are deeply interested in formalisms that can capture nonmonotonicity, therefore purely monotonic approaches can provide ideas but cannot directly serve as frameworks. Furthermore, combining homogeneous systems is not the situation one typically encounters; if we combine knowledge from existing systems, every system usually uses a different formalism. Finding a unifying formalism that can capture all system parts might be impossible, and if possible it might be a huge amount of effort to transform all system parts into that formalism.

In this work we cope with nonmonotonicity as well as with heterogeneity by using the formalisms of heterogeneous nonmonotonic multi-context systems (MCSs) [BE07]. This formalism is the result of a line of research that goes back to important initial papers by McCarthy [McC87, McC93] and was further developed in [GS94, RS05, BRS07] to finally culminate in the development of MCSs as used in this thesis [BE07].

Unlike other formalisms, MCSs support full nonmonotonicity and heterogeneity of contextual reasoning, and MCSs can be applied in real application scenarios without changing existing contexts. (MCSs interface with the reasoning of a context and do not require to convert context knowledge bases.) Therefore we chose MCS as the basic framework for the work described in this thesis.

Dealing with Inconsistency Similar to the topic of combining information and knowledge, there exist many formalisms for dealing with inconsistency, and most focus on monotonic and/or homogeneous settings.

Existing approaches like data integration [HRO06], MCSs with defeasible rules [BA08], and consistent query answering in databases [BC03, LGI$^+$05, EFGL08] and description logics [LR07] solve the problem of inconsistency by automatically restoring consistency, and in the process ignoring a minimal amount of information in the system. All these approaches hide inconsistency, they create a consistent system and do not provide information about the reason of a repaired inconsistency.

Especially when representing nonmonotonic knowledge, automatically ignoring a small part of the system can introduce counter-intuitive and unwanted behavior into a system. Therefore we here follow the idea of *explaining inconsistency* before, or even instead of, counteracting it, and we cannot use the above methods.

The importance of explaining inconsistency and treating inconsistency as information has been emphasized, e.g., in the area of paraconsistency [GH91, dAP07], in formalisms for debugging logic programs [BV05, Syr06, PC88], and in the area of model-based diagnosis [Rei87].

In this work, we use ideas from ASP debugging and model-based diagnosis and we borrow the name *diagnosis* for one of two very important notions in this thesis. Traditional diagnosis in the style of Reiter [Rei87] aims at analyzing faulty operation of system components and identifying which components operate differently from its specification. In contrast to that, our approach aims at analyzing the potentially *faulty design* of a system, and our concept of a faulty component is a component which operates as intended, but by its normal mode of operation causes inconsistency. Therefore our approach can identify errors that arise due to the unexpected interaction of several components which operate normally, i.e., we can identify knowledge representation errors introduced into the system by humans.

As mentioned above, there exist many methods for dealing with inconsistency within a particular context. Therefore we do not consider to repair context knowledge bases in this work, and we assume they are consistent if no knowledge is added by a bridge rule. As a result of these considerations, the focus of this thesis is on the information flow between contexts in a MCS, and we analyze which bridge rules are problematic and cause inconsistency.

Besides *analyzing* inconsistency, an important topic in this thesis is the *management* of inconsistency, and we propose to do this by means of a declarative policy language. There are several languages related to this approach; here we mention a few of them.

Active integrity constraints (AICs) [CGZ09,CT08a] and inconsistency management policies (IMPs) [MPP^{+}08] are approaches that repair data to make it conform with constraints on that data. This would correspond to modifying contexts such that the overall system is consistent, and to leaving bridge rules as they are. As we want to do the exact opposite, i.e., leave contexts unmodified and identify problematic bridge rules, AIC and IMP are orthogonal to our work.

Two policy language formalisms that are related to what we want to do in this work are the language *IMPACT* [SBD^{+}00] which is a declarative formalism for actions in distributed and heterogeneous multi-agent systems, and the policy description language $\mathcal{PDL}$ [CLN00] which is used for resolving conflicts in a communication network. Compared to the language we develop in this work, *IMPACT* and $\mathcal{PDL}$ have much simpler actions, but these actions may have much more complex conditions. The declarative policy language we develop in this thesis has comparatively simple rules and conditions, while individual actions are complex and suited to the application domain. Furthermore user interaction is a core part of our language, while it is not directly supported by *IMPACT* and $\mathcal{PDL}$.

The approach we develop here is not as general as *IMPACT* and $\mathcal{PDL}$, however this gives us the advantage of making our approach simpler and easier to understand. Furthermore simplicity potentially allows for a more efficient implementation of our approach.

Answer Set Programming with External Knowledge Sources Both MCSs and the HEX formalism combine declarative reasoning with external sources of knowledge. We next point to other formalisms in Answer Set Programming and in related areas which provide such capabilities. For an overview article about that topic, see [EBDT^{+}09].

The gringo tool [GST07, GKKS11] is a grounder for answer set programs which can using LUA scripts to access external knowledge during grounding. External computations in gringo operate 'syntactically' and cannot access information of the model building process, because they are executed during grounding. Different from that, MCSs (resp., HEX programs) evaluate context semantics (resp., external atoms) during the model building process and therefore can use their external knowledge sources 'semantically'.

The clasp software is a state-of-the-art ASP solver [GKNS07] which contains an 'ASP modulo theories' (ASP-MT) interface for external reasoning interleaved with the ASP model building process. This interface is, e.g., used by clingcon [GOS09] which combines ASP with the constraint satisfaction solver GECODE [Gec08]. Compared to HEX external computations, ASP-MT operates on ground programs and requires implementation of propagators within the conflict-driven clause learning (CDCL) [Mit05] framework underlying clasp, while HEX external computations operate on nonground programs and implementing an external atom in the HEX framework does not require knowledge about the implementation of the HEX model building process. As for MCSs, both MCSs and ASP-MT combine declarative reasoning with other types of reasoning (theories, resp., contexts), however these formalisms have very different goals: ASP-MT tightly integrates everything in one place, while MCS aims at loosely integrating distributed knowledge sources.

Therefore the HEX framework is more general than the kinds of external reasoning possible with ASP-MT and in gringo. On the other hand, the current implementation of HEX (i.e., dlvhex) does not use the state-of-the-art CDCL methodology for model building, which makes it slower than clasp and gringo.[3]

Beyond the ASP community, satisfiability modulo theories (SMT) [BSST09] is an approach for combining SAT solving and additional theories. This approach inspired ASP-MT and can be regarded external reasoning with respect to SAT problems. There exists also an approach for rewriting ASP to SMT with difference logic as external theory [Nie08].

1.3 Evolution of this Work

In this section we give an account on the process of gaining knowledge in the years 2009 to 2012, we summarize the most important work done during the creation of this thesis and give references to relevant publications. As one would expect when doing scientific work, this history contains some unexpected twists and turns, and we will here give an account of them.

In the beginning we classified inconsistency into two cases: inconsistency within a context, caused by the same context, and inconsistency within a context or within bridge rules, caused by interaction of contexts and bridge rules.

The first class of problems can be solved within the formalism of one context, therefore a multitude of approaches for homogeneous systems (e.g., belief revision [AGM85], repair and consistent query answering [BC03, Ber11], and information integration [LGI$^+$05, HRO06]) can be used to solve such problems. For that reason we decided to disregard the first class of problems, and made the following assumption: a context without bridge rule input is consistent, i.e., it contains no immanent inconsistencies or conflicts.

The second class of problems are inconsistencies that are always at least partially caused by the exchange of knowledge via bridge rules, i.e., at least one bridge rule is involved in an inconsistency of the second class. We decided to focus our work on this class of inconsistencies.

In the following, we developed theoretical notions that allow for repairing inconsistency in a system, and equally important giving reasons for different inconsistencies and separating multiple inconsistencies. Due to the importance of bridge rules for existence of inconsistency, we created two characterizations for inconsistency, and both characterizations consist of sets of bridge rules:

(i) *diagnoses* point out bridge rules that must be changed to restore consistency, while

(ii) *explanations* point out bridge rules that trigger inconsistency independently from other bridge rules in the system.

We published preliminary versions of these two notions [EFSW09], and showed that a useful duality relationship holds between them: both notions point out the same bridge rules as responsible for inconsistency.

Soon it became apparent, that the initially proposed notions of inconsistency explanations can be replaced by a formulation which is more informative and intuitive in practice. Moreover, this new formulation not only preserves the original duality property, but makes it possible to obtain the set of explanations (the new notion) in a MCS if we know just the set of diagnoses in that MCS.[4] Moreover, the converse, i.e., deriving diagnoses from explanations in the same MCS, is also possible, although only in a limited form[5] which nicely completes the picture of the relationship between both notions.

[3]A CDCL-based implementation of HEX evaluation is subject of ongoing work but is out of scope of this thesis.

[4]This is shown in Theorem 1 on page 27.

[5]This is shown in Theorem 2 on page 28.

Our first conference publication in the course of this thesis [EFSW10] introduces the notion of diagnosis and the revised notion of inconsistency explanation and conducts a more comprehensive theoretical analysis of diagnoses and explanations: we describe simplified diagnosis and explanation notions, analyze modularity properties and preference orderings over bridge rules, provide computational complexity results for our notions and their subset-minimal refined notions, and give a method for computing diagnoses by rewriting to HEX programs. In that work, the main responsibility of the author of this thesis was computational complexity and the HEX rewriting. During the complexity analysis, the author's attempt to prove a wrong hardness result for explanations was a very insightful process, which yielded an interesting property of explanations and a completeness result for a lower complexity class.[6]

After introducing diagnoses and explanations for inconsistency in MCSs, our work continued along different paths. The paths followed by the author of this thesis are: approximations for cases where the system is *partially hidden*, and the realization of the theoretical results in a *software prototype*, and implementation of a *policy language* for managing inconsistency.[7]

In typical application scenarios, not all system parts expose all their information to each other, typical examples are credit card systems and authentication systems. For dealing with such systems, we lifted the MCSs formalism from fully known contexts to *partially known* contexts [SEF10]. In [EFS10, EFS11] we (i) describe the partially known MCS formalism, (ii) lift the concepts of diagnoses and explanations to partially known MCSs, (iii) show how to obtain approximations of diagnoses and explanations in such systems, and (iv) provide a method for identifying the most informative queries we can pose to a partially known context such that the information gained from the query answer will improve the precision of our approximation.

In parallel to the theoretical work on approximations, we created the *prototype software tool* MCS-IE [BEFS10, MIE12a] which uses the HEX rewriting previously developed to actually compute diagnoses and explanations for inconsistency in MCSs. MCS-IE is implemented as a plugin to the dlvhex solver software [DHX12]; we also developed a web front end [MIE12b] which received very positive feedback from the community during the JELIA 2010 systems demonstration session.

An important insight from using and testing the MCS-IE system was that the dlvhex evaluation at that time had scalability problems, and that these problems could be solved by a different evaluation strategy. Therefore, at a research visit in Calabria, we started work on a novel theoretical framework for evaluating HEX programs that did not suffer from these scalability issues. The resulting framework was implemented in a new version of dlvhex[8] and published together with empirical experiments on the performance of the result [EFI$^+$11]. The paper showed a significant efficiency improvement and was selected for presentation in the 'Best Papers Sister Conferences Track' of IJCAI 2011.

The final result in this thesis is a combination of the work on inconsistency in MCSs and on HEX: the Inconsistency Management Policy Language (IMPL) for managing inconsistency in MCSs [EFIS11, EFIS12a, EFIS12b] with the goal as described in Section 1.1: IMPL uses the analysis of inconsistencies in MCSs and the power of nonmonotonic reasoning in order to deal with inconsistencies in MCSs in a semi-automated manner. In an IMPL policy, a system designer can specify that certain inconsistencies shall be repaired automatically, while other inconsistencies must be resolved with user interaction, or left untouched. The IMPL language is inspired by the ACTHEX language which is an extension of HEX by actions. We describe how IMPL can be implemented on top of ACTHEX (which can be implemented on top of HEX).

IMPL combines two lines of research that were followed during this thesis: it draws its strengths from theoretical and practical advances in the analysis of inconsistencies in MCSs, as

[6]All complexity results are summarized in Table 3.1 on page 32.

[7]Other investigations which are out of scope of this thesis were conducted by Antonius Weinzierl, for example a framework for preferences on diagnoses and inconsistency explanations, and several extensions of the MCS formalism that directly or indirectly make use of these notions.

[8]Version 2.0.0 of dlvhex was released on March 11, 2012.

well as from performance improvements achieved for the HEX formalism.

1.4 Results and Thesis Outline

We now give an overview of the structure of this thesis and point out the most important results of each section.

Section 2 introduces notation and the formalisms we use throughout this thesis. In particular we describe Answer Set Programming (ASP), HEX programs, Multi-Context Systems (MCSs), and some complexity theoretical notions. This section also contains a formal description of our running example from the medical domain, which will be used throughout this work. (We informally sketched the example on page 1.)

Section 3 describes our formal framework for analyzing inconsistency in MCSs, and shows a comprehensive analysis of these notions. Major results in this section are the notions of *diagnosis* and *inconsistency explanation* and the theoretical analysis of their properties, in particular *duality*, *conversions* between the notions, and analysis of their *computational complexity*. Furthermore we give a possibility for performing inconsistency analysis under *incomplete information*, i.e., if a part of the MCS at hand is not fully known.

Section 4 shows how notions of diagnoses and explanations can be computed by rewriting the problems to computational logic, in particular we use HEX programs as logical formalism. Major results are the *rewriting of an MCS to a* HEX *program* for computing diagnoses, the *implementation* of the rewriting in a software prototype, including a *web front end* to the software, and results of *empirical experiments* with that software prototype.

Section 5 explains the drawbacks of the previous HEX evaluation framework and introduces a new evaluation method for HEX programs that does not have these drawbacks. We give formal proofs of the soundness and completeness of this new framework, and describe benchmark experiments done with a prototype implementation of this framework. Major results are the *novel theoretical evaluation framework* for HEX programs, the *implementation* within the dlvhex software, and an account of the *empirical experiments* we conducted to verify the advantage of this new framework.

Section 6 introduces the policy language IMPL for managing inconsistency in MCSs. IMPL achieves central goals we described in Section 1.1, it uses diagnoses and explanations introduced in Section 3 to manage inconsistency, it is inspired by ACTHEX and can be realized using ACTHEX. Major contributions of this section are the definition of *syntax and semantics of* IMPL, a *rewriting from* IMPL *to* ACTHEX which can be used for realizing IMPL, and *methodologies for applying* IMPL in practical application scenarios.

Section 7 concludes the thesis with a summary and an outlook on future research.

2 Preliminaries

In this section we introduce formal notions that are used in multiple parts of this thesis.

We first introduce heterogeneous nonmonotonic multi-context systems (MCSs), which are the foundation for most work done in this thesis. At that point we also give a formal account of our running example in the medical domain.

We furthermore describe the HEX formalism which is an extension of Answer Set Programs [GL91]. Answer Set Programming (ASP) is a foundational formalism for knowledge representation and reasoning, HEX programs additionally are capable of using external computations as knowledge. We use HEX programs for implementing parts of the software described in this thesis, and HEX is related to the IMPL policy language described in Chapter 6.

Finally we introduce some relevant notions of formal complexity theory.

As we will often encounter sets that contain logic programming rules, we will separate elements in such sets using ';'. For instance we write '$\{d;\ a \leftarrow b,\ c;\ e \leftarrow f,\ not\, c,\ not\, d\}$' to denote the program consisting of fact 'd', and rules '$a \leftarrow b,\ c$' and '$e \leftarrow f,\ not\, c,\ not\, d$'.

Given a family of sets A and a set B, we denote by $A|_B$ the projection of A to B, formally $A|_B = \{X \cap B \mid X \in A\}$.

2.1 Multi-Context Systems

A heterogeneous nonmonotonic multi-context system (MCS) [BE07] is a formalism for representing knowledge based systems which consist of interlinked smaller knowledge based systems. The individual smaller systems are called *contexts*, each composed of a knowledge base with an underlying abstract *logic*, while the information flow between contexts is controlled and described by a set of *bridge rules*.

Contexts and Logics. For each context, we use a an abstract *Logic*, which is a tool that provides a uniform and minimalistic interface to various KR formalisms.

Definition 1 (Logic)**.** *A logic $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ consists, in an abstract view, of the following components:*

- $\mathbf{KB}_L$ *is the set of well-formed knowledge bases of L. We assume each element of $\mathbf{KB}_L$ is a set (of "formulas").*
- $\mathbf{BS}_L$ *is the set of possible belief sets, where the elements of a belief set are statements that possibly hold given a knowledge base.*
- $\mathbf{ACC}_L : \mathbf{KB}_L \to 2^{\mathbf{BS}_L}$ *is a function describing the "semantics" of the logic by assigning to each knowledge base a set of acceptable belief sets.*

This concept of a *logic* captures many monotonic and nonmonotonic logics, e.g., classical logic, description logics, modal, default, and autoepistemic logics, circumscription, and logic programs under the answer set semantics.

The following examples shall illustrate how this abstraction captures some well-known KR formalisms. They all introduce logics we will use to formalize our running example.

Example 1. *To capture logical relational databases with inclusion constraints and equality constraints over a signature $\Sigma = \Sigma_r \cup \Sigma_c \cup \Sigma_v$ of relation, constant, and variable symbols Σ_r, Σ_c, and Σ_v respectively. A formula $p(t_1, \dots, t_k)$ with $p \in \Sigma_r$, $t_1, \dots, t_k \in \Sigma_c \cup \Sigma_v$ is an atom; it is ground if $t_1, \dots, t_k \in \Sigma_c$, nonground otherwise. A literal is either an atom a or its negation $\neg a$. We have that*

- $\mathbf{KB} = \mathbf{KB}_p \cup \mathbf{KB}_f \cup \mathbf{KB}_u$ *contains*
  - *the set $\mathbf{KB}_p$ of ground literals over Σ,*
  - *the set $\mathbf{KB}_f$ of inclusion constraints (these can express foreign key constraints) of the form $a_1 \wedge \cdots \wedge a_k \rightarrow \exists X_1, \dots, X_m\ a$ where $a, a_1, \dots, a_k$ are nonground literals over Σ, $X_1, \dots, X_m \in \Sigma_v$ are variables, and all variables in a appear in $a_1, \dots, a_k$ or in $X_1, \dots, X_m$. Furthermore we have*
  - *the set $\mathbf{KB}_u$ of equality constraints (these can express unique key constraints) of the form $a_1 \wedge \cdots \wedge a_k \rightarrow X = Y$ where $a_1, \dots, a_k$ are nonground literals over Σ, and $X, Y \in \Sigma_v$ are variables which appear in $a_1, \dots, a_k$.*
- $\mathbf{BS}$ *is the collection of deductively closed sets of literals over Σ; and*
- $\mathbf{ACC}(kb)$ *returns for each set $kb \in \mathbf{KB}$*
  - *a singleton set containing the set Λ of facts that are consequences of kb under the closed world assumption [Rei78], i.e., $\mathbf{ACC}(kb) = \{\Lambda\}$ where $\Lambda = CWA(kb)$;*
  - *the empty set $\emptyset$ if some constraint is not satisfied by the ground literals in kb.*

Note that we often omit negative consequences of a knowledge base in the following. The resulting logic over Σ then is $L^{db}_{\Sigma} = (\mathbf{KB}, \mathbf{BS}, \mathbf{ACC})$.

Our running example contains two contexts C_{db} and C_{lab} that employ this logic. We omit the variable signature and assume that variables start with capital letters. C_{db} uses signature $\Sigma_{db} = \Sigma_{db,p} \cup \Sigma_{db,c}$ with $\Sigma_{db,p} \supseteq \{person, allergy\}$ and $\Sigma_{db,c} \supseteq \{sue, ab1, 02/03/1985, 03/02/1985\}$. C_{lab} uses signature $\Sigma_{lab} = \Sigma_{lab,p} \cup \Sigma_{lab,c}$ with $\Sigma_{lab,p} \supseteq \{customer, test\}$ and $\Sigma_{lab,c} \supseteq \{sue, 02/03/1985, 03/02/1985, xray, bloodtest, pneum, lmark, cmark\}$. The corresponding logics are $L^{db}_{\Sigma_{db}}$ and $L^{db}_{\Sigma_{lab}}$. Knowledge bases for these contexts are as follows:

$$\begin{aligned}
kb_{db} &= \{person(sue, 02/03/1985);\ allergy(sue, ab1)\},\\
kb_{lab} &= \{customer(sue, 02/03/1985);\\
&\qquad test(sue, xray, pneum); \neg test(sue, bloodtest, cmark);\\
&\qquad test(\textsc{Id}, X, Y) \rightarrow \exists D : customer(\textsc{Id}, D);\\
&\qquad customer(\textsc{Id}, X) \wedge customer(\textsc{Id}, Y) \rightarrow X = Y\}.
\end{aligned}$$

Knowledge base kb_{db} provides the information that the patient identified as 'Sue' is allergic to a certain antibiotic $ab1$, while kb_{lab} tells us that the blood marker $cmark$ was not present in a blood test, and that pneumonia was detected in an X-ray examination. Both contexts also store Sue's birth date (in different relations). Constraints in kb_{lab} enforce that each test result must be linked to a $customer$ record, and that each customer has a unique birth date.

The corresponding accepted belief sets (we omit negated beliefs) are

$$\begin{aligned}
\mathbf{ACC}(kb_{db}) &= \big\{\{person(sue, 02/03/1985), allergy(sue, ab1)\}\big\} \text{ and}\\
\mathbf{ACC}(kb_{lab}) &= \big\{\{customer(sue, 02/03/1985), test(sue, xray, pneum)\}\big\}.
\end{aligned}$$

□

Example 2. *For ontologies with syntax and semantics of the description logic $\mathcal{ALC}$ [BCM+03] we use the abstract logic $L_{\mathcal{A}}$. It is composed of*

- $\mathbf{KB}$*, being the collection of sets of $\mathcal{ALC}$ axioms. Basic entities of $\mathcal{ALC}$ are concepts, roles, and individuals. An atomic concept is an $\mathcal{ALC}$ concept, and starting from (atomic) concepts C, D and roles R, we can inductively build $\mathcal{ALC}$ concepts $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall R.C$, and $\exists R.C$. $\mathcal{ALC}$ axioms are formulas of the form $C \sqsubseteq D$, called terminological (T-Box) axioms, and formulas of the form $a{:}C$, resp. $(a,b){:}R$ (given individual names a and b), called assertional (A-Box) axioms. In bridge rules where we have multiple ':' symbols in one expression, we write brackets also around individual symbols, such that $a{:}C$ is written as $(a){:}C$.*

- $\mathbf{BS}$*, being the set of possibly believed assertions, i.e.,* $\mathbf{BS}$ *is the powerset of the set of A-Box axioms, and*

- $\mathbf{ACC}$*, being a mapping from knowledge bases to the set of assertions entailed by the knowledge base. As $\mathcal{ALC}$ amounts to a fragment of first-order logic, the semantics of an $\mathcal{ALC}$ knowledge base kb can be given by a rewriting π to first-order logic. For our purpose, $\mathbf{ACC}(kb) = \{S\}$ where S is the set of classically entailed atomic assertions of the first-order rewriting $\pi(kb)$ of kb.*

Note that we will show only positive A-Box axioms in belief sets.

For a running example we use an ontology about diseases, given by context C_{onto} using the above logic $L_{\mathcal{A}}$. Its knowledge base, kb_{onto}, is as follows:

$$\begin{aligned} kb_{onto} = \{ & Pneum \sqsubseteq BacterialDisease; \\ & \exists hasDisease.Pneum \sqcap \exists hasMarker.APMark \sqsubseteq \exists hasDisease.AtypPneum; \\ & mmark{:}APMark;\ cmark{:}APMark\} \end{aligned}$$

This knowledge base consists of two axioms, where the first states that pneumonia is a bacterial disease[1] *and the second states that pneumonia, if it occurs with an associated class of blood-markers, implies atypical pneumonia (a severe form of pneumonia treatable only by certain antibiotics). Furthermore the knowledge base contains assertions about two blood markers which indicate atypical pneumonia. From kb_{onto}, only assertions already within the knowledge base follow, therefore*

$$\mathbf{ACC}(kb_{onto}) = \big\{\{mmark{:}APMark,\ cmark{:}APMark\}\big\}.$$

Adding the assertion that there is a disease d classified as pneumonia, results in the conclusion that d also is a bacterial disease, formally

$$\begin{aligned} & \mathbf{ACC}\big(kb_{onto} \cup \{d{:}Pneum\}\big) = \\ & \qquad \big\{\{d{:}Pneum, d{:}BacterialDisease, mmark{:}APMark, cmark{:}APMark\}\big\}. \end{aligned}$$

□

Example 3. *For extended disjunctive logic programs under answer set semantics over a non-ground signature Σ [GL91] (see also Section 2.2),*

- $\mathbf{KB}$ *is the set of normal disjunctive logic programs over Σ, i.e., each $kb \in \mathbf{KB}$ is a set of rules of the form*

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_i, not\, b_{i+1}, \ldots, not\, b_m,$$

where all a_i, b_j, are atoms over Σ, and $n + m > 0$.

[1]For the purpose of this thesis, we assume that pneumonia can only be caused by bacteria.

- $\mathbf{BS}$ *is the set of Herbrand interpretations over* Σ*, i.e, each* $bs \in \mathbf{BS}$ *is a set of ground atoms from* Σ*, and*
- $\mathbf{ACC}(kb)$ *returns the set of kb's answer sets: for* $P \in \mathbf{KB}$ *and* $T \in \mathbf{BS}$ *let* $P^T = \{r \in grnd(P) \mid T$ *models the body of* $r\}$ *be the FLP-reduct of* P *wrt.* T*, where* $grnd(P)$ *returns the ground version of all rules in* P*. Then* $bs \in \mathbf{BS}$ *is an answer set, i.e.,* $bs \in \mathbf{ACC}(kb)$*, iff* bs *is the minimal model of* kb^{bs}*.*

The resulting abstract logic for answer set programs finally is $L_{\Sigma}^{asp} = (\mathbf{KB}, \mathbf{BS}, \mathbf{ACC})$*. We employ this logic for the decision support system context,* C_{dss}*, where the signature* Σ_{dss} *satisfies* $\Sigma_{dss} \supseteq \{give, need, allow, sue, nothing, ab, ab1, ab2\}$ *and we assume that variables start with capital letters.*

The knowledge base for C_{dss} *is:*

$$\begin{aligned} kb_{dss} = \{ & give(\textsc{Id}, ab1) \vee give(\textsc{Id}, ab2) \leftarrow need(\textsc{Id}, ab); \\ & give(\textsc{Id}, ab1) \leftarrow need(\textsc{Id}, ab1); \\ & \neg give(\textsc{Id}, ab1) \leftarrow not\ allow(\textsc{Id}, ab1), need(\textsc{Id}, \textsc{Med})\}. \end{aligned}$$

If antibiotics are required, C_{dss} *suggests a treatment with one of two possible antibiotics, antibiotic* $ab1$ *is explicitly ruled out if there is no information that indicates to allow giving this antibiotic to the patient. Without further information,* kb_{dss} *thus concludes that nothing is required:*

$$\mathbf{ACC}(kb_{dss}) = \{\emptyset\}.$$

If $need(sue, ab)$ *is added, however,* kb_{dss} *results in one answer sets:*

$$\mathbf{ACC}\big(kb_{dss} \cup \{need(sue, ab) \leftarrow\}\big) = \big\{\{\neg give(\textsc{Id}, ab1), give(\textsc{Id}, ab2), need(sue, ab)\}\big\}$$

□

We now have seen several examples for abstract 'Logics' in the sense of MCSs, and how we can use Logics to represent various knowledge representation formalisms.

Bridge Rules. Bridge rules are the second important ingredient of MCSs. A *bridge rule* can add information to a context, depending on the belief sets which are accepted at other contexts. Let $L = (L_1, \ldots, L_n)$ be a sequence of logics with L_i as above. An L_k-bridge rule r over L is of the form

$$(k : s) \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (c_m : p_m). \quad (2.1)$$

where $1 \leq c_i \leq n$, p_i is an element of some belief set of L_{c_i}, k refers to the context receiving information s. We denote by $h_b(r)$ the belief formula s in the head of r and by $h_c(r)$ the context k where r belongs to. The literals in the body of r are referred to by $body(r)$, $body^+(r)$, $body^-(r)$ which denotes the set $\{(c_1 : p_1), \ldots, (c_m : p_m)\}$, $\{(c_1 : p_1), \ldots, (c_j : p_j)\}$, $\{(c_{j+1} : p_{j+1}), \ldots, (c_m : p_m)\}$, respectively. Furthermore, by $cf(r)$ we denote the *condition-free* bridge rule stemming from r by removing all elements in its body, i.e., $cf(r)$ is '$(k : s) \leftarrow$' and for any set of bridge rules R, we let $cf(R) = \bigcup_{r \in R} cf(r)$.

Multi-Context System. We now have introduced all components of multi-context systems and can define them formally.

Definition 2 (Multi-Context System)**.** *A multi-context system* $M = (C_1, \ldots, C_n)$ *is a collection 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,* $kb_i \in \mathbf{KB}_i$ *a knowledge base, and* br_i *is a set of* L_i*-bridge rules over* $(L_1, \ldots, L_n)$*. Furthermore, for each* $H \subseteq \{h_b(r) \mid r \in br_i\}$ *it holds that* $kb_i \cup H \in \mathbf{KB}_{L_i}$*, i.e., bridge rule heads can be added to knowledge bases.*

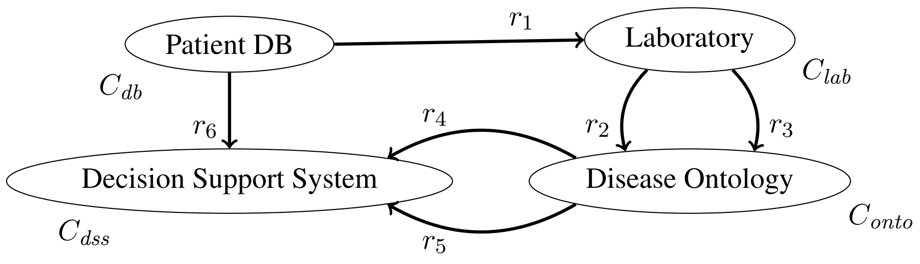


Figure 2.1: Knowledge bases and bridge rules of the Medical Example MCS M.

By $br_M = \bigcup_{i=1}^{n} br_i$ and $c(M) = \{C_1, \dots, C_n\}$ we denote the set of all bridge rules, resp. the set of all contexts of M. We write $br_i(M)$ to denote the set of bridge rules of context i of M, i.e., $br_i(M) = \{r \in br_M \mid h_c(r) = i\}$.

In the following we introduce the *Medical Example* which was already sketched in the introduction. This example it is an extended version of the running examples in [EFSW10] and [EFIS11] and will be the main running example in this thesis. For ease of reading we use the subscripts db, lab, $onto$, dss to denote, respectively, the patient database, the laboratory database, the disease ontology, and the decision support system, in place of the integer subscripts.

Example 4. *Consider a health care decision support system that contains the following contexts: a patient history database (C_{db}), a blood and X-Ray analysis database (C_{lab}), a disease ontology (C_{onto}), and an expert system (C_{dss}) which suggests proper treatments. The corresponding abstract logics and knowledge bases are those in Example 1, 2, and 3. A layout of the information exchange in this MCS is depicted in Figure 2.1 where contexts are shown as ellipses and bridge rules as arrows. We next give schemas for bridge rules, where* ID*,* BIRTHDAY*, and* MARKER *are schema variables.*

$$
\begin{aligned}
r_1&: \quad (lab : customer(\text{ID}, \text{BIRTHDAY})) \leftarrow (db : person(\text{ID}, \text{BIRTHDAY})).\\
r_2&: \; (onto : (\text{ID}){:}\exists hasDisease.Pneum) \leftarrow (lab : test(\text{ID}, xray, pneum)).\\
r_3&: (onto : (\text{ID}, \text{MARKER}){:}hasMarker) \leftarrow (lab : test(\text{ID}, bloodtest, \text{MARKER})).\\
r_4&: \quad (dss : need(\text{ID}, ab)) \leftarrow (onto : (\text{ID}){:}\exists hasDisease.BacterialDisease).\\
r_5&: \quad (dss : need(\text{ID}, ab1)) \leftarrow (onto : (\text{ID}){:}\exists hasDisease.AtypPneum).\\
r_6&: \quad (dss : allow(\text{ID}, ab1)) \leftarrow \mathbf{not}\ (db : allergy(\text{ID}, ab1).
\end{aligned}
$$

Rule r_1 links the patient records with the lab database (so patients do not need to enter their data twice). Rules r_2 and r_3 provide test results from the lab to the ontology. Rules r_4 and r_5 link disease information with medication requirements, and r_6 associates acceptance of the particular antibiotic '$ab1$' with a negative allergy check on the patient database. □

Intuitively our example MCS should suggest to give antibiotic $ab2$ to Sue, because she needs an antibiotic due to her having pneumonia, however she is allergic to $ab1$, so $ab2$ is the only treatment option within the knowledge represented in the system.

We next formally define semantics of MCSs.

Semantics. The semantics of MCSs is defined in terms of accepted belief states.

A *belief state* of an MCS $M = (C_1, \dots, C_n)$ is a sequence $S = (S_1, \dots, S_n)$ of belief sets $S_i \in \mathbf{BS}_i$, $1 \leq i \leq n$. A bridge rule r of form (2.1) is *applicable* in S, denoted $S \models r$, iff for all $(c : p) \in body^+(r)$ it holds that $p \in S_c$, and for all $(c : p) \in body^-(r)$ it holds that $p \notin S_c$. For a set R of bridge rules, $app(R, S) = \{r \in R \mid S \models r\}$ denotes applicable bridge rules with respect to S.

Equilibrium semantics selects certain belief states of an MCS M as acceptable. Intuitively, an equilibrium is a belief state S where each context C_i takes the heads of all bridge rules that are applicable in S into account, and accepts S_i.

Definition 3. *A belief state $S = (S_1, \ldots, S_n)$ of M is an equilibrium iff for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i, S)\})$.*

Given a MCS M, we denote by $\text{Eq}(M)$ the set of equilibria of M. For reasons of better readability, and as the semantics of MCSs is defined on ground systems, we now repeat the whole running example MCS (contexts and bridge rules) and we ground bridge rules such that they keep their names. (This is possible, as every bridge rule creates exactly one ground instance, and other ground instances are irrelevant for the purposes of this work).

Example 5 (Medical Example)**.** *Our running example MCS is formally defined as $M_1 = (C_{db}, C_{lab}, C_{onto}, C_{dss})$, where C_{db} and C_{lab} use logics $L^{db}_{\Sigma_{db}}$ and $L^{db}_{\Sigma_{lab}}$, C_{onto} uses logic $L_{\mathcal{A}}$, and C_{dss} uses logic $L^{asp}_{\Sigma_{dss}}$. The knowledge bases are as follows:*

$$\begin{aligned}
kb_{db} &= \{person(sue, 02/03/1985), allergy(sue, ab1)\},\\
kb_{lab} &= \{customer(sue, 02/03/1985);\\
&\quad test(sue, xray, pneum); \neg test(sue, bloodtest, cmark);\\
&\quad test(\text{Id}, X, Y) \rightarrow \exists D : customer(\text{Id}, D);\\
&\quad customer(\text{Id}, X) \wedge customer(\text{Id}, Y) \rightarrow X = Y\},\\
kb_{onto} &= \{Pneum \sqsubseteq BacterialDisease;\\
&\quad \exists hasDisease.Pneum \sqcap \exists hasMarker.APMark \sqsubseteq \exists hasDisease.AtypPneum;\\
&\quad (mmark){:}APMark,\ (cmark){:}APMark\},\ \textit{and}\\
kb_{dss} &= \{give(\text{Id}, ab1) \vee give(\text{Id}, ab2) \leftarrow need(\text{Id}, ab);\\
&\quad give(\text{Id}, ab1) \leftarrow need(\text{Id}, ab1);\\
&\quad \neg give(\text{Id}, ab1) \leftarrow not\ allow(\text{Id}, ab1), need(\text{Id}, \text{Med})\}.
\end{aligned}$$

M_1 has the following set of ground bridge rules:

$$\begin{aligned}
r_1&: (lab : customer(sue, 02/03/1985)) \leftarrow (db : person(sue, 02/03/1985)).\\
r_2&: (onto : (sue){:}\exists hasDisease.Pneum) \leftarrow (lab : test(sue, xray, pneum)).\\
r_3&: (onto : (sue, cmark){:}hasMarker) \leftarrow (lab : test(sue, bloodtest, cmark)).\\
r_4&: (dss : need(sue, ab)) \leftarrow (onto : (sue){:}\exists hasDisease.BacterialDisease).\\
r_5&: (dss : need(sue, ab1)) \leftarrow (onto : (sue){:}\exists hasDisease.AtypPneum).\\
r_6&: (dss : allow(sue, ab1)) \leftarrow \mathbf{not}\ (db : allergy(sue, ab1).
\end{aligned}$$

□

Applying equilibrium semantics yields the following result.

Example 6. *The Medical Example M_1 has a single equilibrium $S = (S_{db}, S_{lab}, S_{onto}, S_{dss})$, where*

$$\begin{aligned}
S_{db} &= \{person(sue, 02/03/1985), allergy(sue, ab1)\},\\
S_{lab} &= \{customer(sue, 02/03/1985), test(sue, xray, pneum)\},\\
S_{onto} &= \{(mmark){:}APMark, (cmark){:}APMark,\\
&\quad (sue){:}\exists hasDisease.Pneum, (sue){:}\exists hasDisease.BacterialDisease\},\ \textit{and}\\
S_{dss} &= \{need(sue, ab), \neg give(\text{Id}, ab1), give(\text{Id}, ab2)\}.
\end{aligned}$$

Rules r_1, r_2, and r_4 are applicable in S. Note that $\neg give(\text{Id}, ab1)$ is a belief accepted by C_{dss}, because $allow(\text{Id}, ab1)$ is not *added to kb_{dss}, which is because bridge rule r_6 is not acceptable, which is because Sue has an allergy to $ab1$.* □

Background information about MCSs, extensions of the formalism, and related work, can be found in [BEF11]. For a comparison of MCSs with other formalisms, see [EBDT$^+$09].

2.2 HEX: Answer Set Programs with External Computations

Answer Set Programming (ASP), based on Answer Set Semantics [GL91], is a widely used knowledge representation and reasoning formalism. ASP uses logic programming rules in the style of Prolog, however semantics of ASP is defined in a purely declarative way (unlike Prolog, whose semantics is defined in procedural terms).

In this thesis we mainly use the HEX formalism, which is a conservative extension of the original ASP formalism. HEX was introduced in [EIST05] and described in more detail in [Sch06]. HEX adds higher order features and external computations to ASP.

As this thesis is mainly concerned with HEX, we directly introduce HEX, and then show which fragment of HEX corresponds to the original definition of ASP.

Syntax Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ are prefixed with "&". Note that constant names serve both as individual and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An *atom* is a tuple $(Y_0, Y_1, \dots, Y_n)$, where $Y_0, \dots, Y_n$ are terms; $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \dots, Y_n)$. The atom is *ordinary* (*higher-order*), if Y_0 is a constant (variable).

An *external atom* is of the form

$$\&g[Y_1, \dots, Y_n](X_1, \dots, X_m), \tag{2.2}$$

where $Y_1, \dots, Y_n$ and $X_1, \dots, X_m$ are two lists of terms (called *input* and *output* lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively.

Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the input tuple and the interpretation.

Example 7. *(a, b, c), $a(b, c)$, $node(X)$, and $D(a, b)$ are atoms. The first two are the same, the first three are ordinary, the last one is higher-order.*

The external atom $\&reach[edge, a](X)$ may be devised for computing the nodes which are reachable in the graph edge from the node a. We have $in(\&reach) = 2$ and $out(\&reach) = 1$. Intuitively, $\&reach[edge, a](X)$ will be true for all ground substitutions $X \mapsto b$ such that b is a node in the graph given by edge, and there is a path from a to b in that graph. □

A *rule* r is of the form

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_1, \dots, \beta_n, not\ \beta_{n+1}, \dots, not\ \beta_m, \tag{2.3}$$

where $m, k \geq 0$, $\alpha_1, \dots, \alpha_k$ are atoms, and $\beta_1, \dots, \beta_m$ are either atoms or external atoms. We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \dots, \beta_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*, r is *ordinary* if it contains only ordinary atoms, and r is *nondisjunctive* if $H(r)$ contains no more than one atom.

A HEX-*program* is a finite set P of rules. A program is *ordinary* (resp., *nondisjunctive*) if all rules are ordinary (resp., nondisjunctive). We denote by $const(P)$ the set of constant symbols in HEX program P.

Example 8 (Swimming Example). *Imagine Alice wants to go for a swim in Vienna. She knows two indoor pools called Margarethenbad and Amalienbad (denoted mpool and apool, respectively), and she knows that outdoor swimming is possible in the river Danube at two locations* *called Gänsehäufel and Neue Donau (denoted $gdanube$ and $ndanube$, respectively).*[2] *She looks up on the web whether she needs to pay an entrance fee, and what additional equipment she will need. Finally she has the constraint that she does not want to pay for swimming.*

The following HEX *program $P_{swim} = P^{EDB}_{swim} \cup P^{IDB}_{swim}$ represents Alice's reasoning problem. The extensional part P^{EDB}_{swim} contains a set of facts about possible swimming locations (note that in and out are short for $indoor$ and $outdoor$, respectively):*

$$\begin{aligned} P^{EDB}_{swim} = \{&location(in, mpool), location(in, apool), \\ &location(out, gdanube), location(out, ndanube)\}. \end{aligned}$$

The intensional part P^{IDB}_{swim} represents the web research of Alice in an external computation, i.e., an external atom of the form $\&rq\langle choice\rangle\langle resource\rangle$. P^{IDB}_{swim} is as follows.

$$\begin{array}{lrl} r_1: & swim(in) \vee swim(out) & \leftarrow . \\ r_2: & need(inout, C) & \leftarrow \&rq[swim](C). \\ r_3: & goto(X) \vee ngoto(X) & \leftarrow swim(P), location(P, X). \\ r_4: & go & \leftarrow goto(X). \\ r_5: & need(loc, C) & \leftarrow \&rq[goto](C). \\ c_6: & & \leftarrow goto(X), goto(Y), X \neq Y. \\ c_7: & & \leftarrow not\ go. \\ c_8: & & \leftarrow need(X, money). \end{array}$$

Assume Alice finds out that indoor pools general cost money, and that you also have to pay at Gänsehäufel, but not at Neue Donau. Furthermore Alice reads some reviews about swimming locations and finds out that she will need her Yoga mat for Neue Donau because the ground is so hard, and she will need goggles for Amalienbad because they use so much chlorine.

We next explain the intuition behind the rules in P_{swim}: r_1 chooses indoor vs. outdoor swimming locations, and r_2 collects requirements that are caused by this choice. Rule r_3 chooses one of the indoor vs. outdoor locations, depending on the choice in r_1, and r_5 collects requirements caused by this choice. By r_4 and c_7 we ensure that some location is chosen, and by c_6 that only a single location is chosen. Finally c_8 rules out all choices that require money.

The external predicate $\&rq$ has $in(\&rq) = out(\&rq) = 1$; intuitively $\&rq[\alpha](\beta)$ is true if a resource β is required when swimming in a place in the extension of predicate α. For example, $\&rq[swim](money)$ is true if $swim(in)$ is true, because indoor swimming pool charge money for swimming. Note that this only gives an intuitive account of the semantics of $\&rq$ which will formally be defined in the following examples. □

Semantics The semantics of HEX-programs [EIST06, Sch06] generalizes the answer-set semantics [GL91]. Let P be a HEX-program. Then the *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{X}$ and $\mathcal{G}$ are implicitly given by P. Different from 'usual' ASP evaluation, the set of constants $\mathcal{C}$ used for grounding a program is only partially given by the program itself; in HEX, external computations may introduce new constants that are relevant for semantics of the program. Section 5.3.1 identifies fragments of HEX which can be evaluated using the 'usual' grounding with constants from P, while Section 5.4 deals with the more general case and shows how to decompose a program and interleave semantic evaluation and grounding in order to evaluate programs where external atoms invent new constants.

[2]To keep the example simple, we assume Alice does not know about other possibilities to go swimming in Vienna.

Example 9 (ctd). *In P_{swim} the external atom &rq can introduce constants yogamat and goggles which are not contained in P_{swim}, but they are relevant for computing answer sets of P_{swim}.* □

An *interpretation relative to P* is any subset $I \subseteq HB_P$ containing only atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$.

With every external predicate name $\&g \in \mathcal{G}$ we associate an $(n{+}m{+}1)$-ary Boolean function $f_{\&g}$ assigning each tuple $(I, y_1 \dots, y_n, x_1, \dots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \dots, y_n](x_1, \dots, x_m)$, denoted $I \models a$, if and only if $f_{\&g}(I, y_1 \dots, y_n, x_1, \dots, x_m) = 1$.

Note that this definition of external atom semantics is very general, indeed an external atom may depend on every part of the interpretation. Therefore we will later (Section 5) formally restrict external computations such that they depend only on the extension of those predicates in I which are given in the input list. All examples and encodings in this thesis obey this restriction.

Example 10 (ctd.). *The external predicate &rq in P_{swim} represents Alice's knowledge about swimming locations as follows: for any interpretation I and some predicate (i.e., constant) α,*

$$\begin{array}{lll}
\&rq[\alpha](money) & \textit{iff } f_{\&rq}(I, \alpha, money) = 1 & \textit{iff } \alpha(in) \in I \textit{ or } \alpha(gdanube) \in I, \\
\&rq[\alpha](yogamat) & \textit{iff } f_{\&rq}(I, \alpha, yogamat) = 1 & \textit{iff } \alpha(ndanube) \in I, \textit{ and} \\
\&rq[\alpha](goggles) & \textit{iff } f_{\&rq}(I, \alpha, goggles) = 1 & \textit{iff } \alpha(apool) \in I.
\end{array}$$

Due to this definition of $f_{\&rq}$, it holds e.g. that $\{swim(in)\} \models \&rq[swim](money)$. This matches the intuition about &rq indicated in the previous example. □

Let r be a ground rule. We define

(i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$,

(ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and

(iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$.

We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model.

Given a HEX-program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. Then $I \subseteq HB_P$ is an *answer set of P* iff I is a minimal model of fP^I.

Example 11 (ctd.). *The* HEX *program P_{swim} with external semantics as given in the previous example has a single answer set*

$$I = \{swim(out), goto(ndanube), ngoto(gdanube), go, need(loc, yogamat)\}.$$

(Here, and in following examples, we omit P^{EDB}_{swim} from all interpretations and answer sets.) Under I, the external atom $\&rq[goto](yogamat)$ is true, all others (e.g., $\&rq[swim](money)$, $\&rq[goto](money)$, $\&rq[swim](yogamat)$, …) are false. Intuitively, answer set I tells Alice to take her Yoga mat and go for a swim to Neue Donau. □

HEX programs are a conservative extension of disjunctive (resp., normal) logic programs under the answer set semantics [EIST05]: answer sets of *ordinary nondisjunctive* HEX *programs* coincide with stable models of logic programs as originally proposed by Gelfond and Lifschitz [GL88], and answer sets of *ordinary* HEX *programs* coincide with stable models of disjunctive logic programs [Prz91, GL91].

2.2.1 Restrictions

To make reasoning tasks on HEX programs decidable (or more efficiently computable), we here use the following restrictions. We informally introduce them here as they are relevant for all HEX programs in this thesis. A formal account and discussion of these restrictions is given in Chapter 5.

Rule safety. This is a restriction well-known in logic programming, and it is required to ensure finite grounding of a nonground program. A rule is safe if all its variables are safe, and a variable is safe if it is contained in a positive body literal. Formally a rule r is safe iff variables in $H(r) \cup B^{-}(r)$ are a subset of variables in $B^{+}(r)$.

Domain-expansion safety. In an ordinary logic program P, we usually assume that the set of constants $\mathcal{C}$ is implicitly given by P. In a HEX program, external atoms may invent new constant values in their output tuples. We therefore must relax this to '$\mathcal{C}$ is countable and partially given by P', as shown by the following example.

Example 12. *The Swimming Example does not specify all necessary constants in P_{swim}: the atom $need(loc, yogamat)$ is part of answer set I, however constant $yogamat \notin const(P_{swim})$. Grounding P_{swim} with $const(P_{swim})$ is insufficient, as such a grounding would not generate the rule*

$$need(loc, yogamat) \leftarrow \&rq[goto](yogamat),$$

which means that $I \models \&rq[goto](yogamat)$; however there is no nonground rule corresponding to rule r_5 which should fire if this external atom is true in I. □

Therefore grounding P with $const(P)$ can lead to incorrect results. Hence we want to obtain new constants during evaluation of external atoms, and we must use these constants to evaluate the remainder of a given HEX program. However, to ensure decidability, this process of obtaining new constants must always terminate.

Hence, we require programs to be *domain-expansion safe* [EIST06]: there must not be a cyclic dependency between rules and external atoms such that an input predicate of an external atom depends on a variable output of that same external atom, if the variable is not guarded by a domain predicate. Domain expansion safety must be kept in mind when developing HEX programs, therefore we introduce it already here. We will formally define and use this notion in Chapter 5.

Extensional Semantics for External Atoms. For efficiency reasons it is useful to restrict external atoms such that their semantics *depends only on extensions of predicates given in the input tuple* [EIST06]. This restriction is relevant for all chapters, the formal definition becomes relevant only in Chapter 5 where we will defined and discuss it in detail.

2.3 Computational Complexity

We recall the concept of a complexity class, membership and hardness properties for problems in such classes, and then recall complexity classes present in typical monotonic and nonmonotonic KR formalisms. For further background and more examples on computational complexity see [Pap94].

Complexity Classes. A complexity class measures resources needed to solve a computational problem, where resources can be time and/or memory usage.

The computational problems we will deal with are called *decision problems* and they can abstractly be described as membership of words in a language as follows. Given an alphabet Σ of symbols, and $\Sigma^\star$ the set of all expressions that can be formed using Σ, we call a subset of $\Sigma^\star$ a *language* $\mathcal{L} \subseteq \Sigma^\star$, and an element $w \in \Sigma^\star$ a *word*. Then the check whether w is an element of $\mathcal{L}$ is equivalent to a concrete decision problems we can encounter in practice.

Complexity classes are typically characterized using Turing machines, with a bound on the number of steps of the machine, or a bound on the number of memory cells on the tape that was used by the machine. For example $\mathbf{P}$ is the class of problems decidable by a deterministic Turing machine in a polynomial number of steps.

Membership, Hardness, and Completeness. A decision problem is *member* of some complexity class, only if there is an algorithm that solves the problem within the resource bounds of the class. In complexity classes that are closed under polynomial-time reductions, a problem is member of some class only if it is possible to create a polynomial-time reduction from the problem to a problem in that class.

A problem is *hard* for some class only if all problems in the class can be reduced to an instance of the problem, using a polynomial-time reduction Hardness of a problem intuitively means that no other problem in that class requires more resources, and that every problem in that class can be solved by reducing it to the hard problem.

A problem is complete for some class C, if it is both member of class C and hard for class C. Complete problems can solve every problem in their complexity class modulo reductions, and every problem in their class can be solved by reducing it to a complete problem.

For our purposes we use *polynomial-time reductions*, which means that a reduction must be in the class $\mathbf{P}$. (Without that restriction the reduction could solve the problem and thereby 'hide' complexity of the problem in the reduction.)

A complexity class C is *closed under conjunctions* if and only if the following holds: given a problem L in C, it holds that the problem L^n (the n-fold Cartesian product of L, where $I = (I_1, \ldots, I_n)$ is a 'yes' instance of L^n iff every instances I_j, $1 \leq j \leq n$ is a 'yes' instances of L) is also a problem in C.

A decision problem $L \subseteq \Sigma^\star \times \Sigma^\star$ is *polynomially balanced*, if some polynomial p exists such that $|I'| \leq p(|I|)$ for all $(I, I') \in L$. Moreover, L is a *polynomial projection* of $L' \subseteq \Sigma^\star \times \Sigma^\star$ if $L = \{I \mid \text{there exists an } I' \text{ such that } (I, I') \in L'\}$ and L' is polynomially balanced. (Intuitively, I' is a witness of polynomial size for I.) Given a complexity class C, let $\pi(C)$ contain all problems which are a polynomial projection of a problem L' in C. Then a complexity class C is *closed under projection* if and only if $\pi(C) \subseteq C$. For example, classes with an 'outer existential quantifier' such as $\mathbf{NP}$, $\mathbf{\Sigma_i^P}$ are closed under projection, while $\mathbf{P}$ or classes with an 'outer universal quantifier' such as $\mathbf{coNP}$, $\mathbf{\Pi_i^P}$ are not (under common complexity hypotheses).

Typical Complexity Classes. $\mathbf{P}$, $\mathbf{EXPTIME}$, and $\mathbf{PSPACE}$ are the classes of problems that can be decided using a deterministic Turing machine in polynomial time, exponential time, and polynomial space, respectively. $\mathbf{P}$ is considered to be the class of problems that are efficiently solvable, also called tractable. Typical examples for $\mathbf{P}$ problems are evaluation of Boolean circuits (circuit value problem), linear programming (in continuous domains), and satisfiability of a set of Horn clauses (Horn-SAT). $\mathbf{EXPTIME}$ and $\mathbf{PSPACE}$ are classes encountered in description logics, in fact concept satisfiability in the description logic $\mathcal{ALC}$ with a TBox (see our running example) is complete for $\mathbf{EXPTIME}$. Under common assumptions, the class $\mathbf{P}$ is not closed under projection, while $\mathbf{EXPTIME}$ and $\mathbf{PSPACE}$ are. All three classes are closed under conjunction.

$\mathbf{NP}$ (resp., $\mathbf{coNP}$) is the class of problems that can be decided on a nondeterministic Turing machine in polynomial time, where one (resp., all) execution paths accept. Problems complete for $\mathbf{NP}$ are deciding whether a Boolean formula without quantifiers is satisfiable (the SAT problem), and deciding whether an ordinary ground Answer Set Program has an answer set. Conversely, the problem whether a SAT instance is unsatisfiable is complete for $\mathbf{coNP}$. Both $\mathbf{NP}$ and $\mathbf{coNP}$ are closed under conjunction, however only $\mathbf{NP}$ is closed under projection, while $\mathbf{coNP}$ is not.

The *polynomial hierarchy* is a hierarchy of complexity classes, defined recursively as follows:

$$\begin{aligned}
&\mathbf{\Sigma_0^P} = \mathbf{\Pi_0^P} = \mathbf{P},\\
&\mathbf{\Sigma_i^P} = \mathbf{NP} \text{ with a } \mathbf{\Sigma_{i-1}^P} \text{ oracle, and}\\
&\mathbf{\Pi_i^P} = \mathbf{coNP} \text{ with a } \mathbf{\Sigma_{i-1}^P} \text{ oracle.}
\end{aligned}$$

Note that $\mathbf{\Sigma_1^P} = \mathbf{NP}$ and $\mathbf{\Pi_1^P} = \mathbf{coNP}$.

A typical knowledge representation task with $\mathbf{\Sigma_2^P}$ complexity is the check whether a ground disjunctive logic program has an answer set, or the check whether an atom is part of some answer set (i.e., brave query answering [DEGV01]).

All classes in the polynomial hierarchy are closed under conjunction. The $\mathbf{\Sigma_i^P}$ classes with $i \geq 1$ are closed under projection; while the class $\mathbf{\Sigma_0^P} = \mathbf{P}$ and the classes $\mathbf{\Pi_i^P}$ are not.

The family of classes $\mathbf{D_i^P}$. In this thesis we will also use a less commonly known complexity class, $\mathbf{D_i^P}$, which denotes the complexity class of decision problems which are the "conjunction" of a $\mathbf{\Sigma_i^P}$ and an independent $\mathbf{\Pi_i^P}$ decision problem, formally $\mathbf{D_i^P} = \{L_1 \times L_2 \mid L_1 \in \mathbf{\Sigma_i^P},\ L_2 \in \mathbf{\Pi_i^P}\}$. Deciding whether a pair (F_1, F_2) of a SAT instance F_1 and an independent UNSAT instance F_2 is a prototypical problem complete for $\mathbf{D_1^P}$.

We also use a generalized version of this family of classes: given complexity class C, we denote by $\mathcal{D}(C)$ the "difference class" of C, i.e., $\mathbf{D}(C) = \{L_1 \times L_2 \mid L_1 \in C,\ L_2 \in \mathbf{co}\text{-}C\}$ denotes the complexity class of decision problems that are the conjunction of a C decision problem L_1 and an independent $\mathbf{co}$-C decision problem L_2. For example, $\mathbf{D}(\mathbf{\Sigma_i^P}) = \mathbf{D_i^P}$, and $\mathbf{D}(\mathbf{NP}) = \mathbf{D_1^P}$. Note in particular that $\mathbf{D}(\mathbf{PSPACE}) = \mathbf{PSPACE}$ and that $\mathbf{D}(\mathbf{EXPTIME}) = \mathbf{EXPTIME}$.

3 Analyzing Inconsistency in Multi-Context Systems

MCSs enable knowledge integration at a general level, like, e.g., interlinking ontologies, databases, and logic programs. Due to their decentralized nature, information exchange can have unforeseen effects, and in particular cause an MCS to be inconsistent. In this chapter we elaborate on the problem of inconsistency in MCSs, and discuss ways to analyze such inconsistencies on a theoretical level.

Inconsistency in an MCS is the lack of an equilibrium. Suppose that, in our Medical Example, the expert system concludes that Sue must be given a special drug, but her patient record states that she is allergic to that drug, thus counter-indicating its use.

Example 13 (Inconsistent Medical Example)**.** *Consider the MCS M_2 which is a slightly modified version of MCS M_1 in Example 5. We modify kb_{lab} such that the blood analysis shows presence of a particular blood marker, and such that it stores a different birth date for Sue:*

$$\begin{aligned} kb_{lab} = \{ & \mathit{customer}(\mathit{sue}, \mathit{03/02/1985}); \\ & \mathit{test}(\mathit{sue}, \mathit{xray}, \mathit{pneum}), \mathit{test}(\mathit{sue}, \mathit{bloodtest}, \mathit{cmark}); \\ & \mathit{test}(\textsc{Id}, X, Y) \rightarrow \exists D : \mathit{customer}(\textsc{Id}, D)); \\ & \mathit{customer}(\textsc{Id}, X) \wedge \mathit{customer}(\textsc{Id}, Y) \rightarrow X = Y \}. \end{aligned}$$

We call M_2 the Inconsistent Medical Example; M_2 is inconsistent for two reasons:

- *C_{db} and C_{lab} are inconsistent in conjunction with r_1, as this bridge rule is applicable under any accepted belief set of C_{db} and adds to kb_{lab} a belief that violates the uniqueness constraint of the birth date.*
- *Even if we remove r_1 from the system (which fixes the above inconsistency), the remaining system is inconsistent because r_2 and r_3 become applicable due to C_{lab}, which causes C_{onto} to classify the illness as atypical pneumonia; as a consequence r_4 and r_5 become applicable which leads C_{dss} to conclude that $ab1$ is required for Sue; however due to Sue's allergy, r_6 does not become applicable, and C_{dss} infers that $ab1$ must not be given to Sue; therefore C_{dss} does not accept any belief state and the MCS is inconsistent.*

*In this system we can observe two independent inconsistencies: an inconsistency due to wrong data entry, and an inconsistency because the only viable treatment option is in conflict with the patient's allergy. Note that applicability of r_6 would resolve this inconsistency by activating $\mathit{allow}(\mathit{sue}, \mathit{ab1})$. However, the presence of belief $\mathit{allergy}(\mathit{sue}, \mathit{ab1})$ in S_{db} together with body literal '**not** $(\mathit{db} : \mathit{allergy}(\mathit{sue}, \mathit{ab1}))$' in r_6 prevents the applicability of r_6 (due to negation as failure).* $\square$

Inconsistency in an MCS makes inferences trivial, therefore an inconsistent MCS is useless.

In real world applications, system complexity tends to increase, both in terms of contexts and in terms of inter-connectivity. Anticipating all possible states of a system is unfeasible, therefore we need inconsistency handling methods to make such systems more robust.

The approach we introduce in this chapter is an extended elaboration on work published in [EFSW09, EFSW10, EFS10, EFS11] and aims at *analyzing* inconsistencies in MCSs in order to understand where and why such inconsistencies occur, and how they can be removed. This will allow to specify how to handle inconsistencies and to extend systems with inconsistency management mechanisms in later chapters of this thesis.

While the task reminds of a traditional data integration problem, an important point is that we focus on the exchange of information, i.e., adjusting bridge rules instead of modifying data in the contexts; in loose integrations (e.g., if companies link their business logics), changing contexts or their data to restore consistency may not be an option.

Therefore, we identify bridge rules as the source of inconsistency, and their modification as a possibility of counteracting. We assume, that every context is consistent if no bridge rules apply, therefore we can fully characterize the reason for an inconsistency in terms of bridge rules.

We make the following contributions.

- Inspired by debugging approaches used in the nonmonotonic reasoning community, especially in answer set programming [Syr06, BGP+07], we introduce two notions of explaining inconsistency in MCSs: a *consistency-based* notion, which characterizes inconsistency in terms of altered sets of bridge rules that are consistent, and an *entailment-based* notion which derives inconsistency in a given system. Possible nonmonotonicity makes intuitive and sound notions challenging; that our notions have appealing properties may be taken as some evidence for their suitability.

- We establish useful properties of our notions. First, we identify a way to convert between the consistency- and entailment-based notion, which is possible in many cases, although not in all cases. We discuss why such a conversion is not possible in general. From the conversion result, we obtain another useful property we call Duality: both notions identify the same bridge rules as relevant for inconsistency. This result in fact generalizes a similar result by Reiter [Rei87].

- We sharply characterize the computational complexity of identifying explanations for inconsistency, under varying assumptions for the complexity of contexts (note that explanations always do exist). It turns out that this problem has for a range of context complexities no (or only mildly) higher complexity than the contexts themselves. As a consequence, computing explanations is in some cases not harder than consistency checking.

- Finally, we investigate how it is possible to analyze inconsistency if only a part of the contexts of the inconsistent system are known. This information hiding scenario is a practically relevant application scenario, which occurs whenever a system does not reveal all internal information to the outside. Typical applications are credit card or access control systems.

Our results provide a basis for building enhanced MCS systems which are capable of analyzing and reasoning about emerging inconsistencies. Rather than automatically resolving inconsistency, as suggested e.g. in [BA08, BA10, BAH11], the results of this chapter set the stage for the subsequent chapters, where we show how to realize inconsistency analysis using HEX programs, and finally realize a (semi-)automatic approach with user support for locating and resolving inconsistency in MCSs.

In the following, we consider two possibilities for explaining inconsistency in MCSs: first, a consistency-based formulation, which identifies a part of the bridge rules which need to be changed to restore consistency. Second, an entailment-based formulation, which identifies a part of the bridge rules which is required to make the MCS inconsistent. Following common terminology, we call the first formulation *diagnosis* (cf. [Rei87]) and the second *inconsistency explanation*.

3.1 Diagnoses

We will use the following notation. Given an MCS M and a set R of bridge rules (compatible with M), by $M[R]$ we denote the MCS obtained from M by replacing its set of bridge rules br_M with R (e.g., $M[br_M] = M$ and $M[\emptyset]$ is M with no bridge rules). By $M \models \bot$ we denote that M has no equilibrium, i.e., is inconsistent, and by $M \not\models \bot$ the opposite.

As well-known, adding knowledge in nonmonotonic reasoning 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).

Definition 4. *Given an MCS M, a* diagnosis *of M is a pair (D_1, D_2), $D_1, D_2 \subseteq br_M$, s.t. $M[br_M \setminus D_1 \cup cf(D_2)] \not\models \bot$. We denote by $D^{\pm}(M)$ the set of all such diagnoses.*

To obtain a more relevant set of diagnoses, we prefer pointwise subset-minimal diagnoses. For pairs $A = (A_1, A_2)$ and $B = (B_1, B_2)$ of sets, the pointwise subset relation $A \subseteq B$ holds iff $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$.

Definition 5. *Given an MCS M, $D^{\pm}_m(M)$ is the set of all pointwise subset-minimal diagnoses of an MCS M.*

Example 14 (ctd). *In our running example,*

$$D^{\pm}_m(M_2) = \{\, (\{r_1, r_2\}, \emptyset), (\{r_1, r_3\}, \emptyset), (\{r_1, r_5\}, \emptyset), (\{r_1\}, \{r_6\}) \,\}.$$

Accordingly, we always need to deactivate r_1, and we can choose whether to additionally deactivate r_2, or r_3, or r_5, or whether to make r_6 unconditional, to obtain a consistent MCS.

Removing bridge rule r_1 simply removes the import of the different birth date into C_{lab}, but as this information is not used in bridge rules or other rules that infer additional information, the effect of removing r_1 is local to C_{lab}.

Diagnosis $(\{r_1, r_2\}, \emptyset)$ removes bridge rules r_1 and r_2. This way we ignore the X-Ray finding and obtain the following equilibrium:

$$\begin{aligned} S = \big(&\{person(sue,\ 02/03/1985),\ allergy(sue,\ ab1)\},\\ &\{customer(sue,\ 03/02/1985),\ test(sue, xray, pneum),\ test(sue,\ bloodtest,\ cmark)\},\\ &\{(mmark){:}APMark,\ (cmark){:}APMark,\ (sue,\ cmark){:}hasMarker\},\\ &\emptyset\big). \end{aligned}$$

This equilibrium represents that we do not treat the patient as no illness is detected in the equilibrium.

Diagnosis $(\{r_1, r_5\}, \emptyset)$ removes bridge rules r_1 and r_5. This ignores the information that treating the illness requires a special antibiotic. The equilibrium is as follows:

$$\begin{aligned} S = \big(&\{person(sue,\ 02/03/1985),\ allergy(sue,\ ab1)\},\\ &\{customer(sue,\ 03/02/1985),\ test(sue, xray, pneum),\ test(sue,\ bloodtest,\ cmark)\},\\ &\{(mmark){:}APMark,\ (cmark){:}APMark,\\ &\ (sue){:}\exists hasDisease.Pneum,\ (sue){:}\exists hasDisease.AtypPneum,\\ &\ (sue,\ cmark){:}hasMarker\},\\ &\{need(sue,\ ab),\ \neg give(\textsc{Id},\ ab1),\ give(\textsc{Id},\ ab2)\}\big). \end{aligned}$$

(Diagnosis $(\{r_1, r_3\}, \emptyset)$ creates the same accepted belief set at C_{db}, C_{lab} and at C_{dss}: it ignores information about the blood marker and therefore does not detect the atypical pneumonia in C_{onto}.)

Diagnosis $(\{r_1\}, \{r_6\})$ *removes* r_1 *and adds an unconditional copy of bridge rule* r_6. *This forces strong antibiotics to be allowed as a treatment. The modified system has the following equilibrium:*

$$\begin{aligned}S = \big(&\{person(sue, 02/03/1985), allergy(sue, ab1)\},\\ &\{customer(sue, 03/02/1985), test(sue, xray, pneum), test(sue, bloodtest, cmark)\},\\ &\{(mmark){:}APMark, (cmark){:}APMark,\\ &\ (sue){:}\exists hasDisease.Pneum, (sue){:}\exists hasDisease.AtypPneum,\\ &\ (sue, cmark){:}hasMarker\},\\ &\{allow(sue, ab1), need(sue, ab), need(sue, ab1), give(\textsc{Id}, ab1)\}\big).\end{aligned}$$

Any or none of the above possibilities might be the right thing to do: such decisions ought to be taken by a domain specialist (e.g., a doctor) and cannot be done automatically. Therefore analysis of inconsistency is important to identify reasons for it. □

3.2 Inconsistency Explanations

In the spirit of abductive reasoning, we also propose an entailment-based notion of explaining inconsistency: an *inconsistency explanation* (in short, an *explanation*) is a pair of sets of bridge rules, such that their presence or absence entails a relevant (cf. below) inconsistency in the given MCS.

Definition 6. *Given an MCS* M, *an* inconsistency explanation *of* M *is a pair* (E_1, E_2) *of sets* $E_1, E_2 \subseteq br_M$ *of bridge rules such that for all* (R_1, R_2) *where* $E_1 \subseteq R_1 \subseteq br_M$ *and* $R_2 \subseteq br_M \setminus E_2$, *it holds that* $M[R_1 \cup cf(R_2)] \models \bot$. *By* $E^{\pm}(M)$ *we denote the set of all inconsistency explanations of* M, *and by* $E^{\pm}_m(M)$ *the set of all pointwise subset-minimal ones.*

Example 15 (ctd). *In* M_2 *we have two minimal inconsistency explanations, namely*

$$E^{\pm}_m(M_2) = \{\, (\{r_1\}, \emptyset), (\{r_2, r_3, r_5\}, \{r_6\}) \,\}.$$

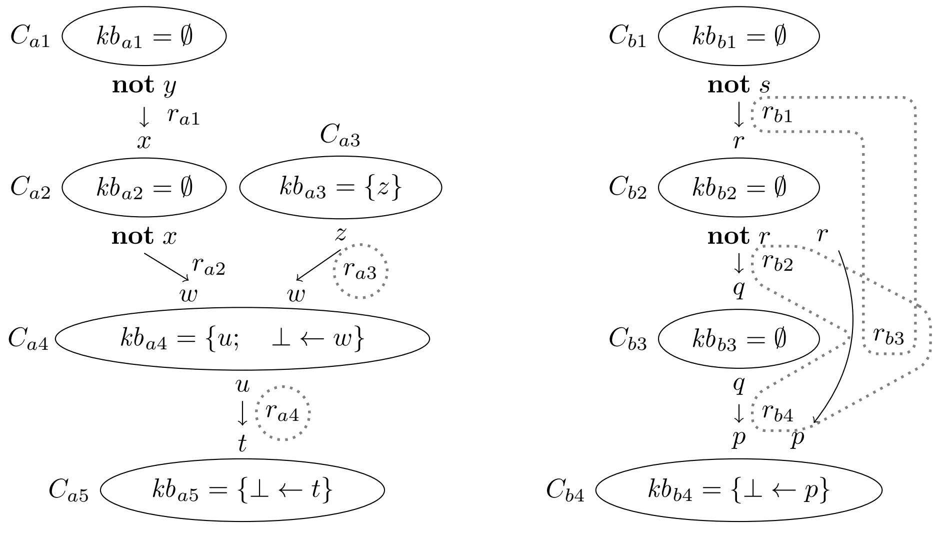
To trigger the inconsistency in C_{lab}, r_1 *and its addition of* $customer(sue, 02/03/1985)$ *to* kb_{lab} *is sufficient. For the inconsistency in* C_{dss}, *we need to import* $need(sue, ab1)$ *by* r_5 *and we must not import* $allow(sue, ab1)$ *by* r_6. *Furthermore,* r_5 *can only fire if* C_{onto} *accepts* $(sue){:}\exists hasDisease.AtypPneum$, *which is only possible if* r_2 *and* r_3 *fire. Therefore, for the second inconsistency,* r_2, r_3, *and* r_5 must be present *to get inconsistency, and the head of* r_6 must not be present. □

The intuition about E_1 is as follows: bridge rules in E_1 create an inconsistency in M ($M[E_1] \models \bot$), and this inconsistency is relevant for M. By relevance we mean that adding some bridge rules from br_M (the set of original bridge rules) to $M[E_1]$ never yields a consistent system.

This condition is necessary; for example the program $P = \{a \leftarrow not\, a\}$ is inconsistent under the answer set semantics, but its superset $P' = \{a \leftarrow not\, a;\ a\}$ is consistent. The inconsistency of P does not matter for P'. In terms of MCSs, a set of bridge rules may create an inconsistency in M, but this inconsistency is irrelevant, as it does not occur if more or all bridge rules are present.

Intuition about E_2 regards inconsistency wrt. the addition of unconditional bridge rules: $M[E_1]$ cannot be made consistent by adding bridge rules unconditionally, unless we use at least one bridge rule from E_2. In summary, bridge rules E_1 create a relevant inconsistency, and at least one bridge rule in E_2 must be added unconditionally to repair that inconsistency.

From Definition 6 we obtain the following Corollary.



(a) Example MCS M_a (b) Example MCS M_b

Figure 3.1: Example MCS topologies for illustrating properties and the usefulness of inconsistency explanations. Dotted areas indicate individual inconsistency explanations.

Corollary 1. *Given an explanation $E = (E_1, E_2)$ of an MCS M, every E' such that $E \subseteq E' \subseteq br_M \times br_M$ is an explanation as well.*

We now give further examples of inconsistency explanations and their properties.

Example 16 (ctd)**.** *Consider a modification of our Medical Example, where bridge rules are added for the administration of anti-allergenics. Bridge rule r_7 encodes that an allergy blocking (anti-allergenic) medication is given, if there is the need to apply strong antibiotics, the patient is allergic to it and nothing was done to block the allergic reaction; r_8 encodes that the patient database is informed if an anti-allergenic is applied:*

$$\begin{aligned}
r_7\colon (dss : give(sue, antiAllergenic)) &\leftarrow (dss : need(sue, ab1)),\\
& (db : allergy(sue, ab1),\\
& \textbf{not}\ (db : allergyBlocked(sue, ab1)).\\
r_8\colon \ (db : allergyBlocked(sue, ab1)) &\leftarrow (dss : give(sue, antiAllergenic)).
\end{aligned}$$

At first sight, this looks like a good idea for solving the allergy problem. However the resulting system now has three minimal inconsistency explanations, because we added a third inconsistency, namely we get the additional explanation

$$\big(\{r_2, r_3, r_5, r_7, r_8\}, \{r_7, r_8\}\big).$$

This illustrates how an inconsistency due to an odd cycle is reported by inconsistency explanations: the odd cycle through r_7 and r_8 causes both rules to be present in both components of the minimal explanation. This is because the instability of the cycle can be broken by either removing one rule in the cycle or by founding the loop (i.e., forcing one belief in the loop to be true) using an unconditional bridge rule. Minimal diagnoses for this modified system are $(\{r_1, r_2\}, \emptyset)$, $(\{r_1, r_3\}, \emptyset)$ $(\{r_1, r_5\}, \emptyset)$, $(\{r_1, r_7\}, \{r_6\})$, $(\{r_1\}, \{r_6, r_8\})$, $(\{r_1, r_8\}, \{r_6\})$, and $(\{r_1\}, \{r_6, r_7\})$. □

Example 17. *To show how explanations separate independent reasons for inconsistency, and to illustrate the fact that they point out only those inconsistencies that are relevant for inconsistency of the overall system, consider $M_a = (C_{b1}, C_{a2}, C_{a3}, C_{a4}, C_{a5})$ depicted in Figure 3.1a. All* *contexts use logic L^{asp}_{Σ} from Example 3 with $\Sigma = \{a, b, \dots, z\}$. This system is inconsistent, because u is a fact in C_{a4} and therefore r_{a4} adds fact t to C_{a5} which makes C_{a5} inconsistent. Furthermore the system is inconsistent, because z is a fact in C_{a3} and therefore r_{a3} adds fact w to C_{a4} which makes C_{a4} inconsistent. The corresponding minimal explanations separate these inconsistencies, we have $E^{\pm}_{m}(M_a) = \{\, (\{r_{a3}\}, \emptyset), (\{r_{a4}\}, \emptyset) \,\}$. The important property is that $M_a[\{r_{a2}\}]$ is an inconsistent system as well, because r_{a2} adds fact w to C_{a4}, making that context inconsistent. However, this inconsistency is not relevant for M_a and it is not reported as relevant by $E^{\pm}_{m}(M_a)$, because $M_a[\{r_{a1}, r_{a2}\}]$ is a consistent system, i.e., adding bridge rule r_{a1} from the original M_a, allows for repairing the inconsistency. This shows that inconsistency explanations characterize only relevant reasons for inconsistency.* □

Example 18. *To show how mutually exclusive bridge rules can be part of the same explanation, and to illustrate the advantage of subset-minimality over cardinality-minimality, consider MCS $M_b = (C_{b1}, C_{b2}, C_{b3}, C_{b4})$ depicted in Figure 3.1b. Again, all contexts use logic L^{asp}_{Σ} from Example 3 with $\Sigma = \{a, b, \dots, z\}$. This MCS is inconsistent, as p causes inconsistency in C_{b4} and p is always true by bridge rule r_{b3} which is always applicable because r is always true by bridge rule r_{b1}. This inconsistency cannot be repaired by using original bridge rules or original bridge rules without conditions. Therefore one minimal inconsistency explanation of M_b is $(\{r_{b1}, r_{b3}\}, \emptyset)$. However, there is another minimal explanation of M_b, it is $(\{r_{b2}, r_{b3}, r_{b4}\}, \emptyset)$: this explanation contains bridge rules r_{b2} and r_{b3} which are mutually exclusive wrt. their body conditions. However only both of them together ensure that C_{b4} becomes inconsistent, regardless of whether r_{b1} is in the system or not and whether fact r is accepted at C_{b2} or not. This example shows that cardinality-minimal explanations are inadequate for resolving inconsistency: removing r_{b1} from the only cardinality-minimal explanation does not make the system consistent. Note that the set of minimal diagnoses is $D^{\pm}_{m}(M_b) = \{\, (\{r_{b1}, r_{b2}\}, \emptyset), (\{r_{b3}\}, \emptyset), (\{r_{b1}, r_{b4}\}, \emptyset) \,\}$.* □

3.2.1 Deletion-Diagnoses / Deletion-Explanations

For domains where removal of bridge rules is preferred to unconditional addition of rules, we specialize $D^{\pm}$ to obtain diagnoses of the form $(D_1, \emptyset)$ only. We again prefer subset-minimal diagnoses.

Definition 7. *Given an MCS M, an s-diagnosis of M is a set $D \subseteq br_M$ s.t. $M[br_M \setminus D] \not\models \bot$. The set of all s-diagnoses (resp., $\subseteq$-minimal s-diagnoses) is $D^{-}(M)$ (resp., $D^{-}_{m}(M)$).*

Example 19. *In our example, $D^{-}_{m}(M) = \{\{r_1\}, \{r_2\}, \{r_4\}\}$.* □

We also specialize the inconsistency explanation to the first component, i.e., we do not consider adding rules unconditionally, so all explanations are of the form (E_1, br_M).

Definition 8. *Given an MCS M, an s-inconsistency explanation of M is a set $E \subseteq br_M$ s.t. each R where $E \subseteq R \subseteq br_M$, satisfies $M[R] \models \bot$. The set of s-inconsistency explanations is denoted by $E^{+}(M)$, and the set of $\subseteq$-minimal s-inconsistency explanations of M is denoted by $E^{+}_{m}(M)$.*

Example 20 (ctd). *The only minimal s-inconsistency explanation in our running example is $\{r_1, r_2, r_4\}$.* □

3.3 Properties

In this section we first show that our notions of diagnoses and explanations are related in a special way, as it is possible to compute one from the other in certain cases. We then use this result to obtain that minimal diagnoses and minimal explanations point out the same bridge rules, a property we call duality. Finally we prove a useful non-intersection property of minimal diagnoses.

3.3.1 Converting between Diagnoses and Explanations

In the following we show that it is possible to characterize explanations in terms of diagnoses, and vice versa minimal diagnoses in terms of minimal explanations.

For the following theorem we generalize the notion of a hitting set from sets [Rei87] to pairs of sets. Given a collection $\mathcal{C} = \{(A_1, B_1), \dots, (A_n, B_n)\}$ of pairs of sets (A_i, B_i), $A_i, B_i \subseteq U$ over a set U, a *hitting set of* $\mathcal{C}$ is a pair of sets (X, Y), $X, Y \subseteq U$ such that for every pair $(A_i, B_i) \in \mathcal{C}$, (i) $A_i \cap X \neq \emptyset$ or (ii) $B_i \cap Y \neq \emptyset$. A hitting set (X, Y) of $\mathcal{C}$ is minimal, if no $(X', Y') \subset (X, Y)$ is a hitting set of $\mathcal{C}$.

We consider hitting sets over pairs of sets of bridge rules, and denote by $\mathit{HS}_M(\mathcal{C})$ (respectively, $\mathit{minHS}_M(\mathcal{C})$) the set of all (respectively, all minimal) hitting sets of $\mathcal{C}$ over br_M. Note that in particular $\mathit{HS}_M(\emptyset) = \{(\emptyset, \emptyset)\}$, and $\mathit{HS}_M(\{(\emptyset, \emptyset)\}) = \emptyset$.

Theorem 1. *For every MCS M,*

(a) a pair (E_1, E_2) with $E_1, E_2 \subseteq br_M$ is an inconsistency explanation of M iff $(E_1, E_2) \in \mathit{HS}_M(D^{\pm}(M))$, i.e., (E_1, E_2) is a hitting set of $D^{\pm}(M)$; and

(b) a pair (E_1, E_2) with $E_1, E_2 \subseteq br_M$ is a minimal inconsistency explanation of M iff $(E_1, E_2) \in \mathit{minHS}_M(D^{\pm}(M))$, i.e., (E_1, E_2) is a minimal hitting set of $D^{\pm}(M)$.

Proof. For convenience we assume in this proof for variables E_i, D_i, and R_i with $i \in \{1, 2\}$ that $E_i, D_i, R_i \subseteq br_M$. Furthermore, we denote by $\overline{X}$ the complement of set X wrt. br_M, i.e., $\overline{X} = br_M \setminus X$.

(a) We transform the condition. Given a pair (E_1, E_2). For all diagnoses $(D_1, D_2) \in D^{\pm}(M)$, $D_1 \cap E_1$ or $D_2 \cap E_2$ or both are nonempty iff

$$\text{for all } (D_1, D_2) \text{ we have that } M[\overline{D_1} \cup cf(D_2)] \not\models \bot \text{ implies } D_1 \cap E_1 \neq \emptyset \text{ or } D_2 \cap E_2 \neq \emptyset$$

which (by reversing the implication and simplifying) is equivalent to

$$\text{for all } (D_1, D_2) \text{ we have that } (D_1 \cap E_1 = \emptyset \text{ and } D_2 \cap E_2 = \emptyset) \\ \text{implies } M[\overline{D_1} \cup cf(D_2)] \models \bot.$$

As $A \cap B = \emptyset$ with $A, B \subseteq br_M$ is equivalent to $A \subseteq \overline{B}$ we next obtain

$$\text{for all } (D_1, D_2) \text{ we have that } (E_1 \subseteq \overline{D_1} \text{ and } D_2 \subseteq \overline{E_2}) \text{ implies } M[\overline{D_1} \cup cf(D_2)] \models \bot.$$

If we let $D_1 = \overline{R_1}$ and $D_2 = R_2$ this amounts to

$$\text{for all } (R_1, R_2) \text{ we have that } (E_1 \subseteq R_1 \text{ and } R_2 \subseteq \overline{E_2}) \text{ implies } M[R_1 \cup cf(R_2)] \models \bot.$$

This proves the result (a) as this last condition is the one of an explanation (E_1, E_2) in Definition 6. Note that, if $(\emptyset, \emptyset) \in D^{\pm}(M)$, then no explanation exists; this is intentional and corresponds to the definitions of diagnosis and explanation for consistent systems.

(b) As $\mathit{minHS}_M(X)$ contains the $\subseteq$-minimal elements in $\mathit{HS}_M(X)$, and $E^{\pm}_m(M)$ contains the $\subseteq$-minimal elements in $E^{\pm}(M)$, (b) follows from (a). □

Clearly, a hitting set of a collection X is the same as a hitting set of the collection of $\subseteq$-minimal elements in X; from Theorem 1. we therefore immediately obtain the following.

Corollary 2. *For every MCS M,*

(a) a pair (E_1, E_2) with $E_1, E_2 \subseteq br_M$ is an inconsistency explanation of M iff $(E_1, E_2) \in HS_M(D^{\pm}_m(M))$; and

(b) a pair (E_1, E_2) with $E_1, E_2 \subseteq br_M$ is a minimal inconsistency explanation of M iff $(E_1, E_2) \in minHS_M(D^{\pm}_m(M))$.

Proof. Let $min(X)$ be the set of $\subseteq$-minimal elements in a collection X of sets. Then for every $(A, B) \in X \setminus min(X)$ there is a pair $(A', B') \in min(X)$ with $(A', B') \subseteq (A, B)$. Given $HS_M(min(X))$, every pair $(A, B) \in X \setminus min(X)$ is hit by every pair $(C, D) \in HS_M(min(X))$. Therefore $HS_M(min(X)) = HS_M(X)$. Then (a) immediately follows from Theorem 1 (a), and (b) immediately follows from Theorem 1 (b). □

We obtain the following generalization of a well-known result for minimal hitting sets [Ber89].

Lemma 1. *For every collection $X = \{X^1, \ldots, X^n\}$ of pairs $X^i = (X^i_1, X^i_2)$ of sets, $1 \leq i \leq n$, such that X is an antichain wrt. $\subseteq$, i.e., elements in X are pairwise incomparable ($X^i \subseteq X^j$ with $1 \leq i, j \leq n$ implies $X^i = X^j$) it holds that $minHS_M(minHS_M(X)) = X$.*

Proof. A collection of sets $C = \{C_1, \ldots, C_n\}$ over a universe, i.e., $C_i \subseteq U$, $1 \leq i \leq n$, can be seen as a *hypergraph* $\mathcal{H} = (U, C)$ with vertices U and hyperedges $C_i \in C$. If no hyperedge C_i is contained in any hyperedge C_j, $i \neq j$, it is called *simple*. A hitting set on C is called *transversal*, and the hypergraph (U, C') containing as hyperedges C' all minimal hitting sets of the hypergraph $\mathcal{H}$ is called *transversal hypergraph* $Tr(\mathcal{H})$.

We can map a collection $X = \{X^1, \ldots, X^n\}$ of pairs $X^i = (X^i_1, X^i_2)$ of sets, X^i_1,$X^i_2 \subseteq U$ bijectively to a collection $\mu(X) = \{\mu(X^1), \ldots, \mu(X^n)\}$ over $U \cup \{u' \mid u \in U\}$ where $\mu(X^i_1, X^i_2) = X^i_1 \cup \{u' \mid u \in X^i_2\}$. Then, (A, B) is a hitting set of X iff $\mu(A, B)$ is a hitting set of $\mu(X)$, and well-known results for transversal hypergraphs [Ber89] carry over to minimal hitting sets over pairs.

In particular, given a simple hypergraph $\mathcal{H} = \mu(X)$, it holds that $Tr(Tr(\mu(X))) = \mu(X)$. This directly translates into the lemma, because $\mu(X)$ is a simple hypergraph due to incomparability (also called the antichain property) of X, and μ is bijective, therefore transversal hypergraphs can be mapped back to minimal hitting sets. □

Combined with Corollary 2 (b) we thus obtain.

Theorem 2. *A pair (D_1, D_2) with $D_1, D_2 \subseteq br_M$ is a minimal diagnosis of M iff (D_1, D_2) is a minimal hitting set of $E^{\pm}_m(M)$, formally $D^{\pm}_m(M) = minHS_M(E^{\pm}_m(M))$.*

Proof. From Corollary 2 (b) we have that $E^{\pm}_m(M) = minHS_M(D^{\pm}_m(M))$. Applying $minHS_M$ on both sides of this formula and then using Lemma 1 yields $minHS_M(E^{\pm}_m(M)) = minHS_M(minHS_M(D^{\pm}_m(M))) = D^{\pm}_m(M)$. □

As for computation, Theorem 1 provides a way to compute the set of explanations $E^{\pm}(M)$ from the set of diagnoses $D^{\pm}(M)$, while Theorem 2 allows us to compute the set of minimal diagnoses $D^{\pm}_m(M)$ from the set of minimal explanations $E^{\pm}_m(M)$. Corollary 2 shows that, for computing $E^{\pm}(M)$ and $E^{\pm}_m(M)$, it is sufficient to know the set of minimal diagnoses $D^{\pm}_m(M)$.

Note that Theorem 2 describes relationships between minimal hitting sets, similar to the relationship between diagnoses and conflict sets in Reiter's approach to diagnosis [Rei87]. In contrast, note that Theorem 1 (a) uses hitting sets without the requirement of $\subseteq$-minimality.

Example 21 (ctd). *In our running example, we had*

$$E^{\pm}_m(M_2) = \{\, (\{r_1\}, \emptyset), (\{r_2, r_3, r_5\}, \{r_6\}) \,\}, \text{ and}$$
$$D^{\pm}_m(M_2) = \{\, (\{r_1, r_2\}, \emptyset), (\{r_1, r_3\}, \emptyset), (\{r_1, r_5\}, \emptyset), (\{r_1\}, \{r_6\}) \,\}.$$

For illustrating Corollary 2, we consider all minimal diagnoses (D_1, D_2). An explanation (E_1, E_2) has a nonempty intersection $E_1 \cap D_1 \neq \emptyset$ or $E_2 \cap D_2 \neq \emptyset$ with every minimal diagnosis. This is easily achieved by using r_1 in the first component E_1, i.e., by $(\{r_1\}, \emptyset)$, which is indeed a minimal explanation. If we do not use r_1 in E_1, we can still hit all minimal diagnoses, which yields the second minimal explanation $(\{r_2, r_3, r_5\}, \{r_6\})$. Furthermore, all component-wise supersets of these explanations are explanations, as they hit every minimal diagnosis as well.

For illustrating Theorem 2, consider the set of minimal explanations; every minimal diagnosis (D_1, D_2) must fulfill $E_1 \cap D_1 \neq \emptyset$ or $E_2 \cap D_2 \neq \emptyset$, and there is no smaller pair (D_1, D_2) with that property. This condition is true for all minimal diagnoses in $D^{\pm}_m(M)$, and as they hit r_1 in the first component (which is the only way to hit the first explanation), and as they also hit exactly one of the rules in the other explanation. □

Asymmetry

We now investigate why it is possible to obtain the set of explanations from the set of diagnoses, while the other direction only works under $\subseteq$-minimality. The following example illustrates this.

Example 22. *Consider the MCS M with one ASP context $C_1 = \{\leftarrow a\}$, and the bridge rules $r_1 = (1{:}a) \leftarrow (1{:}a)$ and $r_2 = (1{:}a) \leftarrow \mathbf{not}\ (1{:}b)$. Then $D^{\pm}(M) = \big\{(\{r_2\}, \emptyset), (\{r_1, r_2\}, \emptyset)\big\}$, while $E^{\pm}_m(M) = \big\{(\{r_2\}, \emptyset)\big\}$, because only r_2 is relevant for inconsistency. $E^{\pm}(M)$ contains all pointwise supersets of $(\{r_2\}, \emptyset)$, i.e., $(\{r_2\}, \emptyset)$, $(\{r_1, r_2\}, \emptyset)$, $(\{r_2\}, \{r_1\})$, $(\{r_2\}, \{r_2\})$, $(\{r_1, r_2\}, \{r_1\})$, $(\{r_1, r_2\}, \{r_2\})$, and $(\{r_1, r_2\}, \{r_1, r_2\})$. Now the (nonminimal) hitting set of the set $E^{\pm}(M)$ of explanations is the set $E^{\pm}(M)$ itself, while the set $D^{\pm}(M)$ of diagnoses only contains two elements.* □

The reason behind this asymmetry is that the notion of explanation is an order-increasing concept, i.e., all supersets of an explanation are also explanations, while the notion of diagnosis is not, i.e., a superset of a diagnosis is not necessarily a diagnosis.

This difference is due to the fact that explanations characterize only relevant inconsistencies (as discussed in Section 3.2) and by its definition, all supersets of an explanation are explanations. Therefore the set of minimal explanations characterizes the set of explanations. For the notion of diagnosis this is not the case: a system might contain inconsistent bridge rule configurations which do not appear in explanations because they are irrelevant in the original system. Non-minimal diagnoses provide modifications of the system which might cause and at the same time suppress such an irrelevant inconsistency in order to achieve overall consistency. Minimal explanations, non-minimal explanations, and minimal diagnoses will never contain such irrelevant inconsistencies.

In summary, a minimal hitting set of the set of diagnoses characterizes the set of minimal explanations (Corollary 2 (b)) and a minimal hitting set of the set of explanations characterizes the set of minimal diagnoses (Theorem 2). With non-minimality it looks different: the non-minimal hitting sets of $D^{\pm}(M)$ characterize the set $E^{\pm}(M)$ of explanations (see Theorem 1 (a)), however the non-minimal hitting sets of $E^{\pm}(M)$ do not characterize the set $D^{\pm}(M)$ of diagnoses (see Example 22 for a counterexample).

3.3.2 Duality

While the previous section showed a fine-grained and detailed relationship between diagnoses and explanations, we next investigate a more coarse relationship between minimal notions: minimal diagnoses and minimal explanations point out the same set of bridge rules as relevant for inconsistency in an MCS.

Intuitively, adding rules E_1 for an explanation (E_1, E_2) to contexts causes inconsistency, while removing rules D_1 for a diagnosis (D_1, D_2) from an MCS can cause consistency; analogous for the second component, i.e., adding rules to E_2 may prevent consistency while adding rules to D_2 may prevent inconsistency; hence explanations and diagnoses represent dual aspects.

Both notions, point out rules that are erroneous in the way that those rules contribute to inconsistency. This naturally gives rise to the question whether diagnoses and explanations point out the same rules of an MCS as erroneous, or if those notions characterize different aspects.

To formalize this question, we introduce relevancy for inconsistency. Let M be an MCS with bridge rules br_M. We call a bridge rule $r \in br_M$ *relevant for diagnosis* (*d-relevant*) iff there exists a diagnosis $(D_1, D_2) \in D^{\pm}_m(M)$ with $r \in D_1 \cup D_2$. Analogously r is *relevant for explanation* (*e-relevant*) iff there exists an explanation $(E_1, E_2) \in E^{\pm}_m(M)$ with $r \in E_1 \cup E_2$. To avoid superfluous rules, both relevance criteria are defined with respect to minimality of the underlying notion.

Example 23 (ctd). *In Example 21 we can see that $D^{\pm}_m(M_2)$ and $E^{\pm}_m(M_2)$ both point out the set $\{r_1, r_2, r_3, r_5\}$ of bridge rules in their first component, and the set $\{r_6\}$ in their second component.* □

Formalizing this, for any set X of pairs (A, B) of sets A and B (e.g., for some set of diagnoses), we write $\bigcup X$ for $(\bigcup\{A \mid (A, B) \in X\}, \bigcup\{B \mid (A, B) \in X\})$.

Proposition 1. *For every inconsistent MCS M, $\bigcup D^{\pm}_m(M) = \bigcup E^{\pm}_m(M)$, i.e., the unions of all minimal diagnoses and all minimal inconsistency explanations coincide.*

Proof. This is a direct specialization of Theorems 1 and 2. □

This strengthens our view that both notions capture exactly those parts of an MCS that are relevant for inconsistency as duality shows that, in total, two very different perspectives on inconsistency state exactly the same parts of the MCS as erroneous.

In practice this allows one to compute the set of all bridge rules which are relevant for making an MCS consistent (i.e., appear in at least one diagnosis) in two ways: either compute all minimal explanations, or compute all minimal diagnoses. In other terms the duality result allows to exclude all bridge rules that are not part of any diagnosis (or explanation) from further investigation as they can be skipped safely.

Our running example suggests that duality also holds for simplified diagnoses and explanations, which indeed is true:

Theorem 3. *Given an inconsistent MCS M, $\bigcup D^{-}_m(M) = \bigcup E^{+}_m(M)$, i.e., the unions of all minimal s-diagnoses and all minimal s-inconsistency explanations coincide.*

Proof. This is a direct consequence of Proposition 1: set in its proof the second components of diagnoses and explanations to $\emptyset$. □

The result of this theorem is similar to the proof of Theorem 4.4 in Reiter's seminal paper [Rei87], which states that diagnoses are minimal hitting sets on the set of conflict sets, where a conflict set is similar to what we call s-inconsistency explanation. The main difference is that Reiter's conflict sets are defined on monotonic (first-order) logic, while our explanations are defined on possibly nonmonotonic logics. However, the condition that an explanation must not be repairable by adding bridge rules of the original system, effectively ensures that explanations become monotonic.

3.3.3 Non-overlap in Minimal Diagnoses

We mention a simple yet useful property of minimal diagnoses. According to Definition 4, given (D_1, D_2) such that $r \in D_2$, whether (D_1, D_2) is a diagnosis is independent from whether $r \in D_1$. Therefore,

Proposition 2. *In a minimal diagnosis (D_1, D_2) of an MCS M, $D_1 \cap D_2 = \emptyset$, i.e., no rule occurs in both components.*

Proof. Let $(D_1, D_2) \in D^{\pm}_{m}(M)$ and let S be a witnessing belief state for it, i.e., S is an equilibrium of $M[br_M \setminus D_1 \cup cf(D_2)]$. For contradiction we assume $D_1 \cap D_2 \neq \emptyset$. Consider any rule $r \in D_1 \cap D_2$ and let $h_c(r) = i$ and $h_b(r) = p$. Let $r' = cf(r) = (i{:}p) \leftarrow .$, then $body(r') = \emptyset$ and r' is applicable in any belief state. Therefore $r' \in app(br_i(M[br_M \setminus D_1 \cup cf(D_2)]), S)$. (Recall that $app(R, S,)$ is the set of bridge rules from R that are applicable wrt. belief state S.) For $(D'_1, D'_2) = (D_1 \setminus \{r\}, D_2)$ we thus obtain that $r' \in app(br_i(M[br_M \setminus D'_1 \cup cf(D'_2)]), S)$. As all other bridge rules are as before, we conclude $app(br_i(M[br_M \setminus D'_1 \cup cf(D'_2)]), S) = app(br_i(M[br_M \setminus D_1 \cup cf(D_2)]), S)$ for all $i \in c(M)$. Consequently S is an equilibrium of $M[br_M \setminus D'_1 \cup cf(D'_2)]$ and $(D'_1, D'_2) \in D^{\pm}(M)$. But $(D'_1, D'_2) \subsetneq (D_1, D_2)$ contradicts (D_1, D_2) being minimal, thus our assumption was wrong and $D_1 \cap D_2 = \emptyset$ for minimal diagnoses. $\square$

This is not true for inconsistency explanations: consider Example 16 where the inconsistency caused by an odd loop yields an explanation (E_1, E_2) with $\{r_7, r_8\}$ being a subset of E_1 as well as of E_2.

3.4 Computational Complexity

We next consider the complexity of consistency checking, and of diagnosis and explanation recognition in MCSs in a parametric fashion. To this end, we first show that we can abstract an MCS to beliefs used in bridge rules. We use *context complexity* as a parameter to characterize the overall complexity of these decision problems. For hardness we establish generic results for all complexity classes that are closed under conjunction and projection. Table 3.1 summarizes our results for complexity classes that are typically used in knowledge representation.

3.4.1 Output-projected Equilibria

Computing equilibria by guessing and verifying so-called "kernels of context belief sets" has been outlined in [EBDT$^+$09]. For the purpose of recognizing diagnoses and explanations, it suffices to check for consistency, i.e., for existence of an arbitrary equilibrium in an MCS.

Here we first define *output beliefs*, which are the beliefs used in bodies of bridge rules. Then we show that, for checking consistency of an MCS, it is sufficient to consider equilibria *projected to output beliefs*.

Definition 9. *Given an MCS $M = (C_1, \dots, C_n)$, the* set of inputs of C_i*, denoted IN_i, is the set of bridge rule heads that can be added by bridge rules in br_i, and the* set of output beliefs of C_i*, denoted OUT_i, is the set of beliefs p of C_i which occur in the body of some bridge rule $r \in br_M$. Formally,*

$$\begin{aligned} IN_i &= \{h_b(r) \mid r \in br_i\}, \textit{ and} \\ OUT_i &= \{p \mid \textit{there exists a bridge rule } r \in br_M \textit{ with } (i:p) \in body(r)\}. \end{aligned}$$

Context complexity $\mathcal{CC}(M)$	Consistency checking MCSEQ	$(A,B) \overset{?}{\in}$			
		$D^{\pm}(M)$ MCSD	$D^{\pm}_m(M)$ MCSD$_m$	$E^{\pm}(M)$ MCSE	$E^{\pm}_m(M)$ MCSE$_m$
P	**NP**	**NP**	$\mathbf{D_1^P}$	**coNP**	$\mathbf{D_1^P}$
NP	**NP**	**NP**	$\mathbf{D_1^P}$	**coNP**	$\mathbf{D_1^P}$
$\mathbf{\Sigma_i^P}$, $i \geq 1$	$\mathbf{\Sigma_i^P}$	$\mathbf{\Sigma_i^P}$	$\mathbf{D_i^P}$	$\mathbf{\Pi_i^P}$	$\mathbf{D_i^P}$
PSPACE	**PSPACE**				
EXPTIME	**EXPTIME**				
Proposition	3	4	5	6	7

Table 3.1: Complexity of consistency checking and recognizing (minimal) diagnoses and explanations, given (A, B) and an MCS M for complexity classes of typical KR formalisms. Membership holds for all cases, completeness holds if at least one context is complete for the respective context complexity.

Example 24 (ctd). *In our running example M_2, we have the following sets of output beliefs:*

$$\begin{aligned} OUT_{db} &= \{person(sue, 02/03/1985), allergy(sue, ab1)\}, \\ OUT_{lab} &= \{test(sue, xray, pneum), test(sue, bloodtest, cmark)\}, \\ OUT_{onto} &= \{(sue){:}\exists hasDisease.BacterialDisease, \\ &\quad (sue){:}\exists hasDisease.AtypPneum\}, \text{ and} \\ OUT_{dss} &= \emptyset. \end{aligned}$$

Note that $OUT_{dss} = \emptyset$ because no bridge rule contains in its body a belief of context C_{dss}. □

Using the notion of output beliefs, we let $S'_i = S_i \cap OUT_i$ be the projection of S_i to OUT_i, and for $S = (S_1, \ldots, S_n)$ we let $S' = (S'_1, \ldots, S'_n)$ be the *output-projected belief state* S' of S.

An output-projected belief state provides sufficient information for evaluating the applicability of bridge rules. We next show how to obtain witnesses for equilibria using this projection.

Definition 10. *An output-projected belief state $S' = (S'_1, \ldots, S'_n)$ of an MCS M is an output-projected equilibrium iff for all $1 \leq i \leq n$,*

$$S'_i \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i, S')\})\big|_{OUT_i}$$

(Recall that $A|_B$ denotes the projection of the family of sets A to the set B.) S' contains information about all (and only about) output beliefs. As these are the beliefs that determine bridge rule applicability, $app(R, S) = app(R, S')$; thus we obtain:

Lemma 2. *For each equilibrium S of an MCS M, S' is an output-projected equilibrium. Conversely, for each output-projected equilibrium S' of M, there exists some equilibrium T of M such that $T' = S'$.*

Given MCS M, we denote by EQ$'(M)$ the set of output-projected equilibria of M.

Proof. ($\Rightarrow$) Let $S = (S_1, \ldots, S_n)$, then $S_i \in \mathbf{ACC}(kb_i \cup H)$, where the set H of active bridge rule heads at each context is $app(br_i, S)$. Bridge rule applicability depends on output beliefs only, therefore $app(br_i, S) = app(br_i, S')$. Thus $S' = (S'_1, \ldots, S'_n)$ with $S'_i = S_i \cap OUT_i$ is an output-projected equilibrium of M.

($\Leftarrow$) The proof is similar to ($\Rightarrow$). Let $S' = (S'_1, \ldots, S'_n)$, then, as S' is an output-projected equilibrium, for each i, $1 \leq i \leq n$, $S'_i \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i, S')\})|_{OUT_i}$, and therefore for each S'_i there exists a belief set S_i such that $S_i \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i, S')\})$ and $S'_i = S_i \cap OUT_i$. If we take for each i some S_i satisfying the above condition, we obtain $T = (S_1, \ldots, S_n)$. As S' and T agree on all ouput beliefs of all contexts, we have that $app(br_i, S') = app(br_i, T)$ and obtain that T is an equilibrium of M. By our construction of T, it also holds that $T' = S'$. □

Example 25 (ctd). *In the consistent Medical Example M_1, the equilibrium*

$$\begin{aligned}S = (&\{person(sue, 02/03/1985), allergy(sue, ab1)\},\\ &\{customer(sue, 02/03/1985), test(sue, xray, pneum), \neg test(sue, bloodtest, cmark)\},\\ &\{(mmark){:}APMark, (cmark){:}APMark,\\ &\ (sue){:}\exists hasDisease.Pneum, (sue){:}\exists hasDisease.BacterialDisease\},\\ &\{need(sue, ab), \neg give(\textsc{Id}, ab1), give(\textsc{Id}, ab2)\}).\end{aligned}$$

is witnessed by the output-projected equilibrium

$$\begin{aligned}S = (&\{person(sue, 02/03/1985), allergy(sue, ab1)\},\\ &\{test(sue, xray, pneum)\},\\ &\{(sue){:}\exists hasDisease.BacterialDisease\},\\ &\emptyset).\end{aligned}$$

Observe that, for consistency of the overall system, it is not relevant which belief set is accepted at C_{dss}, only that some belief set is accepted (as $OUT_{dss} = \emptyset$, all projected belief sets at C_{dss} are empty). □

Therefore each equilibrium is witnessed by a single output-projected equilibrium, and each output-projected equilibrium witnesses at least one equilibrium. For consistency checking (i.e., equilibrium existence) in MCSs it is therefore sufficient to consider output-projected equilibria.

3.4.2 Context Complexity

The complexity of consistency checking for an MCS clearly depends on the complexity of its contexts. We next define a notion of *context complexity* by considering the roles which contexts play in the problem of consistency checking.

For all complexity considerations, we represent logics L_i of contexts C_i *implicitly*; they are fixed and we do not consider these (possibly infinite) objects to be part of the input of the decision problems we investigate. Accordingly, the instance size of a given MCS M will be denoted by $|M| = |kb_M| + |br_M|$ where $|kb_M|$ denotes the size of knowledge bases in M and $|br_M|$ denotes the size of its bridge rules.

Consistency of an MCS M can be decided by a Turing machine with input M which (a) guesses an output-projected belief state $S' \in OUT_1 \times \cdots \times OUT_n$, (b) evaluates the bridge rules on S', yielding for each context C_i a set of active bridge rule heads H_i wrt. S', and (c) checks for each context whether it accepts the guessed S'_i wrt. H_i. We call the complexity of step (c) *context complexity*, formalized as follows.

Definition 11. *Given a context $C_i = (kb_i, br_i, L_i)$ and a pair (H, S'), with $H \subseteq IN_i$ and $S' \subseteq OUT_i$, the* context complexity $\mathcal{CC}(C_i)$ *of C_i is the computational complexity of deciding whether there exists an $S_i \in \mathbf{ACC}_i(kb_i \cup H)$ such that $S_i \cap OUT_i = S'_i$.*

Example 26. *In a context which uses a logical relational database without constraints (i.e., a simplification of Example 1), acceptability checking amounts to looking up a belief in the knowledge base, therefore such a context has complexity $\mathcal{O}(n)$ (in the general case). With constraints the check becomes more expensive in general. A relational database with a fixed set of constraints can be captured by knowledge bases and belief sets which are sets of tuples in relations. Acceptability of a belief set computes whether a belief set is the closure of a knowledge base wrt. a fixed set of (possibly recursive) Datalog view definitions. Such a context is complete for* $\mathbf{P}$ *[DEGV01, Theorem 4.4, data complexity].*

A propositional answer set program can be captured by a context where knowledge bases are sets of rules and belief sets are sets of propositional atoms. Acceptability of such a context then checks whether a set of propositions is the projection of some answer set of the knowledge base of that context to the output beliefs of that context. Such an acceptability check is complete for $\mathbf{NP}$ *[DEGV01, Theorem 5.7]. Similarly, satisfiability checking of Boolean formulas can be captured by* $\mathbf{NP}$ *contexts. In default Logic programs and in disjunctive logic programs (such as introduced in Example 3), the recognition of a projected model is complete for* $\mathbf{\Sigma_2^P}$ *[Got92, Theorem 5.2 (a)], therefore a context using one of these logics is complete for* $\mathbf{\Sigma_2^P}$.

An agent using one of the widely-known modal logics K_n, T_n, or $S4_n$ with n knowledge operators and $n \geq 1$ can be represented as a context. Assuming that such a context has knowledge bases and belief sets consisting of formulas, and the context accepts the closure $\mathcal{C}_X$ of a set of formulas X in the knowledge base, this context is complete for $\mathbf{PSPACE}$ *[HM92, Theorem 6.17].*

For contexts hosting ontological reasoning in the Description Logic $\mathcal{ALC}$ (as in Example 2) we have that acceptability checking corresponds to a set of instance checks. As individual instance checking is $\mathbf{EXPTIME}$*-complete [BCM$^+$03, Section 3.5.1] and* $\mathbf{EXPTIME}$ *is closed under conjunction, such a context is in* $\mathbf{EXPTIME}$. *For $|OUT_i| = 1$ we see that such a context is also* $\mathbf{EXPTIME}$*-hard. Therefore a context using logic $L_{\mathcal{A}}$ has context complexity* $\mathbf{EXPTIME}$. □

Given an MCS M, we say that M has *upper context complexity* C, denoted $\mathcal{CC}^+(M) = C$, if $\mathcal{CC}(C_i) \subseteq C$ for every context C_i of M. We say M has *lower context complexity* C, denoted $\mathcal{CC}^-(M) = C$, if $C \subseteq \mathcal{CC}(C_i)$ for some context C_i of M. We say that M has *context complexity* C, denoted $\mathcal{CC}(M)$, iff $C = \mathcal{CC}^+(M) = \mathcal{CC}^-(M)$. Accordingly, an MCS contains no context that cannot be decided in $\mathcal{CC}^+(M)$, and an MCS with context complexity $\mathcal{CC}(M)$ contains some context complete for $\mathcal{CC}(M)$ if $\mathcal{CC}(M)$ has complete problems.

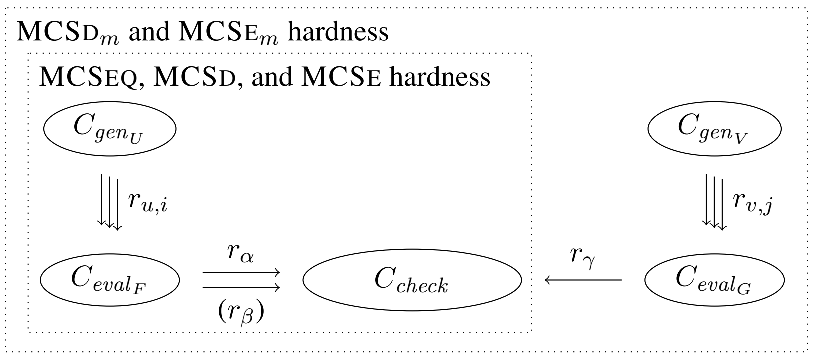
Example 27 (ctd)**.** *In the Medical Example, we have $\mathcal{CC}(C_{db}) = \mathcal{CC}(C_{lab}) = \mathbf{P}$, $\mathcal{CC}(C_{onto}) = \mathbf{EXPTIME}$, and $\mathcal{CC}(C_{dss}) = \mathbf{\Sigma_2^P}$. Overall, we have $\mathcal{CC}^-(M_2) = \mathcal{CC}^+(M_2) = \mathcal{CC}(M_2) = \mathbf{EXPTIME}$. (This complexity is due to C_{onto}).* □

3.4.3 Overview of Results

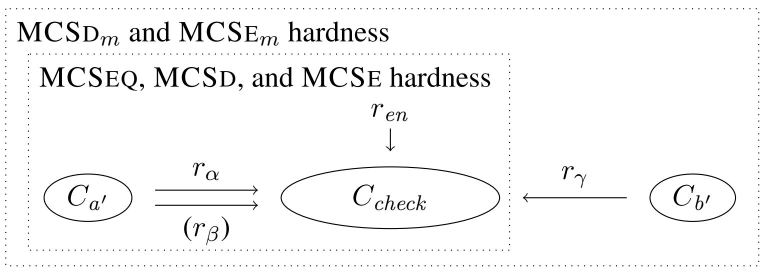
We now give an overview of complexity results, and brief intuition about the proofs.

We study the decision problem for

- consistency of MCSs (MCSEQ);
- recognition of a diagnosis of a MCS (MCSD);
- recognition of a minimal diagnosis of a MCS (MCSD$_m$);
- recognition of an inconsistency explanation of a MCS (MCSE); and
- recognition of a minimal inconsistency explanation of a MCS (MCSE$_m$).



(a) Structures for lower context complexity $\mathcal{CC}^-(M) = \mathbf{P}$



(b) Structures for generic lower context complexity $\mathcal{CC}^-(M)$

Figure 3.2: MCS structures for hardness reductions, where dotted areas indicate parts of the MCS used for respective reductions.

Note that *existence* of diagnoses and explanations is trivial by our basic assumptions that M is inconsistent and that $M[\emptyset]$ is consistent.

Table 3.1 summarizes our results for context complexities that are present in typical monotonic and nonmonotonic KR formalisms. Corresponding theorems are given in Section 3.4.5, which are more general than the results shown in Table 3.1.

For a given context complexity $\mathcal{CC}(M)$ of an MCS M, MCSEQ has the same computational complexity as MCSD. If the context complexity is $\mathbf{NP}$ or above, this complexity is equal to context complexity; for context complexity $\mathbf{P}$, it is $\mathbf{NP}$. Intuitively, this is explained as follows: for context complexity $\mathbf{NP}$ and above, guessing a belief state and checking whether it is an equilibrium can be incorporated into the complexity of the contexts without exceeding checking cost; if the context complexity is $\mathbf{P}$, this complexity is $\mathbf{NP}$.

Recognizing minimal diagnoses MCSD$_m$ is complete for the complexity of MCSD, which captures diagnosis recognition, and an additional complementary problem of refuting MCSD, which captures diagnosis minimality recognition. For context complexity $\mathbf{P}$ we have that the problem MCSD$_m$ is complete for $\mathbf{D}_1^{\mathbf{P}}$.

The complexity of MCSE is in the complementary class of the corresponding problem MCSD. Intuitively this is because diagnosis involves existential quantification, while explanation involves universal quantification. Accordingly, complexity of MCSE$_m$ is complementary to MCSD$_m$. As the complexity classes of MCSD$_m$ are closed under complement, MCSE$_m$ and MCSD$_m$ have the same complexity.

These results show that minimal diagnosis and minimal explanation recognition are harder than checking consistency (under usual complexity assumptions), while they are polynomially reducible to each other.

3.4.4 Proof Outline

We treat context complexity of $\mathbf{NP}$ and above uniformly and the case of $\mathbf{P}$ separately. For hardness results we use MCS structures depicted in Figure 3.2.

For context complexity $\mathbf{P}$ we use reductions from SAT, UNSAT or SAT-UNSAT instances F and/or G to MCSs with context complexity $\mathbf{P}$. These reductions use the structure shown in Figure 3.2a, where contexts C_{gen_U} and C_{gen_V} generate a set of possible truth assignments to sets of variables, C_{eval_F} and C_{eval_G} evaluate formulas F and G under these assignments, and C_{check} checks whether the formulas are satisfiable and/or unsatisfiable. We obtain the hardness via the nondeterministic guess that arises from the different belief sets accepted by contexts C_{gen_U} and C_{gen_V}. (See also the description of logic L_{GUESS} in the following.) Our reductions use an acyclic system topology without negation as failure in bridge rules. Note that hardness can also be obtained using a nonmonotonic guess in cyclic bridge rules which contain negation as failure; in that case all contexts of the reduction can be deterministic, i.e., every context accepts at most one belief set for any input. We give such an alternative hardness reduction in the proof of Proposition 3, where we prove $\mathbf{NP}$ hardness of MCSEq in an MCS of context complexity $\mathbf{P}$.

Hardness results for context complexity $\mathbf{NP}$ and above are established by a generic reduction: we reduce the problem of acceptability checking of contexts C_a (resp., C_b) with context complexity X to decision problems in an MCS M with complexity X. These reductions use the scheme shown in Figure 3.2b, where $C_{a'}$ (resp., $C_{b'}$) evaluates the acceptability checking problem of C_a (resp., C_b), and C_{check} tests whether the original problems are “yes” or “no” instances.

For hardness reductions we use the following context logics.

- L_{ASP} is a logic for contexts that contain stratified propositional ASPs with constraints. More in detail, if $L_{ASP} = (\mathbf{BS}, \mathbf{KB}, \mathbf{ACC})$, then $\mathbf{BS}$ is the collection of sets of atoms over a propositional alphabet Σ, $\mathbf{KB}$ is a set of logic programming rules over Σ, and given a knowledge base $kb \in \mathbf{KB}$, we define $\mathbf{ACC}(kb) = \mathcal{AS}(kb)$, i.e., the context accepts the set of answer sets of the logic program kb. If clear, Σ is omitted. In case of stratified propositional ASPs with constraints, a program has at most one answer set. From [DEGV01, Theorem 4.2] it follows that whether an atom A is part of this model is $\mathbf{P}$-complete. Thus, deciding given OUT_i whether $S'_i \subseteq OUT_i$ is a projected accepted belief set, is $\mathbf{P}$-complete; therefore context complexity is $\mathbf{P}$.

- $L_{GUESS(B)}$ is a trivial logic over the set B that accepts all subsets of its knowledge base. In detail, if logic $L_{GUESS(B)} = (\mathbf{BS}, \mathbf{KB}, \mathbf{ACC})$ then $\mathbf{BS} = \mathbf{KB} = 2^B$ is the powerset of B, and $\mathbf{ACC}(kb) = 2^{kb}$ for $kb \in \mathbf{KB}$. If clear, B is omitted. The check whether belief set S'_i is accepted by knowledge base kb_i can be done in time $\mathcal{O}(|kb_i| + |S'_i|)$.

3.4.5 Detailed Results

We first formally define the decision problems we consider and then report the complexity results.

Definition 12. *Given a MCS M,* MCSEq *is the problem of deciding whether M has an equilibrium.*

Definition 13. *Given a MCS M and a pair (A, B) with $A, B \subseteq br_M$,*

- MCSD *decides whether $(A, B) \in D^{\pm}(M)$, i.e., whether (A, B) is a diagnosis of M;*

- MCSD$_m$ *decides whether $(A, B) \in D^{\pm}_m(M)$, i.e., whether (A, B) is a minimal diagnosis of M;*

- MCSE *decides whether* $(A, B) \in E^{\pm}(M)$*, i.e., whether* (A, B) *is an inconsistency explanation of* M*; and*
- MCSE$_m$ *decides whether* $(A, B) \in E^{\pm}_m(M)$*, i.e., whether* (A, B) *is a minimal inconsistency explanation of* M.

We next formulate the complexity results.

Proposition 3. *The problem* MCSEQ*, given MCS* M*, is*
- $\mathbf{NP}$*-complete if* $\mathcal{CC}(M) = \mathbf{P}$*, and*
- $\mathcal{CC}(M)$*-complete if* $\mathcal{CC}(M)$ *is a class with complete problems that is closed under conjunction and projection.*

Proof. (*Membership*) Given a MCS $M = (C_1, \dots, C_n)$ we compute OUT_i for all C_i in $\mathcal{O}(|br_M|)$, then we guess output projected belief sets $S'_i \subseteq OUT_i$, $1 \leq i \leq n$, yielding an output-projected belief state S'. We evaluate bridge rule applicability of all rules in S' in time $\mathcal{O}(|br_M|)$ and thereby obtain a set of active bridge rule heads H_i for each context C_i, $1 \leq i \leq n$. Finally we check acceptability of S'_i for all contexts C_i, i.e., whether $S'_i \in \mathbf{ACC}_i(kb_i \cup H_i)|_{OUT_i}$. We accept if all contexts accept, otherwise we reject. This check is a conjunction of n independent acceptability checks of maximum complexity equal to the smallest upper bound on context complexities (i.e., upper context complexity) $\mathcal{CC}^+(M)$. If $\mathcal{CC}^+(M)$ is closed under conjunction we can unite these checks into one check of complexity $\mathcal{CC}^+(M)$ over an instance of size $\mathcal{O}(|M|)$. Then the overall acceptability check is in $\mathcal{CC}^+(M)$ as well. This way we check the output-projected equilibrium property for all possible output-projected equilibria. Therefore if no computation path accepts, then the MCS M is inconsistent. If there is one path that accepts, then the output-projected belief state S' corresponding to the guesses on this path is an output-projected equilibrium which fulfills all conditions of Definition 10. Therefore M is consistent iff at least one path accepts. Hence if $\mathcal{CC}^+(M)$ is closed under conjunction and projection, then the guess of size $\mathcal{O}(|br_M|)$ can be projected away (i.e., incorporated into I', see Section 2.3) and the complexity of MCSEQ is in $\mathcal{CC}^+(M)$. For $\mathcal{CC}^+(M) = \mathbf{P}$ (which is not closed under projection) the complexity of MCSEQ is in $\mathbf{NP}$.

($\mathbf{NP}$-*hardness* for $\mathcal{CC}^-(M) = \mathbf{P}$) We show that consistency checking in an MCS M with lower context complexity $\mathcal{CC}^-(M) = \mathbf{P}$ is $\mathbf{NP}$-hard. We use the part of the MCS structure in Figure 3.2a labeled with MCSEQ. We reduce a 3-SAT instance $F = c_1 \wedge \dots \wedge c_n$ on variables $\mathcal{X} = \{x_1, \dots, x_k\}$ and clauses $c_i = c_{i,1} \vee c_{i,2} \vee c_{i,3}$ with $c_{i,j} \in \mathcal{X} \cup \{\neg x \mid x \in \mathcal{X}\}$ to consistency checking in an MCS $M = (C_{gen_U}, C_{eval_F}, C_{check})$. Context $C_{gen_U} = (L_{GUESS}, kb_{gen_U}, br_{gen_U})$ with $kb_{gen_U} = \mathcal{X}$ and $br_{gen_U} = \emptyset$ has linear context complexity, while $C_{eval_F} = (L_{ASP}, kb_{eval_F}, br_{eval_F})$ and $C_{check} = (L_{ASP}, kb_{check}, br_{check})$ have context complexity $\mathbf{P}$. M contains the following bridge rules:

$$r_{u,i}: \quad (eval_F : x_i) \leftarrow (gen_U : x_i). \qquad \forall i : 1 \leq i \leq k \tag{3.1}$$

$$r_\alpha: \quad (check : nsat) \leftarrow \mathbf{not}\ (eval_F : sat). \tag{3.2}$$

Hence $br_{eval_F} = \{r_{u,i} \mid \forall i : 1 \leq i \leq k\}$ and $br_{check} = \{r_\alpha\}$. The knowledge base kb_{eval_F} is as follows:

$$sat_i \leftarrow l_{i,1}. \quad sat_i \leftarrow l_{i,2}. \quad sat_i \leftarrow l_{i,3}. \qquad \forall i : 1 \leq i \leq n \tag{3.3}$$

$$sat \leftarrow sat_1, \dots, sat_n. \tag{3.4}$$

$$\text{where } l_{i,j} \text{ is } \begin{cases} x_v & \text{if } c_{i,j} = x_v \\ not\, x_v & \text{if } c_{i,j} = \neg x_v \end{cases}$$

The knowledge base kb_{check} is as follows:

$$\perp \leftarrow nsat. \tag{3.5}$$

Context C_{gen_U} accepts all possible subsets of $\mathcal{X}$, representing all possible truth assignments for the variables $\mathcal{X}$. (3.1) imports the truth assignment into C_{eval_F}, which evaluates F under that truth assignment using rules (3.3) and (3.4). Then C_{eval_F} puts the belief sat in its belief set iff F is satisfied given the truth assignment accepted by C_{gen_U}. Finally C_{check} imports the belief $nsat$ iff sat is not accepted at C_{eval_F}. Therefore constraint (3.5) makes C_{check} inconsistent, i.e., accepts no belief set, iff sat is not true in C_{eval_F} iff there is no satisfying truth assignment for F. Therefore, if F has a satisfying assignment with variables $\mathcal{T} \subseteq \mathcal{X}$ set to $\mathbf{t}$ and variables $\mathcal{X} \setminus \mathcal{T}$ set to $\mathbf{f}$, then M has an equilibrium $S = (S_{gen_U}, S_{eval_F}, S_{check})$ where $S_{gen_U} = \mathcal{T}$, $S_{eval_F} = \mathcal{T} \cup \{sat_i \mid 1 \leq i \leq n\} \cup \{sat\}$, and $S_{check} = \emptyset$. Conversely, if M has an equilibrium $S = (S_{gen_U}, S_{eval_F}, S_{check})$, then S_{check} does not contain $nsat$ due to constraint (3.5). Hence S_{eval_F} must contain sat, thus S_{eval_F} contains $\{sat\} \cup \{sat_i \mid 1 \leq i \leq n\}$ due to (3.4). It follows that the set of bridge rule heads active at C_{eval_F} corresponds to a satisfying assignment of F. This shows that MCS M is consistent iff F is a satisfiable 3-SAT instance. As the size of M is linear in the size of the formula F and 3-SAT is an $\mathbf{NP}$-hard problem, hardness for equilibrium existence follows.

($\mathcal{CC}^-(M)$-*hardness*) We show that consistency checking in an MCS M with lower context complexity $\mathcal{CC}^-(M)$ is $\mathcal{CC}^-(M)$-hard if $\mathcal{CC}^-(M)$ is a class with complete problems that is closed under conjunction and projection. For that we use part of the MCS structure labeled with MCSEQ in Figure 3.2b. We reduce context acceptability checking, i.e., an instance (H_a, S_a), $C_a = (kb_a, br_a, L_a)$ with IN_a, OUT_a and context complexity $\mathcal{CC}(C_a)$ to consistency checking in an MCS $M = (C_{a'}, C_{check})$ such that the context complexity $\mathcal{CC}(C_{a'}) = \mathcal{CC}(C_a)$ and $\mathcal{CC}(C_{check}) = \mathbf{P}$. Intuitively, $C_{a'}$ gets input H_a, bridge rule r_α is applicable only if S_a is accepted by H_a, and C_{check} verifies whether r_α is applicable. Then M is consistent iff (H_a, S_a), C_a is a 'yes' instance. Formally, $C_{a'} = (kb_a \cup H, \emptyset, L_a)$ uses knowledge base and logic from C_a, while $C_{check} = (kb_{check}, br_{check}, L_{ASP})$ use the specific logic L_{ASP} that can be decided in $\mathbf{P}$. Bridge rules of M are as follows:

$$r_\alpha: \quad (check : equal_{S'_a}) \leftarrow l_1, \dots, l_j, \dots l_{|OUT_a|}.$$

$$\text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_a \wedge s_j \in S_a \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_a \wedge s_j \notin S_a \end{cases} \tag{3.6}$$

$$r_{en}: \quad (check : en) \leftarrow. \tag{3.7}$$

The knowledge base kb_{check} is as follows:

$$\bot \leftarrow not\ equal_{S'_a}, en. \tag{3.8}$$

Bridge rule r_{en} ensures that C_{check} fulfills our assumption that a context without input is consistent. Wlog. we assume that C_a accepts some belief set given input H_a. $C_{a'}$ contains the logic of C_a and its knowledge base already contains bridge rule heads H_a. Therefore $C_{a'}$ accepts a belief set S_a^{full}, such that $S_a^{full} \cup OUT_a = S_a$, iff (H_a, S_a), C_a is a 'yes' instance. Therefore, belief state $S = (S_a^{full}, \{equal_{S'_a}, en\})$ is an equilibrium iff (H_a, S_a), C_a is a 'yes' instance. All belief states where $C_{a'}$ accepts a belief set T with $T \cap OUT_a \neq S_a$ trigger constraint (3.8) and therefore lead to an inconsistency. Therefore M has an equilibrium, and this equilibrium is S iff context (H_a, S_a), C_a is a 'yes' instance for context acceptability checking. We thus have reduced context acceptability checking to consistency checking in M and hardness follows.

(Alternative reduction for $\mathbf{NP}$-hardness with $\mathbf{P}$-contexts) Note that the above reduction for $\mathbf{P}$-contexts uses an acyclic MCS with stratified negation in bridge rules. Furthermore the context C_{gen_U} accepts $2^{|\mathcal{X}|}$ belief sets and the contexts C_{eval_F} and C_{check} accept at most one belief set for any input. In the above reduction $\mathbf{NP}$-hardness arises from the nondeterminism of C_{gen_U}, i.e., from the number of belief sets potentially accepted by context C_{gen_U}. It is possible to obtain the hardness not from nondeterminism of a context but from nondeterminism of bridge rules. To illustrate this, we next give an alternative hardness reduction. (In subsequent proofs we only give one reduction, and there hardness arises from nondeterminism of contexts.) We reduce the same 3-SAT instance F to an MCS $M = (C_1)$ consisting of one context $C_1 = (L_{ASP}, kb_1, br_1)$. It contains the following bridge rules br_1:

$$(1 : x_i) \leftarrow \textbf{not}\ (1 : \bar{x}_i). \qquad \forall i : 1 \leq i \leq k \tag{3.9}$$

$$(1 : \bar{x}_i) \leftarrow \textbf{not}\ (1 : x_i). \qquad \forall i : 1 \leq i \leq k \tag{3.10}$$

$$(1 : \mathit{en}) \leftarrow . \tag{3.11}$$

The knowledge base kb_1 is as follows:

$$\mathit{sat}_i \leftarrow l_{i,1}. \quad \mathit{sat}_i \leftarrow l_{i,2}. \quad \mathit{sat}_i \leftarrow l_{i,3}. \qquad \forall i : 1 \leq i \leq n \tag{3.12}$$

$$\mathit{sat} \leftarrow \mathit{sat}_1, \dots, \mathit{sat}_n. \tag{3.13}$$

$$\bot \leftarrow \mathit{en}, \mathit{not\ sat}. \tag{3.14}$$

$$\text{where } l_{i,j} \text{ is } \begin{cases} x_v & \text{if } c_{i,j} = x_v \\ \bar{x}_v & \text{if } c_{i,j} = \neg x_v \end{cases}$$

Without bridge rules, en is not true in the knowledge base, hence the body of constraint (3.14) is never satisfied. Therefore C_1 satisfies our assumption that a context without bridge rules is consistent. The facts x_i and $\bar{x}_i$ are contained only in heads of bridge rules (3.9) and (3.10) and not in heads of rules in kb_1. Furthermore bridge rules (3.9) and (3.10) are mutually exclusive in their applicability for each $1 \leq i \leq n$. Therefore these bridge rules guess for each x_i whether x_i or $\bar{x}_i$ is part of the set of facts added to kb_1. (3.12) and (3.13) evaluate F wrt. the guess for x_i: if x_i is added by a bridge rule, then $x_i = \mathbf{t}$ in F, otherwise $x_i = \mathbf{f}$. The value of F wrt. the guess for x_i and $\bar{x}_i$ is represented as sat in kb_1. The constraint (3.14) makes the context inconsistent if en is true and sat is not true. Therefore if F is satisfied with variables $\mathcal{T} \subseteq \mathcal{X}$ set to $\mathbf{t}$ and variables $\mathcal{X} \setminus \mathcal{T}$ set to $\mathbf{f}$, then M has an equilibrium (S_1) where $S_1 = \{x_i \mid x_i \in \mathcal{T}\} \cup \{\bar{x}_i \mid x_i \in \mathcal{X} \setminus \mathcal{T}\} \cup \{\mathit{sat}_i \mid 1 \leq i \leq n\} \cup \{\mathit{sat}, \mathit{en}\}$. Conversely, if M has an equilibrium (S_1), then S_1 contains en due to the unconditional bridge rule (3.11). Hence S_1 must contain sat due to constraint (3.14), and thus S_1 contains $\{\mathit{sat}_i \mid 1 \leq i \leq n\}$ due to (3.13). Therefore the guess of bridge rules (3.9) and (3.10) corresponds to a satisfying assignment of F. This shows that M is consistent iff F is satisfiable. Context C_1 uses logic L_{ASP}, therefore $\mathcal{CC}^-(M) = \mathcal{CC}(C_1) = \mathbf{P}$. As the size of M is linear in the size of the formula F and 3-SAT is an $\mathbf{NP}$-hard problem, hardness for equilibrium existence follows. □

Diagnosis recognition can be done by transforming the MCS using the given diagnosis candidate and deciding MCSEQ. On the other hand, MCSEQ can be reduced to diagnosis recognition of the empty diagnosis candidate $(\emptyset, \emptyset)$. Therefore, diagnosis recognition has the same complexity as consistency checking.

Proposition 4. *The problem* MCSD*, given MCS M, is*

- $\mathbf{NP}$*-complete if* $\mathcal{CC}(M) = \mathbf{P}$*, and*
- $\mathcal{CC}(M)$*-complete if* $\mathcal{CC}(M)$ *is a class with complete problems that is closed under conjunction and projection.*

Proof. (*Membership*) Given MCS M and $D_1, D_2 \subseteq br_M$, we compute $M' = M[br_M \setminus D_1 \cup cf(D_2)]$ and return the result of deciding MCSEQ on M'. By Definition 4, this returns 'yes' iff $(D_1, D_2) \in D^{\pm}(M)$. The transformation can be done in time $\mathcal{O}(|M|)$ therefore MCSD is in the same complexity class as MCSEQ.

(*Hardness*) Deciding whether $(\emptyset, \emptyset)$ is a diagnosis of M can be decided by checking consistency of M, because $(\emptyset, \emptyset) \in D^{\pm}(M)$ iff M is consistent. Therefore MCSD is as hard as MCSEQ for respective context complexity. □

Deciding whether a pair (A, B) is a $\subseteq$-minimal diagnosis of an MCS M requires two checks: (a) whether (A, B) is a diagnosis, and (b) whether no pair $(A', B') \subset (A, B)$ is a diagnosis. The pair (A, B) is a minimal diagnosis iff both checks succeed. This intuitively leads to the following complexity result.

Proposition 5. *The problem* MCSD$_m$*, given MCS M, is*

- $\mathbf{D_1^P}$*-complete if* $\mathcal{CC}(M) = \mathbf{P}$*,*
- $\mathbf{D}(\mathcal{CC}(M))$*-complete if* $\mathcal{CC}(M)$ *is a class with complete problems that is closed under conjunction and projection.*

Note that, as shown in Table 3.1, the second item implies that MCSD$_m$ is $\mathbf{D_i^P}$-complete if $\mathcal{CC}(M)$ is complete for $\mathbf{\Sigma_i^P}$ with $i \geq 1$.

Proof. (*Membership*) Given MCS M and $D_1, D_2 \subseteq br_M$, we solve two independent decision problems: (a) we decide whether (D_1, D_2) is a diagnosis of M, and (b) we check whether a smaller diagnosis $(D', D'') \subset (D_1, D_2)$ exists in M. We return 'yes' if (a) returns 'yes' and (b) returns 'no'. Thus, this procedure returns 'yes' iff (a) (D_1, D_2) is a diagnosis and (b) no $\subseteq$-smaller diagnosis exists. Therefore the computation yields the correct result. For (a) we decide MCSD on M and (D_1, D_2). For (b) we guess for each bridge rule in D_1 whether it is contained in D', and for each bridge rule in D_2 whether it is contained in D''. Then we continue with the decision procedure MCSD on M and (D', D''), i.e., we guess presence of output belief sets, evaluate bridge rule applicability, and check acceptability for each context. Consequently for deciding (b) we decide the complement of a polynomial projection of MCSD. Therefore MCSD$_m$ is in the complexity class of solving the MCSD problem and independently solving the complement of a polynomially projected MCSD problem. Hence if $\mathcal{CC}^+(M)$ is closed under conjunction and projection, then the complexity of MCSD$_m$ is in $\mathcal{D}(\mathcal{CC}^+(M))$. For $\mathcal{CC}^+(M) = \mathbf{P}$ (which is not closed under projection) the complexity of MCSD$_m$ is in $\mathbf{D_1^P}$.

($\mathbf{D_1^P}$-*hardness* for $\mathcal{CC}^-(M) = \mathbf{P}$) We reuse ideas from the MCSEQ hardness proof for 3-SAT, but we now use the complete topology shown in Figure 3.2a. We reduce two 3-SAT instances F and G on variables $\mathcal{X}$ and $\mathcal{Y}$, respectively, to minimal diagnosis recognition on MCS $M = (C_{gen_V}, C_{eval_F}, C_{gen_U}, C_{eval_G}, C_{check})$. Intuitively, C_{gen_U} and C_{eval_F} provide $\mathbf{NP}$-hardness for satisfiability of F, while C_{gen_V} and C_{eval_G} provide $\mathbf{coNP}$-hardness for unsatisfiability of G. C_{gen_U} and C_{eval_F} are constructed from F exactly as for the proof of MCSEQ hardness. Similarly, C_{gen_V} and C_{eval_G} are constructed from G with bridge rules $r_{v,j}$ transferring a guessed set $V \subseteq \mathcal{Y}$ from C_{gen_V} to C_{eval_G}. The bridge rules in M are as follows:

$$r_{u,i}: \quad (eval_F : x_i) \leftarrow (gen_U : x_i). \qquad \forall i : 1 \leq i \leq |\mathcal{X}| \tag{3.15}$$

$$r_{v,j}: \quad (eval_G : y_j) \leftarrow (gen_V : y_j). \qquad \forall j : 1 \leq j \leq |\mathcal{Y}| \tag{3.16}$$

$$r_\alpha: \quad (check : nsat_F) \leftarrow \mathbf{not}\ (eval_F : sat). \tag{3.17}$$

$$r_\gamma: \quad (check : nsat_G) \leftarrow \mathbf{not}\ (eval_G : sat). \tag{3.18}$$

Context C_{check} has the following knowledge base kb_{check}:

$$\bot \leftarrow nsat_F. \tag{3.19}$$

$$\bot \leftarrow nsat_G. \tag{3.20}$$

If F and G are both satisfiable, M is consistent so $(\emptyset, \emptyset) \in D_m^{\pm}(M)$. If F is satisfiable and G is unsatisfiable, M is inconsistent and a minimal diagnosis for M is $(\{r_\gamma\}, \emptyset)$. If F and G are both unsatisfiable, M is inconsistent and a minimal diagnosis is $(\{r_\alpha, r_\gamma\}, \emptyset) \in D_m^{\pm}(M)$. If F is unsatisfiable and G is satisfiable, M is inconsistent; $(\{r_\gamma\}, \emptyset)$ is no minimal diagnosis, because every diagnosis containing r_γ in D_1 must also contain r_α in D_1 to restore consistency in M. Therefore $(\{r_\gamma\}, \emptyset)$ is a minimal diagnosis of M iff F is satisfiable and G is unsatisfiable. Therefore recognizing a minimal diagnosis in an MCS with $\mathcal{CC}^-(M) = \mathbf{P}$ is hard for $\mathbf{D_1^P}$.

Note that it is possible to do this reduction with one context that evaluates F and G and checks the result, using bridge rules that guess U and V and bridge rules that individually activate satisfiability checking for F and G. However this would make the reduction less readable.

($\mathbf{D}(\mathcal{CC}^-(M))$*-hardness*) We show that recognizing minimal diagnoses in an MCS M with lower context complexity $\mathcal{CC}^-(M)$ is hard for $\mathbf{D}(\mathcal{CC}^-(M))$ if $\mathcal{CC}^-(M)$ is a class with complete problems that is closed under conjunction and projection. We reduce two context complexity check instances (H_a, S_a), C_a with IN_a, OUT_a and (H_b, S_b), C_b with IN_b, OUT_b to an MCS $M = (C_{a'}, C_{b'}, C_{check})$ with the topology shown in Figure 3.2b. Similar to the generic hardness reduction for MCSEQ, we reduce H_a and $C_a = (kb_a, br_a, L_a)$ to the context $C_{a'} = (kb_a, br_{a'}, L_a)$ with $br_{a'} = \emptyset$ and we reduce H_b and $C_b = (kb_b, br_b, L_b)$ to the context $C_{b'} = (kb_b, br_{b'}, L_b)$ with $br_{b'} = \emptyset$. Then $\mathcal{CC}(C_{a'}) = \mathcal{CC}(C_a)$ and $\mathcal{CC}(C_{b'}) = \mathcal{CC}(C_b)$. Furthermore $C_{a'}$ accepts a belief set S_a^{full} with $S_a^{full} \cap OUT_a = S_a$ iff (H_a, S_a), C_a is a 'yes' instance. Similarly $C_{b'}$ accepts a belief set S_b^{full} with $S_b^{full} \cap OUT_b = S_b$ iff (H_b, S_b), C_b is a 'yes' instance. The bridge rules br_{check} are as follows.

$$r_\alpha: \quad (check : equal_{S'_a}) \leftarrow l_1, \ldots, l_j, \ldots l_{|OUT_a|}. \quad \text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_a \wedge s_j \in S_a \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_a \wedge s_j \notin S_a \end{cases} \tag{3.21}$$

$$r_\gamma: \quad (check : equal_{S'_b}) \leftarrow l_1, \ldots, l_j, \ldots l_{|OUT_b|}. \quad \text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_b \wedge s_j \in S_b \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_b \wedge s_j \notin S_b \end{cases} \tag{3.22}$$

$$r_{en}: \quad (check : en) \leftarrow . \tag{3.23}$$

The knowledge base kb_{check} is as follows:

$$n\_equal \leftarrow not\ equal_{S'_a}. \tag{3.24}$$

$$n\_equal \leftarrow not\ equal_{S'_b}. \tag{3.25}$$

$$\bot \leftarrow not\ n\_equal, en. \tag{3.26}$$

Bridge rule r_{en} ensures that C_{check} fulfills our assumption that a context without input is consistent. Wlog. we assume that C_a and C_b accept some belief set given input H_a and H_b, respectively. Bridge rule r_α adds $equal_{S'_a}$ to C_{check} iff the first instance (H_a, S_a), C_a we reduce from is a 'yes' instance. The same is true for r_γ, $equal_{S'_b}$ and the second instance. Therefore there exists an equilibrium $S = (S_a^{full}, S_b^{full}, \{equal_{S'_a}, equal_{S'_b}, en\})$ in M, i.e., $(\emptyset, \emptyset) \in D_m^\pm(M)$, iff both instances are 'yes' instances. Moreover, if the first instance is a 'yes' instance and the second instance is a 'no' instance, then the system is inconsistent and there is a minimal diagnosis $(\emptyset, \{equal_{S'_b}\}) \in D_m^\pm(M)$. If both instances are 'no' instances, activating $equal_{S'_b}$ is not sufficient for restoring consistency, and a minimal diagnosis for M is then $(\emptyset, \{equal_{S'_a}, equal_{S'_b}\})$. Therefore $(\emptyset, \{equal_{S'_b}\})$ is a minimal diagnosis for M iff (H_a, S_a), C_a is a 'yes' instance and (H_b, S_b), C_b is a 'no' instance of context acceptability checking. Therefore we have established that MCSD$_m$ is hard for $\mathbf{D}(\mathcal{CC}^-(M))$. Note that by nesting contexts $C_{a'}$ and $C_{b'}$ into a new context it is possible, although more complicated, to obtain a reduction with just one context that is hard for $\mathbf{D}(\mathcal{CC}^-(M))$. □

Refuting a candidate (A, B) as an explanation of M can be done by guessing a pair of sets (R_1, R_2) from Definition 6 and checking that $M[R_1 \cup cf(R_2)]$ is inconsistent. Then (A, B) is a yes instance iff all guesses succeed, which leads to complementary complexity of consistency checking for that problem. Hardness for context complexity classes C that are closed under conjunction and projection is established via reducing two contexts of complexity C to an MCS which (a) is consistent if both instances are 'yes' instances, (b) has a minimal diagnosis D if both instances are 'no' instances, and (c) has a nonempty minimal diagnosis which is a subset of D if one is a 'yes' and the other a 'no' instance. For context complexity $\mathbf{P}$ a similar approach is used with two SAT instances.

Proposition 6. *The problem* MCSE*, given MCS M, is*

- $\mathbf{coNP}$*-complete if $\mathcal{CC}(M) = \mathbf{P}$, and*
- $\mathbf{co}$-$\mathcal{CC}(M)$*-complete if $\mathcal{CC}(M)$ is a class with complete problems that is closed under conjunction and projection.*

Note that, as shown in Table 3.1, the second item implies that MCSE is $\mathbf{\Pi^P_i}$-complete if $\mathcal{CC}(M)$ is complete for $\mathbf{\Sigma^P_i}$ with $i \geq 1$.

Proof. (*Membership*) For deciding $(E_1, E_2) \in E^{\pm}(M)$, we guess $R_1, R_2 \subseteq br_M$ and check whether $E_1 \subseteq R_2$ and $R_2 \subseteq br_M \setminus E_2$. If not, we immediately reject, otherwise we decide MCSEQ of $M[R_1 \cup cf(R_2)]$. Then all execution paths reject iff (E_1, E_2) is an explanation. Therefore, if $\mathcal{CC}^+(M)$ is a class with complete problems that is closed under conjunction and projection, the complexity is in $\mathbf{co}$-$\mathcal{CC}^+(M)$. For $\mathcal{CC}^+(M) = \mathbf{P}$ (which is not closed under projection) we obtain that MCSE is in $\mathbf{coNP}$.

($\mathbf{coNP}$-*hardness* for $\mathcal{CC}^-(M) = \mathbf{P}$) We reuse the MCSEQ hardness proof where a 3-SAT instance F was reduced to MCS $M = (C_{gen_U}, C_{eval_F}, C_{check})$. Then satisfiability of F implies consistency, therefore $E^{\pm}(M) = \emptyset$, i.e., no inconsistency explanations exist. Unsatisfiability of F implies inconsistency, and in that case, $(\{r_\alpha\}, \emptyset)$ is an inconsistency explanation of M. Therefore $(\{r_\alpha\}, \emptyset)$ is recognized as inconsistency explanation of M iff F is unsatisfiable. Therefore the problem MCSE in an MCS with $\mathcal{CC}^-(M) = \mathbf{P}$ is hard for $\mathbf{coNP}$.

($\mathbf{co}$-$\mathcal{CC}^-(M)$-*hardness*) We reuse the MCSEQ hardness proof where we reduced an instance $I = (H_a, S_a)$, C_a to a MCS $M_I = (C_{a'}, C_{check})$. If I is a 'yes' instance, then M_I is consistent so no inconsistency explanation exists. If I is a 'no' instance, an inconsistency explanation of M_I is $(\{r_{en}\}, \{r_\gamma\}) \in E^{\pm}(M_I)$. Therefore the problem MCSE in an MCS M with lower context complexity $\mathcal{CC}^-(M)$ is hard for $\mathbf{co}$-$\mathcal{CC}^-(M)$ if $\mathcal{CC}(M)$ is a class with complete problems that is closed under conjunction and projection. □

For complexity results of recognizing minimal explanations we need the following Lemma which limits the number of explanations that need to be checked to verify subset-minimality.

Lemma 3. *An explanation $Q = (Q_1, Q_2)$ is $\subseteq$-minimal iff no pair $(Q_1, Q_2 \setminus \{r\})$ with $r \in Q_2$ is an explanation and no pair $(Q_1 \setminus \{r\}, Q_2)$ with $r \in Q_1$ is an explanation.*

Proof. We write $(A_1, A_2) \subset (B_1, B_2)$ iff $(A_1, A_2) \subseteq (B_1, B_2)$ and $(A_1, A_2) \neq (B_1, B_2)$.

($\Rightarrow$) Assume $Q = (Q_1, Q_2)$ is a minimal explanation. Contrary to the Lemma, assume there exists another explanation Q', such that $Q' = (Q_1, Q_2 \setminus \{r\})$ with $r \in Q_2$ or $Q' = (Q_1 \setminus \{r\}, Q_2)$ with $r \in Q_1$. Then $Q' \subset Q$, therefore Q is not minimal, contradicting the assumption.

($\Leftarrow$) Assume an explanation $Q = (Q_1, Q_2)$, and no pair $(Q_1, Q_2 \setminus \{r\})$ with $r \in Q_2$ or $(Q_1 \setminus \{r\}, Q_2)$ with $r \in Q_1$ is an explanation. Contrary to the Lemma, assume another explanation $P = (P_1, P_2)$ exists with $P \subset Q$. By $P \subset Q$, either a) $P_1 \subset Q_1$ and $P_2 \subseteq Q_2$ or b) $P_1 \subseteq Q_1$ and $P_2 \subset Q_2$. For a) we create $T' = (Q_1 \setminus \{r\}, Q_2)$ for some $r \in Q_1 \setminus P_1$. Then $P \subseteq T' \subset Q$. Due to Corollary 1, T' is an explanation, contradicting the initial assumption. The case b) is similar. □

Hence, we can check subset-minimality of explanations by deciding whether for linearly many subsets of the candidate (A, B), none is an explanation, i.e., whether for each subset, some (R_1, R_2) exists s.t. $M[R_1 \cup cf(R_2)]$ is consistent. As $\mathbf{NP}$ (resp., $\mathbf{\Sigma^P_i}$) is closed under conjunction and projection, this check is in $\mathbf{NP}$ (resp., $\mathbf{\Sigma^P_i}$). In combination with checking whether the candidate is an explanation, this leads to a complexity of $\mathbf{D^P_1}$ (resp., $\mathbf{D^P_i}$). For context complexity $C \in \mathbf{PSPACE}$ (resp., $C \in \mathbf{EXPTIME}$), $\mathbf{D}(C) = C$. The hardness reduction for MCSE_m is very similar to the one for MCSD_m.

Proposition 7. *The problem* MCSE_m*, given MCS* M*, is*

- $\mathbf{D}_1^{\mathbf{P}}$*-complete if* $\mathcal{CC}(M) = \mathbf{P}$*,*
- *complete for* $\mathbf{D}(\mathcal{CC}(M))$ *if* $\mathcal{CC}(M)$ *is a class with complete problems that is closed under conjunction and projection.*

Proof. (*Membership*) We can decide $(E_1, E_2) \in E_m^{\pm}(M)$ using Lemma 3, i.e., we decide (1) whether $(E_1, E_2) \in E^{\pm}(M)$, and (2) whether all of $(E_1, E_2 \setminus \{r \mid r \in E_2\}) \notin E^{\pm}(M)$ and $(E_1 \setminus \{r \mid r \in E_1, E_2\}) \notin E^{\pm}(M)$ are true. Note that the number of $E^{\pm}$-checks in (2) is linear in the size of the instance, hence we obtain the following membership results: if the upper context complexity $\mathcal{CC}^+(M)$ is a class with complete problems that is closed under conjunction and projection, deciding (1) is in $\mathbf{co}$-$\mathcal{CC}^+(M)$ and deciding (2) is in $\mathcal{CC}^+(M)$, therefore MCSE_m is in $\mathcal{D}(\mathcal{CC}^+(M))$. For upper context complexity $\mathcal{CC}^+(M) = \mathbf{P}$ (which is not closed under projection) deciding (1) is in $\mathbf{coNP}$ and deciding (2) is in $\mathbf{NP}$ and therefore MCSE_m is in $\mathbf{D}_1^{\mathbf{P}}$.

($\mathbf{D}_1^{\mathbf{P}}$-*hardness* for $\mathcal{CC}^-(M) = \mathbf{P}$) We use the same topology as for the MCSD_m hardness proof, i.e., the complete topology shown in Figure 3.2a. We reduce two 3-SAT instances F and G on variables $\mathcal{X}$ and $\mathcal{Y}$, respectively, to minimal explanation recognition on MCS $M = (C_{gen_V}, C_{eval_F}, C_{gen_U}, C_{eval_G}, C_{check})$. Again, C_{gen_U} and C_{eval_F} provide $\mathbf{NP}$-hardness for satisfiability of F, while C_{gen_V} and C_{eval_G} provide $\mathbf{coNP}$-hardness for unsatisfiability of G. All contexts except for C_{check} are constructed from F and G exactly as in the MCSD_m hardness proof. Wlog. we assume that F is not valid. The bridge rules in M are as follows:

$$r_{u,i}: \quad (eval_F : x_i) \leftarrow (gen_U : x_i). \qquad \forall i : 1 \leq i \leq |\mathcal{X}| \tag{3.27}$$

$$r_{v,j}: \quad (eval_G : y_j) \leftarrow (gen_V : y_j). \qquad \forall j : 1 \leq j \leq |\mathcal{Y}| \tag{3.28}$$

$$r_\alpha: \quad (check : sat\_or\_nsat_F) \leftarrow (eval_F : sat). \tag{3.29}$$

$$r_\beta: \quad (check : sat\_or\_nsat_F) \leftarrow \mathbf{not}\ (eval_F : sat). \tag{3.30}$$

$$r_\gamma: \quad (check : nsat_G) \leftarrow \mathbf{not}\ (eval_G : sat). \tag{3.31}$$

Context C_{check} has the following knowledge base kb_{check}:

$$\bot \leftarrow sat\_or\_nsat_F, not\ nsat_G. \tag{3.32}$$

If G is satisfiable, M is consistent, so $E_m^{\pm}(M) = \emptyset$. If F and G are unsatisfiable, the belief sat is never accepted at C_{eval_F}; therefore the bridge rule r_β is sufficient for creating inconsistency in M (i.e., $M[\{r_\beta\}] \models \bot$) and forcing r_γ to become applicable is the only way to restore consistency. Therefore if F and G are unsatisfiable, $(\{r_\beta\}, \{r_\gamma\}) \in E^{\pm}(M)$. If F is satisfiable and G is unsatisfiable, the belief sat may or may not be accepted at C_{eval_F}, depending on the input C_{eval_F} gets from C_{gen_U}. Therefore both bridge rules r_α and r_β are required for ensuring inconsistency in M, and they are also sufficient. Again, forcing r_γ to become applicable is the only way to restore consistency. Therefore if F is satisfiable and G is unsatisfiable, then $(\{r_\alpha, r_\beta\}, \{r_\gamma\}) \in E^{\pm}(M)$. Thus $(\{r_\alpha, r_\beta\}, \{r_\gamma\})$ is a minimal inconsistency explanation for M iff F is satisfiable and G is unsatisfiable. Note that if G is satisfiable, no explanations exist, while if F is unsatisfiable, the above explanation exists but is no longer minimal. Therefore recognizing a minimal inconsistency explanation in an MCS with $\mathcal{CC}^-(M) = \mathbf{P}$ is hard for $\mathbf{D}_1^{\mathbf{P}}$.

($\mathbf{D}(\mathcal{CC}^-(M))$-*hardness*) We use the same topology as for the MCSD_m hardness proof, i.e., the complete topology shown in Figure 3.2b. We also use a very similar reduction. The only change is in the checking context C_{check}. We reduce two context complexity check instances (H_a, S_a), C_a with IN_a, OUT_a and (H_b, S_b), C_b with IN_b, OUT_b to an MCS $M = (C_{a'}, C_{b'}, C_{check})$. The bridge rules br_{check} are as follows.

$$r_\alpha: \quad (check : equal_{S'_a}) \leftarrow l_1, \dots, l_j, \dots l_{|OUT_a|}.$$

$$\text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_a \wedge s_j \in S_a \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_a \wedge s_j \notin S_a \end{cases} \tag{3.33}$$

$$r_\beta: \quad (check : make\_inc) \leftarrow l_1, \dots, l_j, \dots l_{|OUT_a|}.$$

$$\text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_a \wedge s_j \in S_a \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_a \wedge s_j \notin S_a \end{cases} \tag{3.34}$$

$$r_\gamma: \quad (check : equal_{S'_b}) \leftarrow l_1, \dots, l_j, \dots l_{|OUT_b|}.$$

$$\text{where } l_j \text{ is } \begin{cases} s_j & \text{if } s_j \in OUT_b \wedge s_j \in S_b \\ \mathbf{not}\ s_j & \text{if } s_j \in OUT_b \wedge s_j \notin S_b \end{cases} \tag{3.35}$$

$$r_{en}: \quad (check : en) \leftarrow. \tag{3.36}$$

Note that r_α and r_β have the same body but different heads, moreover only r_β differs from the MCSD$_m$-reduction. The knowledge base kb_{check} is as follows:

$$n\_equal_a \leftarrow not\ equal_{S'_a}. \tag{3.37}$$

$$n\_equal_a \leftarrow make\_inc. \tag{3.38}$$

$$\bot \leftarrow en, n\_equal_a, not\ equal_{S'_b}. \tag{3.39}$$

The bridge rule r_{en} ensures that C_{check} fulfills our assumption that a context without input is consistent. Wlog. we assume that C_a and C_b accept some belief set given input H_a and H_b, respectively. The bridge rule r_α adds $equal_{S'_a}$ to C_{check} iff the first instance (H_a, S_a), C_a is a 'yes' instance. Under the same condition, r_β adds $make\_inc$. The bridge rule r_γ adds $equal_{S'_b}$ to C_{check} iff the second instance (H_b, S_b), C_b is a 'yes' instance. In that case, M is consistent, i.e., $E^{\pm}_m(M) = \emptyset$, because r_γ becomes applicable and this deactivates constraint (3.39) such that C_{check} can no longer become inconsistent. If both instances are 'no' instances, M is inconsistent and for explaining this inconsistency it is sufficient to have r_{en} present and the heads of the bridge rules r_α and r_γ absent. Therefore, in that case, $(\{r_{en}\}, \{r_\alpha, r_\gamma\})$ is a minimal inconsistency explanation for M. Finally, if (H_a, S_a),C_a is a 'yes' instance and (H_b, S_b),C_b is a 'no' instance, M is inconsistent and for this inconsistency it is sufficient to have r_{en} and r_β present and heads of bridge rules r_α and r_γ absent, so $(\{r_{en}, r_\beta\}, \{r_\alpha, r_\gamma\}) \in E^{\pm}_m(M)$. Therefore $(\{r_{en}, r_\beta\}, \{r_\alpha, r_\gamma\}) \in E^{\pm}_m(M)$ iff the first instance is a 'yes' instance and the second instance is a 'no' instance. Note that if the second instance is a 'yes' instance, no explanations exist, while if the first instance is a 'no' instance, the above explanation exists but is no longer minimal. Therefore we have established that MCSE$_m$ is hard for $\mathbf{D}(\mathcal{CC}^-(M))$ if $\mathcal{CC}(M)$ is a class with complete problems that is closed under conjunction and projection. □

3.5 Approximating Inconsistency Analyses

So far, we assumed an omniscient view of the system, where the user has full information about all contexts including their knowledge bases and semantics. However, in many real world scenarios full information is not available [BO07], and some contexts are black boxes with internal knowledge bases or semantics that are not disclosed due to intellectual property or privacy issues (e.g., banks will not disclose their full databases to credit card companies). Partial behavior of such contexts may be known, however querying the contexts might be limited, e.g., by contracts or costs. In such scenarios, inconsistencies can only be explained given the knowledge of the system one has, and since this is partial, the explanations obtained just approximate the actual situation, i.e., those explanations one would obtain if one would have full insight.

In other words, this calls for explaining inconsistency in an MCS with *partial knowledge* about contexts, which raises the following technical challenges:

- how to represent partial knowledge about the system, and
- how to obtain reasonable *approximations* for diagnoses and inconsistency explanations in the actual system (under full knowledge), ideally in an efficient way.

The first issue depends on the nature of this knowledge, and a range of possibilities exists. The second issue requires an assessment method to determine such approximations. We tackle both issues and make the following contributions (published in [SEF10, EFS10, EFS11]).

- We develop a representation of partially known contexts, which is based on context abstraction with Boolean functions. Partially defined Boolean functions [Val84, CHI88] are then used to capture partially known behavior of a context.
- We exploit these representations to define *over-* and *underapproximations* of diagnoses and explanations for inconsistency in the presence of partially known contexts. The approximations target either the whole set of diagnoses, or one diagnosis at a time; analogously for explanations.
- For scenarios where partially known contexts can be asked a limited number of queries, we consider query selection strategies.
- Finally, we discuss computational complexity of recognizing approximate explanations. In contrast to semantic approximations for efficient evaluation [SC95], our approximations handle incompleteness, which usually increases complexity. Fortunately, our approach does not incur higher computational cost than in the case of full information.

Our results extend methods for inconsistency handling in MCSs to more realistic settings. This allows us to identify reasons for inconsistency even if it is impossible to obtain full system knowledge, and without increasing computational cost.

The following running example involves an access control system which uses a separate credit card validity checking system for granting permissions to certain users. This scenario involves policies and trust information which are often non-public and distributed [BO07]. It demonstrates the necessity of reasoning under incomplete information.

Example 28 (Credit Card Example)**.** *Consider a MCS which consists of a permission database C_{perm}, a credit card clearing context C_{cc}, and the following bridge rules schemas:*

$$\begin{array}{lll}
r_a: & (perm : person(\text{PERSON})) \leftarrow \top. & \\
r_b: & (cc : card(\text{CREDITCARD})) \leftarrow & (perm : person(\text{PERSON})), \\
 & & \mathbf{not}\ (perm : grant(\text{PERSON})), \\
 & & (perm : ccard(\text{PERSON}, \text{CREDITCARD})). \\
r_c: & (perm : ccValid(\text{CREDITCARD})) \leftarrow & (cc : valid(\text{CREDITCARD})).
\end{array}$$

Here r_a defines a set of persons which is relevant for permission evaluation in C_{perm}; r_b specifies that, if some person is not granted access, credit cards of that person have to be checked; and r_c translates validation results to C_{perm}.

We next describe the context internals: C_{perm} is a Datalog program over signature $\Sigma_{perm} = \{group, ccard, igrant, grant, ccValid\}$ and hence uses $L^{asp}_{\Sigma_{perm}}$ as introduced in Example 3. The knowledge base kb_{perm} is as follows:

$$\begin{array}{rl}
kb_{perm} = \{ & group(nina, vip);\ ccard(nina, cnr1);\ ccard(moe, cnr2); \\
 & igrant(\text{PERSON}) \leftarrow person(\text{PERSON}), group(\text{PERSON}, vip); \\
 & grant(\text{PERSON}) \leftarrow igrant(\text{PERSON}); \\
 & grant(\text{PERSON}) \leftarrow ccValid(\text{CREDITCARD}), ccard(\text{PERSON}, \text{CREDITCARD})\}.
\end{array}$$

Context C_{cc} is a credit card clearing facility, which typically is neither fully disclosed to the operator, nor can it be queried without significant cost. Hence, one obviously has to deal with partial knowledge: C_{cc} accepts $valid(X)$ iff card X is valid and validation is requested by $card(X)$. Without full insight or a history of past requests, we only know the behavior of C_{cc} when no bridge rules are applicable: $\mathbf{ACC}_{cc}(kb_{cc} \cup \emptyset) = \{\emptyset\}$.

The MCS formalism is defined on ground bridge rules. We next give the set of relevant groundings of bridge rules.

$$\begin{array}{lrl}
r_1\colon & (perm : person(nina)) \leftarrow & \top. \\
r_2\colon & (perm : person(moe)) \leftarrow & \top. \\
r_3\colon & (cc : card(cnr1)) \leftarrow & (perm : person(nina)), \\
 & & \mathbf{not}\ (perm : grant(nina)), \\
 & & (perm : ccard(nina, cnr1)). \\
r_4\colon & (cc : card(cnr2)) \leftarrow & (perm : person(moe)), \\
 & & \mathbf{not}\ (perm : grant(moe)), \\
 & & (perm : ccard(moe, cnr2)). \\
r_5\colon & (perm : ccValid(cnr1)) \leftarrow & (cc : valid(cnr1)). \\
r_6\colon & (perm : ccValid(cnr2)) \leftarrow & (cc : valid(cnr2)).
\end{array}$$

Assume that MCS M_1^{cc} contains the above contexts and the following bridge rules: r_1, r_3, r_4, r_5, r_6, i.e., we want to check permissions of $nina$. As $nina$ is in the vip group, there is no need to verify a credit card, and M_1 has the following equilibrium (we omit facts in kb_{perm}):

$$(\{person(nina), igrant(nina), grant(nina)\}, \emptyset).$$

For an example of inconsistency, consider M_2^{cc} which contains both contexts and bridge rules r_2, r_3, r_4, r_5, r_6, i.e., we want to check permissions of moe. Furthermore assume the following full knowledge about C_{cc}: all credit cards are valid.

Observe that M_2^{cc} is inconsistent: moe is not in the vip group, card verification is required by r_4, and C_{cc} accepts $valid(cnr2)$. This allows C_{perm} to derive $grant(moe)$, which blocks applicability of r_4. This inconsistency is due to an unstable cycle in M_2^{cc}. Under full knowledge, the set of $\subseteq$-minimal diagnoses of M_2^{cc} is as follows:

$$D_m^{\pm}(M_2^{cc}) = \{(\{r_2\}, \emptyset), (\{r_4\}, \emptyset), (\emptyset, \{r_4\}), (\{r_6\}, \emptyset), (\emptyset, \{r_6\})\}$$

The first points out that the reason for inconsistency is our request. The second (resp., third) diagnosis suggests to never (resp., always) check the credit card of moe. Finally the fourth (resp., fifth) diagnosis suggests to never (resp., always) use the output of the credit card check. As we obviously want to use the system, we will not remove the request. Furthermore, as we need to use the output of the credit card check, the interesting diagnoses are those that point out r_4 as the reason for inconsistency. Indeed, schema r_b should intuitively contain $igrant$(PERSON) in its body instead of $grant$(PERSON). This would make the system behave as expected and grant access, because moe has a valid credit card. □

In the remainder of this section, we develop an approach which is able to point out a problem in r_4 without requiring complete knowledge.

3.5.1 Information Hiding

We first introduce an abstraction of contexts which allows us to calculate diagnoses and explanations. We then generalize this abstraction to represent partial knowledge, i.e., contexts C_i where either kb_i, or $\mathbf{ACC}_i$ is only partially known.

Context Abstraction

We abstract from a context's knowledge base kb_i and logic L_i by a Boolean function over the context's inputs IN_i and output beliefs OUT_i (see Definition 9).

Recall that a Boolean function (BF) is a map $f : \mathbb{B}^k \to \mathbb{B}$ where $k \in \mathbb{N}$ and $\mathbb{B} = \{0, 1\}$. Such a BF can also be characterized either by its true points $T(f) = \{\vec{x} \mid f(\vec{x}) = 1\}$, or by its false points $F(f) = \{\vec{x} \mid f(\vec{x}) = 0\}$.

Given a set $X \subseteq U = \{u_1, \dots, u_k\}$, we denote by $\vec{x}_U$ the characteristic vector of X wrt. some universe U (i.e. $\vec{x}_U = (b_1, \dots, b_k)$, where $b_i = 1$ if $u_i \in X$, 0 otherwise). We write $\vec{x}$ if U is implicitly clear. Using this notation, we characterize sets of bridge rule heads $I \subseteq IN_i$ and sets of output beliefs $O \subseteq OUT_i$ by vectors $\vec{\imath}_{IN_i}$ and $\vec{o}_{OUT_i}$, respectively. For example, given $O = \{a, c\}$, and $OUT_i = \{a, b, c\}$, we have $\vec{o} = (1, 0, 1)$.

Example 29 (ctd)**.** *We use the following (ordered) sets for inputs and output beliefs of C_{cc}: $IN_{cc} = \{card(cnr1), card(cnr2)\}$, and $OUT_{cc} = \{valid(cnr1), valid(cnr2)\}$.* □

Definition 14. *The unique BF $f^{C_i} : \mathbb{B}^{|IN_i|+|OUT_i|} \to \mathbb{B}$ corresponds to the semantics of context C_i in an MCS M as follows:*

$$\text{for all } I \subseteq IN_i, O \subseteq OUT_i \text{ it holds that } f^{C_i}(\vec{\imath}, \vec{o}) = 1 \text{ iff } O \in \mathbf{ACC}_i(kb_i \cup I)\big|_{OUT_i}.$$

Example 30 (ctd)**.** *With full knowledge (see Example 28), C_{cc} has as corresponding BF the function $f^{C_{cc}}(X, Y, X, Y) = 1$ for all $X, Y \in \mathbb{B}$, 0 otherwise.* □

If a context accepts a belief set O' for a given input I, we obtain the true point $(\vec{\imath}, \vec{o})$ of f with $O = O' \cap OUT_i$. Similarly, each non-accepted belief set yields a false point of f. Due to projection, different accepted belief sets can characterize the same true point.

In the following, we show that this context abstraction provides sufficient information to calculate *output-projected equilibria* of the given MCS. Due to Lemma 2 we know that checking consistency is possible using output-projected equilibria only. Hence, the abstraction also allows for checking consistency and calculating diagnoses and explanations.

Intuitively, the representation of a context by a BF provides an input/output oracle, projected to output beliefs. Therefore, the BF is sufficient for deciding whether an output-projected belief state is an output-projected equilibrium as well.

To provide this result, and towards a representation of an MCS with partial knowledge of certain contexts, we next provide a notation for an MCS M where the knowledge of a context C_i is given by BF f, rather than kb_i.

Definition 15. *For every MCS $M = (C_1, \dots, C_n)$, BF f and index $1 \leq i \leq n$, we denote by $M[i/f]$ the MCS M where context C_i is replaced by some context $C(f)$ which contains the set br_i of bridge rules, a logic with a signature that contains $IN_i \cup OUT_i$, and $kb_{C(f)}$ and $\mathbf{ACC}_{C(f)}$ are such that $f^{C(f)} = f^{C_i}$.*

For instance, $C(f)$ could be based on classical logic or logic programming, with $kb_{C(f)}$ over $IN \cup OUT$ as atoms encoding f by clauses (rules) that realize the correspondence.

We now show that a BF representation of a context is sufficient for calculating output-projected equilibria. We denote by $M[i_1, \dots, i_k/f_1, \dots, f_k]$ the substitution of pairwise distinct contexts $C_{i_1}, \dots, C_{i_k}$ by $C(f_1), \dots, C(f_k)$, respectively.

Theorem 4. *Let $M = (C_1, \dots, C_n)$ be an MCS, and let $f_{i_1}, \dots, f_{i_k}$ be BFs that correspond to $C_{i_1}, \dots, C_{i_k}$. Then, $\text{EQ}'(M) = \text{EQ}'(M[i_1, \dots, i_k/f_{i_1}, \dots, f_{i_k}])$.*

Proof. Let $M^\star = M[i_1, \dots, i_k/f_{i_1}, \dots, f_{i_k}]$. By construction $M^\star = (C_1^\star, \dots, C_n^\star)$, such that $C_i = C_i^\star$ for non-substituted contexts, and $br_i = br_i^\star$ for $1 \leq i \leq n$. The latter also implies $IN_i = IN_i^\star$ and $OUT_i = OUT_i^\star$, for $1 \leq i \leq n$. By Definition 10, $\text{EQ}'(M) =$

$\text{EQ}'(M^\star)$ if the following condition (i) holds: for each pair $(C_i, C_i^\star)$ of contexts with $C_i = ((\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i), kb_i, br_i)$ and $C_i^\star = ((\mathbf{KB}_i^\star, \mathbf{BS}_i^\star, \mathbf{ACC}_i^\star), kb_i^\star, br_i^\star)$, and for all $H \subseteq IN_i$: $\mathbf{ACC}_i(kb_i \cup H)|_{OUT_i} = \mathbf{ACC}_i^\star(kb_i^\star \cup H)|_{OUT_i}$.

This trivially holds for non-substituted contexts. So let $C_i^\star = C(f_i)$ be an arbitrary substituted context. By construction it holds that $f^{C_i^\star} = f_i$. Furthermore, each f_i corresponds to its respective C_i, so $f_i = f^{C_i}$. Thus, $f^{C_i^\star} = f^{C_i}$. Since f^{C_i} is defined in a 1–1 relationship to $\mathbf{ACC}_i(kb_i \cup H)|_{OUT_i}$ for all $H \subseteq IN_i$ (see Definition 14), we obtain that (i) holds for all substituted contexts. □

Partially Known Contexts. As the BF representation concerns only output beliefs, it already hides part of the context, while we are still able to analyze inconsistency. Now we generalize the BF representation to *partially defined Boolean functions* (pdBFs) (cf. [Val84, CHI88]), to represent contexts where we have only partial knowledge about their output-projected behavior.

In applications, existence of such partial knowledge is realistic: for some bridge rule firings one may know an accepted belief set of a context, but not whether other accepted belief sets exist. Similarly one may know that a context is inconsistent for some input combination, but not whether it accepts some belief set for other input combinations.

Formally, a pdBF pf is a function from $\mathbb{B}^k$ to $\mathbb{B} \cup \{\star\}$, where $\star$ stands for undefined (cf. [Val84]). It is equivalently characterized by two sets [CHI88]: its true points $T(pf) = \{\vec{x} \mid pf(\vec{x}) = 1\}$ and its false points $F(pf) = \{\vec{x} \mid pf(\vec{x}) = 0\}$. We denote by $U(pf) = \{\vec{x} \mid pf(\vec{x}) = \star\}$ the *unknown points* of pf. A BF f is an *extension* of a pdBF pf, formally $pf \leq f$, iff $T(pf) \subseteq T(f)$ and $F(pf) \subseteq F(f)$.

We connect partial knowledge of context semantics and pdBFs as follows.

Definition 16. *A pdBF $pf : \mathbb{B}^k \to \mathbb{B} \cup \{\star\}$ is compatible with a context C_i in an MCS M iff $pf \leq f^{C_i}$ (where f^{C_i} is as in Definition 14).*

Therefore, if a pdBF is compatible with a context, one extension of this pdBF is exactly f^{C_i}, which corresponds to the context's exact semantics.

Example 31 (ctd). *Partial knowledge as given in Example 28 can be expressed by the pdBF pf_{cc} with $T(pf_{cc}) = \{(0,0,0,0)\}$ and $F(pf_{cc}) = \{(0,0,A,B) \mid A, B \in \mathbb{B}, (A,B) \neq (0,0)\}$. (See Example 29 for the variable ordering.)* □

In the following, a *partially known MCS* (M, i, pf) consists of an MCS M, where the context C_i is partially known, given by pdBF pf which is compatible with C_i.

3.5.2 Approximations for Diagnoses

In this section, we develop a method for calculating under- and overapproximations of diagnoses and explanations, using the pdBF representation for a partially known context C_i. For simplicity, we only consider the case that a single context in the system is partially known (the generalization is straightforward).

Each diagnosis is defined in terms of consistency, which is witnessed by an output-projected equilibrium. Such an equilibrium requires a certain set of output beliefs O to be accepted by the context C_i, in the presence of certain bridge rule heads I. This means that f_{C_i} has true point $(\vec{\imath}, \vec{o})$. For existence of an equilibrium where C_i gets I as input and accepts O, no more information is required from f_{C_i} than this single true point. We thus can approximate the set of diagnoses of M as follows:

- Completing pf with false points, we obtain the extension $\underline{pf}$ with $T(\underline{pf}) = T(pf)$. The set of diagnoses witnessed by $T(pf)$ contains a *subset* of the diagnoses which actually occur in M, therefore we obtain an *under*approximation.

- Completing pf with true points, we obtain the extension $\overline{pf}$ as the extension of pf with the largest set of true points. The set of diagnoses witnessed by $\overline{pf}$ contains a *superset* of the diagnoses which actually occur in M, providing an *over*approximation. Formally,

Theorem 5. *Given a partially known MCS (M, i, pf), the following holds:*

$$D^{\pm}(M[i/\underline{pf}]) \subseteq D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{pf}]).$$

Proof. We prove the direction $D^{\pm}(M[i/\underline{pf}]) \subseteq D^{\pm}(M)$ as follows: each diagnosis $(D_1, D_2) \in D^{\pm}(M[i/\underline{pf}])$ induces a consistent MCS $M^{\star}$ by removing bridge rules D_1 and making bridge rules D_2 unconditional. Since (D_1, D_2) is a diagnosis, $M^{\star}$ has at least one witnessing output-projected equilibrium S'. At context C_i, S' contains a certain set of output beliefs $O_i = S'_i$, furthermore the set of active bridge rule heads at C_i is $H_i = app(br_i, S')$.

Because S' is an output-projected equilibrium, we have that $O_i \in \mathbf{ACC}_i(kb_i \cup H_i)|_{OUT_i}$, so $\underline{pf}$ has a true point at $(\vec{\imath}, \vec{o})$. Since pf is compatible with C_i, some extension of pf is equal to f^{C_i}. Moreover, every true point of $\underline{pf}$ is a true point of pf, therefore every true point of $\underline{pf}$ is a true point of f^{C_i}. Consequently, C_i accepts some S_i for input H_i where $O_i = S_i \cap OUT_i$, which proves that (D_1, D_2) is a diagnosis of M.

$D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{pf}])$ is proved similarly: no true point of f^{C_i} is a false point of pf, and thus neither of $\overline{pf}$. Consequently, all true points of f^{C_i} are true points of $\overline{pf}$. Hence, all accepted input–output "behaviors" of context C_i are accepted in the overapproximation, and therefore each diagnosis in $D^{\pm}(M)$ is in $D^{\pm}(M[i/\overline{pf}])$ as well. □

Example 32 (ctd)**.** *The extensions $\overline{pf}_{cc}$ and $\underline{pf}_{cc}$ are as follows:*

$$\begin{array}{rclcrcl} T(\overline{pf}_{cc}) & = & \mathbb{B}^4 \setminus F(pf_{cc}), & & F(\overline{pf}_{cc}) & = & F(pf_{cc}), \\ T(\underline{pf}_{cc}) & = & T(pf_{cc}), \textit{ and} & & F(\underline{pf}_{cc}) & = & \mathbb{B}^4 \setminus T(pf_{cc}). \end{array}$$

The underapproximation $D^{\pm}(M_2^{cc}[cc/\underline{pf}_{cc}])$ yields several diagnoses, for instance

$$D_\alpha = (\{r_2\}, \emptyset),\ D_\beta = (\{r_4\}, \emptyset), \textit{ and } D_\gamma = (\emptyset, \{r_6\}).$$

The overapproximation $D^{\pm}(M_2^{cc}[cc/\overline{pf}_{cc}])$ contains the empty diagnosis, i.e., $D_\delta = (\emptyset, \emptyset)$, because $M_2^{cc}[cc/\overline{pf}_{cc}]$ is consistent with the following two equilibria:

$$(\{person(moe)\}, \emptyset), \textit{ and } (\{person(moe)\}, \{valid(cnr1)\}).$$

□

Subset-minimality. If we approximate ⊆-minimal diagnoses, the situation is different. Obtaining additional diagnoses (due to overapproximation) may cause an approximated diagnosis to be subset-minimal while that diagnosis is not necessarily a ⊆-minimal diagnosis under full knowledge. However, at least one minimal diagnosis under full knowledge is a superset of every approximated diagnosis. Vice versa, missing certain diagnoses (due to underapproximation) can yield an approximated ⊆-minimal diagnosis which is a superset of (at least one) minimal diagnosis.

In any case, if a diagnosis is ⊆-minimal under both over- and underapproximation, then it is also a minimal diagnosis under full knowledge.

Theorem 6. *Given a partially known MCS (M, i, pf), the following hold:*

$$\textit{for all } D \in D^{\pm}_m(M[i/\underline{pf}]) \textit{ there exists } D' \in D^{\pm}_m(M) \textit{ such that } D' \subseteq D \tag{3.40}$$

$$\textit{for all } D \in D^{\pm}_m(M) \textit{ there exists } D' \in D^{\pm}_m(M[i/\overline{pf}]) \textit{ such that } D' \subseteq D \tag{3.41}$$

$$D^{\pm}_m(M[i/\underline{pf}]) \cap D^{\pm}_m(M[i/\overline{pf}]) \subseteq D^{\pm}_m(M) \tag{3.42}$$

Proof. (3.40) For every diagnosis $D \in D^{\pm}_{m}(M[i/\underline{pf}])$ by definition of $D^{\pm}_{m}$ we know that $D \in D^{\pm}(M[i/\underline{pf}])$. From Theorem 5 we infer that $D \in D^{\pm}(M)$. If D is $\subseteq$-minimal in $D^{\pm}(M)$, then (3.40) follows for $D' = D$, otherwise there exists a $D' \in D^{\pm}_{m}(M)$, such that $D' \subseteq D$, which also implies (3.40).

(3.41) The proof is similar, again using Theorem 5.

(3.42) $D \in D^{\pm}_{m}(M[i/\underline{pf}])$ implies $D \in D^{\pm}(M)$. From $D \in D^{\pm}_{m}(M[i/\overline{pf}])$, we infer that there is no $D' \subset D$ such that $D' \in D^{\pm}(M[i/\overline{pf}])$. Since by Theorem 5, $D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{pf}])$, it follows that there is no $D' \subset D$ such that $D' \in D^{\pm}(M)$. As $D \in D^{\pm}(M)$, this proves $D \in D^{\pm}_{m}(M)$. □

Example 33 (ctd)**.** *Note that the diagnoses in Example 32 are in fact the $\subseteq$-minimal diagnoses of the under- and overapproximation, and they are actual $\subseteq$-minimal diagnoses. Under complete knowledge, additional $\subseteq$-diagnoses exist which are not obtained by underapproximation. Overapproximation, on the other hand, yields consistency and therefore an empty $\subseteq$-minimal diagnosis D_{δ}. In Section 3.5.4 we develop a strategy for improving this approximation if limited querying of the context is possible.* □

We can use the overapproximation to reason about the necessity of bridge rules in actual diagnoses: a bridge rule is necessary, if it is present in all diagnoses.[1]

Definition 17. *For a set of diagnoses $\mathcal{D}$, the set of* necessary bridge rules *is $nec(\mathcal{D}) = \{r \mid \forall (D_1, D_2) \in \mathcal{D} : r \in D_1 \cup D_2\}$.*

Proposition 8. *Given a partially known MCS (M, i, pf), the set of necessary bridge rules for the overapproximation is necessary in the actual set of diagnoses. This is true for both arbitrary and $\subseteq$-minimal diagnoses, i.e.,*

$$nec(D^{\pm}(M[i/\overline{pf}])) \subseteq nec(D^{\pm}(M)), \textit{ and } nec(D^{\pm}_{m}(M[i/\overline{pf}])) \subseteq nec(D^{\pm}_{m}(M)).$$

Proof. We first prove $nec(D^{\pm}(M[i/\overline{pf}])) \subseteq nec(D^{\pm}(M))$: by Theorem 5, we have that $D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{pf}])$. Thus, if a bridge rule is contained in all diagnoses of the latter set, it must also be contained in all diagnoses of the former.

Next, we prove $nec(D^{\pm}_{m}(M[i/\overline{pf}])) \subseteq nec(D^{\pm}_{m}(M))$: towards a contradiction, assume $r \in nec(D^{\pm}_{m}(M[i/\overline{pf}]))$ and $r \notin nec(D^{\pm}_{m}(M))$. Then, there exists some $D \in D^{\pm}_{m}(M)$, $D = (D_1, D_2)$, such that $r \notin D_1 \cup D_2$. By Theorem 6 (3.41), we have that there exists $D' \in D^{\pm}_{m}(M[i/\overline{pf}])$, $D' = (D'_1, D'_2)$, such that $D' \subseteq D$. Consequently, $r \notin D'_1 \cup D'_2$, a contradiction to $r \in nec(D^{\pm}_{m}(M[i/\overline{pf}]))$. □

While simple, this property is useful in practice: in a repair of an MCS according to a diagnosis, necessary bridge rules need to be fixed in any case.

3.5.3 Approximations for Inconsistency Explanations

So far we have only described approximations for *diagnoses*.

Example 34. *In our examples, with complete knowledge there is one subset-minimal inconsistency explanation:*

$$E^{\pm}_{m}(M_2^{cc}) = \big\{(\{r_2, r_4, r_6\}, \{r_4, r_6\})\big\}.$$

□

[1]Note that we do not consider the dual notion of relevance, as it is trivial in our definition of diagnosis: all bridge rules are relevant in any $D^{\pm}(M)$.

As we have seen in Theorem 1, we can characterize the set of explanations from the set of diagnoses. Intuitively, explanations are defined in terms of non-existing equilibria, and witnessing equilibria of diagnoses are counterexamples for the existence of certain explanations.

Using this characterization of explanations, we can infer the following: a subset of the actual set of diagnoses characterizes a superset of the actual set of explanations. This is true since a subset of diagnoses will rule out a subset of explanations, allowing more candidates to become explanations. Conversely, a superset of diagnoses characterizes a subset of the explanations. From this insight and from Theorem 5, we obtain the following:

Theorem 7. *Given a partially known MCS (M, i, pf), the following hold:*

$$E^{\pm}(M[i/\overline{pf}]) \subseteq E^{\pm}(M) \subseteq E^{\pm}(M[i/\underline{pf}]) \tag{3.43}$$

$$\textit{for all } E \in E^{\pm}_m(M[i/\overline{pf}]) \textit{ there exists } E' \in E^{\pm}_m(M) \textit{ such that } E' \subseteq E \tag{3.44}$$

$$\textit{for all } E \in E^{\pm}_m(M) \textit{ there exists } E' \in E^{\pm}_m(M[i/\underline{pf}]) \textit{ such that } E' \subseteq E \tag{3.45}$$

$$E^{\pm}_m(M[i/\overline{pf}]) \cap E^{\pm}_m(M[i/\underline{pf}]) \subseteq E^{\pm}_m(M) \tag{3.46}$$

Proof. By Theorem 1 we have that $E^{\pm}(M) = HS_M(D^{\pm}(M))$ and that $E^{\pm}(M[i/\overline{pf}]) = HS_M(D^{\pm}(M[i/\overline{pf}]))$. By Theorem 5 we have that $D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{pf}])$, and therefore the overapproximation imposes a superset of constraints on the members of the hitting set $HS_M(D^{\pm}(M))$ and therefore imposes a superset of constraints on $E^{\pm}(M)$. Hence a subset of pairs $(E_1, E_2) \in HS_M(D^{\pm}(M))$ are elements of $HS_M(D^{\pm}(M[i/\overline{pf}]))$, therefore we have that $E^{\pm}(M[i/\overline{pf}]) \subseteq E^{\pm}(M)$. By Theorem 5 we also have that $D^{\pm}(M[i/\underline{pf}]) \subseteq D^{\pm}(M)$, hence the underapproximation imposes a subset of constraints on the members of the hitting set $E^{\pm}(M) = HS_M(D^{\pm}(M))$ and by similar argumentation as above we obtain that $E^{\pm}(M) \subseteq E^{\pm}(M[i/\underline{pf}])$.

The results (3.44), (3.45), and (3.46) directly follow from (3.43) and can be proved analogously to the proofs of (3.40), (3.41), and (3.42) in Theorem 6. □

Therefore, the extensions $\overline{pf}$ and $\underline{pf}$ allow to underapproximate and overapproximate diagnoses as well as inconsistency explanations.

Example 35 (ctd)**.** *From $\underline{pf}_{cc}$ as in Example 32, we obtain one $\subseteq$-minimal explanation $E_\mu = (\{r_2, r_4\}, \{r_6\})$, which is a subset of the actual minimal explanation in Example 34.* □

3.5.4 Limited Querying

Up to now we used existing partial knowledge to approximate diagnoses, assuming that more information is simply not available. However, in practical scenarios like our running example, one can imagine that a (small) limited number of queries to a partially known context can be made. Therefore we next aim at identifying queries to contexts, such that incorporating their answers into the pdBF will yield the best guarantee of improvement in approximation accuracy.

Given a partially known MCS (M, i, pf), let $D^{\pm}_{\Delta}(M, i, pf) = D^{\pm}(M[i/\overline{pf}]) \setminus D^{\pm}(M[i/\underline{pf}])$ (in short: $D^{\pm}_{\Delta}(pf)$ or $D^{\pm}_{\Delta}$) be the set of *potential diagnoses*, which are possible from the overapproximation but unconfirmed by the underapproximation. A large set of potential diagnoses provides less information than a smaller set. Hence, we aim at identifying unknown points of pf which remove from $D^{\pm}_{\Delta}$ as many potential diagnoses as possible. To this end we introduce the concept of a *witness* as an unknown point and a potential diagnosis that is supported by this point if it is a true point.

Definition 18. *Given a partially known MCS (M, i, pf), a* witness *is a pair $(\vec{x}, D)$ s.t. $\vec{x} \in U(pf)$ and $D \in D^{\pm}(M[i/f_{\vec{x}}]) \cap D^{\pm}_{\Delta}$, where $f_{\vec{x}}$ is the BF with the single true point $T(f_{\vec{x}}) = \{\vec{x}\}$. We denote by $W_{(M,i,pf)}$ the set of all witnesses wrt. (M, i, pf). If clear from the context, we omit the subscript (M, i, pf).*

Based on W we define the set $wnd(\vec{X}) = \{D \mid (\vec{X}, D) \in W\}$ of potential diagnoses witnessed by unknown point $\vec{X}$, and the set $ewnd(\vec{X}) = wnd(\vec{X}) \setminus \{D \mid (\vec{X}', D) \in W,\ \vec{X}' \neq \vec{X}\}$ of potential diagnoses exclusively witnessed by $\vec{X}$. These sets are used to investigate how much the set of potential diagnoses is reduced when adding information about the value of an unknown point $\vec{X}$ to pf.

Lemma 4. *Given a partially known MCS (M, i, pf) and $\vec{X} \in U(pf)$, let $pf_{\vec{X}:0}$ (resp., $pf_{\vec{X}:1}$) be the pdBF that results from pf by making $\vec{X}$ a false (resp., true) point. Then $D^{\pm}_{\triangle}(pf_{\vec{X}:1}) = D^{\pm}_{\triangle}(pf) \setminus wnd(\vec{X})$, and $D^{\pm}_{\triangle}(pf_{\vec{X}:0}) = D^{\pm}_{\triangle}(pf) \setminus ewnd(\vec{X})$.*

Proof of Lemma 4. We first prove $D^{\pm}_{\triangle}(pf_{\vec{X}:1}) = D^{\pm}_{\triangle}(pf) \setminus wnd(\vec{X})$. Intuitively, diagnoses in $wnd(\vec{X})$ leave the set of potential diagnoses and enter the underapproximation.

Changing $\vec{X}$ from an unknown point into a true point has no effect on the result of the overapproximation: in $D^{\pm}(M[i/\overline{pf}])$ all unknown points are transformed into true points, therefore $D^{\pm}(M[i/\overline{pf}_{\vec{X}:1}]) = D^{\pm}(M[i/\overline{pf}])$. Each potential diagnosis requires some unknown point of pf to be a true point, therefore from the definition of wnd we get that (a) $wnd(\vec{X})$ is the subset of potential diagnoses, where each diagnosis is witnessed by an equilibrium with context C_i accepting the input-output state $\vec{X}$, and (b) $wnd(\vec{X})$ is not contained in the underapproximation but is contained in the overapproximation. Adding a known true point $\vec{X}$ creates a new known accepted input-output state on the partially known context. From (a) follows that $wnd(\vec{X}) \subseteq D^{\pm}(M[i/\underline{pf}_{\vec{X}:1}]$. From (b) we get that $D^{\pm}_{\triangle}(pf_{\vec{X}:1}) = D^{\pm}_{\triangle}(pf) \setminus wnd(\vec{X})$.

We now prove $D^{\pm}_{\triangle}(pf_{\vec{X}:0}) = D^{\pm}_{\triangle}(pf) \setminus ewnd(\vec{X})$. Intuitively, diagnoses in $ewnd(\vec{X})$ leave the set of potential diagnoses and also leave the overapproximation.

Changing $\vec{X}$ from an unknown point into a false point has no effect on the result of the underapproximation: in $D^{\pm}(M[i/\underline{pf}])$ all unknown points are transformed into false points, and as a consequence, $D^{\pm}(M[i/\underline{pf}_{\vec{X}:0}]) = D^{\pm}(M[i/\underline{pf}])$. From the definition of $ewnd$, we get that (a) $ewnd(\vec{X})$ is the subset of potential diagnoses, where each diagnosis is witnessed by an equilibrium with context C_i accepting the input-output state $\vec{X}$, that requires $\vec{X}$ to be a true point, and (b) no other unknown point $\vec{X}$ induces a witness for any diagnosis in $ewnd(\vec{X})$, and (c) $ewnd(\vec{X})$ is not contained in the underapproximation and is contained in the overapproximation. Adding a known false point $\vec{X}$ removes a potentially accepted input-output state on the partially known context. From (a) and (b) it follows that $ewnd(\vec{X}) \cap D^{\pm}(M[i/\overline{pf}_{\vec{X}:0}]) = \emptyset$. From (c) we get, that $D^{\pm}_{\triangle}(pf_{\vec{X}:0}) = D^{\pm}_{\triangle}(pf) \setminus ewnd(\vec{X})$. □

Note that $ewnd(\vec{X}) \subseteq wnd(\vec{X}) \subseteq D^{\pm}_{\triangle}$. If $\vec{X}$ is a true point, $|wnd(\vec{X})|$ many potential diagnoses become part of the underapproximation; otherwise $|ewnd(\vec{X})|$ many potential diagnoses are no longer part of the overapproximation. Knowing the value of $\vec{X}$ therefore guarantees a reduction of $D^{\pm}_{\triangle}$ by $|ewnd(\vec{X})|$ diagnoses.

Proposition 9. *Given a partially known MCS (M, i, pf), for all $\vec{X} \in U(pf)$ such that the cardinality of $ewnd(\vec{X})$ is maximal, the following holds:*

$$\max_{u \in \mathbb{B}} \left| D^{\pm}_{\triangle}\left(pf_{\vec{X}:u}\right) \right| \leq \min_{\vec{Y} \in U(pf)} \max_{v \in \mathbb{B}} \left| D^{\pm}_{\triangle}\left(pf_{\vec{Y}:v}\right) \right|. \tag{3.47}$$

Proof of Proposition 9. Given (M, i, pf) and $\vec{X}$ such that $|ewnd(\vec{X})|$ is maximal among $\vec{X} \in U(pf)$, (3.47) expresses the following: regardless of whether we obtain that $\vec{X}$ is a true or a false point of context C_i, we have a guaranteed reduction of the set of potential diagnoses, and no other $\vec{X}' \in U(pf)$ can guarantee a greater reduction. The reason for this guaranteed reduction is that getting information about an additional point either adds at least one diagnosis to the underapproximation $D^{\pm}(M[i/\underline{pf}])$ or it removes at least one diagnosis from the overapproximation $D^{\pm}(M[i/\overline{pf}])$. In both cases at least one diagnosis is removed from the set of potential diagnoses which is defined as $D^{\pm}(M[i/\overline{pf}]) \setminus D^{\pm}(M[i/\underline{pf}])$.

Quantitatively, acquiring information about unknown point $\vec{x}$ has two possible outcomes: (i) $\vec{x} \in T(pf)$ and the reduction is $wnd(\vec{x})$; or (ii) $\vec{x} \in F(pf)$ and the reduction is $ewnd(\vec{x})$ (see Lemma 4). As $ewnd(\vec{x}) \subseteq wnd(\vec{x})$ (Definition 18), the guaranteed reduction in size is $|ewnd(\vec{x})|$. The proposition follows, since $\vec{x}$ is chosen s.t. $|ewnd(\vec{x})|$ is maximal. □

Proposition 9 suggests to query unknown points $\vec{x}$ where $|ewnd(\vec{x})|$ is maximum.

If there are more false points than true points (e.g., for contexts that accept only one belief set for each input), using $ewnd$ instead of wnd is even more suggestive. If the primary interest are necessary bridge rules (cf. previous section), we can base query selection on the number of bridge rules which become necessary if a certain unknown point is a false point. Let $nwnd(\vec{x}) = nec(\overline{D^{\pm}} \backslash ewnd(\vec{x})) \backslash nec(\overline{D^{\pm}})$, where $\overline{D^{\pm}} = D^{\pm}(M[i/\overline{pf}])$, then $|nwnd(\vec{x})|$ many bridge rules become necessary if $\vec{x}$ is identified as a false point.

Another possible criterion for selecting queries can be based on the likelihood of errors, similar to the idea of *leading diagnoses* [dK91]. Although a different notion of diagnosis is used there, the basic idea is applicable to our setting: if multiple problematic bridge rules are less likely than single ones, or if we have confidence values for bridge rules (e.g., some were designed by an expert, others by a less experienced administrator), then we can focus confirming or discarding diagnoses that have a high probability. If we have equal confidence in all bridge rules, this amounts to using *cardinality-minimal* potential diagnoses for determining witnesses and guiding the selection of queries.

Example 36 (ctd). *In our example, the set of potential diagnoses is large, but the cardinality-minimal diagnosis is the empty diagnosis, which has the following property: at C_{cc} the bridge rules add heads $\{card(cnr2)\}$, and C_{cc} either accepts $\emptyset$ or $\{valid(cnr1)\}$ (the unrelated credit card). Therefore, points $(0,1,0,0)$ and $(0,1,1,0)$ are the only witnesses for D_δ, and querying these two unknown points is sufficient for verifying or falsifying D_δ. (Note that pf_{cc} has 12 unknown points, the four known points (one true and three false points) are $(0,0,X,Y)$ s.t. $X,Y \in \mathbb{B}$.) After updating pf with these points (false points, if all credit cards are valid), the overapproximation yields the $\subseteq$-minimal diagnoses; this result is optimal.* □

Estimating Approximation Quality. The previously defined notation wnd and $ewnd$ relates unknown points to sets of potential diagnoses.

This correspondence allows to obtain an estimate for the quality of an approximation, simply by calculating the ratio between known and potential true (resp., false) points: a high value of $\frac{|T(pf)|}{|T(pf)|+|U(pf)|}$ indicates a high underapproximation quality, while a low value indicates an underapproximation distant from the actual system. The same can be done for the overapproximation, by exchanging $T(pf)$ with $F(pf)$.

These estimates can be calculated efficiently and prior to calculating an approximation. A decision between calculating an under- vs. an overapproximation could be based on this heuristics.

Stronger Queries. Instead of membership queries which check whether $O \in \mathbf{ACC}(kb \cup I)$ for given $(\vec{\imath}, \vec{o})$, one could use stronger queries that provide the *value* of $\mathbf{ACC}(kb \cup I)$ for a given $\vec{\imath}$. On the one hand this allows for a better query selection, roughly speaking because combinations of unknown points witness more diagnoses exclusively than they do individually. On the other hand, considering such combinations increases computational cost. Another extension of limited querying is the usage of meta-information, e.g., monotonicity, or consistency properties, of a partially known context.

3.5.5 Computational Complexity

As our approximation methods deal with incomplete knowledge, it is important how their computational complexity compares to the full knowledge case which is shown in Table 3.1 on page 32. With context complexity in $\mathbf{P}$ (resp., $\mathbf{NP}$, $\mathbf{\Sigma_i^P}$), recognizing correct diagnoses under full knowledge is in $\mathbf{NP}$ (resp., $\mathbf{NP}$, $\mathbf{\Sigma_i^P}$) while recognizing minimal diagnoses and minimal explanations under full knowledge is in $\mathbf{D_1^P}$ (resp., $\mathbf{D_1^P}$, $\mathbf{D_i^P}$); completeness holds in all cases.

Let us first consider the case where some contexts C_i are given by their corresponding BF f_i (such that $f_i(\vec{\imath}, \vec{o})$ can be evaluated efficiently). As we know that context C_i accepts only input/output combinations which are true points of f, we simply guess all possible output beliefs O_i of all contexts and evaluate bridge rules to obtain I_i; if for some C_i as above, $f_i(\vec{\imath}_i, \vec{o}_i)=0$ we reject, otherwise we continue checking context acceptance for other contexts. Overall, this method leads to the same complexity as if all contexts were total. Thus, detecting explanations of inconsistency for an MCS M, where some contexts of M are given as BFs, has the same worst case complexity as if M were given regularly.

Consider now the case of approximation. Given a MCS, where a pdBF pf is given instead of a BF in a representation such that the value of $pf(\vec{\imath}, \vec{o})$ can be computed efficiently. This implies that the extensions $\underline{pf}$ and $\overline{pf}$ can be computed efficiently as well. Hence, approximations of diagnoses and explanations have the same complexity as the exact concepts.

Dealing with incomplete information usually increases complexity, as customary for many nonmonotonic reasoning methods. Our approach, however, exhibits no such increase in complexity, even though it provides faithful under- and overapproximations.

3.6 Discussion and Related Work

We presented the notions of diagnosis and inconsistency explanation for explaining inconsistencies in multi-context systems, showed relationships between these notions, analyzed their computational complexity, and studied possibilities to approximate these notions in partially known MCSs.

Due to presence of nonmonotonic contexts (witnessed by the decision support system in our example) and the possibility of negation in bridge rules, the problem we deal with in this chapter is more general than many other approaches for dealing with inconsistency in the literature.

Moreover, a decision for repair may need to take domain knowledge into account, as illustrated by our example, where it is not obvious how to resolve the dilemma. Therefore the work in this chapter is the foundation for the subsequent work in this thesis, which ultimately culminates in the final chapter, where we apply the notions introduced in this chapter for specifying a declarative policy language for (semi-)automatic inconsistency management in multi-context systems.

Regarding our approach for approximating explanations of inconsistency, note that even if nothing is known about the behavior of some context C, the overapproximation accurately characterizes inconsistencies that do not involve C. Further work on approximations could use meta-information about context properties to improve approximation accuracy. In particular learning a BF seems suggestive for our setting of incomplete information. However, explaining inconsistency requires correct information, therefore pac-learning methods [Val84] are not applicable. On the other hand, exact methods [Ang88, HPRW96] require properties of the contexts which are beneficial to learning and might not be present.[2] Moreover, contexts may only allow membership queries, which are insufficient for efficient learning of many concept domains [Ang88]. Furthermore, partially known contexts may not allow many, even less a polynomial number of queries (which is the target for learnability). Most likely it will thus not be possible to learn the

[2]Note that even if a context's logic is monotonic (resp., positive), this does not imply that the BF corresponding to the context is monotonic (resp., positive).

complete function. Hence learning cannot replace our approach, but it can be useful as a pre-processing step to increase the amount of partial information. The incorporation of probabilistic information into the pdBF representation is another interesting topic for future research.

Consecutive and Orthogonal Work. In conjunction with diagnoses and inconsistency explanations, *modularity properties* for diagnoses and inconsistency explanations, *refined notions* of diagnoses and explanations (i.e., bridge rule bodies are modified instead of removing them or making them completely unconditional), and possibilities for using ceteris paribus orders for selecting *preferred diagnoses* are described in [EFSW10], This work is continued in [EFW10] with additional *filter and preference notions* over diagnoses, and the usage of *conditional preference networks* for selecting preferred diagnoses from the set of all diagnoses. Quantitative assessment of diagnoses using inconsistency measures, and preference orders over *categories* of bridge rules is described in [Wei10].

3.6.1 Related Work

We first discuss scientific literature that is tightly related to the work of this chapter. Then we give a broader overview of related work about analyzing and/or repairing inconsistency.

Managing inconsistency in a *homogeneous MCS setting* is addressed in [BAH11, BA08]. The authors manage inconsistency by making bridge rules defeasible for inconsistency removal, i.e., a rule is applicable only if its conclusion does not cause inconsistency. This concept is described in terms of an argumentation semantics in [BA10]. The decision which bridge rules to ignore is based, for every context, on a *strict total order* of all contexts. This set of ignored rules then corresponds to a unique deletion-diagnosis whose declarative description is more involved than our notion, but which is polynomially computable. However, the second component of diagnoses, i.e., rules that are forced to be applicable, have no counterpart in the defeasible MCS inconsistency management approach. Furthermore, the strict total order over contexts forces the user to make (perhaps unwanted) decisions at design time; alternative orders require redesigns and separate runs. Our approach does not require this, yet it has be combined with preference orders [EFW10].

Debugging answer set programs, i.e., finding out why some program has no answer set, is a task similar to computing diagnoses. Practical and theoretical work on this topic is described in [Syr06,BGP$^+$07]. Both approaches are similar to removal-based diagnoses. These results can be used to compute (possibly constrained) diagnoses of an MCS, given that it has ASP contexts and uses the more restrictive grounded equilibria semantics (cf. [BE07]).

Abduction was used in [IS95] to repair theories in (nonmonotonic) autoepistemic logics, knowledge bases, and extended logic programs, by means of two notions of ‘explanation’ and ‘anti-explanation’. Given a theory K and abducibles Γ, an explanation (anti-explanation) in the sense of [IS95] removes $O \subseteq \Gamma$ and adds $I \subseteq \Gamma$ to entail (resp. not entail) an observation F. I.e., $(K \cup I) \setminus O \models F$ (explanation), resp. $(K \cup I) \setminus O \not\models F$ (anti-explanation). A repair of an inconsistent K is given by an anti-explanation of $F = \bot$. Our notion of diagnosis amounts to a 2-sorted variant of such anti-explanations, where $O \subseteq \Gamma_O$ and $I \subseteq \Gamma_I$; under suitable conditions, it is reducible to ordinary anti-explanations. However, the notion of explanation introduced in this thesis has no counterpart in [IS95].

Model-based diagnosis is a method for detecting faulty components given a logical model for correct and faulty behavior of system components, a logical model of the system, and a set of observations (i.e., measurements) of a concrete system at hand where a problem shall be found. This approach was introduced by Reiter in [Rei87]; a nice introduction and overview about diagnosis and probing, including some criticism of the approach, can be found in [dKK03]. Model-based diagnosis of nonmonotonic systems is discussed in [PEB94], where the authors point out that with a nonmonotonic modeling language, consistency-based and abductive diagnosis can no longer be regarded as separate techniques, and show how to integrate both types of diagnosis into an abductive framework.

Related to our work on *approximating diagnoses* is [tTvH96], who aimed at approximating abductive diagnoses of a single knowledge base. They replaced classical entailment with approximate entailment of [SC95], motivated by computational efficiency. However, there is no lack of information about the knowledge base as in our case, moreover they approximate semantics to gain efficiency, different from our approach that uses unmodified semantics of the system. Model-based diagnosis for systems where no complete model is available has been described in [CDT89], here some causal dependencies are "may-dependencies" and they are used as abducibles in the diagnosis process, leading to strongly (resp. weakly) confirmed diagnoses, where all (resp. some) may-dependencies lead to a certain diagnosis.

Lower and upper bounds of classical theories (viewed as sets of models [SK96]), known as cores and envelopes, are related to our over- and underapproximations of $D^{\pm}$ and $E^{\pm}$. Envelopes also were used for (fast) sound, resp. complete, reasoning from classical theories.

The limited querying approach is related to *optimal probing strategies* [BRMO03]. However, we do not require probing to localize faults in the system, but to obtain information about the behavior of system parts, which have a much more fine grained inner structure and more intricate dependencies than the systems in [BRMO03]. (Those system parts have as possible states 'up', and 'down', while in MCSs each partially known context possibly accepts certain belief sets for certain inputs.)

Dealing with Inconsistency in Knowledge-based Systems

Inconsistency in knowledge-based systems and database systems follows three main approaches in the literature: (a) *repairing* modifies the content of knowledge bases which are combined into one system, (b) *consistent query answering* ignores a minimal subset of beliefs or subsystems and operating on the resulting consistent system (no knowledge is permanently removed), and (c) *paraconsistent reasoning* where no knowledge is ignored but instead there are at least parts of an overall solution which use inconsistency as an additional information and reason using inconsistent information

Different from all these approaches, the primary aim of this chapter is to provide a solid and useful theoretical framework for analyzing inconsistency. We do not aim at automatically restoring consistency, although our notions can be used to achieve that.

Repair Approaches. A lot of work about dealing with inconsistency has focused on the repair of data, i.e., to change knowledge bases in order to restore consistency. In case of multiple knowledge bases, these approaches regard the mappings (or similar concepts) between knowledge bases as fixed. We here also include approaches that do not modify original data but copy them and operate on the copy (belief merging and information integration).

Belief revision [AGM85,Pep08] and *belief merging* [KP05] are two approaches for combining beliefs [GRP09]. In belief revision, an existing knowledge base is updated with new beliefs, therefore it might be necessary to remove some beliefs before adding new ones. In contrast to that, belief merging is an approach for combining possibly conflicting beliefs from several knowledge bases into one resulting consistent knowledge base without the notions of prior and new beliefs. In that respect, our approach is related more to belief merging than to belief revision. Belief merging combines homogeneous knowledge bases using expressive mappings and a simple topology, and removes conflicting beliefs by modifying knowledge bases. Different from that, MCSs are decentralized systems with more complex topologies and heterogeneous components, and our work concentrates on the mappings between these components. Overall, belief merging is weakly related to our notion of deletion diagnosis.

Information integration approaches like Infomix [LGI$^+$05] wrap several sources, materialize that information in one schema and combine it in a Global-as-View approach; inconsistencies are resolved by modifying the materialized information. This can be regarded similar to this work's approach of modifying the mapping between information systems, as the source is not modified. Different from MCSs, information integration approaches use hierarchical system topologies which are acyclic (usually only a star topology) and semantics that can be evaluated using fixpoint algorithms. On the other hand they contain a more expressive mapping formalism than bridge rules in MCSs. Therefore inconsistency management in information integration systems can be seen as implicit suppression of mappings and implicit generation of missing tuples which corresponds to deactivating bridge rules and forcing bridge rules to become applicable, respectively.

Federated Databases are a distributed formalism for linked databases [HM85]. Objects can be *exported* and *imported* using a decentralized negotiation between two databases. The architecture can be described as a MCS with stratified (mostly monotonic) contexts including constraints, and positive bridge rules. Instability is rarely possible and not addressed, incoherence is handled in a database-typical manner of *cascading* or *rejecting* local or distributed constraints. [SL90] is a survey from 1990 focusing on the issues of heterogeneity (mostly referring to the integration of different query languages) and autonomy (access granting and revoking). Several protocols for global integrity constraint enforcement are presented in [GW96]. These protocols define the *quiescent* state of the system — when it is *at rest* — and ensure that no constraints are violated in such states. Consistency is defined as consistency during certain points in time, whereas this work is not specific to a certain evaluation protocol and does not consider transient states of the system. Even though SQL and stratified Datalog allows for nonmonotonicity, the possibility of instability in a distributed database system — due to a cycle going through such a construct — has not been addressed by the literature.

Ontology mapping [CSH06] and the related fields ontology merging, alignment, and integration, aim at the reuse of ontologies by combining them and identifying mappings between concepts, roles, and individuals that denote the same entity in different ontologies. This is usually done by automatic statistical methods which 'discover' mappings. Such mappings may introduce inconsistency in the (global or local) view on the resulting ontology, even though all individual ontologies are consistent. Consistency is achieved by not adding a mapping in case it would add an inconsistency. Ontology mapping considers heterogeneous ontologies, however heterogeneity there often refers to different nomenclatures expressed in different ontologies, different from MCSs where heterogeneity refers to combining systems based on different logical formalisms. Due to the integration of knowledge from different sources and the treatment of inconsistencies arising from that integration, ontology mapping is related to our work. However, in MCSs, mappings are initially given and not automatically discovered by statistical methods. Furthermore our approach does not avoid but it explains inconsistency.

Consistent Query Answering. In this section, we relate to approaches that do not remove information but ignore it temporarily to reason on a consistent system.

Consistent query answering [ABC03, Ber11] is related to this work because it can be seen as an approach that automatically applies deletion-diagnoses (and sometimes even adds missing tuples) to suppress inconsistencies for answering queries over inconsistent relational databases. In contrast, diagnoses and explanations aim at making inconsistencies visible, as they can hint at problems that should be investigated.

Consistent query answering over description logic ontologies is described in [LR07], where the TBox is considered as correct, while the ABox as possibly inconsistent, and queries are answered on maximal consistent subsets of the ABox wrt. the TBox.

Peer-to-peer data integration [CGL+08] is an automatic approach for repairing inconsistency by ignoring inconsistent components, or by ignoring beliefs held by a minority of peers in the system to deal with inconsistency. Different from that, our approach explains inconsistency by pointing out mappings that must be changed to achieve consistency; we do not automatically fix the system, in particular we do not ignore contexts or minorities. Different from MCSs, peer-to-peer data integration deals with systems that have a dynamically changing architecture, i.e., peers may enter or leave the system anytime.

Modular Ontologies are a framework where consistent description logic modules utilize and realize a set of interfaces [ED08]. These interfaces are connected by Distributed Description Logic (DDL) bridge rules [BS03]. Consistent query answering in a module is achieved by using the maximum consistent set of interfaces utilized by this module only, therefore whole interfaces will be ignored if they would cause any inconsistency in the module. Similarly, a peer-to-peer approach for propositional knowledge bases [BM08] answers queries over a maximal consistent subset of knowledge bases. In terms of MCSs, both approaches correspond to ignoring all beliefs of certain contexts from all bridge rules in the system to fix inconsistency. Contrary to that, our approach allows heterogeneous systems and a much more fine-grained modification of bridge rules.

Paraconsistent Approaches. This section considers related approaches that do not eliminate inconsistency but use it for reasoning. For that reason, paraconsistent approaches are related to the approach of this work more than data cleaning and consistent query answering.

Gabbay and Hunter argue strongly for *managing inconsistency*, in contrast to avoiding, removing, or ignoring it [GH91]. They give various real life examples where humans deal with inconsistency without problems, in particular because there is no need to restore consistency at all, or because it is advantageous to postpone restoring consistency to a later point in time. In general they argue that "inconsistency implies action" and consider as important issues (a) to allow for contradictions, (b) to use relevance information, (c) to provide meta-inference (rules about rules), (d) to not completely reject information (it might become useful again), (e) to connect learning with inconsistency ("Learning is a process that is directed by inconsistency."), (f) to consider argumentation, and (g) to consider several possibilities as solution without deciding immediately which one to prefer. They propose a framework of a database together with a supplementary database, that can act on the database using an action language. Proposed actions are (a) learning, (b) information acquisition, (c) inconsistency removal (belief revision, truth maintenance), (d) inference preference (applying inference strategies and nonmonotonic reasoning), and (e) argumentation (directing a dialogue). They propose to use Labeled Deductive Systems (LDS) [Gab93] as underlying logical formalism. LDSs are a formalism which captures different kinds of monotonic and nonmonotonic logics, similar as the MCS logic abstraction. However LDSs are calculus-based while MCS logics are model-based.

The *Data and Action* formalism [GH93] is a more concrete realization of the initial ideas in [GH91]. This formalism uses an 'object level' logic (Da Costa's paraconsistent logic C_ω [DCC86]) and a 'meta level' logic (temporal logic). The main principle is that the meta-level reflects the state of the object level over time. The meta level is always consistent, and it may modify an inconsistent object level (which would in turn create a feedback into the meta-level). Gabbay and Hunter describe their goal as a generalization of several approaches to deal with inconsistency, namely paraconsistency (which localizes inconsistency), truth maintenance systems (which force consistency on data by removing/adding certain beliefs), and integrity constraints (which fully prevent inconsistent data from entering a database).

Many-valued logics and *paraconsistent logic programming* are two further approaches for dealing with inconsistency. Early work in this field was done by Blair and Subrahmanian [BS89] and by Kifer and Lozinskii [KL92]. They define annotated predicate calculus (APC) for reasoning in the presence of inconsistency, based on a lattice-valued logic where values correspond with degrees of beliefs and each atomic formula is annotated with such a value. Recent work that focusses on multi-valued logic is due to Schenk, who models the trust on different information sources in the semantic web as an extension of the four-valued logic $\mathcal{FOUR}$ [Sch08]. He uses external predicates which lead to bridge-rule like constructions, to model information flow between trusted (direct access) and less trusted (cache) information sources. This framework is then generalized to a finite number of trust levels, along with a definition of stable and well-founded semantics. The truth values in extended $\mathcal{FOUR}$ are $\top_{i,j}$, $t_{i,j}$, $f_{i,j}$ and $\bot_{i,j}$, with i and j designating the information source. Inconsistency is the assignment of $\top_{i,j}$; the source of an inconsistency can be detected from the indices which indicate single-module inconsistency ($\top_{i,i}$) or a module-interaction inconsistency ($\top_{i,j}, i \neq j$). Recent work on paraconsistency is by de Amo and Sakuray Pais who use the paraconsistent logic LFI1 to define P-Datalog as query language for inconsistent databases [dAP07]. They assume that data is labeled either as "sure" or as "controversial", their truth values are true, false, inconsistent, and unknown. They present an evaluation method based on an alternating fixed-point operator.

4 Realizing Inconsistency Analysis in MCSs with HEX

In this chapter, we describe a method for computing the notions introduced in the previous chapter and also describe an implementation of that method.

We first show how diagnoses and equilibria of MCSs can be computed by rewriting the MCS at hand into a formalism of computational logic, in particular into HEX-programs. For the rewriting we establish how diagnoses and equilibria correspond to answer sets of the rewritten HEX-program.

We then describe the architecture and functionality of the tool MCS-IE [MIE12a], which is an open source prototype realized in the framework of the dlvhex [DHX12] software. MCS-IE supports various analyses of inconsistency in MCSs using the the HEX-rewritings discussed in this chapter. We discuss experiences about performance of the MCS-IE tool and describe how these experiences prompted the research described in the subsequent chapter.

Our contributions, which have been published in less detail than here in [EFSW10,BEFS10] are as follows.

- We give a (naive) rewriting from an MCS M to a HEX program $P_D(M)$ such that answer sets of $P_D(M)$ correspond 1-1 to diagnoses of M.
- We give a rewriting from an MCS M to a HEX program $P_p(M)$ such that answer sets of $P_p(M)$ correspond 1-1 to output-projected equilibria of M.
- We combine the previous two rewritings which yields a more efficient rewriting $P_p^D(M)$ such that answer sets of $P_p^D(M)$ correspond 1-1 to the set of diagnoses plus witnessing output-projected equilibria of M.
- We describe the architecture of MCS-IE, show examples for input and output files corresponding to our Inconsistent Medical Example, and briefly show how the tool is used from the command line.
- Finally we discuss experiences of using the tool, in particular experiences regarding evaluation performance and scalability.

4.1 Computing Diagnoses by Rewriting to HEX

We next use HEX programs to describe a generic approach for computing diagnoses, and a way for checking consistency of MCS. In order to compute diagnoses more efficiently, we then integrate both approaches into one HEX encoding.

4.1.1 Generic Approach

We can compute diagnoses for some MCS M by guessing a candidate diagnosis and checking whether it yields a consistent system.

Due to Proposition 2, we only consider diagnoses (D_1, D_2) where $D_1 \cap D_2 = \emptyset$; diagnoses with $D_1 \cap D_2 \neq \emptyset$ can always trivially be reconstructed from such diagnoses. Furthermore the omitted diagnoses are never minimal, and we are often interested only in the latter.

Given an MCS M, we assemble a HEX-program $P_D(M)$ as follows. For each bridge rule $r \in br_M$, we add the following guessing rule (here and in the following, we use constant r as a name for rule r):

$$norm(r) \vee d_1(r) \vee d_2(r). \tag{4.1}$$

Intuitively, predicates d_1 and d_2 hold bridge rules that are removed from M, respectively bridge rules that are added to M in unconditional form; $norm$ denotes unmodified bridge rules.

Furthermore we create a check for the diagnosis property, which is 'outsourced' to an external atom $\&eq_M[d_1, d_2]()$ with the following evaluation function:

$$f_{\&eq_M}(I, d_1, d_2) = 1 \text{ iff } M[br_M \setminus \{r \mid d_1(r) \in I\} \cup cf(\{r \mid d_2(r) \in I\})] \not\models \bot. \tag{4.2}$$

Using this external atom, the following constraint eliminates all answer sets that do not correspond to diagnoses:

$$\leftarrow not\ \&eq_M[d_1, d_2](). \tag{4.3}$$

The program $P_D(M)$ comprising (4.1) and (4.3) properly captures diagnoses. The answer sets I of $P_D(M)$ correspond to the diagnoses $(D_{I,1}, D_{I,2})$ of an MCS M as follows.

Theorem 8. *Let M be an MCS and let $P_D(M)$ be as above. Then (i) for each answer set I of $P_D(M)$, the pair $(D_{I,1}, D_{I,2}) = (\{r \in br_M \mid d_1(r) \in I\}, \{r \in br_M \mid d_2(r) \in I\})$ is a diagnosis of M, and (ii) for each diagnosis $(D_1, D_2) \in D^{\pm}(M)$ with $D_1 \cap D_2 = \emptyset$, there is an answer set I of $P_D(M)$ such that $(D_{I,1}, D_{I,2}) = (D_1, D_2)$.*

For proving the correctness of our HEX encodings, we use some lemmas.

Lemma 5. *Let $P = R \cup C$ be a HEX program consisting of an ordinary HEX-program R and a set of constraints C which contain external atoms. Then for every $I \in \mathcal{AS}(P)$ it holds that $I \in \mathcal{AS}(R)$ and I does not satisfy the body of any constraint in C.*

Proof. From $I \in \mathcal{AS}(P)$ we know that $I \models P$ and therefore $I \models R$ and $I \models C$. From the latter we infer that I does not satisfy the body of any constraint in C (i.e., the second claim). Thus the reduct fP^I does not contain any constraint from C. Hence $fP^I = fR^I$ and I is a minimal model of fR^I as it is a minimal model of fP^I. □

Lemma 6. *Let P be a HEX program, and let $I \in \mathcal{AS}(P)$ be an answer set of P. Then for every atom $a \in I$ it holds that there is a rule $r \in P$ of form* (2.3) *with $a \in \{a_1, \dots, a_k\}$ and I satisfies the body of r.*

Proof. Assume towards a contradiction that $I \in \mathcal{AS}(P)$, $a \in I$ and no rule $r \in P$ is such that a is in the head of r and I satisfies the body of r. Due to the latter assumption, no rule that contains a in the head is contained in fP^I. Since I is an answer set of P, $I \models fP^I$, therefore the bodies of all rules in fP^I are satisfied by I. Hence every rule in fP^I has a nonempty intersection of its head with I (otherwise $I \not\models fP^I$). Because no rule in fP^I contains a in the head, it follows that $I \setminus \{a\} \models fP^I$, therefore I is no minimal model of fP^I and no answer set, which is a contradiction. □

Proof of Theorem 8. ($\Rightarrow$) Given $I \in \mathcal{AS}(P_D(M))$, due to Lemma 5 we have (a) $I \in AS(R)$ where R contains rules (4.1), and (b) constraint (4.3) has an unsatisfied body. Due to (a) the pair $(D_{I,1}, D_{I,2})$ is such that $D_{I,1}, D_{I,2} \subseteq br(M)$. From (b) we know that the external atoms in (4.3) evaluate to true, therefore from (4.2) we know $M[br(M) \setminus D_{I,1} \cup cf(D_{I,2})] \not\models \bot$, hence $(D_{I,1}, D_{I,2}) \in D^{\pm}(M)$.

($\Leftarrow$) Given $(D_1, D_2) \in D^{\pm}(M)$ with $D_1 \cap D_2 = \emptyset$, the corresponding $Q = \{norm(r) \mid r \in br_M \setminus (D_1 \cup D_2)\} \cup \{d_1(r) \mid r \in D_1\} \cup \{d_2(r) \mid r \in D_2\}$ satisfies rules (4.1), furthermore $P_D(M)$ contains only constraint (4.3) apart from (4.1), and this constraint, per definition of $f_{\&eq_M}$, has an unsatisfied body if $(D_1, D_2) \in D^{\pm}(M)$. Therefore the reduct $fP_D(M)^Q$ contains only the rules (4.1). For each rule $r \in br(M)$, Q contains exactly one atom from the set $\{norm(r), d_1(r), d_2(r)\}$. Hence Q satisfies the reduct $fP_D(M)^Q$, furthermore for each atom we remove from Q, Q no longer satisfy one rule in (4.1). Therefore Q is a minimal model of $fP_D(M)^Q$ and hence $Q \in \mathcal{AS}(P_D(M))$. $\square$

Note that to compute all answer sets of $P_D(M)$, the function f_{eq_M}, which amounts to consistency checking in an MCS, will be called $3^{|br(M)|}$ times. The approach we develop in the following can drastically reduce the computational effort by performing large parts of the MCS consistency checking within the HEX encoding, with external atoms used only for evaluating $\mathbf{ACC}$ of each context in M.

4.1.2 Consistency Checking

Consistency of an MCS can be checked by computing output-projected equilibria. For that, we assemble a program $P_p(M)$ as follows.

We guess presence or absence of each output belief of each context in M, where we represent each belief p using a constant p:

$$pres_i(p) \lor abs_i(p). \qquad \text{for all } p \in OUT_i,\ 1 \leq i \leq n. \tag{4.4}$$

Given an interpretation I of $P_p(M)$, we use $A_i(I) = \{p \mid pres_i(p) \in I\}$, $1 \leq i \leq n$, to denote the set of output beliefs at context C_i, corresponding to the guess in (4.4).

We evaluate each bridge rule (2.1) by two corresponding HEX rules, depending on output beliefs guessed in (4.4):

$$\begin{aligned} b_i(s) \leftarrow & \, not\ d_1(r), pres_{c_1}(p_1), \ldots, pres_{c_j}(p_j), \\ & not\ pres_{c_{j+1}}(p_{j+1}), \ldots, not\ pres_{c_m}(p_m). \end{aligned} \tag{4.5}$$

$$b_i(s) \leftarrow d_2(r). \tag{4.6}$$

Given an interpretation I of $P_p(M)$, we use $B_i(I) = \{s \mid b_i(s) \in I\}$ to denote the set of bridge rule heads at context C_i, activated by the output-projected belief state $\mathcal{A}(I) = (A_1(I), \ldots, A_n(I))$. Note that atoms $d_2(r)$ and $d_1(r)$ will become useful in the next step (Theorem 9), when we integrate $P_p(M)$ with $P_D(M)$. For now, they do not occur in any rule head in the program, therefore (4.5) will never be deactivated by $d_1(r)$ and (4.6) will never become applicable.

Finally, we ensure that answer sets of the program correspond to output-projected equilibria by checking whether each context C_i accepts the guessed $A_i(I)$ wrt. the set $B_i(I)$ of bridge rule heads activated by bridge rules. For that we create an external atom

$$\&con\_out_i[pres_i, b_i]()$$

which realizes $\mathbf{ACC}_i$ in an external computation. This external atom returns true iff context C_i, when given $B_i(I)$, accepts a belief set S_i such that S_i projected to output-beliefs OUT_i is equal to $A_i(I)$. Formally,

$$f_{\&con\_out_i}(I, pres_i, b_i) = 1 \text{ iff } A_i(I) \in \mathbf{ACC}_i(kb_i \cup B_i(I))|_{OUT_i}. \tag{4.7}$$

We complete $P_p(M)$ by adding the following constraints:

$$\leftarrow not\ \&con\_out_i[pres_i, b_i](). \qquad \text{for all } i \text{ with } 1 \leq i \leq n. \tag{4.8}$$

Let the program $P_p(M)$ comprise the rules (4.4), (4.5), (4.6), and (4.8). Then the answer sets I of $P_p(M)$ correspond to output-projected equilibria of M as follows.

Proposition 10. *Let M be an MCS, and let $P_p(M)$ as above. Then (i) for each answer set I, the belief state $\mathcal{A}(I)$ is an output-projected equilibrium of M, and (ii) for each output-projected equilibrium $\mathcal{S}'$ of M there is an answer set I of $P_p(M)$ such that $\mathcal{A}(I) = \mathcal{S}'$.*

As existence of output-projected equilibria characterizes consistency of MCSs (Theorem 2), we obtain the following.

Corollary 3. *Given an MCS M, $P_p(M)$ has some answer set iff M is consistent.*

Proof of Proposition 10. (i) Given $I \in \mathcal{AS}(P_p(M))$, due to Lemma 5 we have (a) $I \in \mathcal{AS}(R)$ where R contains rules (4.4), (4.5), and (4.6); and (b) no constraint (4.8) has a satisfied body. In R, (4.4) are the only rules with $pres_i$ and abs_i atoms in the head, therefore $A_i(I) \subseteq OUT_i$ for each context $C_i \in c(M)$. Hence $\mathcal{A}(I)$ is an output-projected belief state of M. Due to Lemma 6, I does not contain $d_1(r)$ or $d_2(r)$ for any $r \in br(M)$, as no rule contains these atoms in the head; therefore (4.6) never has a satisfied body and I always satisfies $not\, d_1(r)$ in (4.5). Due to Lemma 6, I contains $b_i(s)$ iff there is at least one bridge rule $r \in br(M)$ such that in the corresponding rule (4.5), for all i, $1 \leq i \leq j$, $pres_{c_i}(p_i) \in I$, and for all l, $j < l \leq m$, $pres_{c_l}(p_l) \notin I$. This in turn is the case iff for all $(c_i : p_i)$ in the body of r, $p_i \in A_{c_i}(I)$, and for all $\mathbf{not}$ $(c_l : p_l)$ in the body of r, $p_l \notin A_{c_l}(I)$. The same is true iff bridge rule r is applicable in $\mathcal{A}(I)$, therefore we have $B_i(I) = \{h_b(r) \mid r \in app(br_i, \mathcal{A}(I))\}$ for each $C_i \in c(M)$. From (b) we can infer that for every context $C_i \in c(M)$, constraint (4.8) has an unsatisfied body, therefore the external atom returns false, hence $A_i(I) \in \mathbf{ACC}_i(kb_i \cup B_i(I))|_{OUT_i}$. We further obtain $A_i(I) \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i, \mathcal{A}(I))\})|_{OUT_i}$ for every $C_i \in c(M)$, which exactly satisfies Definition 10. Therefore $\mathcal{A}(I)$ is an output-projected equilibrium of MCS M.

(ii) Given an output-projected equilibrium $S' = (S'_1, \ldots, S'_n)$ of M, we assemble an interpretation I of $P_p(M)$ as follows: $I = \{a_i(p) \mid p \in S'_i,\ 1 \leq i \leq n\} \cup \{\bar{a}_i(p) \mid p \in OUT_i \setminus S'_i,\ 1 \leq i \leq n\} \cup \{b_i(s) \mid s \in H_i,\ 1 \leq i \leq n\}$, with $H_i = app(br_i, S')$. Facts (4.4) are contained in the reduct $fP_p(M)^I$. By construction of I and by the definition of bridge rule applicability, and because d_1 has an empty extension in I, all bodies of rules (4.5) which correspond to an applicable bridge rule in S' are satisfied, therefore these rules are part of $fP_p(M)^I$. Because d_2 has an empty extension in I, no rule from (4.6) is part of $fP_p(M)^I$. Since S' is an output-projected equilibrium, for each C_i there exists some $S'_i \in \mathbf{ACC}_i(kb_i \cup H_i)|_{OUT_i}$. As $B_i(I) = H_i$ and $A_i(I) = S'_i$, we obtain that $A_i(I) \in \mathbf{ACC}_i(kb_i \cup B_i(I))|_{OUT_i}$, therefore $f_{\&con\_out_i}(I, a_i, b_i) = 1$ for all C_i, and I does not satisfy the body of any constraint (4.8). Hence none of the constraints (4.8) is part of $fP_p(M)^I$. I satisfies all rules of $P_p(M)$ and all rules of $fP_p(M)^I$. $P_p(M)$ does not contain loops, neither does $fP_p(M)^I$, hence I is a $\subseteq$-minimal model of $fP_p(M)^I$ and therefore $I \in \mathcal{AS}(P_p(M))$. □

4.1.3 Combining Diagnosis Guess and Consistency Checking

To implement $f_{\&eq_M}$ in the program $P_D(M)$, we could use $P_p(M)$: given a candidate diagnosis (D_1, D_2), we simply add a representation of this candidate using facts $d_1(X)$ and $d_2(X)$ to $P_p(M)$. The resulting program is equivalent to $P_p(M[br_M \setminus D_1 \cup cf(D_2)])$. By returning 1 iff that program has some answer set, we obtain a faithful implementation of $f_{\&eq_M}$.

However, it is possible (and moreover more efficient) to directly integrate the programs $P_p(M)$ and $P_D(M)$: let $P_p^D(M)$ be the program comprising $P_p(M)$ and the rules (4.1). In $P_p^D(M)$ the diagnosis guess in predicates d_1 and d_2 directly activates or deactivates bridge rule evaluation in (4.5) and (4.6). The answer sets I of $P_p^D(M)$ then correspond to the diagnoses $(D_{I,1}, D_{I,2})$ and to the output-projected equilibria $(S'_1(I), \ldots, S'_n(I))$ of the MCS $M[br_M \setminus D_{I,1} \cup cf(D_{I,2})]$ as follows.

Theorem 9. *Let M be an MCS, and let $P_p^D(M)$ be as above. Then* *(i) for each answer set I of $P_p^D(M)$, the pair $(D_{I,1}, D_{I,2}) = (\{r \in br_M \mid d_1(r) \in I\}, \{r \in br_M \mid d_2(r) \in I\})$ is a diagnosis of M and the tuple $\mathcal{A}(I) = (A_1(I), \ldots, A_n(I))$ (see above) is an output-projected equilibrium of $M[br_M \setminus D_{I,1} \cup cf(D_{I,2})]$; and*

(ii) for each diagnosis $(D_1, D_2) \in D^{\pm}(M)$ where $D_1 \cap D_2 = \emptyset$, and for each output-projected equilibrium S' of $M[br_M \setminus D_1 \cup cf(D_2)]$, there exists an answer set I of $P_p^D(M)$ such that $(D_1, D_2) = (D_{I,1}, D_{I,2})$ and $S' = \mathcal{A}(I)$.

Proof. (i) Given $I \in \mathcal{AS}(P_p^D(M))$, due to Lemma 5 we have (a) $I \in \mathcal{AS}(R)$ where R contains rules (4.1), (4.4), (4.5), and (4.6); and (b) no constraint (4.8) has a satisfied body. As in $P_p(M)$, every I corresponds to a unique belief state $\mathcal{A}(I)$ of M, and as in $P^D(M)$, every I corresponds to a unique pair $(D_{I,1}, D_{I,2})$, $D_{I,1}$, $D_{I,2} \subseteq br_M$. Due to Lemma 6, I contains $b_i(s)$ iff at least one of the following is true: $d_2(r) \in I$ and accordingly $r \in D_{I,2}$, or there is at least one bridge rule $r \in br(M)$ such that $d_1(r) \notin I$ and in the corresponding rule (4.5) we have that for all i, $1 \leq i \leq j$, $pres_{c_i}(p_i) \in I$, and for all l, $j < l \leq m$, $pres_{c_l}(p_l) \notin I$; this holds iff $r \notin D_{I,1}$ and $r \in app(br_i(M), \mathcal{A}(I))$, which holds iff $r \in app(br_i(M) \setminus D_{I,1}, \mathcal{A}(I))$. Therefore, for each context $C_i \in c(M)$ we have $B_i(I) = \{h_b(r) \mid r \in app(br_i(M) \setminus D_{I,1}, \mathcal{A}(I))\} \cup \{h_b(r) \mid r \in D_{I,2}\}$. The condition-free bridge rules are always applicable, therefore $B_i(I) = \{h_b(r) \mid r \in app(br_i(M[br_M \setminus D_{I,1} \cup cf(D_{I,2})]), \mathcal{A}(I))\}$. Note that in this expression, first all bridge rules of M are modified using $D_{I,1}$ and $D_{I,2}$, then the bridge rules of context C_i of the result are extracted using $br_i(\cdot)$. From (b) we know that for every context $C_i \in c(M)$, the external atom in (4.8) returns false, therefore $A_i(I) \in \mathbf{ACC}_i(kb_i \cup B_i(I))|_{OUT_i}$ for every $C_i \in c(M)$. Substituting $B_i(I)$ we obtain $A_i(I) \in \mathbf{ACC}_i(kb_i \cup \{h_b(r) \mid r \in app(br_i(M[br_M \setminus D_{I,1} \cup cf(D_{I,2})]), \mathcal{A}(I))\})|_{OUT_i}$. Therefore $\mathcal{A}(I)$ is an output-projected equilibrium of MCS $M[br_M \setminus D_{I,1} \cup cf(D_{I,2})]$ and $(D_{I,1}, D_{I,2}) \in D^{\pm}(M)$.

(ii) Given a diagnosis $(D_1, D_2) \in D^{\pm}(M)$ and given an output-projected equilibrium $S' = (S'_1, \ldots, S'_n)$ of $M' = M[br_M \setminus D_1 \cup cf(D_2)]$ we assemble the interpretation

$$I = \{d_1(r) \mid r \in D_1\} \cup \{d_2(r) \mid r \in D_2\} \cup \{norm(r) \mid r \notin (D_1 \cup D_2)\} \cup \\ \{a_i(p) \mid p \in S'_i\} \cup \{\bar{a}_i(p) \mid p \in OUT_i \setminus S'_i\} \cup \{b_i(s) \mid s \in H_i\}$$

where $H_i = app(br_i(M'), S')$. Since S' is an output-projected equilibrium, I satisfies constraints (4.8), therefore they are not part of the reduct $fP_p^D(M)^I$. By construction of I, those rules in (4.5) where $r \in D_1$ or r is not applicable in $\mathcal{A}(I)$ have an unsatisfied rule body, so these rules are not part of the reduct. Those rules in (4.6) where $r \in D_2$ have a satisfied rule body, so these rules are always part of the reduct. Other rules in (4.5) or (4.6) are satisfied by I as their body is not satisfied. For each applicable bridge rule r, the according head atom $b_i(s)$ is part of I, and $P_p^D(M)$ contains no cyclic dependencies between rules (hence neither does the reduct $fP_p^D(M)^I$). Therefore I is a minimal model of rules (4.5) and (4.6) in the reduct. Rules (4.1) and (4.4) are contained in the reduct, and I by construction is a minimal model of these rules. Therefore I is a model of $P_p^D(M)$ and a minimal model of $fP_p^D(M)^I$, and we have that $I \in \mathcal{AS}(P_p^D(M))$. □

Creating one HEX-program that contains both guessing of diagnosis candidates and evaluation of bridge rule semantics has the advantage of one level less of HEX indirection than the naive approach of using $P_p(M)$ in $f_{\&eq_M}$. This allows to reduce the number of external computations, e.g., if the inconsistency of one context makes acceptability checks on other contexts unnecessary.

In Section 4.2 we discuss an implementation of $P_p^D(M)$ and practical results on scalability.

4.1.4 Explanations

So far we only discussed how to compute diagnoses. For obtaining explanations there are two ways: either compute them from the set of diagnoses using Theorem 1, or compute them via an encoding directly in HEX. Such an encoding uses the saturation technique (see, e.g., [EIK09]) to do a check over all subsets, resp., subsets of rules in an explanation candidate, as per definition of explanation (Definition 6).

An appropriate rewriting has been developed, primarily by Antonius Weinzierl. As we discuss this rewriting in subsequent sections, for completeness' sake we here give the full encoding. We also give some intuition why the encoding works but we omit the formal proof of correctness which can be found in the PhD thesis of Weinzierl.

The encoding $P^E(M)$ is as follows. We guess an explanation candidate (E_1, E_2), where we represent E_1, resp., E_2 by predicates $e1$, resp., $e2$:

$$\mathit{e1}(r) \lor \mathit{ne1}(r). \qquad\qquad \mathit{e2}(r) \lor \mathit{ne2}(r).$$

We guess a pair (R_1, R_2) with $E_1 \subseteq R_1 \subseteq br_M$ and $R_2 \subseteq br_M \setminus E_2$ as follows:

$$\begin{aligned} \mathit{r1}(R) &\leftarrow \mathit{e1}(R). \\ \mathit{r1}(R) \lor \mathit{nr1}(R) &\leftarrow \mathit{ne1}(R). \\ \mathit{nr2}(R) &\leftarrow \mathit{e2}(R). \\ \mathit{r2}(R) \lor \mathit{nr2}(R) &\leftarrow \mathit{ne2}(R). \end{aligned}$$

We further guess a belief state of M; to this end we add for every $a \in \mathit{OUT}_i$ with $1 \leq i \leq |c\,(M)\,|$ the following rule:

$$\mathit{pres}_i(a) \lor \mathit{abs}_i(a).$$

Saturation invalidates an answer set if some classical model of the reduct is smaller than the saturated model of the program. Therefore we need to encode bridge rules in a special way, such that a bridge rule head is in the classical model iff at least one bridge rule body of a bridge rule with that head is satisfied. For each bridge rule r of the form

$$(i : b) \leftarrow (i_1 : b_1), \ldots, (i_{k-1} : b_{k-1}), \mathit{not}(i_k : b_k), \ldots, \mathit{not}(i_m : b_m)$$

we add the following rules:

$$\begin{aligned} \mathit{body}(r) \leftarrow\ & \mathit{r1}(r), \mathit{pres}_{i_1}(b_1), \ldots, \mathit{pres}_{i_{k-1}}(b_{k-1}), \\ & \mathit{abs}_{i_k}(b_k), \ldots, \mathit{abs}_{i_m}(b_m). \\ \mathit{r1}(r) \leftarrow\ & \mathit{body}(r). \end{aligned}$$

$$\begin{array}{rlrl} \mathit{pres}_{i_1}(b_1) \leftarrow & \mathit{body}(r). & \mathit{abs}_{i_k}(b_k) \leftarrow & \mathit{body}(r). \\ \vdots & & \vdots & \\ \mathit{pres}_{i_{k-1}}(b_{k-1}) \leftarrow & \mathit{body}(r). & \mathit{abs}_{i_m}(b_m) \leftarrow & \mathit{body}(r). \end{array}$$

Next, if a body of a rule is satisfied, the head is active at the respective context C_i (indicated by predicate in_i). We thus add for each bridge rule r the rules

$$\begin{aligned} \mathit{in}_i(b) &\leftarrow \mathit{body}(r). \\ \mathit{in}_i(b) &\leftarrow \mathit{r2}(r). \end{aligned}$$

Furthermore, if the head is added to the context, either one of the bridge rules has a satisfied body, or one of the bridge rules is unconditionally added (i.e., in R_2). Therefore, for the set $\{r_1, \ldots, r_k\}$ of bridge rules with head $(i : b)$, we have the following rules:

$$\mathit{body}(r_1) \lor \ldots \lor \mathit{body}(r_k) \lor r2(r_1) \lor \ldots \lor r2(r_k) \leftarrow \mathit{in}_i(b).$$

We derive $spoil$ whenever the guess for R_1, R_2, or the belief state is invalid:

$$\begin{aligned}spoil &\leftarrow r1(r), nr1(r).\\ spoil &\leftarrow r2(r), nr2(r).\\ spoil &\leftarrow pres_i(a), abs_i(a).\end{aligned}$$

We also derive $spoil$ if some context does not accept its belief state given its bridge rule inputs:

$$spoil \leftarrow \&con\_out'_i[spoil, pres_i, in_i, out_i]().$$

For that we create the external atom $\&con\_out'_i$ which is true iff the respective context would not accept or if $spoil$ is in the answer set: formally it has the following semantics:

$$f_{\&con\_out'_i}(I, pres_i, b_i) = 1 \text{ iff } f_{\&con\_out_i}(I, pres_i, b_i) = 0 \text{ or } spoil \in I. \tag{4.9}$$

To saturate all guesses, we add the following rules for all $r \in br_M$, $i \in c(M)$, $a \in OUT_i$, and $b \in IN_i$:

$$\begin{aligned} r1(r) &\leftarrow spoil. & r2(r) &\leftarrow spoil.\\ nr1(r) &\leftarrow spoil. & nr2(r) &\leftarrow spoil.\\ abs_i(a) &\leftarrow spoil. & pres_i(a) &\leftarrow spoil.\\ in_i(b) &\leftarrow spoil. & body(r) &\leftarrow spoil.\end{aligned}$$

Finally, we require that only spoiled answer sets survive.

$$\leftarrow not\ spoil.$$

As an interpretation I of $P^E(M)$ is an answer set if and only if it is a minimal model of $fP^E(M)^I$, I can not be an answer set if there exists some $I' \subset I$ with I' being a model of $fP^E(M)^I$. Therefore, if the guess for R_1, R_2 and the belief state is not acceptable at some context, then $spoil$ is derived and saturation takes place. On the other hand, if any such guess yields an equilibrium of M, all contexts accept. Therefore 4.9 yields 0 for all external atoms, $spoil$ is not in the answer set, and the corresponding interpretation I' is a subset of the spoiled answer set. This means an answer set is eliminated if and only if the guessed explanation candidate is not an explanation. Therefore, intuitively, answer sets of program $P^E(M)$ correspond 1-1 to explanations of MCS M.

4.2 Implementation: MCS-IE System

We have implemented the rewriting P_p^D and P_p and the saturation encoding for computing explanations in the MCS-IE [MIE12a] tool, the *MCS Inconsistency Explainer* [BEFS10], which is an experimental prototype based on the dlvhex [DHX12] solver. MCS-IE solves the reasoning tasks of enumerating output-projected equilibria, diagnoses, minimal diagnoses, explanations, and minimal explanations of a given MCS.

Figure 4.1 shows the architecture of the MCS-IE system, which is implemented as a plugin to the dlvhex solver. The MCS M at hand is described by the user in a master input file, which specifies all bridge rules and contexts (it may refer to context knowledge base files). Depending on the configuration of MCS-IE, the desired reasoning tasks are solved using one of the three rewritings $P_p(M)$, $P_p^D(M)$, resp. $P^E(M)$, on the input MCS M. MCS-IE enumerates answer sets of the rewritten program, and potentially uses a $\subseteq$-minimization module, and a module which realizes the conversions between diagnosis and explanation notions as described in Theorem 2 and Corollary 2. As a consequence of the asymmetry of duality (see Section 3.3.1),

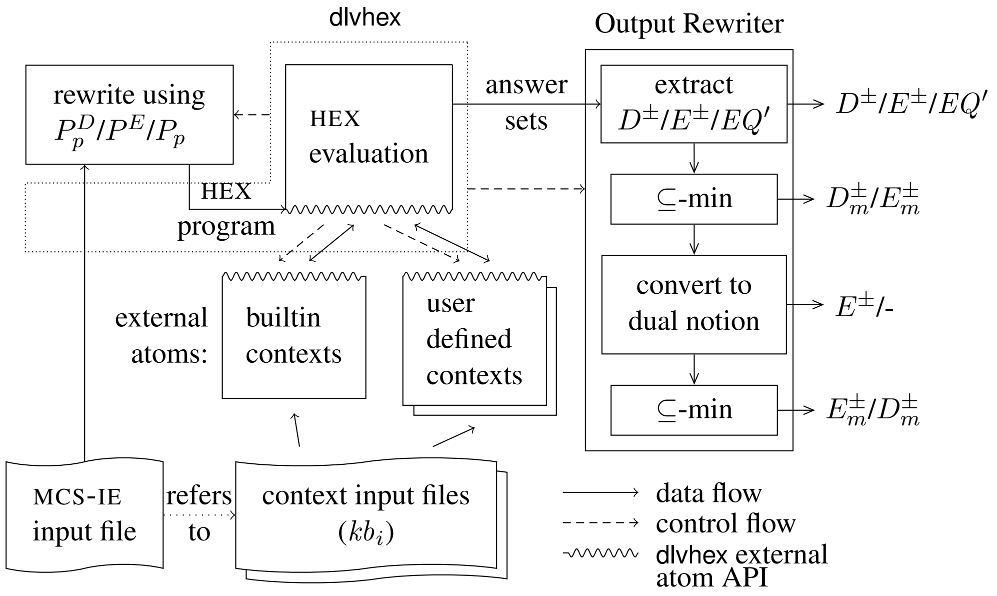


Figure 4.1: Architecture of the MCS-IE system.

master.hex
```
#context(1,"dlv_asp_context_acc", "db.dlv").
#context(2,"dlv_asp_context_acc", "lab.dlv").
#context(3,"ontology_context3_acc", "").
#context(4,"dlv_asp_context_acc", "dss.dlv").

r1: (2:customer02031985) :- (1:person02031985).
r2: (3:pneumonia) :- (2:xray_pneumonia).
r3: (3:marker) :- (2:blood_marker).
r4: (4:need_ab) :- (3:bacterial_disease).
r5: (4:need_strong) :- (3:atyppneumonia).
r6: (4:allow_strong_ab) :- not (1:allergy_strong_ab).
```

db.dlv
```
allergy_strong_ab.
person02031985.
```

lab.dlv
```
customer03021985.
xray_pneumonia.
blood_marker.
:- customer03021985, customer02031985.
```

dss.dlv
```
give_strong v give_weak :- need_ab.
give_strong :- need_strong.
ngive_strong :- need_strong, not allow_strong_ab.
:- give_strong, ngive_strong.
```

Figure 4.2: Examples for MCS topology and knowledge base input files of the MCS-IE tool, encoding our running example (except for C_{onto} which is realized in C++, see Figure 4.3).

there are two distinct possibilities for obtaining the set of explanations, minimal explanations, and minimal diagnoses, while there is only one possibility for obtaining the set of diagnoses and output-projected equilibria.

Contexts can be realized in MCS-IE as ASP programs, or by writing a context reasoning module using a C++ interface which allows for implementing arbitrary formalisms that can be captured by MCS contexts.

Example 37 (ctd). *Figure 4.2 shows files which encode the Medical Example MCS M_2 in the* MCS-IE *input format. Contexts C_{db}, C_{lab}, and C_{dss} are formalized in ASP, with knowledge bases* `db.dlv`, `lab.dlv`, *and* `dss.dlv`, *these contexts are evaluated using the* DLV *solver. On the other hand, ontology reasoning in C_{onto} is implemented as a user-defined* MCS-IE *context in C++: the source code in Figure 4.3 can be compiled to yield a* dlvhex *plugin which realizes semantics of C_{onto}. For more details about the format and the interface we refer to [BEFS10] and to the source code documentation of* MCS-IE. *Inconsistency in our running example can be explained by calling the* MCS-IE *plugin for* dlvhex *with the following command line (assuming the plugin in the current directory):*

```
$ dlvhex2 --plugindir=./ --ieenable --ieexplain=Dm,Em master.hex
```

MCS-IE *calculates the following output, containing minimal diagnoses plus witnessing equilibria (*`Dm:EQ:`*), and minimal inconsistency explanations (*`Em`*):*

```
Dm:EQ:({r1,r5},{}):({allergy_strong_ab,person02031985},{blood_mark
   er,xray_pneumonia},{atyppneumonia,bacterial_disease},{})
Dm:EQ:({r1},{r6}):({allergy_strong_ab,person02031985},{blood_marke
   r,xray_pneumonia},{atyppneumonia,bacterial_disease},{})
Dm:EQ:({r1,r3},{}):({allergy_strong_ab,person02031985},{blood_mark
   er,xray_pneumonia},{bacterial_disease},{})
Dm:EQ:({r1,r2},{}):({allergy_strong_ab,person02031985},{blood_mark
   er,xray_pneumonia},{},{})
Em:({r2,r3,r5},{r6})
Em:({r1},{})
```

□

For researching notions of inconsistency analysis in MCSs without the need of building and installing the software, an online version [MIE12b] of MCS-IE is available. Figure 4.4 shows a screenshot of this web interface.

4.3 Discussion

We discuss scalability then other possibilities for computing diagnosis and explanations.

Scalability

Even though it is not primarily geared towards efficiency, MCS-IE is a useful research tool for the notions introduced in this work. As expected, MCS-IE shows the following behavior wrt. efficiency: rewriting $P_p^D(M)$, which uses guess-and-check, shows better performance than rewriting $P^E(M)$, which expresses the **coNP** task of recognizing explanations in the $\mathbf{\Sigma_2^P}$ formalism of full-fledged disjunctive HEX programs.

However, experiments with MCS-IE showed that even $P_p^D(M)$ does not scale well with reasonably sized systems. We looked for alternative rewritings, and soon found out that our rewriting is not the problem; instead, the HEX evaluation algorithm itself was the main reason for lack of scalability, and we also identified possibilities to improve the algorithm.

```
#include "ContextInterfaceAtom.h"
#include "ContextInterfacePlugin.h"

DLVHEX_MCSEQUILIBRIUM_PLUGIN(7,0,0,MedExamplePluginContext3,0,1,0)

namespace
{
  DLVHEX_MCSEQUILIBRIUM_CONTEXT(Context3,"ontology_context3_acc")

  std::set<std::set<std::string> >
  Context3::acc(
      const std::string& param,
      const std::set<std::string>& input)
  {
    std::set<std::set<std::string> > ret;
    // accept all input
    std::set<std::string> s(input.begin(),input.end());
    if( input.count("pneumonia") == 1
        && input.count("marker") == 1 )
    {
      // additionally accept atyppneumonia
      s.insert("atyppneumonia");
    }
    if( input.count("pneumonia") == 1 )
    {
      // additionally accept bacterial_disease
      s.insert("bacterial_disease");
    }
    ret.insert(s);
    return ret;
  }

  void MedExamplePluginContext3::registerAtoms(
      ProgramCtxData& pcd) const
  {
    registerAtom<Context3>(pcd);
  }
}
```

Figure 4.3: Examples for C++ implementation of context semantics for context C_{onto}. This file can be compiled into a MCS-IE context plugin.

This led to the work of the next chapter, where we develop a better HEX evaluation formalism, which divides and conquers the guessing space more efficiently [EFI$^+$11]. In terms of performance, the evaluation of $P_p^D(M)$ previously scaled exponentially in the total number of output beliefs and bridge rules, while the improved method scales exponentially only in the number of output beliefs and bridge rules of the largest context of M. Formally, the previous method processed $\mathcal{O}(2^{\Sigma_i |OUT_i|+|br_i|})$ guesses, while the improved method needs to process only $\mathcal{O}(max_i(2^{|OUT_i+br_i|}))$ guesses when enumerating all diagnoses in $P_p^D(M)$. Section 5.5.3 on page 104 and Figures 5.9 and 5.10 show the difference between the original performance of dlvhex (and therefore MCS-IE) and the performance of the new evaluation algorithm of dlvhex which was developed due to the insights gained with MCS-IE.

Other Approaches for Realizing Inconsistency Analysis

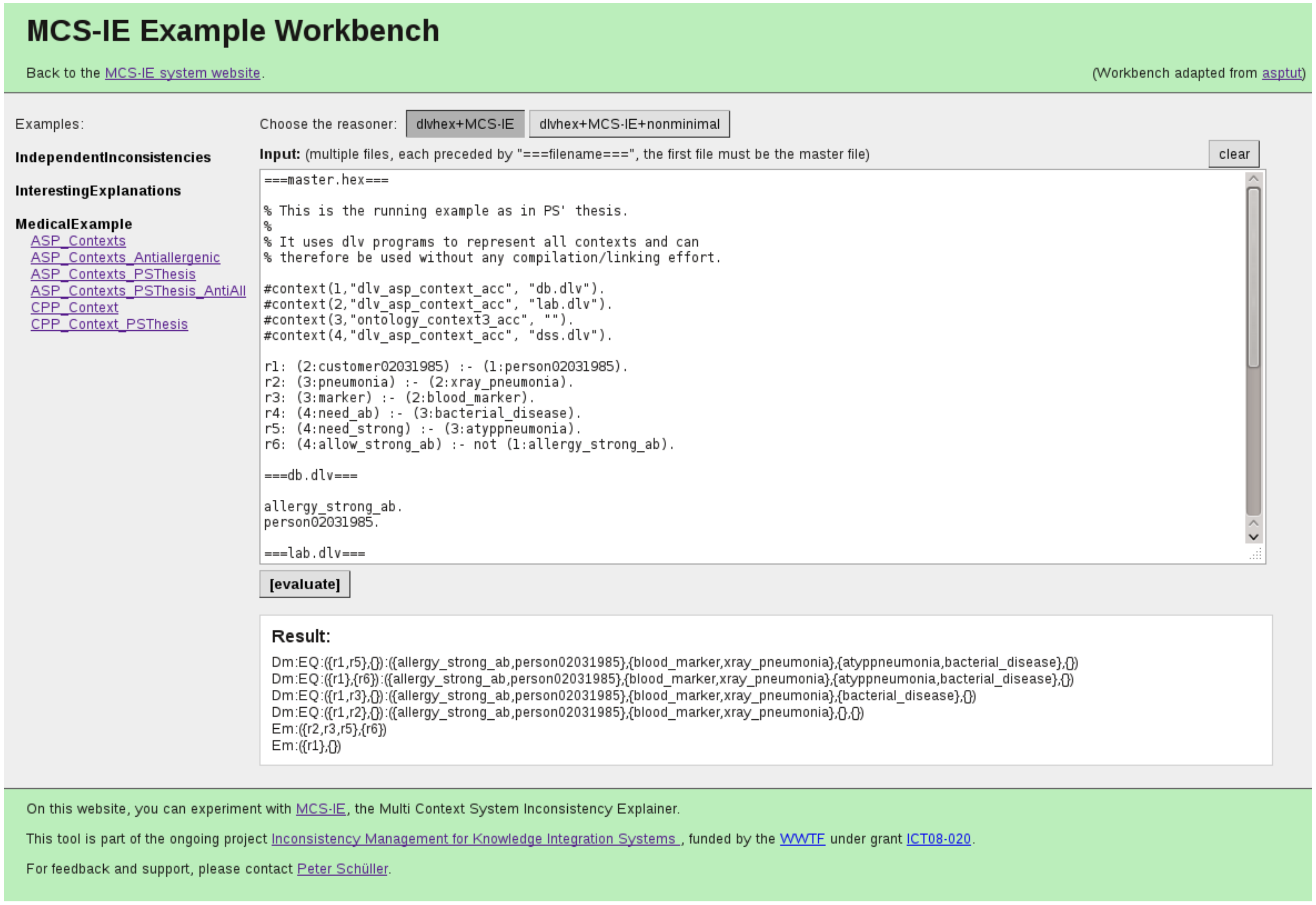
Apart from rewriting the computation of diagnoses and inconsistency explanations to a formalism of computational logic, there exist other possibilities for obtaining these notions.

Most prominently we want to mention the *DMCS algorithm* [DTEFK10, BDTE$^+$10a] and its research prototype [BDTE$^+$10b], which allows to compute equilibria of MCSs in a truly distributed way. The integration of diagnosis computation into the DMCS algorithm, and a corresponding implementation, is currently investigated in a masters thesis project, co-supervised by the author of this thesis.

In the field of model-based/consistency-based diagnosis [Rei87], numerous approaches for computation exist. We here focus on the discussion of distributed approaches for consistency-based diagnosis (these approaches are more closely related to our work as most other approaches in the field of model-based diagnosis). As discussed in Chapter 3, both approaches are only remotely related, as they focus on detecting correct or faulty operation of system parts, based on observations on the system, whereas our approach focuses on finding bridge rules that cause inconsistency (and they cause inconsistency by operating as specified, not by faulty behavior).

An algorithm for *decentralized computation of consistency-based diagnosis* is described in [CPD07]; the algorithm uses a global supervisor module and local diagnoser modules which communicate using a distributed algorithm. This work is motivated similarly as multi-context systems, i.e., the combination of existing components yields an overall system where we want to find problems. Furthermore the authors stress the privacy of each local diagnoser's information about the component it is responsible for, which is a similar argument as we gave for information hiding in Section 3.5.

Another algorithm for distributed consistency-based diagnosis is described in [ADS08]; this approach no longer requires a centralized supervisor and works with a true peer-to-peer architecture, which is, architecturally, very similar to the approach taken in the DMCS algorithm mentioned above. Furthermore, that approach operates on prime implicants of disjunctive normal form representations of the knowledge of each peer, while DMCS operates on partial models and does not require to convert each context knowledge base into a specific formalism.



MCS-IE Example Workbench

Back to the MCS-IE system website.

(Workbench adapted from asptut)

Examples:

IndependentInconsistencies

InterestingExplanations

MedicalExample
- ASP_Contexts
- ASP_Contexts_Antiallergenic
- ASP_Contexts_PSThesis
- ASP_Contexts_PSThesis_AntiAll
- CPP_Context
- CPP_Context_PSThesis

Choose the reasoner: dlvhex+MCS-IE | dlvhex+MCS-IE+nonminimal

Input: (multiple files, each preceded by "===filename===", the first file must be the master file)

```
===master.hex===

% This is the running example as in PS' thesis.
%
% It uses dlv programs to represent all contexts and can
% therefore be used without any compilation/linking effort.

#context(1,"dlv_asp_context_acc", "db.dlv").
#context(2,"dlv_asp_context_acc", "lab.dlv").
#context(3,"ontology_context3_acc", "").
#context(4,"dlv_asp_context_acc", "dss.dlv").

r1: (2:customer02031985) :- (1:person02031985).
r2: (3:pneumonia) :- (2:xray_pneumonia).
r3: (3:marker) :- (2:blood_marker).
r4: (4:need_ab) :- (3:bacterial_disease).
r5: (4:need_strong) :- (3:atyppneumonia).
r6: (4:allow_strong_ab) :- not (1:allergy_strong_ab).

===db.dlv===

allergy_strong_ab.
person02031985.

===lab.dlv===
```

[evaluate]

Result:

```
Dm:EQ:({r1,r5},{}):({allergy_strong_ab,person02031985},{blood_marker,xray_pneumonia},{atyppneumonia,bacterial_disease},{})
Dm:EQ:({r1},{r6}):({allergy_strong_ab,person02031985},{blood_marker,xray_pneumonia},{atyppneumonia,bacterial_disease},{})
Dm:EQ:({r1,r3},{}):({allergy_strong_ab,person02031985},{blood_marker,xray_pneumonia},{bacterial_disease},{})
Dm:EQ:({r1,r2},{}):({allergy_strong_ab,person02031985},{blood_marker,xray_pneumonia},{},{})
Em:({r2,r3,r5},{r6})
Em:({r1},{})
```

On this website, you can experiment with MCS-IE, the Multi Context System Inconsistency Explainer.

This tool is part of the ongoing project Inconsistency Management for Knowledge Integration Systems , funded by the WWTF under grant ICT08-020.

For feedback and support, please contact Peter Schüller.

Figure 4.4: Screenshot of the MCS-IE web interface.

5 Modular Evaluation Framework for HEX-Programs

As we have seen in the previous chapter, the HEX language is a useful tool for representing and reasoning with knowledge that is present in a non-monolithic form, i.e., part of the knowledge is represented as logic programming rules, while other parts are abstracted away as external computations.

While developing the MCS-IE tool in the previous chapter was useful for investigating the notions of diagnoses and explanations, we found out that MCS-IE does not scale with the size of the MCS at hand. We looked into the reason of this scalability problem and tried to solve it by creating alternative HEX rewritings.

However, we soon came to the conclusion that the scalability problem had its roots deep within the way dlvhex evaluates semantics of HEX programs. At that time, dlvhex evaluated HEX programs roughly as follows [EIST06]: the non-ground program is split into subprograms (strongly connected components (SCCs)) with and without external access, where the former are as large and the latter as small as possible. Subprograms with external atoms are evaluated with various specific techniques, depending on their structure [EIST06, Sch06].

The problem we identified was as follows: because the subprograms which are evaluated together are as large as possible, answer sets of such subprograms can contain products of many independent guesses in the program. In real-world applications such as the one described in the previous chapter, these independent guesses can independently cause a global constraint to fire and to invalidates the whole model candidate. Ideally, every program component that can trigger a global constraint is evaluated independently from other program components, and if a guess triggers a global constraint then no more (pointless) computation that depends on such a guess is performed.

Therefore the HEX evaluation approach had severe scalability limitations compared to an ideal behavior, and we decided to develop a new approach for evaluating HEX programs which is described in this chapter.

We make the following theoretical and practical contributions to the HEX formalism, which we published in [EFI^{+}11] in slightly different form and in less detail than presented here.

- We present a new notion of dependencies between rules of a HEX program and use this notion to reformulate the original HEX splitting theorem, and to formulate a generalization of the splitting theorem. Our *Generalized* HEX *Splitting Theorem* allows a new decomposition approach where program parts may overlap by sharing constraints which can prune away irrelevant partial answer set candidates earlier than in previous approaches.

- We present a novel evaluation framework for HEX-programs, which allows for more flexible decomposition of the nonground program than was previously possible. The new framework comprises an *evaluation graph*, which captures a modular decomposition and has a tight correspondence with the Generalized Splitting Theorem. This graph allows us to realize customized divide-and-conquer evaluation strategies which are a generalization of the former HEX decomposition and evaluation approach.

- Based on the evaluation graph, we describe an *answer set graph* which comprises for each node sets of partial input interpretations that are evaluated with the program of the node, and sets of output interpretations that are passed on to subsequent nodes where they are combined into new input interpretations. We show that a complete answer set graph can easily be transformed into the set of all answer sets of the program that gave rise to the evaluation graph. Furthermore we describe an algorithm for building a complete answer set graph, given an evaluation graph of a HEX program.

- We describe a prototype of the evaluation framework in the dlvhex solver engine. This implementation is generic and can be instantiated with different ASP solvers (in our case, with DLVand clasp + gringo). It features also *model streaming*, i.e., computing one answer set at a time.

- We present results of an experimental evaluation which shows that our new framework considerably reduces memory consumption and avoids timeouts in a larger number of settings compared to the previous approach. In some cases, the new approach uses exponentially less memory (which can be explained by the difference in evaluation strategies). In particular, the MCS-IE tool described in the previous chapter can now feasibly be used on considerably larger instances.

5.1 Preliminaries

In the following we repeat several important concepts connected to HEX programs. For some notions we adjust existing definitions to make them more precise, or more suitable for our subsequent formal work. In particular we revise definitions of atom dependencies and safety restrictions.

5.1.1 Restriction to Extensional Semantics for HEX External Atoms

To make HEX programs computable in practice, it is useful to restrict external atoms, such that their semantics depends only on extensions of predicates given in the input tuple [EIST06]. This restriction is relevant for all subsequent considerations.

Syntax Each *&g* is associated with an input type signature $t_1, \dots, t_n$ such that every t_i is the type of input Y_i at position i in the input list of *&g*. A *type* is either $\mathbf{const}$ or a non-negative integer.

Consider *&g*, its type signature $t_1, \dots, t_n$, and a ground external atom $\&g[y_1, \dots, y_n](x_1, \dots, x_m)$. Then, in this setting, the signature of *&g* enforces certain constraints on $f_{\&g}(I, y_1, \dots, y_n, x_1, \dots, x_m)$ such that its truth value depends only on (a) the constant value of y_i whenever $t_i = \mathbf{const}$, and (b) the extension of predicate y_i, of arity t_i, in I whenever $t_i \in \mathbb{N}$.

Example 38 (ctd)**.** *Continuing Example 7, for* $\&reach[edge, a](x)$*, we have* $t_1 = 2$ *and* $t_2 = \mathbf{const}$*. Therefore the truth value of* $\&reach[edge, a](x)$ *depends on the extension of binary predicate* $edge$*, on the constant* a*, and on* x*.*

Continuing Example 10, the external predicate $\&rq$ *has* $t_1 = 1$*, therefore the truth value of* $\&rq[swim](x)$ *for various* x *wrt. an interpretation* I *depends on the extension of the unary predicate* $swim$ *in the input list.* ☐

Note that the truth value of an external atom with only constant input terms, i.e., $t_i = \mathbf{const}$, $1 \leq i \leq n$, does not depend on I at all.

Semantic constraints enforced by signatures are formalized next.

Semantics Let a be a type, let I be an interpretation and $p \in \mathcal{C}$. We define the *projection function* Π_a as the following binary function: for $a = \mathbf{const}$, $\Pi_{\mathbf{const}}(I, p) = p$; for $a \in \mathbb{N}$, $\Pi_a(I, p) = \{(x_1, \ldots, x_a) \mid p(x_1, \ldots, x_a) \in I\}$. For $\mathcal{C}^a$ the a-th cartesian power of $\mathcal{C}$, let D_a be the family of sets of tuples with arity $a+1$, i.e., $D_a = 2^{\mathcal{C}^{a+1}}$ (we use this for the family of sets of atoms with arity a). As a special case, we conventionally set: $D_{\mathbf{const}} = \mathcal{C}$. (Note that, e.g., atom $p(u, v)$ is of arity 2 and therefore its tuple representation (p, u, v) is of arity 3. Hence the powerset of all possible atoms of arity 2 is $D_2 = 2^{\mathcal{C}^3}$.)

Let *&g* be an external predicate with oracle function $f_{\&g}$, $in(\&g) = n$, $out(\&g) = m$, and type signature $t_1, \ldots, t_n$, then the *extensional evaluation function* function $F_{\&g} : D_{t_1} \times \cdots \times D_{t_n} \to 2^{\mathcal{C}^m}$ of *&g* is defined such that

$$(a_1, \ldots, a_m) \in F_{\&g}(\Pi_{t_1}(I, p_1), \ldots, \Pi_{t_n}(I, p_n)) \text{ iff } f_{\&g}(I, p_1, \ldots, p_n, a_1, \ldots, a_m) = 1.$$

Note that $F_{\&g}$ makes the possibility of new constants invented by external atoms more explicit: tuples returned by $F_{\&g}$ may contain constants that are not contained in P.

Example 39 (ctd)**.** *Consider I from Example 11, then we have*

$$\Pi_1(I, swim) = \{(swim, out)\} \textit{ and}$$
$$\Pi_1(I, goto) = \{(goto, ndanube)\}.$$

The extensional evaluation function of &rq then is as follows:

$$\begin{aligned} F_{\&rq}(U) = &\{(money) \quad \mid (X, in) \in U \textit{ or } (X, gdanube) \in U\} \cup \\ &\{(yogamat) \mid (X, ndanube) \in U\} \cup \\ &\{(goggles) \quad \mid (X, apool) \in U\} \end{aligned}$$

Observe that neither of the constants yogamat and goggles is contained in P (we have that $const(P) = \{swim, goto, ngoto, need, go, inout, loc, in, out, apool, gdanube, ndanube, mpool, money, location\}$). The constants yogamat and goggles are invented by external atom semantics. Note that $(money)$ is a unary tuple, as &rq has a unary output list. □

5.1.2 Atom Dependencies

To account for dependencies between heads and bodies of rules is a common approach for realizing semantics of ordinary logic programs, as done, e.g., by means of the notions of *stratification* and its refinements like *local* [Prz88] or *modular stratification* [Ros94], or by *splitting sets* [LT94]. In HEX programs, head-body dependencies are not the only possible source of predicate interaction. Therefore new types of (nonground) dependencies were considered in [EIST06, Sch06]. In the following we recall these definitions. We also slightly reformulate and extend them, to prepare for the following sections where we lift atom dependencies to rule dependencies.

In contrast to the traditional notion of dependency that in essence hinges on propositional programs, we need to consider relationships between non-ground atoms. Two nonground atoms a and b may inter-depend when they unify, which we denote by $a \sim b$.

For analyzing program properties it is relevant whether a dependency is positive or negative. With an external atom a, it depends on the semantic evaluation function whether the truth value of the external evaluation function $f_{\&a}$ depends on the presence or absence of an atom b in interpretation I. Depending on other atoms in I, in some cases the *presence* of b might make a true, in some cases its *absence*. Therefore we will in the following not speak of *positive* and *negative* dependencies (as in [EFI$^+$11]); instead we use the more adequate wording of *monotonic* and *nonmonotonic* dependencies.[1]

[1]Note that antimonotonicity (i.e., a larger input of an external atom can only make the external atom false, but never true) could be a third useful distinction which has been exploited in [EFKR12]. We here only require the distinction between monotonic vs. nonmonotonic external atoms and therefore classify antimonotonic external atoms as nonmonotonic.

Definition 19. *An external predicate &g is called* monotonic *iff for all interpretations I, I', and all tuples of constants $\vec{X}$, $f_{\&g}(I, \vec{X}) = 1$ and $I \subseteq I'$ implies $f_{\&g}(I', \vec{X}) = 1$, otherwise it is called* nonmonotonic.

Example 40 (ctd). *Consider $F_{\&rq}(U)$ in Example 39: adding tuples to U cannot remove tuples from $F_{\&rq}(U)$, therefore &rq is a monotonic external predicate.* □

Next we define relations for dependencies from external atoms to other atoms.

Definition 20 (External Atom Dependencies). *Let P be a* HEX *program, let a be an external atom of the form $\&g[X_1, \dots, X_n](\vec{Y})$ in P with the type signature $t_1, \dots, t_n$ and let b be an atom in the head of a rule in P such that b is of the form $p(\vec{Z})$ or b is a higher order atom of the form $U(\vec{Z})$. Then a* depends external monotonically (resp., nonmonotonically) on b, *formally $a \rightarrow^e_m b$ (resp., $a \rightarrow^e_{nm} b$) iff &g is monotonic (resp., nonmonotonic), $t_i \in \mathbb{N}$, $\vec{Z}$ has arity t_i, and either (i) b is of form $p(\vec{Z})$ and $X_i = p$; or (ii) b is of form $U(\vec{Z})$. We define $a \rightarrow^e b$ iff $a \rightarrow^e_m b$ or $a \rightarrow^e_{nm} b$.*

Example 41 (ctd). *In our running example, we have external dependencies $\&rq[swim](C) \rightarrow^e_m swim(in)$, $\&rq[swim](C) \rightarrow^e_m swim(out)$, and $\&rq[goto](C) \rightarrow^e_m goto(X)$.* □

As in ordinary ASP, atoms in HEX programs are interdependent because of rules in the program.

Definition 21. *For a* HEX*-program P and atoms α, β occurring in P, we say that*

(a) α depends monotonically on β ($\alpha \rightarrow_m \beta$), if one of the following holds:

(i) some rule $r \in P$ has $\alpha \in H(r)$ and $\beta \in B^+(r)$;
(ii) there are rules $r_1, r_2 \in P$ such that $\alpha \in B(r_1)$ and $\beta \in H(r_2)$ and $\alpha \sim \beta$; or
(iii) some rule $r \in P$ has $\alpha \in H(r)$ and $\beta \in H(r)$.

(b) α depends nonmonotonically on β ($\alpha \rightarrow_n \beta$), if there is some rule $r \in P$ such that $\alpha \in H(r)$ and $\beta \in B^-(r)$.

Note that combinations of Definitions 20 and 21 were already introduced in [Sch06,EFK09], however these works represent nonmonotonicity of external atoms in rule body dependencies and use a single ‘external dependency’ relation. On the contrary, we represent nonmonotonicity of external atoms where it really happens, namely in dependencies from external atoms to ordinary atoms. Therefore we obtain a simpler dependency relation between rule bodies and heads.

We say that atom α depends on atom β ($\alpha \rightarrow \beta$) where $\rightarrow$ is the union of $\rightarrow_m$, $\rightarrow_n$, and $\rightarrow^e$.

We next define the atom dependency graph.

Definition 22. *For a* HEX*-program P, the* atom dependency graph *$ADG(P) = (V_A, E_A)$ of P has as vertices V_A the (nonground) atoms occurring in non-facts of P and as edges E_A the dependency relations $\rightarrow_m$, $\rightarrow_n$, $\rightarrow^e_m$, and $\rightarrow^e_{nm}$ between these atoms in P.*

Example 42 (ctd). *Figure 5.1 depicts the atom dependency graph of P_{swim}. Note that the nonmonotonic body literal in c_7 does not show up as a nonmonotonic dependency, as c_7 has no head atoms. (The rule dependency graph we introduce later in Section 5.2 will make this negation apparent.)* □

Next we use the dependency notions to define safety conditions on HEX programs.

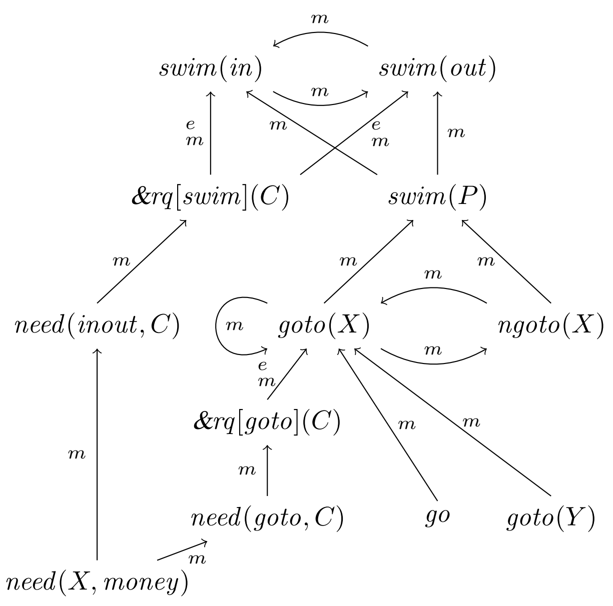


Figure 5.1: Atom dependency graph of running example P_{swim}.

5.1.3 Rule Unfolding for External Atom Input Grounding

As in previous work about HEX evaluation, we perform a process reminiscent of rule unfolding whenever an external atom input tuple contains a variable that is contained in the output of another external atoms in the same rule body. This is necessary for grounding external atom inputs. We here briefly repeat the definition and give an example.

Definition 23. *(Definition 4.6.11 from [Sch06].) Let P be a* HEX*-program and let $\&g[\vec{Y}](\vec{X})$ be some external atom with input list $\vec{Y}$ occuring in a rule $r \in P$. Then, for each such atom, a rule $r^{\&g}_{inp}$ is composed as follows:*

- *The head $H(r^{\&g}_{inp})$ contains an atom $g_{inp}(\vec{Y})$ with a fresh predicate symbol g_{inp}.*
- *The body $B(r^{\&g}_{inp})$ of the auxiliary rule contains all body literals of r other than $\&g[\vec{Y}](\vec{X})$ that have at least one variable in its arguments (resp., in its output list if b is another external atom) that occurs also in $\vec{Y}$.*

For each external atom in P we can create such a rule. We denote the set of all such rules with P_{inp}.

Example 43. *Consider the* HEX *program P consisting of one rule*

$$foo(X) \leftarrow \&concat[a, Y](X), \&concat[b, c](Y)$$

where $\&concat$ performs string concatenation of the input arguments to the output. Intuitively $\mathcal{AS}(P) = \{\{foo(abc)\}\}$. The following auxiliary program P_{inp} is created for evaluating P:

$$g^{\&concat}_{inp}(Y) \leftarrow \&concat[b, c](Y).$$

Intuitively, for evaluating P we first evaluate P_{inp}, use the answer set $\{g^{\&concat}_{inp}(bc)\}$ of P_{inp} for grounding P, and then evaluate P. □

Note that the process of creating auxiliary rules sometimes must be iterated, i.e., we need to create auxiliary rules for auxiliary rules. For example consider the program

$$foo(X) \leftarrow \&concat[a, Y](X), \&concat[b, Z](Y), \&concat[c, d](Z)$$

which intuitively yields the single answer set $\{foo(abcd)\}$ and requires two iterations of rule unfolding for grounding external atom inputs.

5.1.4 Safety Restrictions

With HEX we need the usual notion of rule safety, i.e., a syntactic restriction which ensures that each variable in a rule only has a finite set of relevant constants for grounding. As external computations can introduce new constants in their output lists, ensuring safety in HEX is not as straightforward as in ordinary ASP.

We first recall the definition of safe variables and a safe rule for HEX.

Definition 24 (Def. 5 in [EIST06]). *Given a rule r, the set of* safe variables *in r is the smallest set X of variables such that*

(i) X appears in a positive ordinary atom in the body of r, or

(ii) X appears in the output list of an external atom $\&g[Y_1, \dots, Y_n](X_1, \dots, X_m)$ in the positive body of r and $Y_1, \dots, Y_n$ are safe.

A rule r is safe *iff each variable in r is safe.*[2]

However, safety alone does not guarantee finite grounding of HEX programs, because an external atom might create new constants, i.e., constants not part of the program itself (see Example 39), in its output list. These constants can become part of the extension of an atom in the rule head, and by grounding and evaluation of semantics of other rules become part of the extension of a predicate which is an input to the very same external atom.

Example 44 (adapted from [Sch06]). *The following* HEX *program is safe according to Definition 24 and nevertheless cannot be finitely grounded:*

$$\begin{aligned} source(\text{"}\texttt{http://some\_url}\text{"}) &\leftarrow . \\ url(X) &\leftarrow \&rdf[source](X, \text{"}rdf{:}subClassOf\text{"}, C). \\ source(X) &\leftarrow url(X). \end{aligned}$$

Suppose the $\&rdf[source](S, P, O)$ atom retrieves all triples (S, P, O) from all RDF triplestores specified in the extension of source, and suppose that every triplestore contains a triple with S being a URL that is not contained in another triplestore. As a result, all these URLs are collected in the extension of source which leads to even more URLs being retrieved and a potentially infinite grounding.

However, we could change the rule with the external atom to

$$url(X) \leftarrow \&rdf[source](X, \text{"}rdf{:}subClassOf\text{"}, C), limit(X) \tag{5.1}$$

and add an appropriate set of limit facts. This addition of a range predicate $limit(X)$ which does not depend on the external atom output ensures a finite grounding. □

To obtain a syntactic restriction that ensures finite grounding for HEX, a stronger notion of safety called *strong safety* has been proposed. We next give a definition which is different from existing ones [EIST06, Sch06] as those definitions have limitations.[3]

[2]This is stated in [EIST06] as 'if each variable appearing in a negated atom and in any input list is safe, and variables appearing in $H(r)$ are safe'. However, if all variables in $H(r)$ are safe, and all variables in all input lists are safe, and all variables in negated body atoms are safe, then all variables in r are safe because then atoms in all positive body atoms are safe. Therefore we can simplify the definition.

[3]The definition in [EIST06, Def. 7] is too strict because it does not allow variables in output lists of external atoms which are not contained in positive ordinary atoms. This rules out cases like $command(X) \leftarrow$

Definition 25. *An external atom b in a rule r in a* HEX *program P is* strongly safe *wrt. r and P iff*

(a) $b \not\to^{+} b$, i.e., there is no cyclic dependency over b;[4] *or*

(b) for each variable X in the output list of b, there exists a positive ordinary atom $a \in B^{+}(r)$ containing X such that $a \not\to^{+} b$, i.e., a does not depend on b.

Strong safety of a HEX external atom is a non-local property which is defined wrt. a program. Two rules that are strongly safe in isolation might not be strongly safe if they are put into one program P. Therefore it can be misleading to define the strong safety property for a rule without mentioning P (as done in [EIST06, Sch06]) and we here define strong safety in terms of the safety of an external atom wrt. a rule in a program.

Using this definition we now obtain the following notion of domain-expansion safety, which characterizes a class of HEX programs that has a finite grounding.

Definition 26. *A* HEX *program P is* domain-expansion safe *if each rule in P is safe, and each external atom in each rule $r \in P$ is strongly safe wrt. r and P.*

In the following we consider only domain-expansion safe HEX programs. Note that every *ordinary* HEX program that is safe is also domain-expansion safe.

Example 45 (ctd). *Our running example P_{swim} is domain-expansion safe as every rule is safe and no external atom is contained in a cyclic dependency to itself (see Figure 5.1). Therefore condition* (a) *in Definition 25 is satisfied for all external atoms.* □

Because we changed the definition of domain-expansion safety, we now prove that every domain-expansion safe program indeed has a finite grounding that is sufficient for evaluating its semantics. We denote by $grnd_X(P)$ the grounding of HEX program P with ground atoms X.

Proposition 11. *(Adapted from [Sch06, Theorem 4.6.1].) For any domain-expansion safe* HEX*-program P, there exists a finite set $D \subseteq \mathcal{C}$ such that $\mathcal{AS}(grnd_D(P)) = \mathcal{AS}(grnd_{\mathcal{C}}(P))$.*

Proof. (Adapted from the proof of Theorem 4.6.1 in [Sch06].) A program that incrementally builds D can be sketched as follows: we repeatedly update a set A of *active* atoms by means of a function $ins(r, A)$ which is repeatedly invoked over all rules $r \in P$. We start from the set A of ground atoms present in the program P, and D is the least fixpoint of this iteration, i.e., the least fixpoint where $A = ins(r, A)$. The function $ins(r, A)$ is such that, given a safe rule r and a set A of ground atoms, it returns a set that contains A and the following atoms: (a) all ordinary ground atoms that are created by grounding r with A, and (b) all ground external atoms that are created by grounding r with A, and (c) for each nonground external atom $\&g[\vec{Y}](\vec{Z})$ that obtains a ground input tuple $\vec{Y} = (y_1, \ldots, y_n)$ by grounding r with A, the set of ground external atoms

$$\{\&g[\vec{Y}](\vec{X}) \mid \vec{X} \in F_{\&g}(\Pi_{t_1}(J, y_1), \ldots, \Pi_{t_n}(J, y_1)) \text{ for some } J \subseteq A\}.$$

Iterating $ins(r, A)$ yields a finite fixpoint for A, because

(i) a ground external atoms a yielded by (c) cannot cyclically cause more atoms to be added by (c), as each a either is not part of a cyclic dependency over itself (condition (a) in Definition 25) or the range of output variable grounding of a is limited by an ordinary atom that does not depend on a (condition (b) in Definition 25) which means that a will not be processed by step (c) of $ins(r, A)$ but by step (b); furthermore

$\&concat[\text{" cat "}, Filename](X), showFile(Filename)$, where $\&concat$ has only constant inputs and therefore the rule can be finitely grounded as long as $showFile(X)$ does not depend on $command(X)$. The definition in [Sch06, Def. 4.6.7] is more strict in another way: it does not detect cases as (5.1) as finitely groundable because it rules out any cycles over external atoms, i.e., it is not possible to ‘save’ the cycle over $url(X)$ by adding a range predicate $limit(X)$ as shown in (5.1). The implementation in dlvhex is less strict than both definitions.

[4]This implies that b does not depend on any head atom of r.

(ii) steps (a) and (b) yield a finite amount of new atoms for each atom generated by step (c) (as we do not have function symbols, (a) and (b) can be reduced to ordinary ASP grounding which is finite).

We have shown that $D = A = \mathit{ins}(r, A)$ is a finite fixpoint that exists. This fixpoint provides a finite set of atoms that can be used for grounding P, hence it remains to show that indeed $\mathcal{AS}(\mathit{grnd}_D(P)) = \mathcal{AS}(\mathit{grnd}_{\mathcal{C}}(P))$. We split $\mathit{grnd}_{\mathcal{C}}(P)$ into $N_1 = \mathit{grnd}_D(P)$ and $N_2 = \mathit{grnd}_{\mathcal{C}}(P) \setminus \mathit{grnd}_D(P)$. As the above fixpoint iteration terminated, N_2 contains no rules that depend on N_1 and vice versa N_1 contains no rules that depend on N_2. Therefore $\mathcal{AS}(P)$ is the cross product of answer sets $\mathcal{AS}(N_1)$ and $\mathcal{AS}(N_2)$. As every rule in P is safe, no rule in N_2 can be applicable due to an atom within N_2, as N_2 does not depend on N_1, no rule in N_2 can be applicable and $\mathcal{AS}(N_2) = \{\emptyset\}$. Therefore $\mathcal{AS}(P) = \mathcal{AS}(N_1) = \mathcal{AS}(\mathit{grnd}_D(P))$ and the result follows. $\square$

5.2 Rule Dependencies and Generalized Rule Splitting Theorem

So far we just recalled and slightly adapted existing definitions. Next we introduce a new notion of dependencies in HEX programs, namely between *rules* in a program. Then we lift the existing HEX splitting theorem [EIST06, Global Splitting Theorem] to this new dependency notion, and we will generalize and improve it. This will help us to obtain a more efficient evaluation algorithm in Section 5.4.

The former HEX evaluation algorithm [EIST06] is based on the atom dependency graph consisting of non-ground atoms and dependencies; based on this graph, gradual evaluation is carried out on appropriate selections of sets of rules (the 'bottoms' of a program). In contrast with that, we consider dependencies between rules of the program at hand. This simplifies the view on dependencies and corresponding theoretical results considerably.

5.2.1 Rule Dependencies

We define rule dependencies as follows.

Definition 27 (Rule dependencies)**.** *Let P be a program with rules $r, s \in P$, $r \neq s$, and a, b atoms. Then r depends on s according to the following cases:*
(i) if $a \in B^+(r)$ and $b \in H(s)$, and $a \sim b$: then $r \rightarrow_m s$;
(ii) if $a \in B^-(r)$, $b \in H(s)$, and $a \sim b$: then $r \rightarrow_n s$;
(iii) if $a \in H(r)$, $b \in H(s)$, and $a \sim b$: then both $r \rightarrow_m s$ and $s \rightarrow_m r$;
(iv) if $a \in B(r)$ is an external atom, $b \in H(s)$, and $a \rightarrow^e b$, then
- *$r \rightarrow_m s$ if $a \in B^+(r)$ and $a \rightarrow^e_m b$, and*
- *$r \rightarrow_n s$ otherwise.*

Intuitively, conditions (i) and (ii) reflect the fact that the applicability of a rule r depends on the applicability of a rule s with a head that unifies with a literal in the body of rule r; condition (iii) exists because rules r and s cannot be evaluated independently if they share a common head atom (e.g., $u \vee v \leftarrow$ cannot be evaluated independently from $v \vee w \leftarrow$); and (iv) defines dependencies due to predicate inputs of external atoms.

In the following we denote by $\rightarrow_{m,n} = \rightarrow_m \cup \rightarrow_n$ the union of monotonic and nonmonotonic rule dependencies.

We next define graphs of rule dependencies.

Definition 28. *Given a HEX-program P, the* rule dependency graph *$DG(P) = (V_D, E_D)$ of P is the labeled graph with vertex set $V_D = P$ and edge set $E_D = \rightarrow_{m,n}$.*

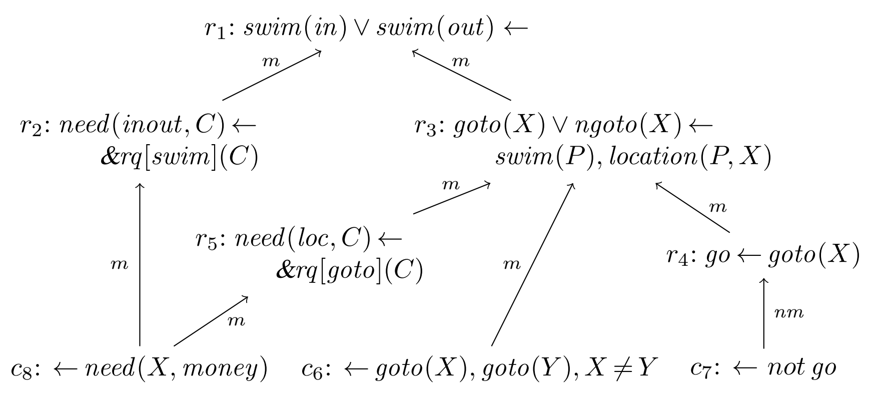

r_1: $swim(in) \lor swim(out) \leftarrow$
r_2: $need(inout, C) \leftarrow \&rq[swim](C)$
r_3: $goto(X) \lor ngoto(X) \leftarrow swim(P), location(P, X)$
r_5: $need(loc, C) \leftarrow \&rq[goto](C)$
r_4: $go \leftarrow goto(X)$
c_8: $\leftarrow need(X, money)$
c_6: $\leftarrow goto(X), goto(Y), X \neq Y$
c_7: $\leftarrow not\ go$
m
nm

Figure 5.2: Rule dependency graph of running example P_{swim}.

Example 46 (ctd.). *Figure 5.2 depicts the rule dependency graph of our running example. According to Definition 27 we have the following rule dependencies in P^{IDB}_{swim}:*

- *due to (i) we have $r_3 \rightarrow_m r_1$, $r_4 \rightarrow_m r_3$, $c_6 \rightarrow_m r_3$, $c_8 \rightarrow_m r_2$, and $c_8 \rightarrow_m r_5$;*
- *due to (ii) we have $c_7 \rightarrow_n r_4$;*
- *due to (iii) we have no dependencies; and*
- *due to (iv) we have $r_2 \rightarrow_m r_1$ and $r_5 \rightarrow_m r_3$.*

Note that &rq is monotonic (see Example 40). □

5.2.2 Splitting Sets and Theorems

Splitting sets are a notion that allows for describing how a program can be decomposed into parts and how semantics of the overall program can be obtained from semantics of these parts in a divide-and-conquer manner.

We lift the original HEX splitting theorem [EIST06, Theorem 2] and the according definitions of global splitting set, global bottom, and global residual [EIST06, Definitions 8 and 9] to our new definition of dependencies among rules.

A *rule splitting set* is a subset of the original program that does not depend on the rest of the program. This has a correspondence with global splitting sets in [EIST06].

Definition 29 (Rule Splitting Set). *A* rule splitting set *R for a* HEX*-program P is a set $R \subseteq P$ of rules such that whenever $r \in R$, $s \in P$, and $r \rightarrow_{m,n} s$ then $s \in R$.*

Example 47 (ctd). *The following are some rule splitting sets of P_{swim}: $\{r_1\}$, $\{r_1, r_2\}$, $\{r_1, r_3\}$, $\{r_1, r_2, r_3\}$, $\{r_1, r_2, r_3, r_5, c_8\}$. The set $\{r_1, r_2, c_8\}$ is not a rule splitting set, because $c_8 \rightarrow_m r_5$ but r_5 is not part of the set.* □

In the HEX evaluation algorithm described in [EIST06], a constraint can only kill models once all its dependencies to rules (that might make the constraint body become applicable) are fulfilled. Our framework increases evaluation efficiency by duplicating nonground constraints, allowing them to kill models earlier than possible in the former approach. A constraint can only be shared among units, if all its nonmonotonic dependencies are fulfilled, otherwise the constraint could kill partial models too early.

Because of constraint duplication, we no longer partition the input program, and the customary notion of splitting set, bottom, and residual, is not appropriate for sharing constraints between bottom and residual. Instead, we next define a *generalized bottom* of a program, which splits a program into two parts with a nonempty intersection that may contain certain constraints.

Definition 30 (Generalized Bottom). *Given a rule splitting set R of a* HEX*-program P, a* generalized bottom *B of P wrt. R is a set B with $R \subseteq B \subseteq P$ such that all rules in $B \setminus R$ are constraints that do not depend nonmonotonically on any rule in $P \setminus B$.*

Example 48 (ctd)**.** *A rule splitting set R of P_{swim} (e.g., those given in Example 47) is also a generalized bottom of P_{swim} wrt. R. The set $\{r_1, r_2, c_8\}$ is not a rule splitting set, but it is a generalized bottom of P_{swim} wrt. rule splitting set $\{r_1, r_2\}$, as c_8 is a constraint that depends only monotonically on rules in $P_{swim} \setminus \{r_1, r_2, c_8\}$.* □

Next, we describe how interpretations of a generalized bottom B of a program P lead to interpretations of P without re-evaluating rules in B. This is a generalization of the Splitting Set Theorem [LT94], of a modularity result for disjunctive logic programs [EGM97, Lemma 5.1] and of the splitting theorem for (nonground) HEX-programs in [Sch06, Theorem 4.6.2] and [EIST06, Global Splitting Theorem].

Intuitively, this is a relaxation of the previous nonground HEX splitting theorem regarding constraints: a constraint may be put both into the bottom and into the residual if it has no nonmonotonic dependencies to the residual. The benefit of sharing such constraints between bottom and residual is a reduced number of answer sets of the bottom and therefore fewer evaluations of the residual program.

Given a set of ground ordinary atoms I, we denote by $facts(I)$ the corresponding set of ground facts. Given a set of rules P, we denote by $gh(P)$ the set of ground head atoms appearing in $grnd(P)$.

Theorem 10 (Splitting Theorem)**.** *Given a* HEX*-program P and a rule splitting set R of P, $M \in \mathcal{AS}(P)$ iff $M \in \mathcal{AS}(P \setminus R \cup facts(X))$ with $X \in \mathcal{AS}(R)$.*

Proof. Given a set of ground atoms M and a set of rules R we denote by $M|_R = M \cap gh(R)$ the projection of M to ground heads of rules in R.

($\Rightarrow$) Let $M \in \mathcal{AS}(P)$. We first show that $M|_R \in \mathcal{AS}(R)$ and then that $M \in \mathcal{AS}(P \setminus R \cup facts(M|_R))$.

We first show that $M|_R$ satisfies the reduct $fR^{M|_R}$, and then that it is indeed a minimal model of the reduct. M satisfies fP^M and $R \subseteq P$. Observe that, by definition of FLP reduct, $fR^M \subseteq fP^M$. By definition of rule splitting set, satisfiability of rules in R does not depend on heads of rules in $P \setminus R$ (due to the restriction of external atoms to extensional semantics, this is in particular true for external atoms in rules in R). Therefore $fR^{M|_R} = fR^M$, M satisfies $fR^{M|_R}$, and $M|_R$ satisfies $fR^{M|_R}$. For showing $M|_R \in \mathcal{AS}(R)$ it remains to show that $M|_R$ is a minimal model of $fR^{M|_R}$.

Assume towards a contradiction that some $S \subset M|_R$ is a model of $fR^{M|_R}$. Then there is a nonempty set $A = M|_R \setminus S$ of atoms with $A \subseteq gh(R)$. Let $M^\star = M \setminus A$. We next show that $M^\star$ is a model of fP^M, which implies that $M \notin \mathcal{AS}(P)$. Assume on the contrary that $M^\star$ is not a model of fP^M. Hence there exists some rule $r \in fP^M$ such that $H(r) \cap M^\star = \emptyset$, $B^+(r) \subseteq M^\star$, and $B^-(r) \cap M^\star = \emptyset$. S agrees with $M^\star$ on atoms from $gh(R)$, and S satisfies $fR^{M|_R}$. Therefore $r \notin fR^{M|_R}$ and $r \in f(P \setminus R)^M$. Since $r \in P \setminus R$, $H(r) \subseteq gh(P \setminus R)$, and because M and $M^\star$ agree on atoms from $gh(P \setminus R)$, $H(r) \cap M^\star = \emptyset$ from above implies that $H(r) \cap M = \emptyset$. Because $r \in fP^M$, its body is satisfied in M, and since its head has no intersection with M, we get that fP^M is not satisfied by M which is a contradiction. Therefore $M^\star$ is a model of fP^M. As $M^\star \subset M$, this contradicts our assumption that $M \in \mathcal{AS}(P)$. Therefore $S = M|_R = X$ is a minimal model of fR^M.

We next show that M satisfies the reduct $f(P \setminus R \cup facts(M|_R))^M$, and then that it is indeed a minimal model of the reduct. By definition of reduct, $f(P \setminus R \cup facts(M|_R))^M = f(P \setminus R)^M \cup facts(M|_R)$. M satisfies $facts(M|_R)$ because $M|_R \subseteq M$. Furthermore $f(P \setminus R)^M \subseteq fP^M$, hence M satisfies $f(P \setminus R)^M$. Therefore M satisfies $f(P \setminus R \cup facts(M|_R))^M$.

To show that M is a minimal model of $f(P \setminus R \cup facts(M|_R))^M$, assume towards a contradiction that some $S \subset M$ is a model of $f(P \setminus R \cup facts(M|_R))^M$. Since $facts(M|_R)$ is part of the reduct, $M|_R \subseteq S$, therefore $S|_{gh(R)} = M|_R$. By definition of rule splitting set, satisfiability of rules in R does not depend on heads of rules in $P \setminus R$, hence S satisfies fR^M. Because S

satisfies $f(P \setminus R \cup \mathit{facts}(M|_R))^M = f(P \setminus R)^M \cup \mathit{facts}(M|_R)$, it also satisfies $f(P \setminus R)^M$. Since S satisfies both fR^M, S satisfies $fP^M = f(P \setminus R)^M \cup fR^M$. This is a contradiction to $M \in \mathcal{AS}(P)$. Therefore $S = M$ is a minimal model of $f(P \setminus R \cup \mathit{facts}(M|_R))^M$.

($\Leftarrow$) Let $M \in \mathcal{AS}(P \setminus R \cup \mathit{facts}(X))$ and let $X \in \mathcal{AS}(R)$. We first show that M satisfies fP^M, and then that it is a minimal model of fP^M.

As facts X are part of the program $P \setminus R \cup \mathit{facts}(X)$, and by definition of rule splitting set, $P \setminus R$ contains no rule heads unifying with $\mathit{gh}(R)$, hence we have $X = M|_R$. Furthermore $f(P \setminus R \cup \mathit{facts}(X))^M \setminus \mathit{facts}(X) \cup fR^M = fP^M$, and as M satisfies the left side, it satisfies the right side. To show that M is a minimal model of fP^M, assume $S \subset M$ is a smaller model of fP^M. By definition of reduct, S also satisfies $f(P \setminus R)^M$ and fR^M. Since R is a splitting set, satisfiability of rules in R does not depend on heads of rules in $P \setminus R$, therefore $fR^M = fR^{M|_R} = fR^X$ and $S|_{\mathit{gh}(R)}$ satisfies fR^X. Since $S \subset M$, we have $S|_{\mathit{gh}(R)} \subseteq X$. Because X is a minimal model of fR^X, $S|_{\mathit{gh}(R)} \subset X$ is impossible and $S|_{\mathit{gh}(R)} = X$. Therefore $S|_{\mathit{gh}(P \setminus R)} \subset M|_{\mathit{gh}(P \setminus R)}$. Because S satisfies $f(P \setminus R)^M$ and $S|_{\mathit{gh}(R)} = X$, S also satisfies $f(P \setminus R \cup \mathit{facts}(X))^M$. Since $S \subset M$, this contradicts the fact that M is a minimal model of $P \setminus R \cup \mathit{facts}(X)$. Therefore $S = M$ is a minimal model of fP^M. □

Using the definition of generalized bottom, we generalize the above theorem.

Theorem 11 (Generalized Splitting Theorem). *Let P be a* HEX*-program, let R be a rule splitting set of P, and let B be a generalized bottom of P wrt. R. Then*

$$M \in \mathcal{AS}(P) \textit{ iff } M \in \mathcal{AS}(P \setminus R \cup \mathit{facts}(X)) \textit{ where } X \in \mathcal{AS}(B).$$

Proof. By definition of generalized bottom, the set $C = B \setminus R$ contains only constraints, therefore $\mathit{gh}(B) = \mathit{gh}(R)$ and $M|_{\mathit{gh}(B)} = M|_{\mathit{gh}(R)}$. As $R \subseteq B$ and $B \setminus R$ contains only constraints, $\mathcal{AS}(B) \subseteq \mathcal{AS}(R)$. The only difference between Theorem 10 and Theorem 11 is, that for obtaining X, the latter takes additional constraints into regard.

($\Rightarrow$) It is sufficient to show that $M|_{\mathit{gh}(B)}$ does not satisfy the body of any constraint in $C \subseteq P$ if M does not satisfy the body of any constraint in P. Since B is a generalized bottom, no negative dependencies of constraints C to rules in $P \setminus B$ exist; therefore if the body of a constraint $c \in C$ is not satisfied by M, the body of c is not satisfied by $M|_{\mathit{gh}(B)}$. M satisfies all rules in P, therefore it does not satisfy any constraint body in P, hence the projection $M|_{\mathit{gh}(B)}$ does not satisfy any constraint body in $B \setminus R$.

($\Leftarrow$) It is sufficient to show that an answer set of R that satisfies a constraint body in C would also satisfy that constraint body in P. As constraints in C have no negative dependencies to rules in $P \setminus B$, a constraint with a satisfied body in $M|_{\mathit{gh}(R)}$ also has a satisfied body in M, therefore the result follows. □

Note that $B \setminus R$ contains shareable constraints which are used twice in the Generalized Splitting Theorem: they are used to compute X and to compute M.

The Generalized Splitting Theorem is useful for early elimination of answer sets of the bottom, caused by these constraints which depend on the bottom but also depend on rule heads not in the bottom. Such constraints can be shared between the bottom and the remaining program.

Example 49 (ctd). *We apply Theorems 10 and 11 to P_{swim} and compare them. Using the rule splitting set $\{r_1, r_2\}$ we can obtain answer sets of P_{swim} by first computing the answer sets $\mathcal{AS}(\{r_1, r_2\}) = \big\{\{\mathit{swim}(\mathit{in}), \mathit{need}(\mathit{inout}, \mathit{money})\}, \{\mathit{swim}(\mathit{out})\}\big\}$ and then using Theorem 10: X is an answer set of P_{swim} iff $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup \{\mathit{swim}(\mathit{out}) \leftarrow\})$ or $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup \{\mathit{swim}(\mathit{in}) \leftarrow;\ \mathit{need}(\mathit{inout},\ \mathit{money}) \leftarrow\})$. Note that the computation with $\mathit{need}(\mathit{inout}, \mathit{money})$ in the input does not yield any answer set, because the body of c_8 is always satisfied, which kills the model. On the contrary, if we use the generalized bottom $\{r_1, r_2, c_8\}$ we have $\mathcal{AS}(\{r_1, r_2, c_8\}) = \big\{\{\mathit{swim}(\mathit{out})\}\big\}$ and we can use Theorem 11 to* *obtain answer sets of P_{swim} without only one further answer set computation: X is an answer set of P_{swim} iff $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup \{swim(out) \leftarrow\})$. Note that we use c_8 in both computations, i.e., c_8 is shared between the generalized bottom and the overall answer set computation.* $\square$

Comparison of Splitting Theorems

As there exist quite a number of related splitting theorems, we now give reasons for differences between this theorem and related splitting theorems and show advantages of the theorems in this work.

Our theorem is similar to [Sch06, Theorem 4.6.2] however we do not use splitting sets defined on atoms, but splitting sets defined on rules.

The Splitting Set Theorem in [LT94] splits the interpretation of P into disjoint sets X and Y, where X is an answer set of the 'bottom' $gb_A(P) \subseteq P$ and Y is an answer set of a 'residual' program obtained from $P \setminus gb_A(P)$ and X. In the residual program, all references to atoms in X are removed, in a way that the residual program semantically behaves as if facts X were added to $P \setminus gb_A(P)$, while the answer sets of the residual do not contain any atom in X. This works nicely for answer set programs, but it is problematic when applied to HEX programs, because external atoms may depend on the bottom and on atoms in heads of the residual program, and therefore they cannot be eliminated from rule bodies. The only way to eliminate bottom facts from the residual program would be to split semantics of external atoms into the part depending on the bottom and the remaining part, and by replacing external atoms in rules by external atoms that have been partially evaluated wrt. a bottom answer set. Therefore formulating a splitting theorem for HEX programs with two disjoint interpretations X and Y is not straightforward. This fact may very well be the reason that led to the third splitting theorem, which we compare next.

Compared to the above two splitting theorems, the Global Splitting Theorem in [EIST06] does not split the interpretation of the program P into two disjoint interpretations X and Y. The same is true for the theorem in this work. However, the Global Splitting Theorem in [EIST06] involves the definition of a residual program which specifies how external atoms are evaluated via 'replacement atoms'. These replacement atoms create superfluous facts D in the residual program, which then need to be removed from the answer sets of the residual program. Both the specification of replacement atoms and the superfluous facts make the Global Splitting Theorem in [EIST06] cumbersome to work with when proving correctness of HEX encodings. Furthermore these replacement atoms hint at a certain implementation technique which is not mandatory and can be avoided.

Lemma 5.1 in [EGM97] does not consider external atoms but it is structurally similar to our theorem: answer sets of the bottom program are evaluated together with the program depending on the bottom (here called the residual), hence answer sets of the residual are answer sets of the original program.

Some advantages of our new Theorems 10 and 11 over other HEX splitting theorems in the literature are:

- our theorems rely on the semantics definition of HEX only, and not on implementation details like replacement atoms;
- the residual program is simply a subset of the original program plus facts; and
- our theorems do not introduce superfluous facts in the residual program or in the bottom.

The only (rather negligible) disadvantage of our theorems is that answer sets of the bottom and the residual program are no longer disjoint. (The residual answer set is always a superset of some bottom answer set.)

5.3 Evaluating HEX by Rewriting to ASP

In this section we describe how to evaluate a fragment of HEX programs where the search for an answer set and the grounding process need not be interleaved, i.e., the program can be grounded prior to semantic evaluation of HEX rules. We describe this fragment of HEX and provide an algorithm that evaluates the fragment by rewriting to an ordinary answer set program and verifying external predicate semantics after evaluating answer sets of that program.

In the subsequent section we extend this to a more interesting class of HEX programs where external computations may introduce new constant values. Evaluation of external computations with value invention is more involved, because the set of constants that are relevant for grounding may grow during evaluation.

5.3.1 Pre-Groundable HEX Fragment

The property that makes a HEX program groundable prior to external atom evaluation is closely related to strong safety and to domain-expansion safety (see Def. 25 and 26). It can be captured by the following definition. (Note that we here consider atom-atom dependencies, different from the previous section where we considered mainly rule-rule dependencies.)

Definition 31. *An external atom b in a rule r in a* HEX *program P is* pre-groundable *wrt. r and P iff for each variable X in the output list of b there exists a positive ordinary atom $a \in B^+(r)$ containing X such that $a \not\to^+ b$, i.e., a does not transitively depend on b. A* HEX *program P is* pre-groundable *iff all external atoms in all rules $r \in P$ are pre-groundable wrt. r and P.*

Note that the definition of a 'pre-groundable' external atom exactly reflects condition (b) in Definition 25.

Example 50 (ctd)**.** *If we look at some subsets of P_{swim}, we have that $\{r_1\}$, $\{r_1, r_3\}$, and $\{r_1, r_3, r_4, c_7\}$ are pre-groundable, however $\{r_1, r_2\}$ and $\{r_1, r_2, c_8\}$ are not pre-groundable. This is clear, because C in r_2 must be grounded with a constant that is not contained in P_{swim} but relevant for the evaluation of the semantics of P_{swim}.* $\square$

We have the following proposition which shows acyclicity for non-pregroundable external atoms in a domain-expansion safe program.

Proposition 12. *Let P be a domain-expansion safe* HEX *program which contains an external atom b in a rule $r \in P$ such that b is not pre-groundable. Then $b \not\to^+ b$, i.e., b is not contained in a dependency cycle.*

Proof. As P is domain-expansion safe, b is strongly safe wrt. r and P. As b is not pre-groundable, condition (b) in Def. 25 is not satisfied, therefore condition (a) is satisfied, therefore $b \not\to^+ b$ and the result holds. $\square$

This result will be useful in Section 5.4, where we show that our new evaluation algorithm can evaluate the semantics of every domain-expansion safe HEX program. (Wlog. it might be necessary to perform rule unfolding as a preprocessing step, as described in Section 5.1.3.)

In the following we first extend the pre-groundable fragment and recall a method for evaluating this fragment. In Section 5.4 we then introduce a HEX decomposition and evaluation formalism which can evaluate all domain-expansion safe HEX programs independent of whether they are pre-groundable or not.

Algorithm 5.1: EVALUATEUNIT(P: HEX program, I: HEX interpretation)

```
**Output**: answer sets of $P \cup facts(I)$ without $I$
// determine non-disjunctive facts in $P$ and $I$
$F := I \cup \{a \mid r \in P$ such that $H(r) = \{a\}$ and $B(r) = \emptyset\}$
// determine external atoms that get input only from $F$
$A_{in} := \big\{\&g[\vec{x}](\vec{y}) \in r \mid r \in P$ and for every $r' \in P$ such that $\&g[\vec{x}](\vec{y}) \rightarrow^e b$
                                          it holds that $b \in F\big\}$
// evaluate external atom semantics and create corresponding ground replacement atoms
$I_{aux} := \big\{d_{\&g}(\vec{x}, \vec{z}) \mid \&g[\vec{x}](\vec{y}) \in A_{in}$ with $\vec{x} = (x_1, \dots, x_k)$, signature $t_i$,
                        $1 \leq i \leq k,\ \vec{z} \in F_{\&g}(\Pi_{t_1}(F), \dots, \Pi_{t_k}(F))$, and $\vec{z} \sim \vec{y}\big\}$
$P' := P$ with external atoms $\&g[\vec{x}](\vec{y}) \in A_{in}$ replaced by auxiliaries $d_{\&g}(\vec{x}, \vec{y})$
**return** $\{I' \setminus (I \cup I_{aux}) \mid I' \in$ EVALUATEPREGROUNDABLE$(P', I \cup I_{aux})\}$
```

5.3.2 Extended Pre-Groundable Fragment and Evaluation Algorithm

If we have a pregroundable HEX program, we can ground it, guessing the truth value of each external atom, and evaluate the grounded program over each guess. As a result, we can evaluate pre-groundable HEX programs by rewriting them to plain answer set programs, evaluating plain ASP semantics and evaluating external semantics.

Furthermore we can also evaluate a slightly extended fragment of HEX programs which is not pre-groundable, but apart from pre-groundable external atoms it only contains external atoms that depend on non-disjunctive facts. We define this fragment in the following.

Definition 32. *A* HEX *program P is* extended pre-groundable *iff for each external atom b in a rule $r \in P$ it holds that either b is pre-groundable wrt. r and P, or every atom a that b depends on is the head of a non-disjunctive fact in P. (I.e., if atom a occurs in a rule head in P, this rule must be of the form $a \leftarrow .$)*

Note that this is a weakening of Definition 31.

Example 51 (ctd). *We have previously seen, that $\{r_1, r_2\}$ and $\{r_1, r_2, c_8\}$ are not pre-groundable while $\{r_1\}$ is pre-groundable. Using the Generalized Splitting Theorem, we can split $\{r_1, r_2, c_8\}$ into $\{r_1\}$ and $\{r_2, c_8\}$. The result are two programs $P_1 = \{r_2, c_8\} \cup \{swim(in) \leftarrow\}$ and $P_2 = \{r_2, c_8\} \cup \{swim(out) \leftarrow\}$. Both are not pre-groundable, however they are extended pre-groundable because the external atom depends only on nondisjunctive facts. Therefore we can first evaluate the external atoms, thereby obtain the new constant '$money$', hence we have all constants required to ground and evaluate these programs.* □

Algorithm 5.1 evaluates an extended pre-groundable HEX program P as follows: it (a) computes the set F of non-disjunctive facts in P and I, (b) computes the set of external atoms A_{in} which only depend on atoms in F, (c) evaluates these external atoms wrt. F, (d) replaces these atoms in P by corresponding replacement atoms, and (e) obtains a pre-groundable program P' which can be evaluated together with auxiliary atoms that correspond to truth values of external atoms that were evaluated in (c).

This algorithm, together with EVALUATEPREGROUNDABLE, was introduced in [Sch06, Sec. 4.6.4, Algorithm $\text{eval}(comp, I)$]. These algorithms are not the focus of this thesis, therefore we limit ourselves to the informal description above and next define the behavior of these algorithms. Intuitively, EVALUATEPREGROUNDABLE(P, I) returns $\mathcal{AS}(P \cup I)$. For our subsequent considerations we assume input and output of EVALUATEUNIT as follows.

Proposition 13. *Given an extended pre-groundable* HEX *program P, and a set of ground atoms I,* EVALUATEUNIT(P, I) *returns $\{I' \setminus I \mid I' \in \mathcal{AS}(P \cup facts(I))\}$.*

Example 52 (ctd). *Continuing Example 51, calling* EVALUATEUNIT$(\{r_2, c_8\}, \{swim(in)\})$ *first computes* $F = \{swim(in)\}$*; then gets* $A_{in} = \{\&rq[swim](C)\}$*; next obtains that* $I_{aux} = \{d_{\&rq}(swim, money)\}$ *from evaluating &rq on* F*. Then the algorithm creates the pre-groundable* HEX *program*

$$P' = \{need(inout, C) \leftarrow d_{\&rq}(swim, C); \leftarrow need(X, money)\}$$

and finally calls EVALUATEPREGROUNDABLE$(P', \{swim(in), d_{\&rq}(swim, money)\})$ *which returns* $\emptyset$ *as the constraint body is satisfied because* $need(inout, money)$ *is true.*

Calling EVALUATEUNIT$(\{r_2, c_8\}, \{swim(out)\})$ *computes* $F = \{swim(out)\}$*, then* $A_{in} = \{\&rq[swim](C)\}$*, next obtains* $I_{aux} = \emptyset$ *(the external atom is not true for any output tuple on input* $swim(out)$*), creates the same pre-groundable* HEX *program* P' *as before, and finally calls* EVALUATEPREGROUNDABLE$(P', \{swim(out)\})$ *which returns* $\{\emptyset\}$*, i.e.,* $\{r_1, r_2, c_8\}$*, has one answer set which is* $\{swim(out)\}$*.* □

5.4 Decomposition and Evaluation Techniques

This section introduces our new HEX evaluation framework, which is based on selections of sets of rules of the original program. We call such groups of rules *evaluation units* (in short: units). Compared to the former evaluation approach, units are not necessarily maximal. Instead, we require that partial models of units, i.e., atoms in heads of their rules, do not interfere with those of other units. This allows for independence, efficient storage, and easy composition of partial models of distinct units. Furthermore, in certain cases this allows to share constraints among several units, leading to performance benefits.

5.4.1 Evaluation Graph

Using our notion of rule dependencies, we next define the evaluation graph which consists of evaluation units that depend on one another. We then relate evaluation graphs to the well-known notion of splitting sets [LT94] and show how evaluation graphs permit to evaluate semantics of HEX-programs by evaluating semantics of evaluation units and combining the results.

Definition 33. *Every* evaluation unit *(in short 'unit') is an extended pre-groundable* HEX *program* P*.*

We say a unit u depends on another unit v, if there is an edge from u to v. An important point of the following evaluation graph definition is, that we impose different conditions for dependencies between rules, depending on whether a rule is a constraint or not: constraints cannot (directly) make atoms true, hence they can be shared between evaluation units in certain cases, while sharing non-constraints could violate modularity conditions.

Given a rule $r \in P$ and a set U of evaluation units, we denote by $U|_r$ the set $\{u \in U \mid r \in u\}$ of units that contain rule r.

Definition 34 (Evaluation graph). *An* evaluation graph $\mathcal{E} = (U, E)$ *of a program* P *is a directed acyclic graph; vertices* U *are evaluation units and* $\mathcal{E}$ *has the following properties:*

(a) $P = \bigcup_{u \in U} u$*, i.e., every rule* $r \in P$ *is contained in at least one unit;*

(b) for every non-constraint $r \in P$*, it holds that* $\left|U|_r\right| = 1$*, i.e.,* r *is contained in exactly one unit;*

(c) for each nonmonotonic dependency $r \to_n s$ *between rules* $r, s \in P$ *and for all* $u \in U|_r$*,* $v \in U|_s$*,* $u \neq v$*, there exists an edge* $(u, v) \in E$ *(intuitively, nonmonotonic dependencies between rules have corresponding edges everywhere in* $\mathcal{E}$*); and*

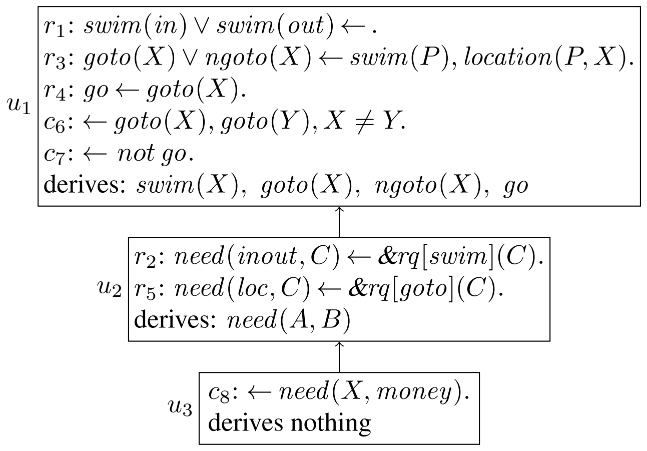

u_1
r_1: $swim(in) \vee swim(out) \leftarrow .$
r_3: $goto(X) \vee ngoto(X) \leftarrow swim(P), location(P,X).$
r_4: $go \leftarrow goto(X).$
c_6: $\leftarrow goto(X), goto(Y), X \neq Y.$
c_7: $\leftarrow not\ go.$
derives: $swim(X),\ goto(X),\ ngoto(X),\ go$
u_2
r_2: $need(inout, C) \leftarrow \&rq[swim](C).$
r_5: $need(loc, C) \leftarrow \&rq[goto](C).$
derives: $need(A,B)$
u_3
c_8: $\leftarrow need(X, money).$
derives nothing

Figure 5.3: Evaluation graph $\mathcal{E}_1$ for running example HEX program P_{swim}.

(d) for each monotonic dependency $r \rightarrow_m s$ between rules $r, s \in P$, there exists one $u \in U|_r$ such that E contains all edges (u,v) with $v \in U|_s$, $v \neq u$ (intuitively, for each rule r there is (at least) one unit in $\mathcal{E}$ where all monotonic dependencies from r to other rules have corresponding outgoing edges in $\mathcal{E}$).

As a non-constraint can only be contained in a single unit, the above definition implies that all dependencies of non-constraints have corresponding edges in $\mathcal{E}$, which is formally expressed in the following proposition.

Proposition 14. *Let $\mathcal{E} = (U, E)$ be an evaluation graph of a program P, then for every dependency $r \rightarrow_{m,n} s$ between a non-constraint $r \in P$ and a rule $s \in P$ and for all $u \in U|_r$, $v \in U|_s$ there exists an edge $(u,v) \in E$.*

Proof of Proposition 14. Assume towards a contradiction that there exists a non-constraint $r \in P$, a rule $s \in P$ with $r \rightarrow_{m,n} s$, and there exist units $u' \in U|_r$, $v' \in U|_s$ such that $(u',v') \notin E$. Due to Definition 27, $r \rightarrow_{m,n} s$ implies that s has $H(s) \neq \emptyset$ and therefore that s is a non-constraint. Definition 34 (b) then implies that $U|_r = \{u'\}$ and $U|_s = \{v'\}$ (non-constraints are present in exactly one unit).

Case (i): for $r \rightarrow_n s$, Definition 34 (c) specifies that for all $u \in U|_r$, $v \in U|_s$ there exists an edge $(u,v) \in E$, therefore also $(u',v') \in E$, which is a contradiction.

Case (ii): for $r \rightarrow_m s$, Definition 34 (d) specifies that there exists some $u \in U|_r$ such that for all $v \in U|_s$ there exists an edge $(u,v) \in E$, and since $U|_r = \{u'\}$ and $U|_s = \{v'\}$ it must hold that $(u',v') \in E$, which is a contradiction. $\square$

Example 53 (ctd)**.** *Figures 5.3 and 5.4 show two possible evaluation graphs for our running example. $\mathcal{E}_1$ is an evaluation graph which contains every rule of P_{swim} in exactly one unit. In contrast, $\mathcal{E}_2$ contains constraint c_8 both in u_2 and in u_4. Condition* (d) *of Definition 34 is particularly interesting in these two graphs; it is fulfilled as follows. In $\mathcal{E}_1$ each rule is contained in exactly one unit, and at that unit all rule dependencies have corresponding unit dependencies. In $\mathcal{E}_2$ the constraint c_8 is part of u_2 and of u_4. The dependencies of c_8, i.e., $c_8 \rightarrow_m r_2$ and $c_8 \rightarrow_m r_5$, have corresponding unit dependencies at u_4, but not at u_2. As condition* (d) *only requires that rule dependencies have corresponding unit dependencies at a single unit, u_4 in $\mathcal{E}_2$ fulfills this condition for dependencies of c_8.* $\square$

Evaluation graphs have the important property that partial models of evaluation units do not intersect, i.e., evaluation units do not mutually depend on each other. This is achieved by requiring acyclicity, and that rule dependencies are covered in the graph.

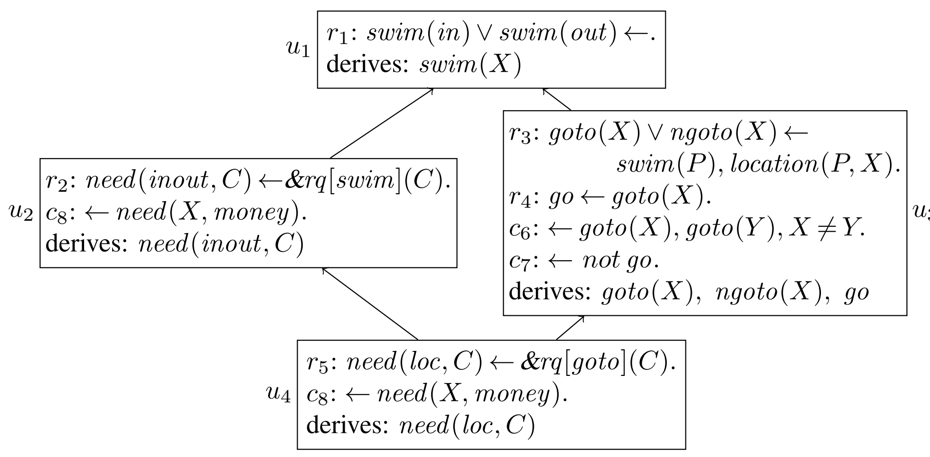

u_1
r_1: $swim(in) \lor swim(out) \leftarrow.$
derives: $swim(X)$
u_2
r_2: $need(inout, C) \leftarrow \&rq[swim](C).$
c_8: $\leftarrow need(X, money).$
derives: $need(inout, C)$
u_3
r_3: $goto(X) \lor ngoto(X) \leftarrow swim(P), location(P, X).$
r_4: $go \leftarrow goto(X).$
c_6: $\leftarrow goto(X), goto(Y), X \neq Y.$
c_7: $\leftarrow not\ go.$
derives: $goto(X),\ ngoto(X),\ go$
u_4
r_5: $need(loc, C) \leftarrow \&rq[goto](C).$
c_8: $\leftarrow need(X, money).$
derives: $need(loc, C)$

Figure 5.4: Evaluation graph $\mathcal{E}_2$ for running example HEX program P_{swim}.

Due to acyclicity of an evaluation graph, mutually dependent rules of a program are contained in the same unit, therefore each strongly connected component of the program's dependency graph is fully contained in a single evaluation unit. Furthermore, a unit can have in the heads of its rules only atoms that do not unify with atoms in heads of rules in other units, as rules which have unifiable heads mutually depend on one another. This ensures that under any grounding, the set of heads of rules in one evaluation unit is disjoint from the set of heads of rules in all other evaluation units. We call this the property of *disjoint unit outputs*.

Proposition 15 (Disjoint unit outputs)**.** *Given an evaluation graph $\mathcal{E} = (U, E)$ of a program P, then for each pair of distinct units $u_1 \in U$, $u_2 \in U$, $u_1 \neq u_2$, it holds that $gh(u_1) \cap gh(u_2) = \emptyset$.*[5]

Proof. Given two units $u_1 \in U$, $u_2 \in U$, $u_1 \neq u_2$, assume towards a contradiction that there exists $\gamma \in gh(u_1) \cap gh(u_2)$. Then there exists some $r \in u_1$ with $\alpha \in H(r)$ and $\alpha \sim \gamma$, and there exists some $s \in u_2$ with $\beta \in H(s)$ and $\beta \sim \gamma$. As $\alpha \sim \gamma$ and $\beta \sim \gamma$ and γ is ground, we obtain $\alpha \sim \beta$, therefore, due to Def. 27 (iii) we have $r \rightarrow_m s$ and $s \rightarrow_m r$. As r and s have nonempty heads, they are non-constraints, therefore, due to Prop. 14, there is an edge $(u_1, u_2) \in E$ and an edge $(u_2, u_1) \in E$. As an evaluation graph is an acyclic graph, $u_1 = u_2$ which is a contradiction. ☐

Example 54 (ctd)**.** *Figures 5.3 and 5.4 show for each unit which atoms can become true due to rule heads in the respective units, denoted as 'derived' atoms. Observe, that both graphs have strictly non-intersecting atoms in rule heads of distinct units.* ☐

As the evaluation graph will be central for our evaluation algorithm, we now show that every domain-expansion safe HEX program has at least one corresponding evaluation graph.

As we have seen, splitting a HEX program is possible if there are no cyclic dependencies. Due to Proposition 12 we know that all non-pregroundable external atoms in a domain-expansion safe program are not contained in a cycle. Therefore we can split a domain-expansion safe program at all points where non-pregroundable external atoms occur. The result is a set of extended pre-groundable HEX programs which inter-depend acyclically and therefore can be represented as an evaluation graph.

Proposition 16. *Every domain-expansion safe HEX program P has some evaluation graph of P.*

Proof of Proposition 16. The strongly connected components (SCCs) of the rule dependency graph of P partition the program P into sets of rules. External atoms in such a partition are either

[5]See page 82 for the definition of notation $gh(P)$.

pre-groundable, or they are not contained in a dependency cycle within P (Proposition 12). Therefore each SCC is an extended pregroundable HEX program, and the SCCs inter-depend acyclically (by definition of SCC). Therefore we can create an evaluation graph where each unit is an SCC and unit dependencies are dependencies between rules in each SCC. Note that such an evaluation graph contains no shared constraints. □

Therefore a HEX evaluation approach which is based on the notion of evaluation graph is applicable to all domain-expansion safe HEX programs.

Evaluation Graph Splitting

We next show that evaluation units and their predecessors in an evaluation graph correspond to generalized bottoms. We then use these properties to formulate an algorithm for piece-wise and efficient evaluation of HEX-programs using evaluation graphs.

Given an evaluation graph $\mathcal{E} = (U, E)$, we write $u < w$ iff there exists a path from u to w in $\mathcal{E}$, and $u \leq w$ iff either $u < w$ or $u = w$.

For a unit $u \in U$, let $u^< = \bigcup_{w \in U, u < w} w$ be the set of rules in 'preceding' units of u, i.e., units which u transitively depends on, and let $u^\leq = u^< \cup u$. Note, that for a leaf unit u (i.e., u has no predecessors) we have $u^< = \emptyset$ and $u^\leq = u$.

Theorem 12. *Given an evaluation graph $\mathcal{E} = (U, E)$ of a HEX-program Q and an evaluation unit $u \in U$, it holds that $u^<$ is a generalized bottom of $u^\leq$ wrt. the rule splitting set R comprising all non-constraints in $u^<$.*

Proof. Given a HEX program S, we write $\mathit{constr}(S)$ to denote the subset of constraints in a set of rules S. We say that the *dependencies of $r \in Q$ are covered at unit $u \in U$* iff for all rules $s \in Q$ with $r \rightarrow_{m,n} s$ and $s \notin u$, it holds that r has an edge to all units containing s, formally $(u, u') \in E$ for all $u' \in U|_s$.

To prove that $B = u^<$ is a generalized bottom of $P = u^\leq$ wrt. the rule splitting set $R = u^< \setminus \mathit{constr}(u^<)$ as by Definition 30, we prove that (a) $R \subseteq B \subseteq P$, (b) $B \setminus R$ contains only constraints, (c) no constraint in $B \setminus R$ has nonmonotonic dependencies to rules in $P \setminus B$, and (d) R is a rule splitting set of P.

Statement (a) corresponds to $u^< \setminus \mathit{constr}(u^<) \subseteq u^< \subseteq u^\leq$ and $u^\leq$ is defined as $u^\leq = u^< \cup u$, therefore the relations all hold. For (b), $B \setminus R = u^< \setminus (u^< \setminus \mathit{constr}(u^<))$, and as $A \setminus (A \setminus B) = A \cap B$. We easily see that $B \setminus R = u^< \cap \mathit{constr}(u^<)$ and thus $B \setminus R$ only contains constraints. For (c), we show a stronger property, namely that no rule (constraint or non-constraint) in B has nonmonotonic dependencies to rules in $P \setminus B$. $B = u^<$ is the union of evaluation units $V = \{v \in U \mid v < u\}$. By Definition 34 (c) all nonmonotonic dependencies are covered at all units. Therefore a rule $r \in w$, $w \in V$ with $r \rightarrow_n s$, $s \in V$ implies that either $s \in w$, or that s is contained in a predecessor unit of w and therefore in V. As $P \setminus B = u^\leq \setminus u^<$ it contains exactly those rules at units u such that $u \notin V$. Hence no nonmonotonic dependencies from B to $P \setminus B$ exist and (c) holds. For (d) we know that $R = u^< \setminus \mathit{constr}(u^<)$ contains no constraints, and by Prop. 14 all dependencies of non-constraints in R are covered by $\mathcal{E}$. Therefore $r \in R$, $r \rightarrow_{m,n} s$, and $s \in P$ implies that $s \in R$. Consequently, (d) holds which proves the theorem. □

Example 55 (ctd). *According to Theorem 12, in $\mathcal{E}_1$ we have that $u_2^\leq = u_1 = \{r_1, r_3, r_4, c_6, c_7\}$ is a generalized bottom of $u_2^\leq = u_1 \cup u_2 = \{r_1, r_2, r_3, r_4, r_5, c_6, c_7\}$ wrt. rule splitting set $R = \{r_1, r_3, r_4\}$. This is intuitively true because we can split the rule dependency graph (Figure 5.2) into two disconnected parts by cutting through $r_2 \rightarrow_m r_1$ and through $r_5 \rightarrow_m r_3$ and both cut arrows have the same direction. Also, in $\mathcal{E}_1$, $u_3^\leq = u_1 \cup u_2$ and $u_3^\leq = P_{swim}$; Theorem 12 says that $u_3^\leq$ is a generalized bottom of P_{swim} wrt. rule splitting set $R = \{r_1, r_2, r_3, r_4, r_5\}$. (We can again verify this in the rule dependency graph.)*

In $\mathcal{E}_2$, we have $u_4^< = u_1 \cup u_2 \cup u_3$ and $u_4^\leq = P_{swim}$ and Theorem 12 says that $u_4^<$ is a generalized bottom of P_{swim} wrt. rule splitting set $R = \{r_1, r_2, r_3, r_4\}$. We can verify this on Definition 30: we have $P = P_{swim}$, $B = u_4^< = \{r_1, r_2, r_3, r_4, c_6, c_7, c_8\}$, and R as above. Then indeed $R \subseteq B \subseteq P$; furthermore all rules in $B \setminus R = \{c_6, c_7, c_8\}$ are constraints and none of these constraints depends nonmonotonically on any rule in $P \setminus B = \{r_5\}$. (Note that $c_7 \rightarrow_n r_4$ but this is not a problem as $r_4 \in B$.) □

Theorem 13. *Given an evaluation graph $\mathcal{E} = (U, E)$ of a* HEX*-program Q, an evaluation unit $u \in U$, and an evaluation unit $u' \in preds_{\mathcal{E}}(u)$, it holds that $u'^{\leq}$ is a generalized bottom of the subprogram $u^<$ wrt. the rule splitting set R comprising all non-constraints in $u'^{\leq}$.*

Proof. Similar to the proof of Theorem 12, we show this in four steps; given $P = u^<$, $R = u'^{\leq} \setminus constr(u'^{\leq})$, and $B = u'^{\leq} = u' \cup u'^<$, we show that (a) $R \subseteq B \subseteq P$, (b) $B \setminus R$ contains only constraints, (c) no constraint in $B \setminus R$ has nonmonotonic dependencies to rules in $P \setminus B$, and (d) R is a rule splitting set of P. Predecessors of u are $\{u_1, \dots, u_k\} = preds_{\mathcal{E}}(u)$. Let $V = \{v \in U \mid v < u'\}$ be the set of evaluation units which u' transitively depends on. (Note that $V \subset preds_{\mathcal{E}}(u)$ and $u \notin V$.) As $u'^<$ contains all units u' transitively depends on, we have $B = u' \cup \bigcup_{w \in V} w$.

For (a), $R \subseteq B$ holds trivially, and $B \subseteq P$ holds by definition of $u^<$ and $u'^{\leq}$ and because $u' \in preds_{\mathcal{E}}(u)$. Statement (b) holds, because $B \setminus R$ removes R from B, i.e., it removes everything that is not a constraint in B from B, therefore only constraints remain. For (c) we show that no rule in B has a nonmonotonic dependency to rules in $P \setminus B$. By Definition 34 (c), all nonmonotonic dependencies are covered at all units. Therefore a rule $r \in w$, $w \in \{u'\} \cup V$ with $r \rightarrow_n s$, $s \in U$ implies that either $s \in w$, or that s is contained in a predecessor unit of w and therefore in u' or in V. Hence there are no nonmonotonic dependencies from rules in B to any rules not in B, and hence also not to rules in $P \setminus B$ and (c) holds. For (d) we know that R contains no constraints and by Proposition 14 all dependencies of non-constraints in R are covered by $\mathcal{E}$. Therefore $r \in R$, $r \rightarrow_{m,n} s$, $s \in P$ implies that $s \in R$ and the theorem holds. □

Example 56 (ctd)**.** *Compared to Example 55, Theorem 13 states similar relationships in $\mathcal{E}_1$: we have that $u_1 \in preds_{\mathcal{E}_1}(u_2)$, hence Theorem 13 says that $u_1^{\leq} = u_1$ is a generalized bottom of $u_2^< = u_1$ wrt. rule splitting set $R = \{r_1, r_3, r_4\}$. Furthermore, $u_2 \in preds_{\mathcal{E}_1}(u_3)$, hence $u_2^{\leq} = u_1 \cup u_2$ is a generalized bottom of $u_3^< = u_1 \cup u_2$ wrt. rule splitting set $R = \{r_1, r_2, r_3, r_4, r_5\}$.*

Less straightforward is the case of $\mathcal{E}_2$ and u_4. Unit u_2 is a predecessor of u_4, i.e., $u_2 \in preds_{\mathcal{E}_2}(u_4)$, therefore Theorem 13 states that $u_2^{\leq} = u_1 \cup u_2 = \{r_1, r_2, c_8\}$ is a generalized bottom of $u_4^< = u_1 \cup u_2 \cup u_3$ wrt. rule splitting set $R = \{r_1, r_2\}$. If we compare with Definition 30, we have $P = u_1 \cup u_2 \cup u_3$ and $B = u_1 \cup u_2$, therefore indeed $R \subseteq B \subseteq P$ and the set $B \setminus R = \{c_8\}$ contains only constraints that do not depend nonmonotonically on any rule in $P \setminus B = \{r_3, r_4, c_6, c_7\}$. □

First Ancestor Intersection Units

An evaluation graph is a directed acyclic graph, and from a unit in such a graph there may exist multiple paths reaching another unit. We will use the evaluation graph for model building, and as unit dependencies reflect semantic dependencies between units, units where multiple paths of unit dependencies meet are of special importance. We call such units *first ancestor intersection units;* these are units where distinct paths from the dependency relation of some other unit have their first intersection.

Definition 35. *Given an evaluation graph $\mathcal{E} = (U, E)$, and distinct units v, $w \in U$, we say that unit w is a* first ancestor intersection unit (FAI) *of v iff there exist paths p_1, p_2, $p_1 \neq p_2$, from v to w in E such that p_1 and p_2 overlap only in vertices v and w. We denote by $fai(v)$ the set of all FAIs of a unit v.*

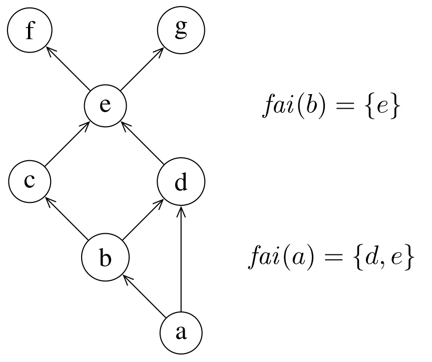


Figure 5.5: First Ancestor Intersection units (FAIs) in an evaluation graph.

Note that for evaluation graphs that are trees, no distinct paths from one to another unit exist, therefore no unit is a FAI of any other unit.

Example 57. *Figure 5.5 sketches an evaluation graph with dependencies (edges) $a \to b \to c \to e \to f$, $a \to d \to e \to g$, and $b \to d$. We have that $fai(a) = \{d, e\}$ and $fai(b) = \{e\}$ and all other units have an empty set of FAIs. In particular, note that f and g are not FAIs of b, because all pairs of distinct paths from b to f or g overlap in more than two units.* $\square$

Example 58 (ctd)**.** *The evaluation graph $\mathcal{E}_1$ (see Fig. 5.3) of P_{swim} is trivially a tree, therefore $fai(u) = \emptyset$ for $u \in \{u_1, u_2, u_3\}$. On the other hand, evaluation graph $\mathcal{E}_2$ (see Fig. 5.4) is not a tree; we have that $fai(u_4) = \{u_1\}$ and no other unit in $\mathcal{E}_2$ has FAIs.* $\square$

5.4.2 Interpretation Graph

We now define the Interpretation Graph (short i-graph), which is the foundation of our model building algorithm. An i-graph is a labeled directed graph which is defined with respect to an evaluation graph: each vertex is associated with a specific evaluation unit, has a type which makes it either an input or an output interpretation, and has an associated set of ground atoms.

We do not use the content of interpretations directly as vertices, because we need distinct vertices to be associated with the same interpretation. Nevertheless we will call vertices of the i-graph interpretations.

We first define the auxiliary Interpretation Structure, then formulate conditions on that structure, and define the i-graph using these conditions. Given an evaluation graph $\mathcal{E} = (U, E)$ and a unit $v \in U$, we define the set of units that v depends on as $preds_{\mathcal{E}}(v) = \{w \in U \mid (v, w) \in E\}$.

Definition 36. *Let $\mathcal{E} = (U, E)$ be an evaluation graph for a program P. An* interpretation structure $\mathcal{I}$ for $\mathcal{E}$ *is a directed acyclic graph $\mathcal{I} = (M, F, unit, type, int)$ where $M \subseteq \mathcal{I}_{id}$ is from a countable set $\mathcal{I}_{id}$ of identifiers, e.g., from $\mathbb{N}$, and $unit\colon M \to U$, $type\colon M \to \{\text{I}, \text{O}\}$, and $int\colon M \to 2^{HB_P}$ are total node labeling functions.*

On interpretation structures we introduce additional notation. Given unit $u \in U$ in the evaluation graph associated with an i-graph $\mathcal{I}$, we denote by $i\text{-}ints_{\mathcal{I}}(u) = \{m \in M \mid unit(m) = u$ and $type(m) = \text{I}\}$, respectively, $o\text{-}ints_{\mathcal{I}}(u) = \{m \in M \mid unit(m) = u$ and $type(m) = \text{O}\}$, the input (i-)interpretations respectively, output (o-)interpretations of $\mathcal{I}$ at unit u. Given vertex $m \in M$, we denote by

$$int^+(m) = int(m) \cup \bigcup \{int(m') \mid m' \in M \text{ and } m' \text{ is reachable from } m \text{ in } \mathcal{I}\}$$

the *expanded interpretation* of m.

Given an interpretation structure $\mathcal{I} = (M, F, unit, type, int)$ for $\mathcal{E} = (U, E)$ and a unit $u \in U$ we define the following conditions:

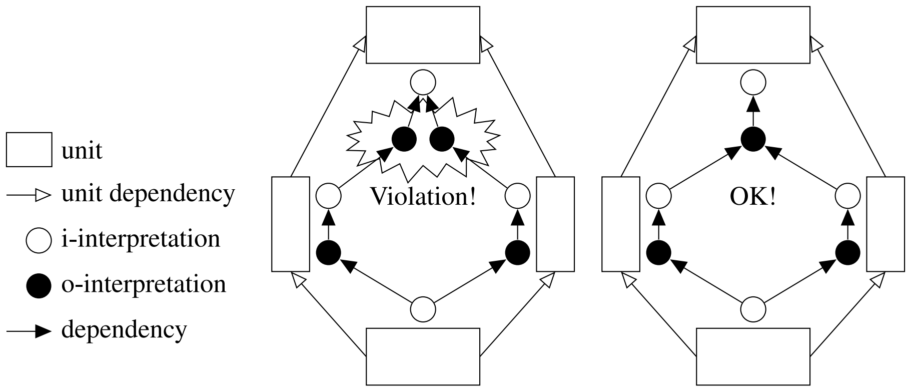


Figure 5.6: Interpretation Graphs: violation of the FAI condition on the left, correct situation on the right.

(IG-I) *I-connectedness:* for every $m \in o\text{-}ints_{\mathcal{I}}(u)$ the structure contains exactly one outgoing edge $(m, m') \in F$ and $m' \in i\text{-}ints_{\mathcal{I}}(u)$ is an i-interpretation at unit u;

(IG-O) *O-connectedness:* for every $m \in i\text{-}ints_{\mathcal{I}}(u)$ and for every predecessor unit $u_i \in preds_{\mathcal{E}}(u)$ of u, there is exactly one outgoing edge $(m, m_i) \in F$ and $m_i \in o\text{-}ints_{\mathcal{I}}(u_i)$ (every m_i is an o-interpretation at the respective unit u_i);

(IG-F) *FAI intersection:* for every $m \in i\text{-}ints_{\mathcal{I}}(u)$, let $\mathcal{I}'$ be the subgraph of $\mathcal{I}$ reachable from m, and let $\mathcal{E}'$ be the subgraph of $\mathcal{E}$ reachable from u. Then $\mathcal{I}'$ contains exactly one o-interpretation at each evaluation unit of $\mathcal{E}'$. (Note that both graphs are acyclic, therefore $\mathcal{I}'$ does not include m and $\mathcal{E}'$ does not include u.)

(IG-U) *Uniqueness:* for every pair of distinct vertices $m_1, m_2 \in M$, $m_1 \neq m_2$, with $unit(m_1) = unit(m_2) = u$, the expanded interpretation of m_1 and m_2 differs, formally $int^+(m_1) \neq int^+(m_2)$.

Definition 37 (Interpretation Graph). *Let $\mathcal{E} = (U, E)$ be an evaluation graph for a program P, then an* interpretation graph *$\mathcal{I} = (M, F, unit, type, int)$ for $\mathcal{E}$ is an interpretation structure that fulfills for every unit $u \in U$ the conditions (IG-I), (IG-O), (IG-F), and (IG-U).*

Intuitively, the conditions make every i-graph 'live' on its associated evaluation graph: an i-interpretation must conform to all dependencies of the unit it belongs to, by depending on exactly one o-interpretation at that unit's predecessor units (IG-I); moreover an o-interpretation must depend on exactly one i-interpretation at the same unit (IG-O). Furthermore, every i-interpretation depends directly or indirectly on exactly one o-interpretation at each unit it can reach in the i-graph (IG-F); this ensures that no expanded interpretation $int^+(m)$ 'mixes' two or more i-interpretations or two or more o-interpretations from one evaluation unit. (The effect of condition (IG-F) is visualized in Figure 5.6.) Finally, redundancies in an i-graph are ruled out by the uniqueness condition (IG-U).

Example 59 (ctd.). *Figures 5.7 and 5.8 show two interpretation graphs: $\mathcal{I}_1$ is an i-graph for $\mathcal{E}_1$, and $\mathcal{I}_2$ is a i-graph for $\mathcal{E}_2$. (We will later see that $\mathcal{I}_1$ and $\mathcal{I}_2$ are two very special i-graphs, namely they are answer set graphs; the symbol ↯ will be explained in Example 64.)*

The unit label is depicted as rectangle labeled with the respective unit. The type label is indicated after interpretation names, i.e., m_1/I denotes that interpretation m_1 is an input interpretation. For $\mathcal{I}_1$, the set $\mathcal{I}_{id}$ of identifiers is $\{m_1, \ldots, m_{18}\}$ and for $\mathcal{I}_2$ it is $\{m_1, \ldots, m_{15}\}$.

Dependencies are shown as arrows between interpretations. Observe that in both graphs I-connectedness is fulfilled, as every o-interpretation depends on exactly one i-interpretation at the same unit. O-connectedness is similarly fulfilled, in particular consider i-interpretations of

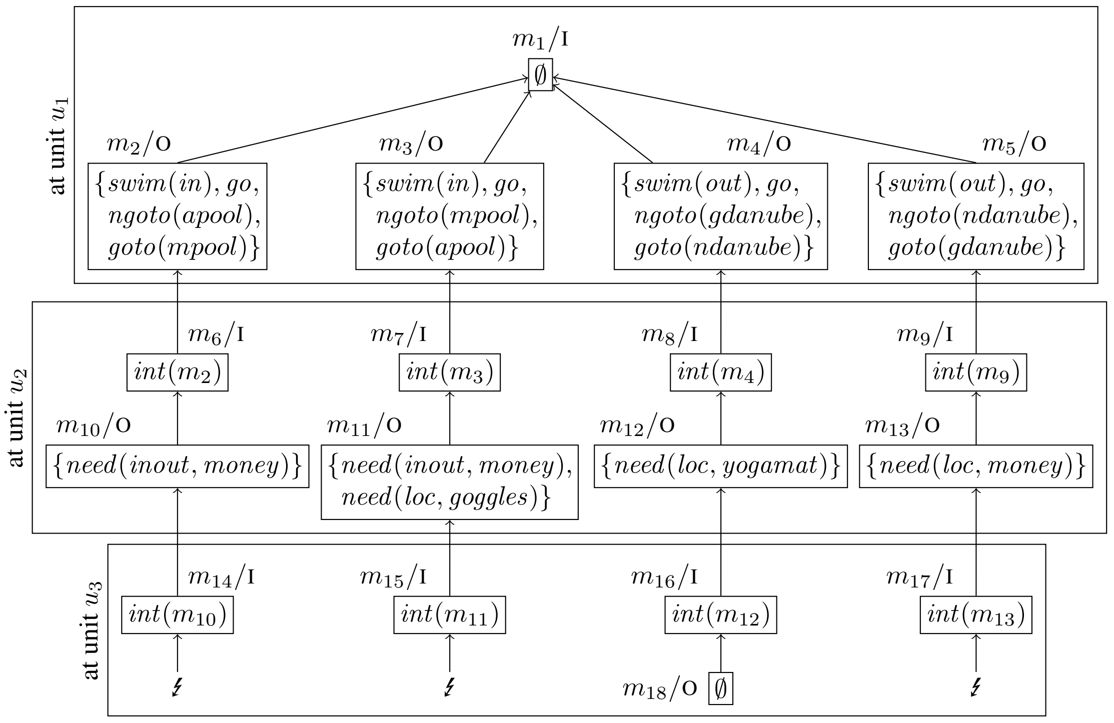


Figure 5.7: Interpretation graph $\mathcal{I}_1$ for $\mathcal{E}_1$

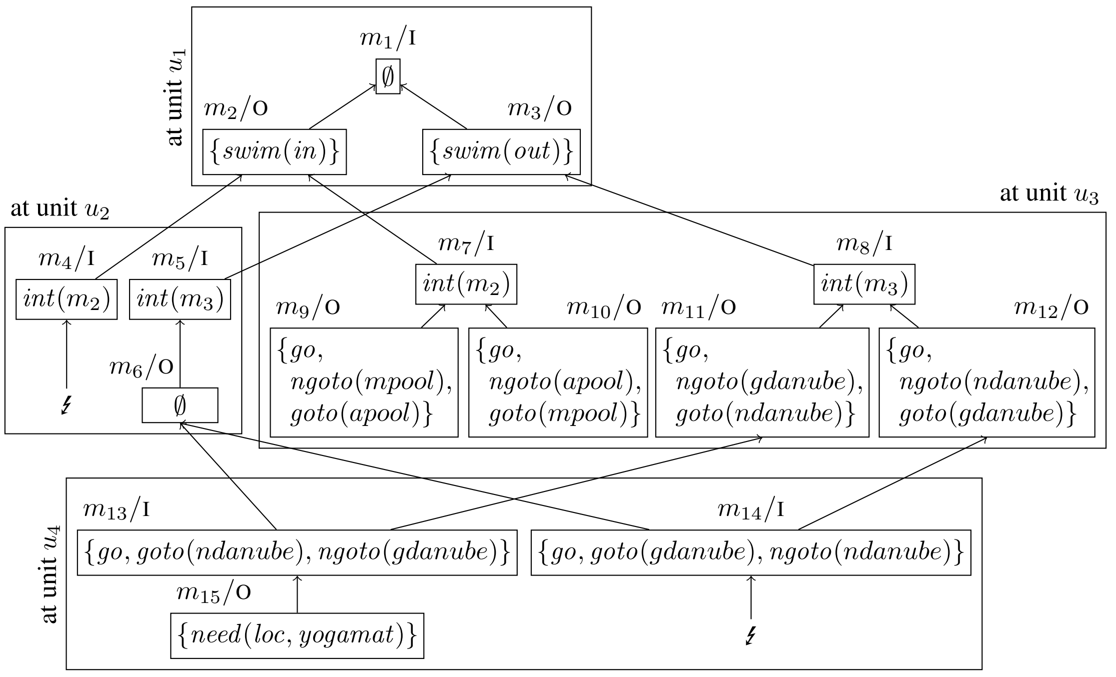


Figure 5.8: Interpretation graph $\mathcal{I}_2$ for $\mathcal{E}_2$

u_4 in $\mathcal{I}_2$: u_4 has two predecessor units (u_2 and u_3) and every i-interpretation at u_4 depends on exactly one o-interpretation at u_2 and exactly one o-interpretation at u_3. The condition on FAI intersection is trivially fulfilled in $\mathcal{I}_1$, as $\mathcal{E}_1$ is a tree. In $\mathcal{I}_2$ the only i-interpretations where FAI intersection could be violated are i-interpretations at u_4. We can verify that from m_{13} and from m_{14} we can reach exactly one o-interpretation at each evaluation unit, therefore the condition is fulfilled. Note that if we had an i-interpretation at u_4 with dependencies to m_6 and to m_9 (or m_{10}), the FAI intersection condition would be violated, because we could reach both m_2 and

m_3 at u_1. Uniqueness is satisfied, as in both graphs no unit has two output models with the same content. □

Note that the empty graph is an i-graph. This is by intent, as our model building algorithm will progress from an empty i-graph to a graph with interpretations at every unit (if and only if the program has an answer set).

Join

We will build i-graphs by adding one vertex at a time, always preserving the i-graph conditions. Adding an o-interpretation requires to add a dependency to one i-interpretation at the same unit. Adding an i-interpretation similarly requires addition of dependencies. However this is more involved because condition (IG-F) could be violated by such additions. Therefore, we next define an operation that captures all necessary conditions.

We call the combination of o-interpretations which yields an i-interpretation a '*join*'. Formally, the join operation '$\bowtie$' is defined as follows.

Definition 38. *Let $\mathcal{I} = (M, F)$ be an i-graph for an evaluation graph $\mathcal{E} = (V, E)$ of a program P. Let $u \in V$ be a unit, let $\{u_1, \dots, u_k\} = preds_{\mathcal{E}}(u)$ be the predecessor units of u, and let $m_i \in o\text{-}ints_{\mathcal{I}}(u_i)$, $1 \leq i \leq k$ be o-interpretations at respective units u_i. Then* the join $m_1 \bowtie \cdots \bowtie m_k = \bigcup_{1 \leq i \leq k} int(m_i)$ at u is defined *iff for each $u' \in fai(u)$ the set of o-interpretations at u' that are reachable (in F) from some o-interpretation m_i, $1 \leq i \leq k$ contains exactly one o-interpretation $m' \in o\text{-}ints_{\mathcal{I}}(u')$.*

Intuitively, a set of interpretations can only be joined if all interpretations depend on the same (and on a single) interpretation at every unit.

Example 60 (ctd). *In $\mathcal{I}_1$, m_1 is created by a trivial join operation with no predecessor units, therefore the join is trivially defined and results in $\emptyset$. Every other i-interpretation in $\mathcal{I}_1$ is created by a trivial join operation with one predecessor unit: such a join is always defined and the interpretation is simply copied. As there are no FAIs in $\mathcal{E}_1$, every join is defined, e.g., m_6 is an i-interpretation created from joining m_2 (with no other interpretation).*

In $\mathcal{I}_2$, i-interpretations m_1, m_4, m_5, m_7, and m_8 are created by trivial join operations with none or one predecessor unit. For m_{13} and m_{14} we have a nontrivial join: $int(m_{13}) = int(m_6) \cup int(m_{11})$ and the join is defined because $fai(u_4) = \{u_1\}$ and it holds that from m_6 and m_{11} we can reach in $\mathcal{I}_2$ exactly one o-interpretation at u_1. Observe that the join $m_6 \bowtie m_9$ is not defined, as we can reach (in $\mathcal{I}_2$) from $\{m_6, m_9\}$ the set of o-interpretations $\{m_2, m_3\}$ at u_1, which means we can reach more than exactly one o-interpretation at each FAI of u_4. Similarly, the join $m_6 \bowtie m_{10}$ is undefined, as we can reach $\{m_2, m_3\}$ at u_1. □

The result of a join is the union of predecessor interpretations; this becomes important next where we introduce answer set graphs and investigate the join operation wrt. them. Note that a leaf unit (i.e., a unit without predecessors) has exactly one well-defined join result $\emptyset$.

If we use the result of a join operation to add a new i-interpretation to the i-graph, and add dependencies to all o-interpretations that were part of that join, then the resulting graph is again an i-graph (i.e., the join is sound operation wrt. the i-graph conditions). Conversely, the join operation creates all i-interpretations that can be added to the i-graph (i.e., the join is a complete operation wrt. the i-graph conditions).

Proposition 17. *Let $\mathcal{I} = (M, F, unit, type, int)$ be an i-graph for an evaluation graph $\mathcal{E} = (V, E)$, a unit $u \in V$ with $\{u_1, \dots, u_k\} = preds_{\mathcal{E}}(u)$, a set $\{m_1, \dots, m_k\}$ with $m_i \in o\text{-}ints_{\mathcal{I}}(u_i)$, $1 \leq i \leq k$, and provided that there is no vertex $m \in i\text{-}ints_{\mathcal{I}}(u)$ such that $\{(m, m_1), \dots, (m, m_k)\} \subseteq F$. Then the join $J = m_1 \bowtie \cdots \bowtie m_k$ is defined at u iff $\mathcal{I}' = (M', F', unit', type', int')$ is an i-graph for $\mathcal{E}$ where (a) $M' = M \cup \{m'\}$ for some new vertex $m' \in \mathcal{I}_{id} \setminus M$,* *(b)* $F' = F \cup \{(m', m_i) \mid 1 \leq i \leq k\}$*, (c)* $\mathit{unit}' = \mathit{unit} \cup \{(m', u)\}$*, (d)* $\mathit{type}' = \mathit{type} \cup \{(m', \mathrm{I})\}$*, and (e)* $\mathit{int}' = \mathit{int} \cup \{(m', J)\}$*.*

Proof. ($\Rightarrow$) The added vertex m' is assigned to one unit and gets assigned a type, furthermore the graph stays acyclic as only outgoing edges from m' are added. I-connectedness is satisfied, as it is satisfied in $\mathcal{I}$ and we add no o-interpretation. O-connectedness is satisfied, as m' gets appropriate edges to o-interpretations at its predecessor units, and for other i-interpretations the condition is already satisfied in $\mathcal{I}$.

For FAI intersection, observe that, if we add an edge (m', m_i) to $\mathcal{I}$, and $m_i \in \textit{o-ints}_{\mathcal{I}}(u_i)$, then m' reaches in $\mathcal{I}$ only one o-interpretation at u_i, and due to O-connectedness that o-interpretation is connected to exactly one i-interpretation at u_i, which is part of the original graph $\mathcal{I}$ and therefore satisfies FAI intersection. Therefore it remains to show that the union of subgraphs of $\mathcal{I}$ reachable in $\mathcal{I}$ from m_1,...,m_k, contains one o-interpretation at each unit in the subgraph of $\mathcal{E}$ reachable in $\mathcal{E}$ from u_1,...,u_k. We make a case distinction.

Case (I): two o-interpretations $m_i \in \textit{o-ints}_{\mathcal{I}}(u_i)$, $m_j \in \textit{o-ints}_{\mathcal{I}}(u_j)$ in the join, with $1 \leq i < j \leq k$, have no common unit that is reachable in $\mathcal{E}$ from u_i and from u_j: then the condition is trivially satisfied, as the subgraphs of $\mathcal{I}$ reachable in $\mathcal{I}$ from m_i and m_j do not intersect at any unit.

Case (II): two o-interpretations $m_i \in \textit{o-ints}_{\mathcal{I}}(u_i)$, $m_j \in \textit{o-ints}_{\mathcal{I}}(u_j)$ in the join, with $1 \leq i < j \leq k$, have at least one common unit that is reachable from u_i and from u_j in $\mathcal{E}$. Let u^f be a unit reachable from in $\mathcal{E}$ both u_i and u_j on two paths that do not intersect before reaching u^f. From u_i to u^f, and from u_j to u^f, exactly one o-interpretation is reachable in $\mathcal{I}$ from m_i and m_j, respectively, as these paths do not intersect. u^f is a FAI of u, and as the join is defined, we reach in $\mathcal{E}$ exactly one o-interpretation at unit u^f from m_i and m_j. Due to O-connectedness, we also reach in $\mathcal{I}$ exactly one i-interpretation m'' at u^f from m_i and m_j. Now m'' is common to subgraphs of $\mathcal{I}$ that are reachable in $\mathcal{I}$ from m_i and m_j, and already satisfies FAI intersection in $\mathcal{I}$.

Therefore FAI intersection is satisfied in $\mathcal{M}'$ for all pairs of predecessors of m' and therefore in all cases. As no vertex m with $\{(m, m_1), \ldots, (m, m_k)\} \subseteq F$ exists, and $\mathcal{M}$ satisfies Uniqueness, $\mathcal{M}'$ also satisfies Uniqueness.

($\Leftarrow$) I-connectedness, O-connectedness, and Uniqueness are satisfied if we add m' as specified, as they are satisfied in $\mathcal{I}$ and due to the way we add m' to $\mathcal{I}$. It remains to show that, whenever FAI intersection is satisfied in $\mathcal{I}'$, then the join is defined. Assume on the contrary that $\mathcal{I}'$ is an i-graph but the join is not defined. As the join is not defined, there exists a FAI $u' \in \mathit{fai}(u)$ such that none or more than one o-interpretation from $\textit{o-ints}_{\mathcal{I}}(u)$ is reachable in $\mathcal{I}$ from m_i, $1 \leq i \leq k$. Due to I-connectedness and O-connectedness, if a unit u' is a FAI and therefore u' is reachable in $\mathcal{E}$ from u_i, then at least one i-interpretation and one o-interpretation at u' is reachable in $\mathcal{I}$ from m_i. If more than one o-interpretation is reachable in $\mathcal{I}$ from m_i, $1 \leq i \leq k$, this means that more than one o-interpretation at u' is reachable in $\mathcal{I}$ from the newly added i-interpretation m. This violates FAI intersection in $\mathcal{I}'$, therefore we reached a contradiction and the result follows. $\square$

Note that the i-graph definition specifies topological properties of an i-graph wrt. an evaluation graph. In the following we extend this specification to the contents of interpretations.

5.4.3 Answer Set Graph

We next restrict the notion of i-graph to the notion of *answer set graph*, such that interpretations correspond with answer sets of certain HEX programs, which are induced by the evaluation graph.

Definition 39 (Answer Set Graph). *Given an evaluation graph* $\mathcal{E} = (U, E)$*, an* answer set graph $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ for $\mathcal{E}$ *is an i-graph for* $\mathcal{E}$ *such that for each unit* $u \in U$ *it holds that* *(a) every expanded i-interpretation at u is an answer set of $u^<$, formally for each i-interpretation $m \in$ i-ints$_\mathcal{I}(u)$ it holds that int$^+(m) \in \mathcal{AS}(u^<)$;*

(b) every expanded o-interpretation at u is an answer set of $u^\leq$, formally for each o-interpretation $m \in$ o-ints$_\mathcal{I}(u)$ it holds that int$^+(m) \in \mathcal{AS}(u^\leq)$; and

(c) every i-interpretation at u is the union of o-interpretations it depends on, formally for each i-interpretation $m \in$ i-ints$_\mathcal{I}(u)$ it holds that int$(m) = \bigcup_{(m,m_i)\in F}$ int(m_i).

Note that for a leaf unit u, we have $u^< = \emptyset$ and therefore the only possible i-interpretation is $\emptyset$ (which coincides with the only possible join result for a leaf unit). Moreover, condition (c) is necessary to ensure that an i-interpretation at unit u contains all atoms of answer sets of predecessor units that are relevant for evaluating u. Furthermore, note that the empty graph is an answer set graph.

Example 61 (ctd)**.** *Both example i-graphs we have discussed so far are in fact answer set graphs. In $\mathcal{I}_2$, int$^+(m_1) = \emptyset$ and $u_1^< = \emptyset$ and indeed $\emptyset \in \mathcal{AS}(\emptyset)$ which satisfies condition* (a). *Less trivial is the case of o-interpretation m_6 in $\mathcal{I}_2$: int$^+(m_6) = \{swim(out)\}$ and $u_2^\leq = \{r_1, r_2, c_8\}$; as c_8 kills all answer sets where money is required, $\mathcal{AS}(\{r_1, r_2, c_8\}) = \{\{swim(out)\}\}$ which means that the expanded interpretation of m_6 is the only possible expanded interpretation of an o-interpretation at u_2. The condition (IG-U) on i-graphs (uniqueness) furthermore implies that m_6 is the only possible o-interpretation at u_2. Consider next m_{13}:*

$$u_4^< = \{r_1, r_2, r_3, r_4, c_6, c_7, c_8\} \text{ and}$$
$$int^+(m_{13}) = \{go, goto(ndanube), ngoto(gdanube), swim(out)\}.$$

The answer sets of $u_4^<$ are

$$\mathcal{AS}(u_4^<) = \{\{go, goto(ndanube), ngoto(gdanube), swim(out)\},$$
$$\{go, goto(gdanube), ngoto(ndanube), swim(out)\}\}$$

and int$^+(m_{13})$ is one of them, the other one is int$^+(m_{14})$. Finally

$$int^+(m_{15}) = \{swim(out), goto(ndanube), go, ngoto(gdanube), need(loc, yogamat)\},$$

which is the single answer set of $u_4^\leq = P_{swim}$. □

The join operation has a useful property with respect to answer set graphs: adding a join result as i-interpretation to an answer set graph again yields an answer set graph (soundness), and all possible answer sets can be created this way (completeness).

Proposition 18. *Given an answer set graph $\mathcal{A} = (M, F, unit, type, int)$ for an evaluation graph $\mathcal{E} = (V, E)$, a unit $u \in V$ with $\{u_1, \ldots, u_k\} = preds_\mathcal{E}(u)$, a set $\{m_1, \ldots, m_k\}$ with $m_i \in$ o-ints$_\mathcal{A}(u_i)$, $1 \leq i \leq k$ and provided that there is no vertex $m \in$ i-ints$_\mathcal{A}(u)$ such that $\{(m, m_1), \ldots, (m, m_k)\} \subseteq F$. Then the join $J = m_1 \bowtie \cdots \bowtie m_k$ is defined at u iff $\mathcal{A}' = (M', F', unit', type', int')$ is an answer set graph for $\mathcal{E}$ where (a) $M' = M \cup \{m'\}$ for some new vertex $m' \in \mathcal{I}_{id} \setminus M$, (b) $F' = F \cup \{(m', m_i) \mid 1 \leq i \leq k\}$, (c) $unit' = unit \cup \{(m', u)\}$, (d) $type' = type \cup \{(m', \mathrm{I})\}$, and (e) $int' = int \cup \{(m', J)\}$.*

Proof of Proposition 18. ($\Rightarrow$) Whenever the join is defined, due to Proposition 17 $\mathcal{A}'$ is an i-graph. It remains to show that $int(m')^+ \in \mathcal{AS}(u^<)$. By Theorem 13 we know that for each u_i, $u_i^\leq$ is a generalized bottom of $u^<$ wrt. a set R_i comprising all non-constraints in $u_i^\leq$. For each u_i, therefore $Y \in \mathcal{AS}(u^<)$ iff $Y \in \mathcal{AS}(u^< \setminus R_i \cup facts(X))$ for some $X \in \mathcal{AS}(u_i^\leq)$. As $\mathcal{A}'$ is an answer set graph, for each m_i we know that $int(m_i)^+ \in \mathcal{AS}(u_i^\leq)$. Therefore $Y \in \mathcal{AS}(u^<)$

iff $Y \in \mathcal{AS}(u^< \setminus R_i \cup \mathit{int}(m_i)^+)$ is an answer set of $u^<$ that contains $\mathit{int}(m_i)^+$. This is true for all i, $1 \leq i \leq k$, from the evaluation graph properties we know that $u^< = u_1^{\leq} \cup \cdots \cup u_k^{\leq}$, and from the construction of $\mathit{int}(m')$ and its dependencies in $\mathcal{A}'$ we obtain that $\mathit{int}(m')^+ = \mathit{int}(m_1)^+ \cup \cdots \cup \mathit{int}(m_k)^+$. Therefore $\mathit{int}(m')^+ \in \mathcal{AS}(u^<)$ which satisfies condition (a). Due to the definition of join, condition (c) is also satisfied and $\mathcal{A}'$ is indeed an answer set graph.

($\Leftarrow$) As $\mathcal{M}'$ is an answer set graph it is also a model graph, therefore the join is defined. As condition (c) holds in $\mathcal{M}'$, J is the union of the o-interpretations it depends on, therefore this direction also holds. $\square$

Example 62 (ctd)**.** *Proposition 18 states that, given an answer set graph, the 'syntactic' conditions of the join operation are sufficient to ensure that an i-interpretation which is the result of a join can be added to the graph and the result is again an answer set graph. In $\mathcal{I}_2$, imagine that there are no interpretations at u_4. The following candidate pairs of o-interpretations exist for creating i-interpretations at u_4: $m_6 \bowtie m_9$, $m_6 \bowtie m_{10}$, $m_6 \bowtie m_{11}$, and $m_6 \bowtie m_{12}$. We have already seen in Example 60 that $m_{13} = m_6 \bowtie m_{11}$ and $m_{14} = m_6 \bowtie m_{12}$ are the only joins at u_4 that are defined. In Example 61 we have seen that $\mathcal{AS}(u_4^<) = \{\mathit{int}^+(m_{13}), \mathit{int}^+(m_{14})\}$, and due to uniqueness (IG-U) of i-graphs we cannot have additional i-interpretations with the same content.* $\square$

Complete Answer Set Graphs

We next introduce a notion of completeness for answer set graphs.

Definition 40. *Let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph for an evaluation graph $\mathcal{E} = (U, E)$ and let $u \in U$ be a unit in U. Then*

- $\mathcal{A}$ is input-complete for u *iff* $\{\mathit{int}^+(m) \mid m \in \textit{i-ints}_{\mathcal{A}}(u)\} = \mathcal{AS}(u^<)$*, and*
- $\mathcal{A}$ is output-complete for u *iff* $\{\mathit{int}^+(m) \mid m \in \textit{o-ints}_{\mathcal{A}}(u)\} = \mathcal{AS}(u^{\leq})$.

If an answer set graph is complete for all units of its corresponding evaluation graph, answer sets of the program can be obtained as follows.

Theorem 14. *Let $\mathcal{E} = (U, E)$ be an evaluation graph of a program P, with $U = \{u_1, \ldots, u_n\}$; and let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph that is output-complete for all units $u \in U$. Then*

$$\mathcal{AS}(P) = \big\{\mathit{int}(m_1) \cup \cdots \cup \mathit{int}(m_n) \mid m_i \in \textit{o-ints}_{\mathcal{A}}(u_i),\ 1 \leq i \leq n, \text{ and } |\textit{o-ints}_{\mathcal{A}'}(u_i)| = 1\big\}, \quad (5.2)$$

where $\mathcal{A}'$ is the subgraph of $\mathcal{A}$ which consists of all interpretations that are reachable in $\mathcal{A}$ from some interpretation m_i, $1 \leq i \leq n$.

Note that the condition implies that, for an answer set, the subgraph $\mathcal{A}'$ contains exactly one o-interpretation at each unit.

To avoid repetition of large parts of this proof in another proof, we delay the proof of Theorem 14 until we have proved Proposition 19.

Example 63 (ctd)**.** *In $\mathcal{I}_2$ we first choose $m_{15} \in \textit{o-ints}(u_4)$ which is the only o-interpretation at u_4. The subgraph reachable from m_{15} must contain exactly one o-interpretation at each unit; therefore we have to choose all o-interpretations m such that $m_{15} \to^+ m$. Hence we obtain*

$$\begin{aligned}
&\big\{\mathit{int}(m_3) \cup \mathit{int}(m_6) \cup \mathit{int}(m_{11}) \cup \mathit{int}(m_{15})\big\}\\
&= \big\{\{\mathit{swim}(\mathit{out})\} \cup \emptyset \cup \{\mathit{goto}(\mathit{ndanube}), \mathit{ngoto}(\mathit{gdanube}), \mathit{go}\} \cup \{\mathit{need}(\mathit{loc}, \mathit{yogamat})\}\big\}\\
&= \big\{\{\mathit{swim}(\mathit{out}), \mathit{goto}(\mathit{ndanube}), \mathit{ngoto}(\mathit{gdanube}), \mathit{go}, \mathit{need}(\mathit{loc}, \mathit{yogamat})\}\big\}
\end{aligned}$$

which is indeed the set of answer sets of P_{swim}. $\square$

The rather involved set construction in (5.2) establishes a relationship between answer sets of a program and complete answer set graphs and resembles condition (IG-F) of i-graphs. To obtain a more straightforward way to enumerate answer sets (and a simpler theorem), we next describe how to extend an evaluation graph with a single evaluation unit u_{final} that depends on all other units in the graph. Answer sets of P then directly correspond to i-interpretations at u_{final}.

Proposition 19. *Given an evaluation graph $\mathcal{E} = (U, E)$ of a program P, where $\mathcal{E}$ contains a final unit $u_{\mathit{final}} = \emptyset$ with dependencies to all other units in U, formally $u_{\mathit{final}} \in U$, $\{(u_{\mathit{final}}, u) \mid u \in U,\ u \neq u_{\mathit{final}}\} \subseteq E$; and given an answer set graph $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ for $\mathcal{E}$ that is input-complete for U and output-complete for $U \setminus \{u_{\mathit{final}}\}$, then*

$$\mathcal{AS}(P) = \{\mathit{int}(m) \mid m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})\}. \tag{5.3}$$

Proof. As u_{final} depends on all units in $U \setminus \{u_{\mathit{final}}\}$, due to O-connectedness an i-interpretation $m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})$ depends on one o-interpretation at every unit in $U \setminus \{u_{\mathit{final}}\}$. Given an i-interpretation $m \in \textit{i-ints}(u_{\mathit{final}})$, let $\{u_1, \ldots, u_k\}$ denote the set of units $U \setminus \{u_{\mathit{final}}\}$, and $\{m_1, \ldots, m_k\}$ denote the set of o-interpretations with $(m, m_i) \in F$, and $m_i \in \textit{o-ints}_{\mathcal{A}}(u_i)$. Then, due to FAI intersection, the set $\{m_1, \ldots, m_k\}$ contains all o-interpretations that are reachable from m in $\mathcal{A}$, and it contains only those interpretations. Therefore $\mathit{int}(m)^+ = \mathit{int}(m_1) \cup \cdots \cup \mathit{int}(m_k)$, and due to condition (c) in Definition 39 we have $\mathit{int}(m) = \mathit{int}(m)^+$. By the dependencies of u_{final}, we have $u^<_{\mathit{final}} = P$, and as u_{final} is input-complete we have that $\mathcal{AS}(P) = \mathcal{AS}(u^<_{\mathit{final}}) = \{\mathit{int}(m)^+ \mid m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})\}$. Using the above intermediate result that $\mathit{int}(m) = \mathit{int}(m)^+$ for all i-interpretations m at u_{final}, we obtain the result. □

Proof of Theorem 14. We prove this theorem here because we make use of Proposition 19. We construct $\mathcal{E}'' = (U'', E'')$ with $U'' = U \cup \{u_{\mathit{final}}\}$, $u_{\mathit{final}} = \emptyset$, and $E'' = E \cup \{(u_{\mathit{final}}, u) \mid u \in U\}$. As u_{final} contains no rules, and $\mathcal{E}''$ is acyclic, no evaluation graph properties are violated and $\mathcal{E}''$ is an evaluation graph. As $\mathcal{A}$ contains no interpretations at u_{final}, and dependencies from units in U are the same in $\mathcal{E}$ and $\mathcal{E}''$, $\mathcal{A}$ is an answer set graph for $\mathcal{E}''$. We now modify $\mathcal{A}$ and yield $\mathcal{A}''$: we add the set $M_{new} = \{m \mid \text{the join } m = m_1 \bowtie \cdots \bowtie m_n \text{ is defined at } u_{\mathit{final}}\}$ as i-interpretations of u_{final} and add according dependencies (from each m to its respective o-interpretations m_i, $1 \leq i \leq n$). Due to Proposition 18 this makes u_{final} input-complete and $\mathcal{A}''$ is an answer set graph for $\mathcal{E}''$. Due to Proposition 19 we have $\mathcal{AS}(P) = \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}}) = M_{new}$. As the result of a join is the union of its joined interpretations, for showing the theorem it remains to show that the join between $m_1, \ldots, m_n$ is defined at u_{final} iff the subgraph $\mathcal{A}'$ of $\mathcal{A}$ reachable from o-interpretations m_i in F contains exactly one o-interpretation $m \in \textit{o-ints}_{\mathcal{A}}(u_i)$ for each unit $u_i \in U$. Due to Proposition 17, adding a joined i-interpretation preserves the i-graph properties, and all i-interpretations we can add have a corresponding join that is defined. FAI intersection states that exactly one o-interpretation at each unit is reachable from an i-interpretation, therefore the result follows. □

Note that it is not necessary to expand i-interpretations at u_{final} because u_{final} depends on all other units, therefore for every $m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})$ it holds that $\mathit{int}^+(m) = \mathit{int}(m)$.

We will use the technique with u_{final} for our model enumeration algorithm; as the join condition must be checked anyways, this technique is an efficient and simple method for obtaining all answer sets of a program using an answer set graph.

5.4.4 Answer Set Enumeration

We build answer set graphs as follows: we start with an empty graph, create o-interpretations by evaluating a unit on an i-interpretation, and create i-interpretations by joining o-interpretations of predecessor units.

Algorithm 5.2: BUILDANSWERSETS

```
    Input: $\mathcal{E} = (V, E)$: evaluation graph for HEX program $P$, which contains a unit $u_{final}$
           that depends on all other units in $V$
    Output: a set of all answer sets of $P$
    $M := \emptyset$, $F := \emptyset$, $unit := \emptyset$, $type := \emptyset$, $int := \emptyset$, $U := V$
(a) while $U \neq \emptyset$ do
        choose $u \in U$ s.t. $preds_{\mathcal{E}}(u) \cap U = \emptyset$
        let $\{u_1, \ldots, u_k\} = preds_{\mathcal{E}}(u)$
        if $k = 0$ then
(b)         $m := max(M) + 1$
            $M := M \cup \{m\}$
            $unit(m) := u$, $type(m) :=$ I, $int(m) := \emptyset$
        else
(c)         for $m_1 \in o\text{-}ints(u_1), \ldots, m_k \in o\text{-}ints(u_k)$ do
                if $J = m_1 \bowtie \cdots \bowtie m_k$ is defined then
                    $m := max(M) + 1$
                    $M := M \cup \{m\}$, $F := F \cup \{(m, m_i) \mid 1 \leq i \leq k\}$
                    $unit(m) := u$, $type(m) :=$ I, $int(m) := J$
(d)     if $u = u_{final}$ then
            return $i\text{-}ints(u_{final})$
(e)     for $m' \in i\text{-}ints(u)$ do
            $O :=$ EVALUATEUNIT$(u, int(m'))$
            for $o \in O$ do
                $m := max(M) + 1$
                $M := M \cup \{m\}$, $F := F \cup \{(m, m')\}$
                $unit(m) := u$, $type(m) :=$ O, $int(m) := o$
(f)     $U := U \setminus \{u\}$
```

Top-down Algorithm

We are now equipped to formulate an algorithm for evaluating HEX programs that have been decomposed into an evaluation graph. Roughly speaking, answer sets can be built by first obtaining an evaluation graph and then computing an answer set graph accordingly.

Algorithm 5.2 shows our model building algorithm, which creates an answer set graph $\mathcal{A} = (M, F, unit, type, int)$ for an evaluation graph $\mathcal{E}$ and returns all answer sets. Intuitively the algorithm operates as follows. U contains units for which $\mathcal{A}$ is not yet output-complete (see Definition 40), and we start with an empty answer set graph $\mathcal{A}$, therefore we start with $U = V$. In each iteration of the while loop (a), one unit u that is not output-complete and depends only on output-complete units is selected, the first for loop (c) makes u input-complete, if u is the final unit we return the answer sets in (d), otherwise the second for loop (e) makes u output-complete, we remove u from U. Each iteration makes one unit input- and output-complete, therefore once the algorithm reaches u_{final} and makes it input-complete all answer sets can directly be returned in (d).

Theorem 15. *Given an evaluation graph $\mathcal{E} = (V, E)$ of a* HEX *program P,* BUILDANSWERSETS$(\mathcal{E})$ *returns $\mathcal{AS}(P)$.*

Proof. We show by induction that the graph $\mathcal{M} = (M, F, unit, type)$ is an answer set graph for $\mathcal{E}$, and that at the beginning of the while loop $\mathcal{M}$ is input- and output-complete for $V \setminus U$.

(Base) $\mathcal{M}$ is initially empty and $V = U$, therefore the base case trivially holds.

(Step) Assuming $\mathcal{M}$ is input- and output-complete for $V \setminus U$, the chosen u is such that it only depends on units in $V \setminus U$, i.e., only on output-complete units. For a leaf unit, (b) creates an empty i-interpretation and therefore makes u input-complete. For a non-leaf unit, the first for loop (c) builds all possible joins of interpretations at predecessors of u, and adds them as i-interpretations to $\mathcal{M}$. As all predecessors of u are output-complete, this makes u input-complete. Condition (d) is true only if we just made u_{final} input-complete, which means that all predecessors of u_{final} are output-complete. As u_{final} depends on all other units, this means that $U = \{u_{\mathit{final}}\}$ and, the algorithm returns $\mathit{i\text{-}ints}_{\mathcal{A}}(u)$ and according to Proposition 19 this is equal to $\mathcal{AS}(P)$. For all other units, the second for loop (e) evaluates u wrt. every i-interpretation at u and adds the result to u as an o-interpretation. Due to Proposition 13, $\textsc{evaluateUnit}(u, \mathit{int}(m'))$ returns all o such that $o \in \{X \setminus \mathit{int}(m') \mid X \in \mathcal{AS}(u \cup \mathit{facts}(\mathit{int}(m))\}$. (As u depends on all units its rules depend on, and as i-interpretations contain all atoms from o-interpretations of predecessor units (due to condition (c) of Definition 39), we have $\textsc{evaluateUnit}(u, \mathit{int}(m')) = \textsc{evaluateUnit}(u, \mathit{int}(m')^+)$ and it is sufficient to call $\textsc{evaluateUnit}(u, \mathit{int}(m'))$ to obtain O. Due to Theorem 12, $u^<$ is a generalized bottom of $u^\leq$. Therefore, due to the generalized splitting theorem, as $\mathit{int}(m')^+ \in \mathcal{AS}(u^<)$ we have that $\mathit{int}(m')^+ \cup o \in \mathcal{AS}(u^\leq)$. Therefore adding a new o-interpretation m with interpretation $\mathit{int}(m) = o$ and dependency to m' to the graph $\mathcal{M}$ makes $\mathit{int}(m)^+ \in \mathcal{AS}(u^\leq)$, and adding all of them makes $\mathcal{M}$ output-complete for u. Finally, in (f) we remove u from U, therefore at the end of the while loop $\mathcal{M}$ is again input- and output-complete for $V \setminus U$.

Hence, in the $|V|$-th iteration of the while loop, condition (d) returns true and the algorithm terminates by returning $\mathcal{AS}(P)$. □

Example 64 (ctd.). *Consider an evaluation graph $\mathcal{E}_2'$ which is $\mathcal{E}_2$ plus $u_{\mathit{final}} = \emptyset$, which depends on all other units. Algorithm 5.2 first chooses $u = u_1$, and as u_1 has no predecessor units, step (b) creates the i-interpretation m_1 with $\mathit{int}(m_1) = \emptyset$. As $u_1 \neq u_{\mathit{final}}$ we continue and in loop (e) obtain $O = \mathcal{AS}(u_1) = \big\{\{\mathit{swim}(\mathit{in})\}, \{\mathit{swim}(\mathit{out})\}\big\}$. We add both answer sets as o-interpretations m_2 and m_3 and then finish the outer loop with $U = \{u_2, u_3, u_4, u_{\mathit{final}}\}$. In the next iteration we could choose $u = u_2$ or $u = u_3$, assume we choose u_2, then $\mathit{preds}_{\mathcal{E}}(u_2) = \{u_1\}$ and $k = 1$. Therefore we enter loop (c) and build all joins that are possible with o-interpretations at u_1 (all joins are trivial and all are possible), i.e., we copy the interpretations and store them at u_2 as new i-interpretations m_4 and m_5. In loop (e) we obtain $O = \textsc{evaluateUnit}(u_2, \{\mathit{swim}(\mathit{in})\}) = \emptyset$, as indoor swimming requires money which is forbidden by $c_8 \in u_2$. Therefore i-interpretation $\{\mathit{swim}(\mathit{in})\}$ yields no o-interpretation, indicated by $\lightning$. However, we obtain $O = \textsc{evaluateUnit}(u_2, \{\mathit{swim}(\mathit{out})\}) = \{\emptyset\}$, as outdoor swimming does not require money and does not require anything else, therefore i-interpretation $\{\mathit{swim}(\mathit{out})\}$ derives no additional atoms and yields the empty answer set which we store as o-interpretation m_6 at u_2. The iteration ends with $U = \{u_3, u_4, u_{\mathit{final}}\}$. The next iteration chooses $u = u_3$, in loop (c) we add i-interpretations m_7 and m_8 to u_3, in loop (e) we add o-interpretations $m_9, \ldots, m_{12}$ to u_3, and the iteration ends with $U = \{u_4, u_{\mathit{final}}\}$. The next iteration chooses $u = u_4$, this time we have multiple predecessors, and in loop (c) we check join candidates $m_6 \bowtie m_9$ and $m_6 \bowtie m_{10}$ which are both not defined. The other join candidates are $m_6 \bowtie m_{11}$ and $m_6 \bowtie m_{12}$ which are both defined, and we add these join results as i-interpretations m_{13} and m_{14} to u_4. Loop (e) then computes one o-interpretation m_{15} for i-interpretation m_{13} and no o-interpretation for m_{14}. The iteration ends with $U = \{u_{\mathit{final}}\}$, therefore the next iteration has $\mathit{preds}_{\mathcal{E}}(u_{\mathit{final}}) = \{u_1, u_2, u_3, u_4\}$ and loop (c) checks all combinations of one o-interpretation at each unit in $\mathit{preds}_{\mathcal{E}}(u_{\mathit{final}})$. Only one such join candidate is defined: this is $m = m_3 \bowtie m_6 \bowtie m_{11} \bowtie m_{15}$, which we store as a new i-interpretation at u_{final}. The check (d) now succeeds, and we return all i-interpretations at u_{final}, i.e., we return $\{m\} = \big\{\{\mathit{swim}(\mathit{out}), \mathit{goto}(\mathit{ndanube}), \mathit{ngoto}(\mathit{gdanube}), \mathit{go}, \mathit{need}(\mathit{loc}, \mathit{yogamat})\}\big\}$ which is indeed the set of answer sets of P_{swim}.* □

5.5 Implementation and Experimental Evaluation

The presented framework has been implemented and released as version 2.0 of the dlvhex solver [DHX12]. The current implementation supports DLV [DLV12] and (for the aggregate-free subset of HEX) clasp+gringo [PAS12].

In addition to the framework described in this thesis, a 'model streaming' algorithm has been implemented which computes interpretations in an answer set graph in a bottom-up order rather than a top-down order. This means that units are not made complete one after the other, but instead the algorithm attempts to find complete answer sets as soon as possible. This is very useful if only the first answer set is required, e.g., for the query support of dlvhex.

5.5.1 Heuristics

As for creating evaluation graphs, several heuristics have been implemented:

- the former dlvhex evaluation heuristics *H1*, which makes units as large as possible and has several drawbacks as discussed above;
- a 'trivial' heuristics which makes units as small as possible and is useful for debugging. However this heuristics has the drawback of a large overhead in the model graph management because it creates the largest possible number of units in the evaluation graph;
- a simple evaluation heuristics *H2* which has the goal of finding a compromise between the trivial heuristics and *H1*. It places rules into units as follows:

  (i) it puts rules r_1, r_2 into the same unit whenever $r_1 \rightarrow_{m,n} s$ and $r_2 \rightarrow_{m,n} s$ and there is no rule t such that exactly one of r_1, r_2 depends on t;

  (ii) it puts rules r_1, r_2 into the same unit whenever $s \rightarrow_{m,n} r_1$ and $s \rightarrow_{m,n} r_2$ and there is no rule t such that t depends on exactly one of r_1, r_2; but

  (iii) it never puts rules r, s into the same unit if r contains external atoms and $r \rightarrow_{m,n} s$.

  Intuitively, this heuristics builds an evaluation graph that puts all rules with external atoms and their successors into one unit, while separating rules creating input for distinct external atoms. This avoids redundant computation and joining unrelated interpretations.

In our experimental evaluation we compare *H1*, which is exactly the strategy of dlvhex 1.x, with *H2*, which turns out to be a fairly good heuristics for our benchmark instances. We here do not claim that *H2* is a universally good heuristics, as the problem which heuristics is generally better than another one is an open research topic.

5.5.2 Benchmarks

For our experiments we use the following benchmark instances.

Multi-context systems. The first kind of benchmark instances, which also motivated this research, are HEX programs $P_p(M)$ that compute output-projected equilibria of a given MCS M (see Section 4.1.2). Each instance consists of 5–10 guessed atoms of input and output interpretations for each of 7–9 knowledge sources, which are realized by external atoms in constraints. Most guesses are eliminated by these constraints; the remaining guesses are then linked by HEX rules which represent bridge rules of the MCS.

The MCSs used for these benchmarks were generated using the instance generator of the DMCS project [DMC11, BDTE+10b]. In particular we use the diamond (D), house (H), ring (R), and zig-zag (Z) topologies of the DMCS instance generator. To obtain a variety of different instances, we generated different instance groups with different configurations and system sizes for each of the above topologies. Each instance group contains 10 randomized instances. The identifier of the instance group contains first the topology, then the number of contexts, the local alphabet size for each context, the maximum number of beliefs used as output beliefs of each context, and finally the maximum number of rules importing knowledge into each context. For example, 'D-7-7-4-4' denotes a diamond topology with 7 contexts where each context has an alphabet of 7 symbols, exports a maximum of 4 symbols to be used in bridge rules by other contexts, and contains a maximum of 4 bridge rules.

The MCS instances have an average of 400 output-projected equilibria, with values ranging from 4 to around 20000, which corresponds to an equal number of answer sets of the HEX rewriting $P_p(M)$.

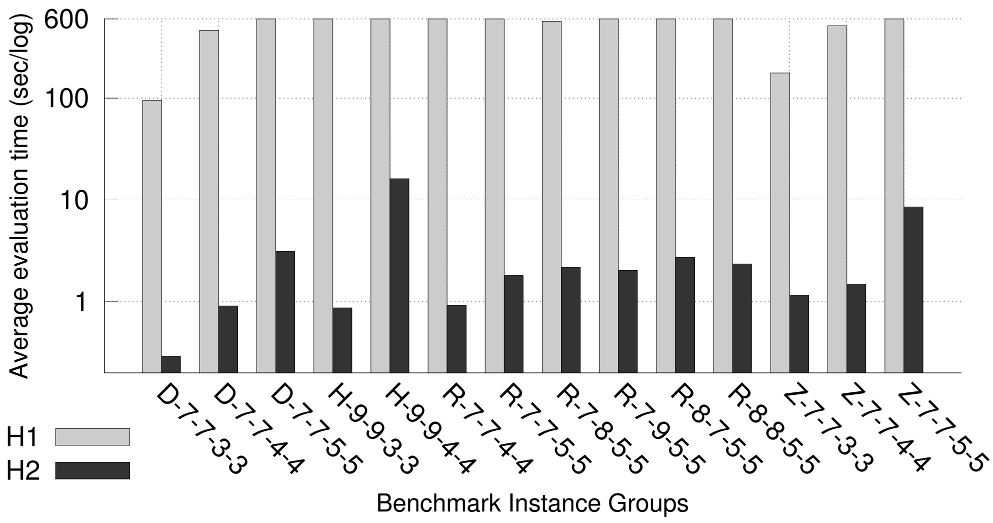
Reviewer Selection. The second class of benchmark instances are synthetic instances that have been crafted to show the effect of our new evaluation framework. These instances encode the selection of reviewers for conference papers—taking conflicts into account, some of which are encoded by external atoms. For these instances, we vary the number T of conference tracks and the number P of papers per track. The number of reviewers available for each track equals P and there is one reviewer who is assigned to all tracks (this establishes a dependency between conference track assignments). Each paper must have 2 reviews and no reviewer gets more than 2 papers assigned. We generated conflicts between reviewers and papers such that we limit the number of overall models, as well as the number of candidate models per conference track, before checking conflicts modeled via external atoms.

For our experiments, we consider two special classes of reviewer selection. In REVSEL 1, we first compared evaluation heuristics *H1* and *H2* and created our instances such that all conference tracks have two solutions before evaluating constraints with external atoms, and there is one overall answer set of the program. For that we used $P = 20$ papers per conference track and varied the number of tracks T. Intuitively this experiment creates one evaluation unit for guessing and one for checking with *H1*, and the size of that guess is exponential in T. On the other hand, *H2* creates one unit for guessing and one unit for evaluating external atoms and checking conflicts per conference track, and one unit to combine the results. This means that *H2* scales linearly with T, which demonstrates that our new evaluation formalism can in some cases provide an exponential speedup.

The other experiment with reviewer selection, REVSEL 2, involved no external atoms. We used $T = 5$ conference tracks and varied the number of papers per track. Conflicts are generated such that there are 1-2 solutions per conference track, with a shared reviewer such that each program Q has 9 answer sets in total. Heuristic *H1* creates one unit in that case, and evaluates it at once in an external solver. In contrast, *H2* keeps each conference track in a separate unit and creates one unit for combining the tracks. This experiment shows first the potential of parallelization that could be applied to the separate conference track units, before combining the results in the final unit. Second, it turned out as a surprise that even with sequential computation, *H2* has better performance than *H1*. In the following, we give detailed results of our evaluation and explain this surprising result.

5.5.3 Results

A series of six concurrent tests were run on a Linux machine with two quad-core Intel Xeon 3GHz CPUs and 32GB RAM. The system resources were limited to a maximum of 3GB memory usage and 600 secs execution time for each run. The computation task for all experiments was to compute all answer sets of the benchmark instances described previously.



	Average time (sec)		Minimum time (sec)		Maximum time (sec)	
Instance group	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
D-7-7-3-3	94.7	0.3	1.9	0.2	—	0.4
D-7-7-4-4	463.5	0.9	7.0	0.3	—	2.2
D-7-7-5-5	—	3.1	—	0.9	—	11.1
H-9-9-3-3	—	0.9	—	0.4	—	2.0
H-9-9-4-4	—	16.2	—	0.6	—	132.0
R-7-7-4-4	—	0.9	—	0.4	—	2.4
R-7-7-5-5	—	1.8	—	0.5	—	2.6
R-7-8-5-5	569.5	2.2	295.2	0.3	—	4.9
R-7-9-5-5	—	2.0	—	0.5	—	4.1
R-8-7-5-5	—	2.7	—	0.6	—	6.9
R-8-8-5-5	—	2.3	—	0.4	—	4.7
Z-7-7-3-3	176.5	1.2	4.1	0.2	—	6.8
Z-7-7-4-4	513.5	1.5	69.9	0.5	—	3.1
Z-7-7-5-5	—	8.5	—	1.8	—	44.6

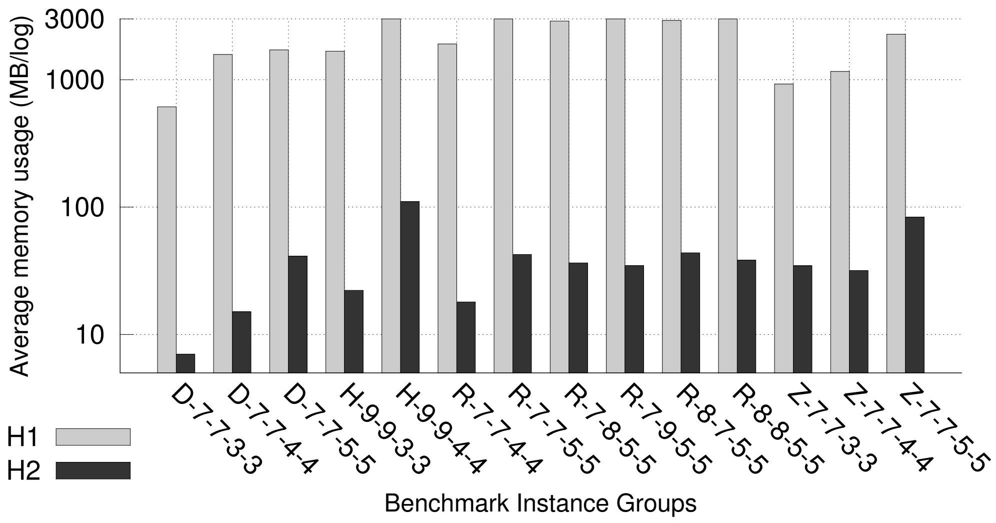
Figure 5.9: Time comparison for enumerating output-projected equilibria of MCS instances. Each instance group contains 10 instances, maximum time/memory was 600 sec/3000 MB, time/memory exhaustion is indicated by '—'.

Multi-context systems

Our application benchmark, the enumeration of output-projected equilibria of a given MCS, shows that the new evaluation approach makes this application feasible for a variety of system topologies.

Figure 5.9 shows the result of time measurements, where *H1* stands for dlvhex 1.x which has heuristics *H1* hard-coded into its evaluation algorithm, and *H2* stands for dlvhex 2.x using heuristics *H2*. Whereas the computations with *H1* very often exhaust time or memory and are not able to finish enumeration of equilibria, *H2* manages to enumerate all equilibria of all instances within the time and memory bounds, and as seen in the table also within reasonable time bounds.

Figure 5.10 shows the memory measurement results of the same experiments. The maximum memory required with heuristic *H1* is always 3000 MB, i.e., computation terminated due to memory exhaustion, while with *H2* the maximum memory required is a modest 332.4 MB. This shows the beneficial effect of our decomposition which keeps separate guesses in separate evaluation units and only combines those guesses which survive a check by external atoms.



	Average memory (MB)		Minimum memory (MB)		Maximum memory (MB)	
Instance group	*H1*	*H2*	*H1*	*H2*	*H1*	*H2*
D-7-7-3-3	612.3	7.0	11.1	4.5	—	9.9
D-7-7-4-4	1579.2	15.1	198.9	6.0	—	32.0
D-7-7-5-5	1714.1	41.2	413.4	11.3	—	90.1
H-9-9-3-3	1673.1	22.2	398.5	7.0	—	59.6
H-9-9-4-4	—	110.5	—	10.8	—	332.4
R-7-7-4-4	1908.2	18.0	494.0	7.1	—	42.6
R-7-7-5-5	—	42.5	—	10.2	—	74.9
R-7-8-5-5	2885.8	36.5	1910.9	6.9	—	120.8
R-7-9-5-5	—	34.8	—	7.2	—	74.2
R-8-7-5-5	2921.9	43.8	2219.2	10.1	—	85.1
R-8-8-5-5	—	38.4	—	14.1	—	80.7
Z-7-7-3-3	925.6	34.6	42.9	6.8	—	260.5
Z-7-7-4-4	1161.8	31.8	251.3	8.5	—	89.8
Z-7-7-5-5	2276.9	83.3	495.3	29.4	—	291.8

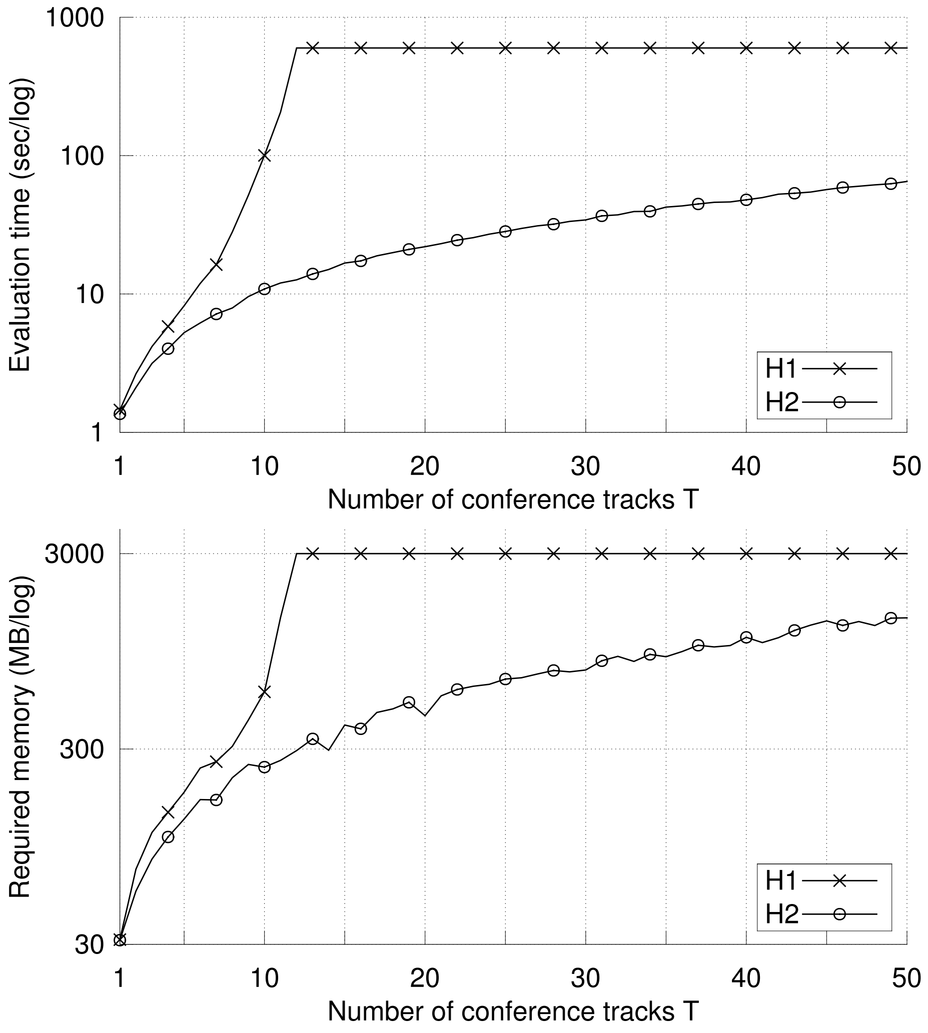
Figure 5.10: Memory usage comparison for enumerating output-projected equilibria of MCS instances. Each instance group contains 10 instances, maximum time/memory was 600 sec/3000 MB, time/memory exhaustion is indicated by '—'.

Reviewer Selection

This benchmark was used in two ways: to demonstrate the effect of our new approach on evaluation with external sources (RevSel 1), and to show that this decomposition can improve over existing ASP solvers for ordinary programs (RevSel 2).

RevSel 1. With the MCS benchmarks, we compared the former dlvhex implementation (which implicitly uses *H1*) to the new dlvhex implementation with heuristics *H2*. This gave us a view on the difference between the former and the new state-of-the-art. Opposed to the MCS experiments, for the RevSel 1 experiments we use evaluation heuristics *H1* and *H2*, both with the new implementation of the dlvhex solver. We make this kind of comparison in order to show the 'raw effect' of the difference between the evaluation heuristics.

Figure 5.11 shows the results of using *H1* and *H2* with the reviewer selection benchmark. In these experiments, we fixed the number of papers per conference track to 20 and varied the number of conference tracks. The former state-of-the-art heuristics *H1* quickly uses too much memory, hence it cannot solve instances with more than 11 conference tracks. Heuristics *H2*,



	Memory usage (MB)		Time usage (sec)	
T	*H1*	*H2*	*H1*	*H2*
1	31.7	31.5	1.45	1.36
2	72.7	56.2	2.65	2.11
3	112.1	82.0	4.17	3.15
4	142.4	106.3	5.82	4.02
5	180.2	131.3	8.21	5.25
6	239.6	165.2	11.92	6.17
7	258.5	164.5	16.31	7.17
8	309.4	213.7	28.08	7.92
9	422.4	250.1	51.74	9.58
10	588.3	242.2	100.26	10.87
11	1413.7	262.4	206.24	12.01
12	—	294.3	—	21.00
20	—	444.4	—	33.45
30	—	761.2	—	46.23
40	—	1118.2	—	62.61
50	—	1408.2	—	65.00

Figure 5.11: RevSel 1 benchmark results: *H1* always exhausts 3000 MB memory (indicated by '—') before it times out; *H2* always terminates successfully within the 600 sec time limit.

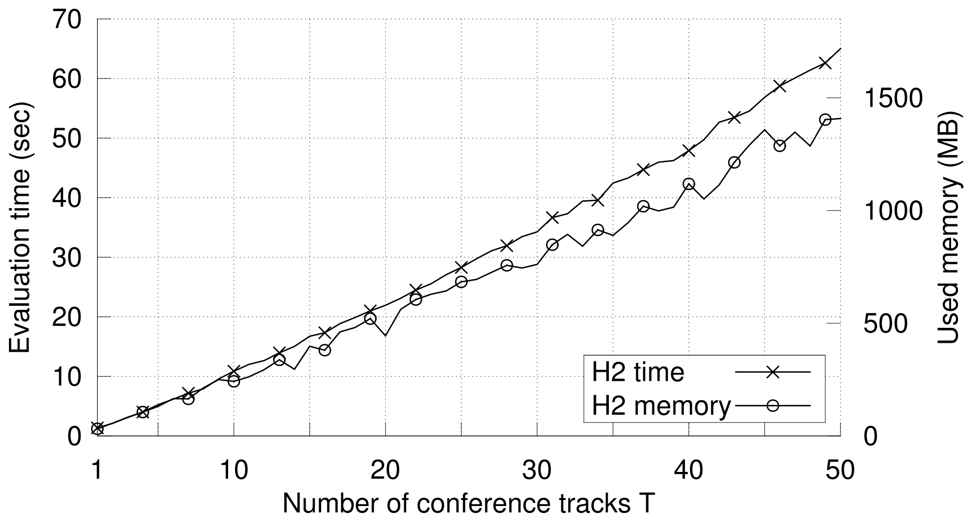


Figure 5.12: Linear plot of time and memory usage of *H2* with RevSel 1: this shows that *H2* scales linearly in both time and space, as opposed to *H1* which scales exponentially or worse.

which can only be applied within the new evaluation framework introduced in this thesis, easily handles up to 50 conference tracks.

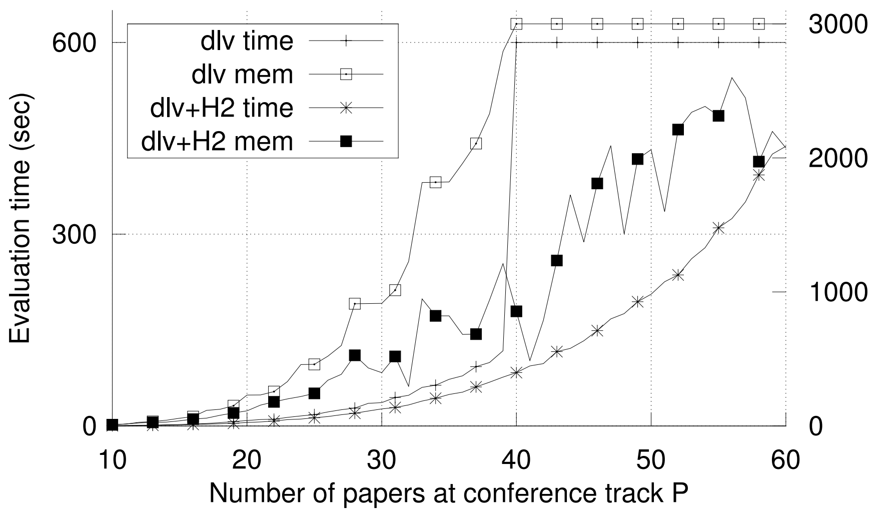
We can observe in Figure 5.12 that heuristics *H2* scales linearly both in time and space wrt. the number of conference tracks, as opposed to exponential growth with *H1*.

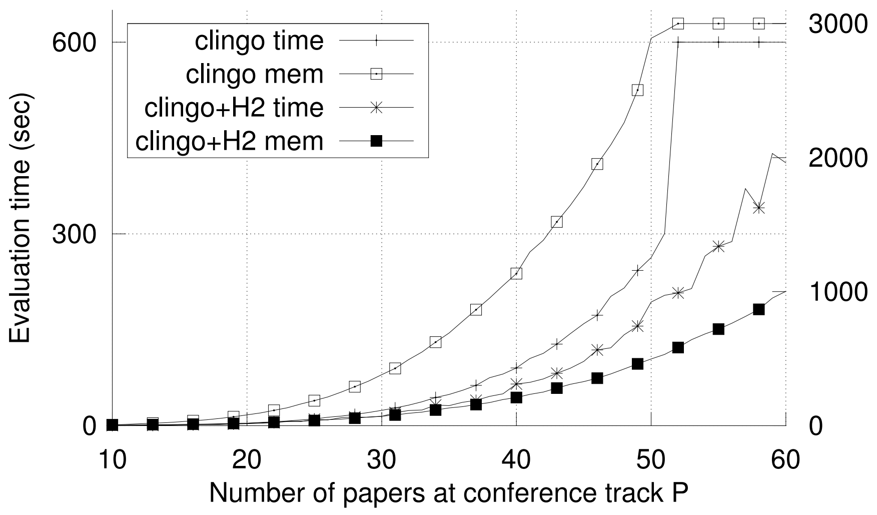
RevSel 2. This benchmark is an experiment that uses HEX programs without external atoms. This serves to investigate the effect of decomposing a program with *H2* and evaluating it using an (external) ASP solver, compared to directly evaluating the program (without decomposition) using that same ASP solver.

Instances of RevSel 2 consist of 5 conference tracks with a varied number of papers per each conference track. Conflicts have been generated such that every instance, regardless of its size, has 9 answer sets. These instances have few dependencies between conference tracks, as there are only two reviewers that are shared between all conference tracks.

Figure 5.13 shows the results of these benchmarks, which were conducted using DLVand clingo as external ASP solvers. (The graphs in that figure show all measurements, while the table gives measured values for multiples of ten.) It is clearly visible that both solvers exhaust the memory limit of 3000 MB above a certain instance size. Using the HEX decomposition with heuristics *H2*, both DLVand clingo are able to solve all instances up to 60 papers per conference track. The reason is evident from the memory measurements: the decomposition makes the external solvers use less memory than in the case when they are given the whole program at once for evaluation.

The benchmark instances have a property that might occur rarely in practice: they are composed of several large program parts that have very few points of interaction. Nevertheless, this experiment shows that decomposition can improve on existing solver technology, and we could improve even more by running the external solvers in parallel and then combining the results. Note that this 'parallelization from the outside' is even possible if the external ASP solver used by dlvhex is not capable of parallelization.





	DLV		clingo		*H2* + DLV		*H2* + clingo	
P	time	memory	time	memory	time	memory	time	memory
10	0.45	9.8	0.20	6.4	0.47	8.8	0.28	5.3
20	8.57	230.8	3.92	80.6	5.48	112.8	2.99	18.7
30	36.62	914.9	23.93	380.8	26.79	398.0	14.61	70.6
40	—	—	90.33	1134.8	83.50	855.4	65.25	210.1
50	—	—	262.91	2890.7	206.17	2061.6	193.36	497.9
60	—	—	—	—	437.47	2074.8	411.15	1002.4

Figure 5.13: RevSel 2 benchmark results (top DLV, bottom clingo): both solvers do not manage to solve instances above a certain size, however if we use *H2* to decompose the computation, it uses less memory and time.

5.6 Discussion and Related Work

We have assessed our new approach by a series of experiments which clearly demonstrate its usefulness. As it is possible to create evaluation graphs that correspond to the previous evaluation strategy, we could directly compare the former and the new approach within the same implementation and show benefits of the new approach. The new approach outperforms the previous one significantly, using (sometimes exponentially) less memory and therefore runs much faster.

The new evaluation framework is able to simulate the previous evaluation method, therefore it can always perform equally good or better. However, the evaluation graph must be configured in an appropriate way to ensure that the new framework really performs favorably. For the examples in this work, the heuristics *H2* ensures that the framework always performs better than the former evaluation method.

Interestingly, also on some ordinary test programs the new method compared well to state-of-the-art ASP solvers: apart from some overhead on fast solved instances, our decomposition approach showed a speed up on top of DLVand clasp. The results indicate a potential for widening the optimization techniques of ordinary logic programs, and possibly also other formalisms like multi-context systems.

The work presented here can be continued and extended in different directions, as the generic notions of evaluation graph and answer set graph allow to specialize and improve our framework in different respects:

- evaluation units (which may contain duplicated constraints) can be chosen according to a proper estimate of the number of answer sets (the fewer, the better);
- evaluation graphs can be built by ad-hoc optimization modules, which may give preference to time, space, or parallelization requirements, or to combinations of these criteria; furthermore
- the data flow (constituted by intermediate answer sets) between evaluation units can be optimized using proper notions of model projection, such as in [GKS09]. Model projections would tailor input data of evaluation units to necessary parts of intermediate answer sets; however, given that different units might need different parts of the same intermediate input answer set, a efficient projection technique that saves space is not straightforward.

Apart from performance in sequential processing mode, our framework can build the base of a coarse-grained distributed computation at the level of evaluation units (in the style of [PRS10]). Evaluation graphs naturally encode parallel evaluation plans, because they make the coupling between evaluation units explicit. We have not yet investigated the potential benefits of this feature in practice, but this property allows to do parallel solving based on solver software that does not have parallel computing capabilities itself ("parallelize from outside"). This applies both to programs with external atoms, as well as to normal ASP programs.

5.6.1 Related Work

HEX programs [Sch06, EIST06, EIST05] have been introduced as a generalization of DL-programs [Sch06, EIP$^+$06, EIST04, EIL$^+$08], which combine Description Logic reasoning with answer set programming. The HEX formalism has been extended to include actions and an environment which gave rise to the ACTHEX formalism [BEFI10]. Modular Nonmonotonic Logic Programs (MLPs) [DTEFK09] do not contain external computations, but instead focus on a modularization approach, including mutually recursive references between program modules.

Our work on decomposition is clearly related to work on program modularity under stable model semantics, including, e.g., the seminal paper [LT94] on the notion of splitting sets, modularity properties of and splitting theorems for disjunctive datalog [EGM97, in particular Lemma

5.1] and more recent works on modularity [OJ08, JOTW09], which lift the concept to modular programs with choice rules and disjunctive rules and allow for "symmetric splitting."

An important difference is that our decomposition approach works for nonground programs and explicitly considers the possibility that modules overlap. It is tailored to efficient evaluation of arbitrary programs, rather than to facilitate module-style logic programming with declarative specifications. In this regard, it is in line with previous work on HEX program evaluation [EIST06] and decomposition techniques for efficient grounding of ordinary programs [CCIL08]. Improving reasoning performance by decomposition has furthermore been investigated in [AM05], however, only wrt. monotonic logics.

Improving HEX evaluation efficiency by using knowledge about *domain restrictions* of external atoms has been discussed in [EFK09]. These rewriting methods yield partially grounded sets of rules which can easily be distributed into distinct evaluation units by an optimizer. This directly provides efficiency gains as described in the above work.

Evaluating HEX with Conflict-Driven Clause Learning. The decomposition we introduced this chapter aims at separating program parts that can be evaluated independently, in order to reduce redundant evaluation. Furthermore, decomposing the program is necessary for evaluating domain-expansion safe programs that are not pre-groundable.

Recently, a new integration of HEX evaluation with conflict-driven clause learning (CDCL) has been introduced [EFKR12] which extends respective works by Gebser et al. [GKS12] from ordinary programs to the HEX setting.

CDCL successively guesses truth values of atoms and handles conflicts (i.e., unsatisfiability of the program) as soon as they can be detected in a partial truth assignment: a conflict is reduced to a (preferrably small) clause that is added to the program (i.e., 'learned') and rules out certain combinations of truth assigments for future guesses. Learned clauses often considerably prune the search space and the CDCL technique was a major breakthrough for ASP and SAT solver efficiency.

The CDCL approach for HEX [EFKR12] integrates external computations and evaluation of programs in external solvers much more tightly, and is able to reduce redundant evaluation even if program parts that could be evaluated independently are evaluated as one big set of rules. The CDCL approach is orthogonal to the decomposition approach described here, and it requires the decomposition to provide pre-groundable HEX programs to the CDCL engine. The combination of both approaches improves evaluation efficiency beyond the results shown in this chapter [EFKR12, EFK$^+$12b, EFK$^+$12a, EFK$^+$12c].

6 Policy Language for Inconsistency Management

In this chapter, we propose the declarative *Inconsistency Management Policy Language* IMPL, which provides means to specify inconsistency management strategies for MCSs.

Remember our inconsistent Medical Example, where someone swapped the digits of Sue's birth date, causing an inconsistency. If such an inconsistency cannot cause harm, e.g., because it only changes the age of Sue by a small amount, or in a case where the inconsistency is due to a mismatch in an address field, then it can be automatically ignored. Such an automatic repair would increase usability of the system as the system stays responsive and gives an answer as unimportant inconsistencies have been recovered automatically.

An entirely different story is the other inconsistency, where we have a patient who needs treatment, but all options conflict with some allergy of the patient. Here attempting an automatic repair may not be a viable option: a doctor should inspect the situation and make a decision.

In the light of such scenarios, tackling inconsistency requires individual strategies and targeted (re)actions, depending on the type of inconsistency and on the application.

Our contributions, presented in [EFIS11, EFIS12a, EFIS12b] in preliminary form, are as follows.

- We define the syntax of IMPL, inspired by Answer Set Programming (ASP) [GL91]. In particular, we specify *input for policy reasoning* in terms of reserved predicates. These predicates encode inconsistency analysis results as introduced in [EFSW10]. Furthermore, we specify *action predicates that can be derived by rules*. Actions provide a means to counteract inconsistency by modifying the MCS, and may involve interaction with a human user.

- We define the semantics of IMPL as a three-step process which first calculates models of a policy, then determines effects of actions which are present in such a model (this possibly involves user interaction), and finally applies these effects to the MCS.

- We provide methodologies for integrating IMPL into application scenarios, and discuss possible modes of reasoning and language extensions that could be useful in practical applications.

- We identify a fragment of IMPL, called *Core* IMPL , which is sufficient for realizing functionality of the full IMPL language. We give a rewriting from Core IMPL to the ACTHEX formalism [BEFI10], which extends the HEX formalism with actions.

- Finally we provide a method of rewriting IMPL to the Core IMPL fragment. This allows for using the ACTHEX rewriting as an implementation for the full IMPL language.

6.1 Policy Language IMPL

Dealing with inconsistency in an application scenario is difficult, because, even if inconsistency analysis provides information how to restore consistency, it is not obvious which choice of system repair is rational. It may not even be clear whether it is wise at all to repair the system by changing bridge rules.

Example 65 (ctd)**.** *In the Medical Example, repairing explanation $e_1 = (\{r_1\}, \emptyset)$ by removing r_1 from M_2 and thereby ignoring the birth date (which differs at the granularity of months) may be the desired reaction and could very well be done automatically. On the contrary, repairing explanation $e_2 = (\{r_2, r_3, r_5\}, \{r_6\})$ by ignoring either the allergy or the illness is a decision that should be left to a doctor, as every possible repair could cause serious harm to Sue.* □

Therefore, managing inconsistency in a controlled way is crucial. To address these issues, we propose the declarative *Inconsistency Management Policy Language* (IMPL), which provides a means to create policies for dealing with inconsistency in MCSs. Intuitively, an IMPL policy specifies (i) which inconsistencies are repaired automatically and how this shall be done, and (ii) which inconsistencies require further external input, e.g., by a human operator, to make a decision on how and whether to repair the system. Note that we do not rule out automatic repairs, but in contrast to previous approaches, automatic repairs are made only if a given policy specifies to do so, and only to the extent specified by the policy.

Since a major point of MCSs is to abstract away context internals, IMPL treats inconsistency by modifying bridge rules. For the scope of this work we delegate any potential repair by modifying the kb of a context to the user. The effect of applying an IMPL policy to an inconsistent MCS M is a *modification* which is a pair (A, R) of sets of bridge rules which are syntactically compatible with M. Intuitively, a modification specifies bridge rules A to be added to M and bridge rules R to be removed from M, similar as for diagnoses without restriction to the original rules of M.

An IMPL policy P for a MCS M is intended to be evaluated on a set of *system and inconsistency analysis* facts, denoted EDB_M, which represents information about M, in particular EDB_M contains atoms which describe bridge rules, minimal diagnoses, and minimal explanations of M.

The evaluation of P yields certain actions to be taken, which potentially interact with a human operator, and modify the MCS at hand. This modification has the potential to restore consistency of M.

In the following we formally define syntax and semantics of IMPL.

6.1.1 Syntax

We assume disjoint sets C, V, Built, and Act, of constants, variables, built-in predicate names, and action names, respectively, and a set of ordinary predicate names $\mathit{Ord} \subseteq C$. Constants start with lowercase letters, variables with uppercase letters, built-in predicate names with $\#$, and action names with @. The set of *terms* T is defined as $T = C \cup V$.

An *atom* is of the form $p(t_1, \ldots, t_k)$, $0 \leq k$, $t_i \in T$, where $p \in \mathit{Ord} \cup \mathit{Built} \cup \mathit{Act}$ is an ordinary predicate name, built-in predicate name, or action name. An atom is *ground* if $t_i \in C$ for $0 \leq i \leq k$. The sets A_{Act}, A_{Ord}, and A_{Built}, called sets of *action atoms*, *ordinary atoms*, and *built-in atoms*, consist of all atoms over T with $p \in \mathit{Act}$, $p \in \mathit{Ord}$, respectively $p \in \mathit{Built}$.

Definition 41. *An* IMPL policy *is a finite set of rules of the form*

$$h \leftarrow a_1, \ldots, a_j, \mathit{not}\, a_{j+1}, \ldots, \mathit{not}\, a_k. \tag{6.1}$$

where h is an atom from $A_{\mathit{Ord}} \cup A_{\mathit{Act}}$, every a_i, $1 \leq i \leq k$, is from $A_{\mathit{Ord}} \cup A_{\mathit{Built}}$, and 'not' is negation as failure.

Given a rule r, we denote by $H(r)$ its head, by $B^+(r) = \{a_1, \dots, a_j\}$ its positive body atoms, and by $B^-(r) = \{a_{j+1}, \dots, a_k\}$ its negative body atoms. A rule is ground if it contains ground atoms only. A ground rule with $k = 0$ is a *fact*. As customary in ASP solvers, we assume that rules are safe, i.e., variables in $H(r)$ or in $B^-(r)$ must also occur in $B^+(r)$. For a set of rules R, we use $\mathit{cons}(R)$ to denote the set of constants from C appearing in R, and $\mathit{pred}(R)$ for the set of ordinary predicate names and action names (elements from $\mathit{Ord} \cup \mathit{Act}$) in R.

We next describe how a policy represents information about the MCS M under consideration.

System and Inconsistency Analysis Predicates.

Entities, diagnoses, and explanations of the MCS M at hand are represented by a suitable finite set $C_M \subseteq C$ of constants which uniquely identify contexts, bridge rules, beliefs, rule heads, diagnoses, and explanations. For convenience, when referring to an element represented by a constant c, we identify it with the constant, e.g., we write 'bridge rule r' instead of 'bridge rule of M represented by constant r'.

Reserved atoms use predicates from the set $C_{\mathit{res}} \subseteq \mathit{Ord}$ of *reserved predicates*, we have $C_{\mathit{res}} = \{\mathit{ruleHead}, \mathit{ruleBody}^+, \mathit{ruleBody}^-, \mathit{context}, \mathit{modAdd}, \mathit{modDel}, \mathit{diag}, \mathit{explNeed}, \mathit{explForbid}\}$. They represent the following information.

- $\mathit{context}(c)$ states that c is a context.
- $\mathit{ruleHead}(r, c, s)$ states that bridge rule r is at context c with head formula s.
- $\mathit{ruleBody}^+(r, c, b)$ (resp., $\mathit{ruleBody}^-(r, c, b)$) states that bridge rule r contains body literal '$(c\!:\!b)$' (resp., '$\mathbf{not}\ (c\!:\!b)$').
- $\mathit{modAdd}(m, r)$ (resp., $\mathit{modDel}(m, r)$) states that modification m adds (resp., deletes) bridge rule r. Note that r is represented using $\mathit{ruleHead}$ and $\mathit{ruleBody}$.
- $\mathit{diag}(m)$ states that modification m is a minimal diagnosis in M.
- $\mathit{explNeed}(e, r)$ (resp., $\mathit{explForbid}(e, r)$) states that the minimal explanation (E_1, E_2) identified by constant e contains bridge rule $r \in E_1$ (resp., $r \in E_2$).
- $\mathit{modset}(ms, m)$ states that modification m belongs to the set of modifications identified by ms.

Example 66 (ctd). *We can represent r_1, r_5 (see Example 5) and the diagnosis $(\{r_1, r_5\}, \emptyset)$ as the following set of reserved atoms:*

$$\begin{aligned} I_{ex} = \{&\mathit{modDel}(d, r_1), \mathit{modDel}(d, r_5), \mathit{diag}(d),\\ &\mathit{ruleHead}(r_1, c_{\mathit{lab}}, \text{`}\mathit{customer}(\mathit{sue}, \mathit{02/03/1985})'),\\ &\mathit{ruleBody}^+(r_1, c_{\mathit{db}}, \text{`}\mathit{person}(\mathit{sue}, \mathit{02/03/1985})'),\\ &\mathit{ruleHead}(r_5, c_{\mathit{dss}}, \text{`}\mathit{need}(\mathit{sue}, \mathit{ab1})'),\\ &\mathit{ruleBody}^+(r_5, c_{\mathit{onto}}, \text{`}(\mathit{sue}){:}\exists \mathit{hasDisease}.\mathit{AtypPneum}')\}, \end{aligned}$$

where constant d identifies the diagnosis. □

Further knowledge used as input for policy reasoning can easily be defined using additional (supplementary) predicates. Note that predicates over all explanations or bridge rules can easily be defined by projecting from reserved atoms. Moreover, to encode preference relations (e.g., as in [EFW10]) between system parts, diagnoses, or explanations, an atom $\mathit{preferredContext}(c_1, c_2)$ could denote that context c_1 is considered more reliable than context c_2. The extensions of such auxiliary predicates need to be defined by the rules of the policy or as additional input facts (ordinary predicates), or they are provided by the implementation (built-in predicates), i.e., the 'solver' used to evaluate the policy. The rewriting to ACTHEX given in Section 6.3.2 provides a good foundation for adding supplementary predicates as built-ins, because the ACTHEX language has generic support for calls to external computational sources. A possible application would be to use a preference relation between bridge rules that is defined by an external predicate and can be used for reasoning in the policy.

Towards a more formal definition of a policy input, we distinguish the following sets.

Definition 42. *Given a MCS M,*

- *the* MCS input base B_M *is the set of ground atoms built from reserved predicates C_{res} and terms from C_M;*
- *the* auxiliary input base B_{Aux} *is the set of ground atoms built from predicates over $Ord \setminus C_{res}$ and terms from C; and*
- *the* policy input base $B_{Aux,M}$ *is defined as $B_{Aux,M} = B_{Aux} \cup B_M$.*

For a set $I \subseteq B_{Aux,M}$, by $I|_{B_M}$ and $I|_{B_{Aux}}$ we denote the restriction of I to predicates from the respective bases.

Now, given an MCS M, we say that a set $S \subseteq B_M$ is a *faithful representation* of M wrt. a reserved predicate $p \in C_{res} \setminus \{modset\}$ iff the extension of p in S exactly characterizes the respective entity or property of M (according to a unique naming assignment associated with C_M as mentioned). For instance, $context(c) \in S$ iff c is a context of M, and correspondingly for the other predicates. Consequently, S is a faithful representation of M iff it is a faithful representation wrt. all p in $C_{res} \setminus \{modset\}$ and the extension of $modset$ in S is empty.

A finite set of facts $I \subseteq B_{Aux,M}$ containing a faithful representation of all relevant entities and properties of an MCS qualifies as input of a policy, as defined next.

Definition 43. *A* policy input *I wrt. MCS M is a finite subset of the policy input base $B_{Aux,M}$, such that $I|_{B_M}$ is a faithful representation of M.*

In the following, we denote by EDB_M a policy input wrt. a MCS M. Note that reserved predicate $modset$ has an empty extension in a policy input (but corresponding atoms will be of use later in combination with actions).

Given a set of reserved atoms I, let c be a constant that appears as a bridge rule identifier in I. Then $rule_I(c)$ denotes the corresponding bridge rule represented by reserved atoms $ruleHead$, $ruleBody^+$, and $ruleBody^-$ in I with c as their first argument. Similarly we denote by $mod_I(m) = (A, R)$ (resp., by $modset_I(m) = \{(A_1, R_1), \ldots\}$) the modification (resp., set of modifications) represented in I by the respective predicates and identified by constant m.

Subsequently, we call a modification m that is projected to rules located at a certain context c 'the *projection* of m to context c'. (We use the same notation for sets of modifications.) Formally we denote by $mod_I(m)|_c$ (resp., $modset_I(m)|_c$) the projection of modification (resp., set of modifications) m in I to context c.

Example 67 (ctd). *In the previous example I_{ex}, by $rule_{I_{ex}}(r_1)$ we refer to bridge rule r_1; moreover $mod_{I_{ex}}(d) = (\{r_1, r_5\}, \emptyset)$ and the projection of modification d to c_{dss} is $mod_{I_{ex}}(d)|_{c_{dss}} = (\{r_5\}, \emptyset)$.* □

Example 68 (ctd). *A proper EDB_{M_2} of our running example is, e.g., as follows:*

$$
\begin{array}{l}
\{\mathit{context}(c_{db}), \mathit{context}(c_{lab}), \mathit{context}(c_{onto}), \mathit{context}(c_{dss}),\\
\ \mathit{ruleHead}(r_1, c_{lab}, \text{‘}\mathit{customer}(\mathit{sue}, \mathit{02/03/1985})'),\\
\ \mathit{ruleBody}^+(r_1, c_{db}, \text{‘}\mathit{person}(\mathit{sue}, \mathit{02/03/1985})'),\\
\ \mathit{ruleHead}(r_2, c_{onto}, \text{‘}(\mathit{sue}){:}\exists \mathit{hasDisease}.\mathit{Pneum}'),\\
\ \mathit{ruleBody}^+(r_2, c_{lab}, \text{‘}\mathit{test}(\mathit{sue}, \mathit{xray}, \mathit{pneum})'),\\
\ \mathit{ruleHead}(r_3, c_{onto}, \text{‘}(\mathit{sue}, \mathit{cmark}){:}\mathit{hasMarker}'),\\
\ \mathit{ruleBody}^+(r_3, c_{lab}, \text{‘}\mathit{test}(\mathit{sue}, \mathit{bloodtest}, \mathit{cmark})'),\\
\ \mathit{ruleHead}(r_4, c_{dss}, \text{‘}(\mathit{sue}){:}\exists \mathit{hasDisease}.\mathit{BacterialDisease}'),\\
\ \mathit{ruleBody}^+(r_4, c_{onto}, \text{‘}\mathit{Pneum}(\mathit{sue})'),\\
\ \mathit{ruleHead}(r_5, c_{dss}, \text{‘}\mathit{need}(\mathit{sue}, \mathit{ab1})'),\\
\ \mathit{ruleBody}^+(r_5, c_{onto}, \text{‘}(\mathit{sue}){:}\exists \mathit{hasDisease}.\mathit{AtypPneum}'),\\
\ \mathit{ruleHead}(r_6, c_{dss}, \text{‘}\mathit{allow}(\mathit{sue}, \mathit{ab1})'),\\
\ \mathit{ruleBody}^-(r_6, c_{lab}, \text{‘}\mathit{allergy}(\mathit{sue}, \mathit{ab1})'),\\
\ \mathit{diag}(d_1), \mathit{modDel}(d_1, r_1), \mathit{modDel}(d_1, r_2),\\
\ \mathit{diag}(d_2), \mathit{modDel}(d_2, r_1), \mathit{modDel}(d_2, r_3),\\
\ \mathit{diag}(d_3), \mathit{modDel}(d_3, r_1), \mathit{modDel}(d_3, r_5),\\
\ \mathit{diag}(d_4), \mathit{modDel}(d_4, r_1), \mathit{modAdd}(d_4, r_6),\\
\ \mathit{explNeed}(e_1, r_1),\\
\ \mathit{explNeed}(e_2, r_2), \mathit{explNeed}(e_2, r_3), \mathit{explNeed}(e_2, r_5), \mathit{explForbid}(e_2, r_6)\}.
\end{array}
$$

Here, the two explanations and four diagnoses given in Examples 14 and 15 are identified by constants e_1, e_2, $d_1, \ldots, d_4$, respectively. □

A policy can create representations of new rules, modifications, and sets of modifications, because reserved atoms are allowed to occur in heads of policy rules. However such new entities require new constants identifying them. To tackle this issue, we next introduce a facility for value invention.

Value Invention via Built-in Predicates ‘$\#id_k$’.

Whenever a policy specifies a new rule and uses it in some action, the rule must be identified with a constant. The same is true for modifications and sets of modifications. Therefore, IMPL contains a family of special built-in predicates which provide policy writers a means to comfortably create new constants from existing ones.

For this purpose, built-in predicates of the form $\#id_k(c', c_1, \ldots, c_k)$ may occur in rule bodies (only). Their intended usage is to uniquely (and thus reproducibly) associate a new constant c' with a tuple $c_1, \ldots, c_k$ of constants (for a formal semantics see the definitions for action determination in Section 6.1.2).

Note that this value invention feature is not strictly necessary, as new constants can be obtained via defining an order relation over all constants, a pool of unused constants, and a guess over an assignment between used and unused constants. However, a dedicated value invention built-in, as introduced here, simplifies policy writing and improves policy readability.

Example 69. *Assume one wants to consider projections of modifications to contexts as specified by the extension of an auxiliary predicate $projectMod(M, C)$. The following policy fragment achieves this using a value invention built-in to assign a unique identifier with every projection (recorded in the extension of another auxiliary predicate $projectedModId(M', M, C)$).*

$$\begin{aligned} projectedModId(M', M, C) &\leftarrow projectMod(M, C), \\ &\quad \#id_3(M', pm_{id}, M, C). \\ modAdd(M', R) &\leftarrow modAdd(M, R), ruleHead(R, C, S), \\ &\quad projectedModId(M', M, C). \\ modDel(M', R) &\leftarrow modDel(M, R), ruleHead(R, C, S), \\ &\quad projectedModId(M', M, C). \end{aligned} \tag{6.2}$$

Intuitively, we identify new modifications by new ids $c_{pm_{id},M,C}$ which are obtained via $\#id_3$ from M, C, and an auxiliary constant $pm_{id} \notin C_M$. The latter simply serves the purpose of disambiguating constants used for projections of modifications. This links new identifiers to constant pm_{id}, therefore we can easily combine (6.2) *with other policy fragments that use $\#id_3$ on modifications and contexts, and values invented in these fragments will not interfere with one another as long as every fragments uses its own auxiliary constant. (We therefore can think of pm_{id} as being 'reserved for value-invention in the projection of modifications'.)* □

Besides representing modifications of a MCS and reasoning about them, an important feature of IMPL is to actually apply them. Actions serve this purpose.

Actions.

Actions alter the MCS at hand and may interact with a human operator. According to the change that an action performs, we distinguish *system actions* which modify the MCS in terms of entire bridge rules that are added and/or deleted, and *rule actions* which modify a single bridge rule. Moreover, the changes can depend on external input, e.g., obtained by user interaction. In the latter case, the action is called *interactive*. Accumulating the changes of all actions yields an overall modification of the MCS. We formally define this intuition when addressing the semantics in Section 6.1.2.

Syntactically, we use @ to prefix action names from Act. The predefined actions listed below are reserved action names. Let M be the MCS under consideration, then the following predefined actions are (non-interactive) system actions:

- $@delRule(r)$ removes bridge rule r from M.
- $@addRule(r)$ adds bridge rule r to M.
- $@applyMod(m)$ applies modification m to M.
- $@applyModAtContext(m, c)$ applies those changes in m to the MCS that add or delete bridge rules at context c (i.e., applies the projection of m to c).

Note that a policy might specify conflicting effects, i.e., the removal and the addition of a bridge rule at the same time. In this case the semantics gives preference to addition.

The predefined actions listed next are rule actions:

- $@addRuleCondition^{+}(r, c, b)$ (resp., $@addRuleCondition^{-}(r, c, b)$) adds body literal $(c : b)$ (respectively, $\mathbf{not}\ (c : b)$) to bridge rule r.
- $@delRuleCondition^{+}(r, c, b)$ (resp., $@delRuleCondition^{-}(r, c, b)$) removes body literal $(c : b)$ (resp., $\mathbf{not}\ (c : b)$) from bridge rule r.
- $@makeRuleUnconditional(r)$ makes bridge rule r unconditional.

Since these actions can modify the same rule, this may also result in conflicting effects, where again addition is given preference over removal by the semantics. (Moreover, rule modifications are given preference over addition or removal of the entire rule.)

Eventually, the subsequent predefined actions are interactive (system) actions, i.e., they involve a human user:

- $@\mathit{guiSelectMod}(ms)$ displays a GUI for choosing from the set of modifications ms. The modification chosen by the user is applied to M.
- $@\mathit{guiEditMod}(m)$ displays a GUI for editing modification m. The resulting modification is applied to M.[1]
- $@\mathit{guiSelectModAtContext}(ms, c)$ projects modifications in ms to c, displays a GUI for choosing among them and applies the chosen modification to M.
- $@\mathit{guiEditModAtContext}(m, c)$ projects modification m to context c, displays a GUI for editing it, and applies the resulting modification to M.

As we define formally in Section 6.1.2, changes of individual actions are not applied directly, but collected into an overall modification which is then applied to M (respecting preferences in case of conflicts as stated above). Before turning to a formal definition of the semantics, we give example policies.

Example 70 (ctd)**.** *Figure 6.1 shows three policies that can be useful for managing inconsistency in our running example. Their intended behavior is as follows.* P_1 *deals with inconsistencies at* C_{lab}*: if an explanation concerns only bridge rules at* C_{lab}*, an arbitrary diagnosis is applied at* C_{lab}*, other inconsistencies are not handled. Applying* P_1 *to* M_2 *removes* r_1 *at* C_{lab}*, the resulting MCS is still inconsistent with inconsistency explanation* e_2*, as only* e_1 *has been automatically fixed.* P_2 *extends* P_1 *by adding an 'inconsistency alert formula' to* C_{lab} *if an inconsistency was automatically repaired at that context. Finally,* P_3 *is a fully manual approach which displays a choice of all minimal diagnoses to the user and applies the user's choice. Note, that we did not combine automatic actions and user-interactions here since this would result in more involved policies (and/or require an iterative methodology; cf. Section 6.2).* □

We refer to the predefined IMPL actions $@\mathit{delRule}$, $@\mathit{addRule}$, $@\mathit{guiSelectMod}$, and $@\mathit{guiEditMod}$ as *core* actions, and to the remaining ones as *comfort* actions. Comfort actions exist for convenience, providing means for projection and for rule modifications. They can be rewritten to core actions as sketched in the following example.

Example 71. *To realize* $@\mathit{applyMod}(M)$ *and* $@\mathit{applyModAtContext}(M, C)$ *using the core language, we replace them by* $\mathit{applyMod}(M)$ *and* $\mathit{applyModAtContext}(M, C)$*, respectively, use rules* (6.2) *from Example 69, and add the following set of rules.*

$$\begin{aligned}
@\mathit{addRule}(R) &\leftarrow \mathit{applyMod}(M),\ \mathit{modAdd}(M, R).\\
@\mathit{delRule}(R) &\leftarrow \mathit{applyMod}(M),\ \mathit{modDel}(M, R).\\
\mathit{projectMod}(M, C) &\leftarrow \mathit{applyModAtContext}(M, C).\\
\mathit{applyMod}(M') &\leftarrow \mathit{applyModAtContext}(M, C),\\
&\qquad \mathit{projectedModId}(M', M, C).
\end{aligned} \tag{6.3}$$

□

This concludes our introduction of the syntax of IMPL. We move on to a formal development of its semantics which so far has only been intuitively conveyed.

[1] It is suggestive to also give the human operator a possibility to abort, causing no modification at all to be made, however we do not specify this here because a useful design choice depends on the concrete application scenario.

Policies (sets of IMPL rules)	Intuitive meaning of rules in each set
$P_1 = \{ \mathit{expl}(E) \leftarrow \mathit{explNeed}(E, R);$	Define domain predicate
$\mathit{expl}(E) \leftarrow \mathit{explForbid}(E, R);$	for explanations.
$\mathit{incNotLab}(E) \leftarrow \mathit{explNeed}(E, R),$	Find out whether one explanation
$\mathit{ruleHead}(R, C, F), C \neq c_{lab};$	only concerns bridge rules at c_{lab}.
$\mathit{incNotLab}(E) \leftarrow \mathit{explForbid}(E, R),$	
$\mathit{ruleHead}(R, C, F), C \neq c_{lab};$	
$\mathit{incLab} \leftarrow \mathit{expl}(E), \mathit{not}\ \mathit{incNotLab}(E);$	
$\mathit{in}(D) \leftarrow \mathit{not}\ \mathit{out}(D), \mathit{diag}(D), \mathit{incLab};$	Guess exactly one diagnosis
$\mathit{out}(D) \leftarrow \mathit{not}\ \mathit{in}(D'),$	if there is a local inconsistency at c_{lab}.
$\mathit{diag}(D), D \neq D', \mathit{incLab};$	
$@\mathit{applyModAtContext}(D, c_{lab}) \leftarrow$	Apply the guessed diagnosis after
$\mathit{useDiag}(D)\}$	projecting it to context c_{lab}.
$P_2 = \{ \mathit{ruleHead}(r_{alert}, c_{lab}, \mathit{alert}) \leftarrow;$	Define new inconsistency alert rule r_{alert}.
$@\mathit{addRule}(r_{alert}) \leftarrow \mathit{incLab}\}$	Add that new rule to c_{lab}.
$\cup\ P_1$	Reuse policy P_1.
$P_3 = \{ \mathit{modset}(md, X) \leftarrow \mathit{diag}(X);$	Create modification set with all diagnoses.
$@\mathit{guiSelectMod}(md) \leftarrow \}$	Let the user choose from that set.

Figure 6.1: Sample IMPL policies for our running example.

6.1.2 Semantics

The semantics of applying an IMPL policy P to a MCS M is defined in three steps:

- *Actions* to be executed are determined by computing a *policy answer set* of P wrt. policy input EDB_M.
- *Effects of actions* are determined by executing actions. This yields modifications (A, R) of M for each action. Action effects can be nondeterministic and thus only be determined by executing respective actions (which is particularly true for user interactions).
- Effects of actions are *materialized* by building the component-wise union over individual action effects and applying the resulting modification to M.

In the remainder of this section, we introduce the necessary definitions for a precise formal account of these steps.

Action Determination.

We define IMPL policy answer sets similar to the stable model semantics [GL91]. Given a policy P and a policy input EDB_M, let id_k be a fixed (built-in) family of one-to-one mappings from k-tuples $c_1, \ldots, c_k$, where $c_i \in \mathit{cons}(P \cup \mathit{EDB}_M)$ for $1 \leq i \leq k$, to a set $C_{id} \subseteq C$ of 'fresh' constants, i.e., disjoint from $\mathit{cons}(P \cup \mathit{EDB}_M)$.[2] Then the *policy base* $B_{P,M}$ of P wrt. EDB_M is the set of ground IMPL atoms and actions, that can be built using predicate symbols from $\mathit{pred}(P \cup \mathit{EDB}_M)$ and terms from $U_{P,M} = \mathit{cons}(P \cup \mathit{EDB}_M) \cup C_{id}$, called the *policy universe*.

The *grounding of* P, denoted by $\mathit{grnd}(P)$, is given by grounding its rules wrt. $U_{P,M}$ as usual. Note that since $\mathit{cons}(P \cup \mathit{EDB}_M)$ is finite, only finitely many mapping functions id_k are

[2]Disjointness ensures finite groundings; without this restriction, for instance the program $\{p(C);\ p(C') \leftarrow \#id_1(C', C)\}$ would not have finite grounding.

used in P. Hence only finitely many constants C_{id} are required, and therefore $U_{P,M}$, $B_{P,M}$, and $grnd(P)$ are finite as well.

An *interpretation* is a set of ground atoms $I \subseteq B_{P,M}$. We say that

- *I satisfies* an atom $a \in B_{P,M}$, denoted $I \models a$, iff (i) a is not a built-in atom and $a \in I$, or (ii) a is a built-in atom of the form $\#id_k(c, c_1, \ldots, c_k)$ and $c = id_k(c_1, \ldots, c_k)$;
- *I satisfies* a set of atoms $A \subseteq B_{P,M}$, denoted $I \models A$, iff $I \models a$ for all $a \in A$;
- *I satisfies* the body of rule r, denoted $I \models B(r)$, iff $I \models a$ for every $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and
- *I satisfies* a ground rule r, denoted $I \models r$, iff $I \models H(r)$ or $I \not\models B(r)$.

Eventually, *I is a model of P*, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. The *FLP-reduct* of P wrt. an interpretation I, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$.[3]

Definition 44 (Policy Answer Set)**.** *Let M be an MCS, let P be an* IMPL *policy, and let EDB_M be a policy input wrt. M. Then an interpretation $I \subseteq B_{P,M}$ is a* policy answer set of P for *EDB_M iff I is a $\subseteq$-minimal model of $fP^I \cup EDB_M$.*

We denote by $\mathcal{AS}(P \cup EDB_M)$ the set of all policy answer sets of P for EDB_M.

Effect Determination.

We define the effects of action predicates $@a \in Act$ by nondeterministic functions $f_{@a}$. Nondeterminism is required for interactive actions. An action is evaluated wrt. an interpretation of the policy and yields an effect according to its type: the effect of a system action is a modification (A, R) of the MCS, in the following sometimes denoted *system modification*, while the effect of a rule action is a *rule modification* $(A, R)_r$ wrt. a bridge rule r of M, i.e., in this case A is a set of bridge rule body literals to be added to r, and R is a set of bridge rule body literals to be removed from r.

Definition 45. *Given an interpretation I, and a ground action α of form $@a(t_1, \ldots, t_k)$, the* effect of α wrt. *I is given by $\mathit{eff}_I(\alpha) = f_{@a}(I, t_1, \ldots, t_k)$, where $\mathit{eff}_I(\alpha)$ is a system modification if α is a system action, and a rule modification if α is a rule action.*

Action predicates of the IMPL core fragment have the following semantic functions.

- $f_{@delRule}(I, r) = (\emptyset, \{rule_I(r)\})$.
- $f_{@addRule}(I, r) = (\{rule_I(r)\}, \emptyset)$.
- $f_{@guiSelectMod}(I, ms) = (A, R)$ where $(A, R) \in modset_I(ms) = \{(A_1, R_1), \ldots\}$ is the user's selection after being displayed a choice among all possible modifications.
- $f_{@guiEditMod}(I, m) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is pre-loaded with modification $(A, R) = mod_I(m)$.

Note that the effect of any core action in I can be determined independently from the presence of other core actions in I, and rule modifications are not required to define the semantics of core actions. However, rule modifications are needed to capture the effect of *comfort* actions. Moreover, adding and deleting rule conditions, and making a rule unconditional can modify the same rule, therefore such action effects yield accumulated rule modifications.

More specifically, the semantics of IMPL comfort actions is defined as follows:

[3]We use the FLP reduct [FPL11] for compliance with ACTHEX (used for realization in Section 6.3), but for the language considered, the Gelfond-Lifschitz reduct would yield an equivalent definition.

- $f_{@\mathit{delRuleCondition}^+}(I, r, c, b) = (\emptyset, \{(c : b)\})_r$.
- $f_{@\mathit{delRuleCondition}^-}(I, r, c, b) = (\emptyset, \{\mathbf{not}\ (c : b)\})_r$.
- $f_{@\mathit{addRuleCondition}^+}(I, r, c, b) = (\{(c : b)\}, \emptyset)_r$.
- $f_{@\mathit{addRuleCondition}^-}(I, r, c, b) = (\{\mathbf{not}\ (c : b)\}, \emptyset)_r$.
- $f_{@\mathit{makeRuleUnconditional}}(I, r) = (\emptyset, \{(c_1 : p_1), \ldots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (c_m : p_m)\})_r$ for r of the form (2.1).
- $f_{@\mathit{applyMod}}(I, m) = \mathit{mod}_I(m)$.
- $f_{@\mathit{applyModAtContext}}(I, m, c) = \mathit{mod}_I(m)|_c$.
- $f_{@\mathit{guiSelectModAtContext}}(I, ms, c) = (A', R')$ where (A', R') is the user's selection after being displayed a choice among all modifications in $\{(A'_1, R'_1), \ldots\} = \mathit{modset}_I(ms)|_c$.
- $f_{@\mathit{guiEditModAtContext}}(I, m, c) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is pre-loaded with modification $\mathit{mod}_I(m)_c$.

In practice, however, it is not necessary to implement action functions on the level of rule modifications, since a policy in the comfort fragment can equivalently be rewritten to the core fragment (which does not rely on rule modifications). Example 71 already sketched a rewriting for $@\mathit{applyMod}$ and $@\mathit{applyModAtContext}$. In Section 6.4 we provide a rewriting from IMPL to the IMPL core fragment.

The effects of user-defined actions have to comply to Definition 45.

Effect Materialization.

Once the effects of all actions in a selected policy answer set have been determined, an overall modification is computed by the component-wise union over all individual modifications. This overall modification is then materialized in the MCS.

Given a MCS M and a policy answer set I (for a policy P and a corresponding policy input EDB_M), let I_M, respectively I_R, denote the set of ground system actions, respectively rule actions, in I. Then, $M_{\mathit{eff}} = \{\mathit{eff}_I(\alpha) | \alpha \in I_M\}$ is the set of effects of system action atoms in I, and $R_{\mathit{eff}} = \{\mathit{eff}_I(\alpha) | \alpha \in I_R\}$ is the set of effects of rule actions in I. Furthermore, $\mathit{Rules}(R_{\mathit{eff}}) = \{r \mid (A, R)_r \in R_{\mathit{eff}}\}$ is the set of bridge rules modified by R_{eff}, and for every $r \in \mathit{Rules}(R_{\mathit{eff}})$, let $\mathcal{R}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} R$, respectively $\mathcal{A}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} A$, denote the union of rule body removals, respectively body additions, wrt. r in R_{eff}.

Definition 46. *Let M be a MCS, let P be an* IMPL *policy, and let I be a policy answer set of P for a policy input EDB_M wrt. M. Then the* materialization of I in M *is the MCS $M' = M[br']$ obtained from M by replacing its set of bridge rules br_M by the set*

$$br' = (br_M \setminus \mathcal{R} \cup \mathcal{A}) \setminus \mathit{Rules}(R_{\mathit{eff}}) \cup \mathcal{M},$$

where

$$\mathcal{R} = \bigcup_{(A,R) \in M_{\mathit{eff}}} R, \qquad \mathcal{A} = \bigcup_{(A,R) \in M_{\mathit{eff}}} A, \text{ and}$$

$$\mathcal{M} = \{(k{:}s) \leftarrow \mathit{Body} \mid r \in \mathit{Rules}(R_{\mathit{eff}}), r \in br_k, h_b(r) = s, \mathit{Body} = B(r) \setminus \mathcal{R}_r \cup \mathcal{A}_r\}.$$

Note that, by definition, the addition of bridge rules has precedence over removal, and the addition of body literals similarly has precedence over removal. There is no particular reason for this choice; one just has to be aware of it when specifying a policy. Apart from that, no order for evaluating individual actions is specified or required.

Eventually, we can define modifications of a MCS that are accepted by a corresponding IMPL policy.

Definition 47. *Given a MCS M, an* IMPL *policy P, and a policy input EDB_M wrt. M, a modified MCS M'* is admissible wrt. M, P and EDB_M *iff M' is the materialization of some policy answer set $I \in \mathcal{AS}(P \cup EDB_M)$.*

Example 72 (ctd). *Evaluating $P_2 \cup EDB_{M_2}$ yields four policy answer sets; one is*

$$I_1 = \{expl(e_1), expl(e_2), incNotLab(e_2), incLab, in(d_1), out(d_2), out(d_3), out(d_4), useOne, ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_1, c_{lab})\}.$$

(Note that we omitted EDB_{M_2} which is obviously contained in every answer set.) From I_1 we obtain a single admissible modification of M_2 wrt. P_2: add bridge rule r_{alert} and remove r_1.

The other policy answer sets are

$$\{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), in(d_2), out(d_3), out(d_4), useOne, ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_2, c_{lab})\},$$

$$\{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), out(d_2), in(d_3), out(d_4), useOne, ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_3, c_{lab})\}, \text{ and}$$

$$\{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), out(d_2), out(d_3), in(d_4), useOne, ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_4, c_{lab})\}.$$

Evaluating $P_3 \cup EDB_{M_2}$ yields one policy answer set, which is $I_2 = EDB_{M_2} \cup \{modset(md, d_1), modset(md, d_2), modset(md, d_3), modset(md, d_4), @guiSelect\text{-}Mod(md)\}$. Determining the effect of I_2 involves user interaction; thus multiple materializations of I_2 exist. For instance, if the user chooses to ignore Sue's allergy and birth date (and probably imposes additional monitoring on Sue), then we obtain an admissible modification of M which adds the unconditional version of r_6 and removes r_1. □

6.2 Methodologies of Applying IMPL and Realization

Based on the simple system design shown in Figure 6.2, we next briefly discuss elementary methodologies of applying IMPL for the purpose of integrating MCS reasoning with potential user interaction in case of inconsistency.

We maintain a representation of the MCS together with a *store of modifications*. The *semantics evaluation* component performs reasoning tasks on the MCS and invokes the *inconsistency manager* in case of an inconsistency. This inconsistency manager uses the *inconsistency analysis* component[4] to provide input for the *policy engine* which computes policy answer sets of a given IMPL *policy* wrt. the MCS and its inconsistency analysis result. This policy evaluation step results in action executions potentially involving user interactions and causes changes to the store of modifications, which are subsequently materialized. Finally the inconsistency manager hands control back to the semantics evaluation component.

6.2.1 Reasoning Modes

Principal modes of operation, and their merits, are the following.

[4]For realizations of this component we refer to [BEFS10, EFSW10].

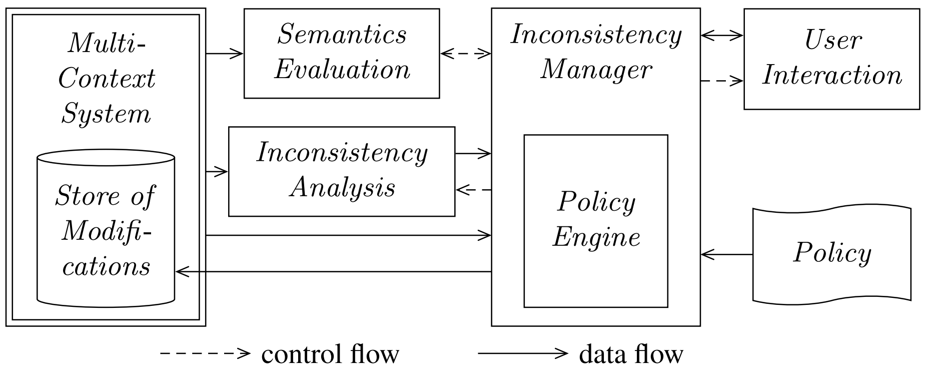


Figure 6.2: Policy integration data flow and control flow block diagram.

Reason and Manage Once. This mode of operation evaluates the policy once, if the effect materialization does not repair inconsistency in the MCS, no further attempts are made and the MCS stays inconsistent. While simple, this mode may not be satisfactory in practice.

Manage Once with Ranked Repair Attempts. In this strategy, the result of evaluating a policy wrt. an inconsistency does not yield a single attempt for restoring consistency; instead it yields multiple attempts, each with a separate set of actions.

This requires to augment actions of the policy language by an *attempt ranking* which specifies an order of actions to be applied: first only the highest-ranked modifications are used; if this repairs the system the process finishes. Otherwise the highest-ranked modifications are removed and the process restarts, looking for the set of actions with the second-best rank and so on. This is repeated until either the system becomes consistent (success), or until no lower rank exists (failure).

Example 73. *An inconsistency management strategy generates some sophisticated policy-generated set of modifications which is attempted first. If this first attempt fails to restore consistency, the policy uses an element of the set of minimal diagnoses as a fallback modification. This guarantees to restore inconsistency. Additionally, this second attempt adds a bridge rule to some context, to notify contexts (and thus users and operators of the MCS). This way reasoning with the system is never impossible due to inconsistency, however consistency may come at the cost of being a "fallback consistency".* □

User interaction in this strategy demands special considerations: (i) user modifications could be the same for all attempt ranks, such that the user does not need to care about ranks, or (ii) the user can produce sets of modifications for multiple ranks. The first option seems easier to use, while the second provides more possibilities to the user and to inconsistency management as a whole.

Overall this mode of using IMPL requires only one reasoning step and easily guarantees termination of the inconsistency management process.

Reason and Manage Iteratively. Another way to deal with failure to restore consistency is to simply invoke policy evaluation again on the modified but still inconsistent system. This is useful if user interaction may involve trial-and-error, especially if multiple inconsistencies occur: some might be more difficult to counteract than others.

Another positive aspect of iterative policy evaluation is, that it allows for policies to be structured, e.g., as follows: (a) classify inconsistencies into automatically versus manually repairable; (b) apply actions to repair one of the automatically repairable inconsistencies; (c) if such inconsistencies do not exist: perform user interaction actions to repair one (or all) of the manually repairable inconsistencies. Such policy structuring follows a divide-and-conquer approach, trying to focus on individual sources of inconsistency and to disregard interactions between inconsistencies as much as possible. If user interaction consists of trial-and-error bug-fixing, fewer components of the system are changed in each iteration, and the user starts from a situation where only critical (i.e. not automatically repairable) inconsistencies are present in the MCS. Moreover, such policies may be easier to write and maintain. On the other hand, termination of iterative methodologies is not guaranteed, and neither is consistency of the MCS. However, one can enforce termination by limiting the number of iterations, possibly by extending IMPL with a *control action* that configures this limit. Consistency of the MCS can be ensured by applying a 'fallback diagnosis'; hence making the system consistent under all circumstances (system consistency then also implies that the iteration terminates).

Manage Iteratively first Automatic then User. This is a specialization of the above 'Manage Iteratively' strategy, with the goal of adding more structure to the inconsistency management process. We accomplish this by deliberately using iterations as a procedural aspect controlled by the declarative policy language. As the name suggests, a policy following this strategy emits either only modification actions, or only user interactions.

This suggests to use the following structure for a policy: detected inconsistencies are categorized as automatically repairable or not, if there exist automatically repairable ones, actions to repair them are emitted, otherwise user interactions for the remaining inconsistencies are emitted. (Additionally, the policy could only emit repair actions for single automatically repairable inconsistencies in one iteration.)

This kind of a policy has the benefit of doing one thing at a time instead of doing everything at once. Therefore, identifying problems (i.e., debugging policies or the whole inconsistency management process) is more easily captured than in the more general case.

Furthermore, if the user interaction consists of trial-and-error bug-fixing, fewer components of the system are changed in each iteration. This should have favorable effects on the performance and maintainability of inconsistency management.

6.2.2 Properties and Extensions

Here we discuss additional properties and features that could be advantageous in practical applications. (And could easily be added to IMPL.)

Iteration-persistent Storage. In iterative mode it may be useful to access information from previous iterations. We call this persistent storage. For instance, a persistent storage (reminiscent of an RDF triplestore) can be added to IMPL as follows: (a) we add a (persistent) triplestore to the policy engine, (b) define actions $@kbAdd(S, P, O)$ and $@kbDel(S, P, O)$ s.t. $@kbAdd$ stores and $@kbDel$ removes triples, and finally (c) define a new ternary predicate $kbTriple(S, P, O)$ that is added to EDB_M for each stored triple.

Stable Identifiers. When an IMPL policy is applied to an MCS, it might remove, add, or change bridge rules. In an iterative mode of operation, it would be useful if changing a bridge rule did not change its identifier.

For example, the bridge rule r_{alert} might be added to M_2 by our example policy P_2 (see Example 72), which yields a new MCS M_2'. If we apply IMPL to M_2', the subsequent $EDB_{M_2'}$ should then use again r_{alert} to identify that bridge rule. If this is the case, we can reason about the existence of that rule in our policy.

When using iteration-persistent storage, we can store rule-identifiers across iterations; however this only makes sense if identifiers remain the same across iterations.

Therefore stable identifiers are a desirable property. This property can be added to IMPL as a simple condition on added and modified rules, namely that they have an associated identifier which remains the same for subsequently created EDB_M's. (In Section 6.4 we will take particular care to provide stable identifiers.)

Automatic Modifications vs User Interactions. In the current declarative semantics definition, a rule might be 'simultaneously' modified both by a user interaction and by another action. However, this means that a modification done by a user can be undone by another action that was triggered by the policy. Therefore, to achieve a system with intuitively clear effects of a user's actions, user interaction actions should be limited to rules that are not modified by other actions.

6.3 Realizing IMPL in acthex

In this section, we demonstrate how IMPL can be realized using ACTHEX. First we give preliminaries about ACTHEX, which is a logic programming formalism that extends HEX programs with executable actions. We then show how to implement the core IMPL fragment by rewriting it to ACTHEX in Section 6.3.2.

6.3.1 Preliminaries on acthex

The ACTHEX formalism [BEFI10] generalizes HEX programs [EIST05] by adding an environment and dedicated action atoms to heads of rules.

An ACTHEX program operates on an *environment*; this environment can influence external sources in ACTHEX, and it can be modified by the execution of actions. The internal structure of an environment is not specified by ACTHEX, it depends on the concrete application scenario.

Example 74. *In a robotics application the environment represents the state of the world, including the state of the robot and its location in the world. External atoms use the environment to represent sensor readings. Actions modify the environment by controlling the robot.*

In a belief revision scenario, the environment represents the knowledge base that shall be revised. External atoms import information from the knowledge base into the program, while actions modify the knowledge base by adding or retracting knowledge. □

Syntax.

As ACTHEX is an extension of HEX, we here only give syntactic elements that ACTHEX adds to HEX. We assume that the set of constants $\mathcal{C}$ contains a finite subset of consecutive integers $\{0, \ldots, n_{max}\}$. By $\mathcal{A}$ we denote the set of action predicate names. We assume that $\mathcal{A}$ is disjoint with the sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$; and we prefix action predicate names with '#'.

An action atom is of the form

$$\#g[Y_1, \ldots, Y_n]\{o, r\}[w : l]$$

where $\#g$ is an action predicate name, $Y_1, \ldots, Y_n$ is a list of terms (called input list), and each action predicate $\#g$ has fixed length $in(\#g) = n$ for its input list. Attribute $o \in \{b, c, c_p\}$ is called the *action option*; depending on o the action atom is called *brave, cautious, and preferred cautious*, respectively. Attributes r, w, and l are called *precedence*, *weight*,[5] and *level*[5] of $\#g$, denoted by $prec(a)$, $weight(a)$, and $level(a)$, respectively. They are optional and range over variables and positive integers.

[5]Weight and level have a similar intuition as the corresponding attributes of weak constraints in ASP [BLR97].

A *rule* r is of the form $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m$, where $m, n, k \geq 0$, $m \geq n$, $\alpha_1, \ldots, \alpha_k$ are atoms or action atoms, and $\beta_1, \ldots \beta_m$ are atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. An ACTHEX *program* is a finite set P of rules.

Example 75. *The* ACTHEX *program*

$$
\begin{aligned}
&\{night \ \vee \ day \leftarrow;\\
&\ \#robot[goto, charger]\{b, 1\} \leftarrow \&sensor[bat](low);\\
&\ \#robot[clean, kitchen]\{c, 2\} \leftarrow night;\\
&\ \#robot[clean, bedroom]\{c, 2\} \leftarrow day\}
\end{aligned}
$$

uses action atom $\#robot$ to command a robot, and an external atom $\&sensor$ to obtain sensor information. Precedence 1 of action atom $\#robot[goto, charger]\{b, 1\}$ makes the robot recharge its battery before executing cleaning actions, if necessary. □

Semantics.

We here first give an intuitive overview of the semantics and then precise formal definitions. Note that our presentation of the influence of an environment on external atom semantics and execution schedule selection is slightly different than in [BEFI10].

Intuitively, an ACTHEX program P is evaluated wrt. an *external environment* E using the following steps: (i) *answer sets* of P are determined wrt. E, the set of *best models* is a subset of the answer sets determined by an objective function; (ii) one best model is selected, and one *execution schedule* S is generated for that model (although a model may give rise to multiple execution schedules); (iii) the *effects of action atoms* in S are applied to E in the order defined by S, yielding an updated environment E'; and finally (iv) the process may be iterated starting at (i), unless no actions were executed in (iii) which terminates an iterative evaluation process. Importantly, the environment can only be changed by action execution, i.e., in step (iii), and we assume it remains constant throughout the other steps. Formally the ACTHEX semantics is defined as follows.

Given an ACTHEX program P the *Herbrand base* HB_P of P is the set of all possible ground versions of atoms, external atoms, and action atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. Given a rule $r \in P$, the grounding $grnd(r)$ of r is defined accordingly; the grounding of P is given as the grounding of all its rules. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ are implicitly given by P.

An *interpretation I relative to P* is any subset $I \subseteq HB_P$ containing ordinary atoms and action atoms. We say that I is a *model* of an ordinary or action atom $a \in HB_P$, denoted by $I \models a$, iff $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n{+}m{+}2)$-ary Boolean function $f_{\&g}$, assigning each tuple $(E, I, y_1, \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, x_i, $y_j \in \mathcal{C}$, $I \subseteq HB_P$, and environment E. Note that this slightly generalizes the external atom semantics such that their truth value may depend on the environment E. This was left implicit in [BEFI10].

We say that an interpretation I is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ wrt. environment E, denoted $I, E \models a$, iff $f_{\&g}(E, I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$. Let r be a ground rule. We define

- $I, E \models H(r)$ iff there is some $a \in H(r)$ such that $I, E \models a$,
- $I, E \models B(r)$ iff $I, E \models a$ for all $a \in B^+(r)$ and $I, E \not\models a$ for all $a \in B^-(r)$, moreover
- $I, E \models r$ iff $I, E \models H(r)$ or $I, E \not\models B(r)$.

We say that I is a *model* of P wrt. E, denoted by $I, E \models P$, iff $I, E \models r$ for all $r \in grnd(P)$. The *FLP-reduct* of P wrt. I and E, denoted as $fP^{I,E}$, is the set of all $r \in grnd(P)$ such that $I, E \models B(r)$. Eventually we can define ACTHEX answer sets as follows.

Definition 48. *Let P be an* ACTHEX *program and let E be an environment. Then $I \subseteq HB_P$ is an* answer set *of P wrt. E iff I is a $\subseteq$-minimal model of $fP^{I,E}$.*

Note that, as for HEX programs we need the FLP-reduct [FPL11], which is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary programs, and in ACTHEX ensures answer-set minimality in the presence of external atoms (see [EIST06] for details). We denote by $\mathcal{AS}(P, E)$ the collection of all answer sets of P wrt. E.

The set of *best models* of P contains those answer sets that minimize an objective function which we define in the following. Let P be an ACTHEX program, then $AA^g_w(P)$ denotes the set of action atoms in $grnd(P)$ with explicit weight and level values. An auxiliary function f_P is recursively defined as follows:

$$\begin{aligned} &f_P(1) = 1, \\ &f_P(n) = f_P(n-1) \cdot (|AA^g_w(P)|) \cdot w^P_{max} + 1, && \text{for } n > 1. \end{aligned}$$

Given an answer set I, the objective function $H_P(I)$ is then defined as

$$H_P(I) = \sum_{i=1}^{l^P_{max}} (f_P(i) \cdot \sum_{a \in M^P_i(I)} weight(a))$$

where

$$w^P_{max} = \max_{a \in AA^g_w(P)} weight(a) \quad \text{and} \quad l^P_{max} = \max_{a \in AA^g_w(P)} level(a)$$

denote the maximum weight and maximum level over weighted action atoms in $grnd(P)$, respectively; and

$$M^P_i(I) = \{\#b[\vec{\mathrm{Y}}]\{o, r\}[w{:}i] \in I\}$$

denotes the set of action atoms with level i that appear in I.

Definition 49. *Let P be an* ACTHEX *program and let E be an environment. Then the set of* best models *of P wrt. E, denoted $\mathcal{BM}(P, E)$, contains all answer sets $I \in \mathcal{AS}(P, E)$ that minimize the objective function $H_P(I)$.*

Intuitively, an answer set I will be among the best models if no other answer set contains only actions with a lower level, and if no other answer set I' that contains only actions with the same level as I has a smaller weight of all contained actions. (See also a similar definition for weak constraint semantics for disjunctive datalog in [BLR97].)

An action $a = \#g[y_1, \ldots, y_n]\{o, r\}[w : l]$ with option o and precedence r is *executable in I wrt. P and E* iff (i) a is brave and $a \in I$, or (ii) a is cautious and $a \in B$ for every $B \in \mathcal{AS}(P, E)$, or (iii) a is preferred cautious and $a \in B$ for every $B \in \mathcal{BM}(P, E)$. An *execution schedule* of a best model I is a sequence of all actions executable in I, such that for all action atoms $a, b \in I$, if $prec(a) < prec(b)$ then a has a lower index in the sequence than b. We denote by $\mathcal{ES}_{P,E}(I)$ the set of all execution schedules of a best model I wrt. ACTHEX program P and environment E; formally

$$\mathcal{ES}_{P,E}(I) = \big\{[a_1, \ldots, a_n] \mid prec(a_i) \leq prec(a_j), \text{for all } 1 \leq i < j \leq n\big\}$$

where $\{a_1, \ldots, a_n\}$ is the set of action atoms that are executable in I wrt. P and E.

Example 76. *In Example 75, if the robot has low battery, then $\mathcal{AS}(P,E) = \mathcal{BM}(P,E)$ contains models*

$$I_1 = \{night, \#robot[clean, kitchen]\{c,2\}, \#robot[goto, charger]\{b,1\}\},\ \textit{and}$$
$$I_2 = \{day, \#robot[clean, bedroom]\{c,2\}, \#robot[goto, charger]\{b,1\}\}.$$

We have $\mathcal{ES}_{P,E}(I_1) = \{\#robot[goto, charger]\{b,1\}, \#robot[clean, bedroom]\{c,2\}\}$. □

Given a model I, the *effect of executing a ground action* $\#g[y_1,\ldots,y_m]\{o,p\}[w:l]$ on an environment E wrt. I is defined for each action predicate name $\#g$ by an associated $(m{+}2)$-ary function $f_{\#g}$ which returns an updated environment $E' = f_{\#g}(E, I, y_1, \ldots, y_m)$. Correspondingly, given an execution schedule $S = [a_1,\ldots,a_n]$ of a model I, the *execution outcome* of S in environment E wrt. I is defined as $EX(S,I,E) = E_n$, where $E_0 = E$, and $E_{i+1} = f_{\#g}(E_i, I, y_1, \ldots, y_m)$, given that a_i is of the form $\#g[y_1,\ldots,y_m]\{o,p\}[w:l]$. Intuitively the initial environment $E_0 = E$ is updated by each action in S in the given order. The set of possible execution outcomes of a program P on an environment E is denoted as $\mathcal{EX}(P,E)$, and formally defined as

$$\mathcal{EX}(P,E) = \{EX(S,I,E) \mid S \in \mathcal{ES}_{P,E}(I) \text{ where } I \in \mathcal{BM}(P,E)\}.$$

In practice, one usually wants to consider a single execution schedule. This requires the following decisions during evaluation: (i) to select one best model $I \in \mathcal{BM}(P,E)$, and (ii) to select one execution schedule $S \in \mathcal{ES}_{P,E}(I)$. Finally, one can then execute S and obtain the new environment $E' = EX(S,I,E)$.

6.3.2 Rewriting the IMPL Core Fragment to ACTHEX

Using ACTHEX for realizing IMPL is a natural and reasonable choice, because ACTHEX already natively provides several features necessary for IMPL: external atoms can be used to access information from a MCS, and ACTHEX actions come with weights for creating ordered execution schedules for actions that occur in the same answer set of an ACTHEX program. Based on this, IMPL can be implemented by a rewriting to ACTHEX, such that ACTHEX actions realize IMPL actions; ACTHEX external predicates provide information about the MCS to the IMPL policy; and ACTHEX external predicates realize the value invention built-in predicates.

We next describe a rewriting from the IMPL core language fragment to ACTHEX. We assume that the environment E contains a pair $(\mathcal{A},\mathcal{R})$ of sets of bridge rules, and an encoding of the MCS M (suitable for an implementation of the external atoms introduced below[6]). A given IMPL policy P wrt. the MCS M is then rewritten to an ACTHEX program P^{act} as follows.

1. Each core IMPL action $@a(t)$ in the head of a rule of P is replaced by a brave ACTHEX action $\#a[t]\{b,2\}$ with precedence 2. These ACTHEX actions implement semantics of the respective IMPL actions according to Def. 45: the interpretation I and the original action's argument t are used as input, the effects are accumulated as $(\mathcal{A},\mathcal{R})$ in E.

2. Each IMPL built-in $\#id_k(C, c_1, \ldots, c_k)$ in P is replaced by an ACTHEX external atom $\&id_k[c_1,\ldots,c_k](C)$. The family of external atoms $\&id_k[c_1,\ldots,c_k](C)$ realizes value invention and has the associated semantics function $f_{\&id_k}$ such that $f_{\&id_k}(E, I, c_1, \ldots, c_k, C) = 1$ for one constant $C = auxc\_c_1\_\ldots\_c_k$ created from the constants in tuple $c_1,\ldots,c_k$.

[6]E.g., in the syntax used by the MCS-IE system, see Chapter 4 and [BEFS10], which provide the corresponding policy input.

3. We add to P^{act} a set P_{in} of ACTHEX rules containing (i) rules that use, for every $p \in C_{res} \setminus \{modset\}$, a corresponding external atom to 'import' a faithful representation of M and its inconsistency analysis (recall that M is encoded in E), and (ii) a preparatory action *#reset* with precedence 1, and a final action *#materialize* with precedence 3:

$$\begin{aligned} P_{in} = \{&p(\vec{t}) \leftarrow \&p[](\vec{t}) \mid p \in C_{res} \setminus \{modset\}\} \cup \\ &\{\#reset[]\{b,1\}; \#materialize[]\{b,3\}\}, \end{aligned}$$

where $\vec{t}$ is a vector of distinct variables of length equal to the arity a of p (i.e., $1 \leq a \leq 3$).

The first two steps transform IMPL actions into ACTHEX actions and $\#id_k$-value invention into external atom calls. The third step essentially creates policy input facts from ACTHEX external sources. External atoms in P_{in} return a representation of M and analyze inconsistency in M, providing minimal diagnoses and minimal explanations. Thus, the respective rules in P_{in} yield an extension of the corresponding reserved predicates which is a faithful representation of M. Moreover, action *#reset* resets the modification $(\mathcal{A}, \mathcal{R})$ stored in E to $(\emptyset, \emptyset)$.[7] Action *#materialize* materializes the modification $(\mathcal{A}, \mathcal{R})$ (as accumulated by actions of precedence 2) in the MCS M (which is part of E).

Example 77 (ctd)**.** *The translation of policy P_3 from Ex. 70 to* ACTHEX *contains the following rules:*

$$P_3^{act} = P_{in} \cup \big\{modset(md, X) \leftarrow diag(X); \#guiSelectMod[md]\{b,2\}\big\}$$

where

$$\begin{aligned} P_{in} = \big\{&ruleHead(R,C,S) \leftarrow \&ruleHead[](R,C,S); \\ &ruleBody^+(R,C,S) \leftarrow \&ruleBody^+[](R,C,S); \\ &ruleBody^-(R,C,S) \leftarrow \&ruleBody^-[](R,C,S); \\ &\dots \\ &\#reset[]\{b,1\}; \#materialize[]\{b,3\}\big\}. \end{aligned}$$

□

Note that actions in the rewriting have no weights; and thus all answer sets are best models. For obtaining an admissible modification, any policy answer set can be chosen, and any execution schedule can be used.

Proposition 20. *Let M be a MCS, let P be a core* IMPL *policy, let EDB_M be a policy input wrt. M, let P^{act} be as given above, and let E be an environment containing M and $(\emptyset, \emptyset)$. Then every execution outcome $E' \in \mathcal{EX}(P^{act} \cup EDB_M|_{B_{Aux}}, E)$ contains an admissible modification M' of M wrt. P and EDB_M.*

Proof. In this proof we denote by $\mathcal{AS}_I$ the IMPL policy answer set function, and by $\mathcal{AS}_A$ the ACTHEX answer set function. Admissible modifications of IMPL are defined using $\mathcal{AS}_I$, and execution outcomes of ACTHEX are defined using $\mathcal{AS}_A$, therefore we first establish a relationship between answer sets

$$I_I \in \mathcal{AS}_I(P \cup EDB_M) \text{ and } I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M|_{B_{Aux}}, E).$$

Let $P_{in}{}' = \{p(\vec{t}) \leftarrow \&p[](\vec{t}) \mid p \in C_{res} \setminus \{modset\}\}$ (recall that C_{res} denotes reserved policy input predicate names which we reuse in our transformation to define external atoms for inconsistency analysis on M which is part of E). The semantics of the external atoms $\&p[](\vec{t})$ of

[7]This reset is necessary if a policy is applied repeatedly, as discussed in Section 6.2.1, i.e., in iterative reasoning modes.

our translation are independent from the answer set; they depend only on inconsistency analysis results on MCS M which is encoded in E. Thus, and by the definition of the semantics of these atoms in our transformation, $\mathcal{AS}_A({P_{in}}', E) = \{EDB_M|_{B_M}\}$. Therefore, and since ${P_{in}}' \subseteq P^{act}$, every $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M|_{B_{Aux}}, E)$ is such that $EDB_M|_{B_M} \subseteq I_A$. Because $EDB_M = EDB_M|_{B_M} \cup EDB_M|_{B_{Aux}}$ we get that $\mathcal{AS}_A(P^{act} \cup EDB_M|_{B_{Aux}}, E) = \mathcal{AS}_A(P^{act} \cup EDB_M, E)$ and therefore

$$I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M, E).$$

We next show the following relationship between IMPL answer sets and ACTHEX answer sets on the rewritten program: given a set A of action atoms $@\alpha(t)$ where $@\alpha \in Act$, $t \in C$, we show that $I_I \in \mathcal{AS}_I(P \cup EDB_M)$ iff $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M, E)$ where $I_I = I \cup A$ and

$$I_A = I \cup \{\#\alpha[t]\{b, 2\} \mid @\alpha(t) \in A\} \cup \{\#reset[]\{b, 1\}, \#materialize[]\{b, 3\}\}$$

and I is a set of ground ordinary atoms (i.e., I neither contains IMPL actions nor ACTHEX actions) with $EDB_M \subseteq I$. The transformation (in particular item 2. on page 127) replaces all built-ins by an external computation that exactly realizes semantics of the replaced built-in (wlog. we assume that $id_k(c_1, \dots, c_k) = auxc\_c_1\_\dots\_c_k$; if this is not the case, the answer sets coincide modulo auxiliary constant replacement). Rules in P_{in} are always satisfied by I_A as it contains the $\#reset$ and $\#materialize$ actions and as it contains EDB_M. Everything else (i.e., rule bodies and rule heads) in P^{act} is equal to P, modulo action renaming, and satisfaction of rules is defined equally in IMPL and ACTHEX. Furthermore, both semantics define answer sets to be minimal models of the reduct, and (in the definition of IMPL policy answer sets) $I_I \models fP^{I_I} \cup EDB_M$ iff $I_I \models f(P \cup EDB_M)^{I_I}$. Therefore the following intermediate result holds: $I_I \in \mathcal{AS}_I(P \cup EDB_M)$ iff $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M, E)$, with I_I and I_A as introduced above. As actions in P^{act} have no weight and no level, all answer sets are best models. An execution schedule of an answer set I_A first executes $\#reset$, then executes actions that originated in IMPL actions, and finally executes $\#materialize$. The reset action sets $(\mathcal{A}, \mathcal{R})$ in E to $(\emptyset, \emptyset)$. The ACTHEX actions, which are created by the transformation from IMPL actions, realize by their definition the semantics of their corresponding IMPL actions and they accumulate the resulting sets of added and removed bridge rules in $(\mathcal{A}, \mathcal{R})$. Before executing $\#materialize$, we have that $(\mathcal{A}, \mathcal{R}) = M_{\mathit{eff}}$ (for M_{eff} see Section 6.1.2). Then $\#materialize$ modifies M in E to yield a materialization M' of I_I in M and therefore an admissible modification of M. □

The results of this section can be used to realize the full IMPL language, using the rewriting technique described in the next section.

6.4 Rewriting IMPL to the IMPL Core Fragment

In this section we provide a rewriting from the full IMPL language to the IMPL core fragment. This allows us to realize the whole IMPL language using the ACTHEX rewriting for the IMPL core fragment.

Our rewriting will be 'identifier-neutral' in the sense that if the original policy would have created a rule with identifier r, the rewritten policy creates the rule exactly with the same identifier. Furthermore rule modifications are realized by removing the original rule and adding a modified version. Here, again, the rewritten policy uses the original identifier to create the modified rule. As a consequence, our rewriting can be used if stable identifiers are required (see Section 6.2.2 for this property and its benefits).

For our rewriting, it is furthermore important that user interactions are limited to rules that are not modified by other actions. This restriction is useful in practice and has been discussed in Section 6.2.2.

For this purpose we introduce auxiliary predicates and constants which do not occur anywhere in a policy before rewriting. Given an IMPL policy P and a policy input EDB_M, we first define the set of *critical constants* which cannot be freely used by the rewriting: $\mathit{critical}(P \cup \mathit{EDB}_M) = C_M \cup C_{\mathit{res}} \cup \mathit{cons}(P \cup \mathit{EDB}_M)$.

Example 78 (ctd.). *We have* $\mathit{critical}(P_1 \cup \mathit{EDB}_{M_2}) = C_{\mathit{res}} \cup \{c_{db}, c_{onto}, c_{lab}, c_{dss}, r_1, \ldots, r_6, d_1, \ldots, d_4, e_1, e_2, \mathit{expl}, \mathit{incNotLab}, \mathit{incLab}, \mathit{in}, \mathit{out}, \mathit{useOne}\}$. *(See Example 68 and Figure 70 for* EDB_{M_2} *and* P_1.*)* □

Without loss of generality we assume that the following sets of 'fresh' constants are disjoint with $\mathit{critical}(P \cup \mathit{EDB}_M)$:

$$\{c' \mid c \in C_M\} \cup \{ra_\alpha \mid \alpha \in \mathit{Act}\} \cup \{\mathit{map}, \mathit{modifiedRule}, \mathit{add}^+, \mathit{add}^-, \mathit{del}^+, \\ \mathit{del}^-, \mathit{cm}_{id}, \mathit{csm}_{id}, \mathit{pm}_{id}, \mathit{psm}_{id}, \mathit{cleanMod}, \mathit{cleanedModId}, \mathit{cleanModSet}, \\ \mathit{cleanedModSetId}, \mathit{projectMod}, \mathit{projectedModId}, \mathit{projectModSet}, \mathit{projectedModSetId}\}.$$

Given a set P of IMPL rules, we define the replacement function $\mathit{tr}_{\mathit{repl}}(P)$ which replaces every constant $c \in C_M$ in every rule in P by its corresponding constant c' and returns the resulting set of rules. Note that facts are also translated by $\mathit{tr}_{\mathit{repl}}$. The replacement of all constants with fresh constants is required to obtain an identifier-stable rewriting.

Given a set P of IMPL rules, we define the replacement function $\mathit{tr}_{\mathit{act}}(P)$ which replaces every action atom $\alpha(\vec{t})$ in every rule in P by an ordinary atom $ra_\alpha(\vec{t})$ and returns the resulting set of rules. (Again, facts are translated.)

Given an IMPL policy P and a policy input EDB_M, then

$$P' = \mathit{tr}_{\mathit{repl}}(\mathit{tr}_{\mathit{act}}(P \cup \mathit{EDB}_M))$$

is an IMPL policy which does not contain any actions (therefore it is in the IMPL Core fragment), furthermore P' does not contain constants from C_M. The policy answer sets of P' correspond 1-1 to policy answer sets of $P \cup \mathit{EDB}_M$ such that the former contain a replacement atom ra_α iff the latter contain a corresponding action α.

We next describe the IMPL code fragment P_{Aux} which realizes semantics of IMPL actions by translating replacement atoms to IMPL core actions.

For mapping replaced constants back to their original value (to achieve stable identifiers), P_{Aux} contains the following facts:

$$\mathit{map}(c, c'). \qquad \text{for every constant } c \in C_M \tag{6.4}$$

We collect all rules which are modified in $\mathit{modifiedRule}$:

$$\begin{aligned}
&\mathit{modifiedRule}(R) \leftarrow ra_{\mathit{addRuleCondition}^+}(R, C, B).\\
&\mathit{modifiedRule}(R) \leftarrow ra_{\mathit{addRuleCondition}^-}(R, C, B).\\
&\mathit{modifiedRule}(R) \leftarrow ra_{\mathit{delRuleCondition}^+}(R, C, B).\\
&\mathit{modifiedRule}(R) \leftarrow ra_{\mathit{delRuleCondition}^-}(R, C, B).\\
&\mathit{modifiedRule}(R) \leftarrow ra_{\mathit{makeRuleUnconditional}}(R).
\end{aligned} \tag{6.5}$$

We accumulate effects of rule modification actions in add^+, add^-, del^+, and del^-:

$$\begin{aligned}
\mathit{add}^+(R, C, B) &\leftarrow ra_{\mathit{addRuleCondition}^+}(R, C, B).\\
\mathit{add}^-(R, C, B) &\leftarrow ra_{\mathit{addRuleCondition}^-}(R, C, B).\\
\mathit{del}^+(R, C, B) &\leftarrow ra_{\mathit{delRuleCondition}^+}(R, C, B).\\
\mathit{del}^+(R, C, B) &\leftarrow ra_{\mathit{makeRuleUnconditional}}(R), \mathit{ruleBody}^+(R, C, B).\\
\mathit{del}^-(R, C, B) &\leftarrow ra_{\mathit{delRuleCondition}^-}(R, C, B).\\
\mathit{del}^-(R, C, B) &\leftarrow ra_{\mathit{makeRuleUnconditional}}(R), \mathit{ruleBody}^-(R, C, B).
\end{aligned} \tag{6.6}$$

We represent rule bodies for modified rules in reserved predicates, using original rule, context, and belief identifiers. (We use primed variable names where primed identifiers will be grounded.)

$$\begin{aligned}
\mathit{ruleBody}^{+}(R,C,B) &\leftarrow \mathit{add}^{+}(R',C',B'),\\
&\qquad \mathit{modifiedRule}(R'),\ \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleBody}^{+}(R,C,B) &\leftarrow \mathit{ruleBody}^{+}(R',C',B'),\ \mathit{not}\ \mathit{del}^{+}(R',C',B'),\\
&\qquad \mathit{modifiedRule}(R'),\ \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleBody}^{-}(R,C,B) &\leftarrow \mathit{add}^{-}(R',C',B'),\\
&\qquad \mathit{modifiedRule}(R'),\ \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleBody}^{-}(R,C,B) &\leftarrow \mathit{ruleBody}^{-}(R',C',B'),\ \mathit{not}\ \mathit{del}^{-}(R',C',B'),\\
&\qquad \mathit{modifiedRule}(R'),\ \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleHead}(R,C,B) &\leftarrow \mathit{ruleHead}(R',C',B'),\\
&\qquad \mathit{modifiedRule}(R'),\ \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').
\end{aligned} \tag{6.7}$$

We represent new rule bodies for unmodified rules in reserved predicates, using original identifiers.

$$\begin{aligned}
\mathit{ruleBody}^{+}(R,C,B) &\leftarrow \mathit{ruleBody}^{+}(R',C',B'),\ \mathit{not}\ \mathit{modifiedRule}(R'),\\
&\qquad \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleBody}^{-}(R,C,B) &\leftarrow \mathit{ruleBody}^{-}(R',C',B'),\ \mathit{not}\ \mathit{modifiedRule}(R'),\\
&\qquad \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').\\
\mathit{ruleHead}(R,C,B) &\leftarrow \mathit{ruleHead}(R',C',B'),\ \mathit{not}\ \mathit{modifiedRule}(R'),\\
&\qquad \mathit{map}(R,R'),\ \mathit{map}(C,C'),\ \mathit{map}(B,B').
\end{aligned} \tag{6.8}$$

For actions that operate on modifications or sets of modifications, we must not use rules that have been changed by rule modifying actions. Therefore we next introduce an IMPL fragment that removes such rules from modifications specified by the extension of $\mathit{cleanMod}$. Identifiers for the changed modifications are created using auxiliary constant cm_{id}.

$$\begin{aligned}
\mathit{cleanedModId}(M',M) &\leftarrow \mathit{cleanMod}(M),\ \#id_2(M',cm_{id},M).\\
\mathit{modAdd}(M',R) &\leftarrow \mathit{modAdd}(M,R'),\ \mathit{cleanedModId}(M',M),\ \mathit{map}(R,R'),\\
&\qquad \mathit{not}\ \mathit{modifiedRule}(R').\\
\mathit{modDel}(M',R) &\leftarrow \mathit{modDel}(M,R'),\ \mathit{cleanedModId}(M',M),\ \mathit{map}(R,R').
\end{aligned} \tag{6.9}$$

We trigger cleaning for every modification that is used by $@\mathit{guiEditMod}$ or $@\mathit{applyMod}$.

$$\begin{aligned}
\mathit{cleanMod}(M) &\leftarrow \mathit{ra}_{\mathit{guiEditMod}}(M).\\
\mathit{cleanMod}(M) &\leftarrow \mathit{ra}_{\mathit{applyMod}}(M).
\end{aligned} \tag{6.10}$$

The following fragment cleans sets of modifications similarly as (6.9):

$$\begin{aligned}
\mathit{cleanedModSetId}(MS',MS) &\leftarrow \mathit{cleanModSet}(MS),\ \#id_2(MS',csm_{id},MS).\\
\mathit{cleanMod}(M) &\leftarrow \mathit{modset}(MS,M),\ \mathit{cleanModSet}(MS).\\
\mathit{modset}(MS',M') &\leftarrow \mathit{cleanedModId}(M',M), \mathit{modset}(MS,M),\\
&\qquad \mathit{cleanedModSetId}(MS',MS).
\end{aligned} \tag{6.11}$$

We trigger this cleaning for all sets of modifications used by $@\mathit{guiSelectMod}$:

$$\mathit{cleanModSet}(MS) \leftarrow \mathit{ra}_{\mathit{guiSelectMod}}(MS). \tag{6.12}$$

For comfort actions that project modifications and sets of modifications, we need a projection feature in the rewriting. Additionally we must remove rules that have been changed by rule modifications.[8]

[8]Examples 69 and 71 already hinted at how to realize $@\mathit{applyMod}$ and $@\mathit{applyModAtContext}$. However they do not guarantee stable identifiers; we therefore give here extended rewritings.

The following IMPL fragment projects modifications specified by the extension of predicate $\mathit{projectMod}$, removes all bridge rules that have been modified from these modifications and maps rule identifier constants back to their original identifiers. We trigger this by actions $@\mathit{guiEditModAtContext}$ and $@\mathit{applyModAtContext}$.

$$\begin{aligned}
\mathit{projectedModId}(M', M, C) &\leftarrow \mathit{projectMod}(M, C),\ \#\mathit{id}_3(M', \mathit{pm}_{id}, M, C).\\
\mathit{modAdd}(M', R) &\leftarrow \mathit{modAdd}(M, R'),\ \mathit{ruleHead}(R', C, S),\\
&\qquad \mathit{projectedModId}(M', M, C),\ \mathit{map}(R, R'),\\
&\qquad \mathit{not}\ \mathit{modifiedRule}(R').\\
\mathit{modDel}(M', R) &\leftarrow \mathit{modDel}(M, R'), \mathit{ruleHead}(R', C, S),\\
&\qquad \mathit{projectedModId}(M', M, C), \mathit{map}(R, R').\\
\mathit{projectMod}(M, C) &\leftarrow \mathit{ra}_{\mathit{guiEditModAtContext}}(M, C).\\
\mathit{projectMod}(M, C) &\leftarrow \mathit{ra}_{\mathit{applyModAtContext}}(M, C).
\end{aligned} \tag{6.13}$$

The next IMPL fragment achieves the same for sets of modifications, triggered by $@\mathit{guiSelectModAtContext}$:

$$\begin{aligned}
\mathit{projectedModSetId}(\mathit{MS}', \mathit{MS}, C) &\leftarrow \mathit{projectModSet}(\mathit{MS}, C),\\
&\qquad \#\mathit{id}_3(\mathit{MS}', \mathit{psm}_{id}, \mathit{MS}, C).\\
\mathit{projectMod}(M, C) &\leftarrow \mathit{modset}(\mathit{MS}, M), \mathit{projectModSet}(\mathit{MS}, C).\\
\mathit{modset}(\mathit{MS}', M') &\leftarrow \mathit{projectedModId}(M', M, C), \mathit{modset}(\mathit{MS}, M),\\
&\qquad \mathit{projectedModSetId}(\mathit{MS}', \mathit{MS}, C).\\
\mathit{projectModSet}(\mathit{MS}, C) &\leftarrow \mathit{ra}_{\mathit{guiSelectModAtContext}}(\mathit{MS}, C).
\end{aligned} \tag{6.14}$$

Program fragments (6.4) to (6.14) prepared everything for executing core actions which realize the original comfort actions.

We trigger action $@\mathit{delRule}$ for every rule that was removed by $@\mathit{delRule}$ in the original program, for every rule that was removed by a cleaned $@\mathit{applyMod}$, for every rule that was removed by a projected $@\mathit{applyModAtContext}$, and for every rule that was modified by a rule modifying action; we use the primed rule identifiers to remove the *original* rules:

$$\begin{aligned}
@\mathit{delRule}(R') &\leftarrow \mathit{ra}_{\mathit{delRule}}(R').\\
@\mathit{delRule}(R') &\leftarrow \mathit{ra}_{\mathit{applyMod}}(M'),\ \mathit{modDel}(M', R').\\
@\mathit{delRule}(R') &\leftarrow \mathit{ra}_{\mathit{applyModAtContext}}(M', C'),\\
&\qquad \mathit{projectedModId}(M'', M', C'),\ \mathit{modDel}(M'', R').\\
@\mathit{delRule}(R') &\leftarrow \mathit{modifiedRule}(R').
\end{aligned} \tag{6.15}$$

We trigger the action $@\mathit{addRule}$ for every rule that was added and not modified, for every rule of an applied and cleaned modification, for every rule of an applied and projected modification, and for every rule that was modified. We map to the original rule identifiers to obtain an identifier stable rewriting. (This is achieved, because rules that are modified are removed with their primed identifiers, while their modified form is added using the original identifiers.)

$$\begin{aligned}
@\mathit{addRule}(R) &\leftarrow \mathit{ra}_{\mathit{addRule}}(R'),\ \mathit{map}(R, R'),\ \mathit{not}\ \mathit{modifiedRule}(R').\\
@\mathit{addRule}(R) &\leftarrow \mathit{ra}_{\mathit{applyMod}}(M'),\ \mathit{cleanedModId}(M'', M'),\ \mathit{modAdd}(M'', R).\\
@\mathit{addRule}(R) &\leftarrow \mathit{ra}_{\mathit{applyModAtContext}}(M', C'),\\
&\qquad \mathit{projectedModId}(M'', M', C'),\ \mathit{modAdd}(M'', R).\\
@\mathit{addRule}(R) &\leftarrow \mathit{modifiedRule}(R'),\ \mathit{map}(R, R').
\end{aligned} \tag{6.16}$$

Finally we realize cleaned and projected GUI actions by activating core GUI actions.[9]

$$\begin{aligned}
@\mathit{guiSelectMod}(M') &\leftarrow \mathit{cleanedModSetId}(M', M), \mathit{ra}_{\mathit{guiSelectMod}}(M).\\
@\mathit{guiSelectMod}(MS') &\leftarrow \mathit{ra}_{\mathit{guiSelectModAtContext}}(MS, C),\\
&\qquad \mathit{projectedModSetId}(MS', MS, C).\\
@\mathit{guiEditMod}(M') &\leftarrow \mathit{cleanedModId}(M', M), \mathit{ra}_{\mathit{guiEditMod}}(M).\\
@\mathit{guiEditMod}(M') &\leftarrow \mathit{ra}_{\mathit{guiEditModAtContext}}(M, C), \mathit{projectedModId}(M', M, C).
\end{aligned} \tag{6.17}$$

This completes P_{Aux} (which consists of (6.4) to (6.17)). We formally define our rewriting as follows.

Definition 50. *Given an* IMPL *policy P and a policy input EDB_M the* rewritten policy $tr(P \cup \mathit{EDB}_M)$ *is defined as*

$$tr(P \cup \mathit{EDB}_M) = tr_{\mathit{repl}}(tr_{\mathit{act}}(P \cup \mathit{EDB}_M)) \cup P_{Aux}.$$

Using this rewriting, we can realize IMPL by implementing the IMPL core fragment.

Proposition 21. *Let M be an MCS, let P be an* IMPL *policy, and let EDB_M be a policy input wrt. M. Then a MCS M' is an admissible modification of M wrt. P and EDB_M iff M' is an admissible modification of M wrt. $tr(P \cup \mathit{EDB}_M)$.*

Proof. We first investigate the internal structure of policy $tr(P \cup \mathit{EDB}_M) = P_{tr} \cup P_{Aux}$ where $P_{tr} = tr_{\mathit{repl}}(tr_{\mathit{act}}(P \cup \mathit{EDB}_M))$. P_{tr} contains no constants from C_M (they all have been replaced). P_{Aux} contains in its rule heads either action atoms, or atoms with predicates that are disjoint with predicates in P_{tr}, or atoms with reserved predicates and constants from $C_M \cup C_{id}$. P_{Aux} contains no constraints and no cyclic dependencies (neither positive nor including default negation). Therefore P_{tr} does not depend on P_{Aux}, and we can intuitively split the policy (similar as with the Splitting Theorem which can be adapted from HEX to apply to ACTHEX) and obtain $I_{tr} \cup I_{Aux} \in \mathcal{AS}(tr(P \cup \mathit{EDB}_M))$ iff $I_{tr} \in \mathcal{AS}(P_{tr})$ and $I_{tr} \cup I_{Aux} \in \mathcal{AS}(P_{Aux} \cup I_{tr})$. Due to the definition of tr_{repl} and tr_{act} which only renames constants and replaces actions by atoms, we can see that $I_{tr} \in \mathcal{AS}(P_{tr})$ iff $tr^{-1}_{\mathit{repl}}(tr^{-1}_{\mathit{act}}(I_{tr})) \in \mathcal{AS}(P \cup \mathit{EDB}_M)$. I.e., answer sets of the translation of P (without P_{Aux}) directly correspond with an inverse translation of an answer sets of P. I_{tr} contains no actions, because P_{tr} contains no actions (only replacements). To show the result, it remains to show the following:

M' is a materialization of an answer set $I_C \in \mathcal{AS}(P \cup \mathit{EDB}_M)$ iff M' is a materialization of an answer set $I_{Aux} \cup tr_{\mathit{repl}}(tr_{\mathit{act}}(I_C)) \in \mathcal{AS}(P_{Aux} \cup tr_{\mathit{repl}}(tr_{\mathit{act}}(I_C)))$, where I_{Aux} contains auxiliary atoms derived by P_{Aux} from $tr_{\mathit{repl}}(tr_{\mathit{act}}(I_C))$.

As the translation removes actions, this amounts to showing that a materialization of actions in I_C is a materialization of actions in I_{Aux}. Therefore we must show that $(br_M \setminus \mathcal{R} \cup \mathcal{A}) \setminus \mathit{Rules} \cup \mathcal{M}$ (see Definition 46) yields the same result for I_C and for I_{Aux}. In the next equation we add subscripts to sets of Definition 46 to indicate from which policy answer set the respective sets were derived. Using this new notation, we need to show that

$$(br_M \setminus \mathcal{R}_{I_{Aux}} \cup \mathcal{A}_{I_{Aux}}) \setminus \mathit{Rules}_{I_{Aux}} \cup \mathcal{M}_{I_{Aux}} = (br_M \setminus \mathcal{R}_{I_C} \cup \mathcal{A}_{I_C}) \setminus \mathit{Rules}_{I_C} \cup \mathcal{M}_{I_C}.$$

As P_{Aux} contains only core actions, I_{Aux} contains only core actions; accordingly $\mathit{Rules}_{I_{Aux}} = \mathcal{M}_{I_{Aux}} = \emptyset$ and we need to show that $\mathcal{R}_{I_{Aux}} = \mathcal{R}_{I_C} \cup \mathit{Rules}_{I_C}$ and $\mathcal{A}_{I_{Aux}} = \mathcal{A}_{I_C} \setminus \mathit{Rules}_{I_C} \cup \mathcal{M}_{I_C}$.

We next show properties of answer sets of P_{Aux}. Given I_C, as P_{Aux} is stratified and contains no constraints, an answer set $I_{Aux} \cup tr_{\mathit{repl}}(tr_{\mathit{act}}(I_C)) \in \mathcal{AS}(P_{Aux} \cup tr_{\mathit{repl}}(tr_{\mathit{act}}(I_C)))$ always exists, is unique, and I_{Aux} has the following properties.

[9]The IMPL core actions $@\mathit{guiEditMod}$ and $@\mathit{guiSelectMod}$ cannot be realized by the simple rule $@\alpha(\vec{t}) \leftarrow \mathit{ra}_\alpha(\vec{t})$, because our usage of $@\mathit{addRule}$ and $@\mathit{delRule}$ for realizing rule modifying actions would lead to incorrect semantics.

(i) Due to (6.5), $\mathit{modifiedRule}(r') \in I_{Aux}$ iff $r \in \mathit{Rules}_{I_C}$.

(ii) Due to (6.6), $\mathit{add}^+(r', c', b') \in I_{Aux}$ iff $(c : b) \in \mathcal{A}_{I_C,r}$; $\mathit{add}^-(r', c', b') \in I_{Aux}$ iff **not** $(c : b) \in \mathcal{A}_{I_C,r}$; $\mathit{del}^+(r', c', b') \in I_{Aux}$ iff $(c : b) \in \mathcal{R}_{I_C,r}$; and $\mathit{del}^-(r', c', b') \in I_{Aux}$ iff **not** $(c : b) \in \mathcal{R}_{I_C,r}$.

(iii) Due to (ii) and (6.7), for every bridge rule $q \in \mathcal{M}_{I_C}$ identified by r in I_C, we have $q = \mathit{rule}_{I_{Aux}}(r)$, i.e., the modified bridge rule q is represented in I_{Aux} and identified by its original constant r.

(iv) Due to (6.8), for every bridge rule $q \in \mathcal{A}_{I_C} \setminus \mathit{Rules}_{I_C}$ we have $q = \mathit{rule}_{I_{Aux}}(r)$, i.e., q is represented in I_{Aux} and identified by constant r.

(v) Due to (6.9) and (6.10), for every modification $(A, R) = \mathit{mod}_{I_C}(m)$ such that $@\mathit{guiEditMod}(m) \in I_C$ or $@\mathit{applyMod}(m) \in I_C$, we have $(A \setminus \mathit{Rules}_{I_C}, R) = \mathit{mod}_{I_{Aux}}(c_{cm_{id},m})$ with $c_{cm_{id},m} \in \mathcal{I}_{id}$ and $\mathit{cleanedModId}(c_{cm_{id},m}, m) \in I_{Aux}$.

(vi) Due to (6.9), (6.11), and (6.12), for every set of modifications $\{(A_1, R_1), \dots, (A_k, R_k)\} = \mathit{modset}_{I_C}(ms)$ such that $@\mathit{guiSelectMod}(ms) \in I_C$, and $\mathit{modset}_{I_{Aux}}(c_{csm_{id},ms}) = \{(A_1 \setminus \mathit{Rules}_{I_C}, R_1), \dots, (A_k \setminus \mathit{Rules}_{I_C}, R_k)\}$ with $c_{csm_{id},ms} \in \mathcal{I}_{id}$, furthermore we have that $\mathit{cleanedModSetId}(c_{csm_{id},ms}, ms) \in I_{Aux}$.

(vii) Due to (6.13), for every modification $\mathit{mod}_{I_C}(m)$ and context identifier c s.t. $(A, R) = \mathit{mod}_{I_C}(m)|_c$ and $@\mathit{guiEditModAtContext}(m, c) \in I_C$ or $@\mathit{applyModAtContext}(m, c) \in I_C$, we have that $(A \setminus \mathit{Rules}_{I_C}, R)|_c = \mathit{mod}_{I_{Aux}}(c_{pm_{id},m,c})$ with $c_{pm_{id},m,c} \in \mathcal{I}_{id}$ and $\mathit{projectedModId}(c_{pm_{id},m,c}, m, c) \in I_{Aux}$.

(viii) Due to (6.14), for every modification set $\mathit{modset}_{I_C}(ms)$ and context identifier c such that $\{(A_1, R_1), \dots, (A_k, R_k)\} = \mathit{modset}_{I_C}(ms)|_c$ and $@\mathit{guiSelectModAtContext}(ms, c) \in I_C$, we have $\{(A_1 \setminus \mathit{Rules}_{I_C}, R_1)|c, \dots, (A_k \setminus \mathit{Rules}_{I_C}, R_k)|c\} = \mathit{modset}_{I_{Aux}}(c_{psm_{id},ms,c})$ with $c_{psm_{id},ms,c} \in \mathcal{I}_{id}$ and $\mathit{projectedModSetId}(c_{psm_{id},ms,c}, ms, c) \in I_{Aux}$.

As the order of evaluating action effects is irrelevant, we can next split the proof into proving correctness for non-GUI actions (indicated by superscript ng), and then proving correctness for GUI actions (indicated by superscript gui).

Due to (6.15), all rules in $\mathcal{R}^{ng}_{I_C}$ (from $@\mathit{delRule}$, $@\mathit{applyMod}$, and $@\mathit{applyModAtContext}$, see (v) and (vii)) and all rules in Rules_{I_C} (see (i)) are deleted in I_{Aux} using $@\mathit{delRule}$, and no other rules are deleted due to (6.15). Therefore, $\mathcal{R}^{ng}_{I_{Aux}} = \mathcal{R}^{ng}_{I_C} \cup \mathit{Rules}_{I_C}$. Due to (6.16), those rules in $\mathcal{A}^{ng}_{I_C}$ which are not in Rules_{I_C} (from $@\mathit{addRule}$, $@\mathit{applyMod}$, and $@\mathit{applyModAtContext}$, see (v) and (vii)) and all rules in $\mathcal{M}_{I_C}$ (see (ii) and (iii)) are added in I_{Aux} using $@\mathit{addRule}$, and no other rules are added due to (6.16). Therefore, $\mathcal{A}^{ng}_{I_{Aux}} = \mathcal{A}^{ng}_{I_C} \setminus \mathit{Rules}_{I_C} \cup \mathcal{M}_{I_C}$. Note that these rules are added using their original identifiers (see (iii) and (iv)) which makes our rewriting identifier-neutral wrt. created rules.

It remains to show, that also GUI actions are realized correctly by the rewriting, i.e., that $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and that $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus \mathit{Rules}_{I_C}$. As semantics of user interaction is nondeterministic, it is not possible (and makes no sense) to directly prove the above equalities. Instead, we split the rest of the proof into two directions: we prove that, given policy answer set I_C and a certain sequence S of user decisions which determine the effects of executing GUI actions in I_C, it is possible to achieve the same effects from executing another sequence S_{tr} of user decisions on the GUI actions in I_{Aux}, and vice versa.

($\Rightarrow$) Given policy answer set $I_C \in \mathcal{AS}(P \cup \mathit{EDB}_M)$, and given the accumulated effect $\mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_C}$ of GUI actions arising from a sequence S of user decisions on GUI actions in I_C, the corresponding I_{Aux} (with $I_{Aux} \cup \mathit{tr}_{repl}(\mathit{tr}_{act}(I_C)) \in \mathcal{AS}(P_{Aux} \cup \mathit{tr}_{repl}(\mathit{tr}_{act}(I_C)))$ as above) contains due to (6.17) a set of GUI actions that corresponds to GUI actions in I_C as follows: $@\mathit{guiEditMod}(m) \in I_C$ is mapped to a modification editor over $(A \setminus \mathit{Rules}_{I_C}, R)$ (see (v)); $@\mathit{guiSelectMod}(ms) \in I_C$ is mapped to a modification selection over $\{(A_1 \setminus \mathit{Rules}_{I_C}, R_1), \dots, (A_k \setminus \mathit{Rules}_{I_C}, R_k)\}$ (see (vi)); $@\mathit{guiEditModAtContext}(m, c)$ and $@\mathit{guiSelectModAtContext}(ms, c)$ are mapped analogously, always removing Rules_{I_C} from the first component of all modifications at hand. As GUI actions in I_{Aux} correspond to GUI actions in I_C with all rules from $Rules_{I_C}$ removed, it is clearly possible to obtain a sequence S_{tr} of user decisions on these GUI actions such that their accumulated effect is $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus Rules_{I_C}$.

($\Leftarrow$) For every GUI action in I_{Aux} there is a corresponding GUI action in I_C which contains the same modification(s) as the action in I_{Aux} and in some cases contains more rules from $Rules_{I_C}$. However, as GUI actions accumulate in $\mathcal{R}_{I_C}$ and in $\mathcal{A}_{I_C}$ and because $Rules_{I_C}$ is always subtracted from $\mathcal{R}_{I_C}$ and from $\mathcal{A}_{I_C}$ to obtain an admissible modification, a rule from $Rules_{I_C}$ which is added by an effect of a GUI action in I_C is not added in the materialization of the overall accumulated action effects. Therefore, from a sequence S_{tr} of user decisions on GUI actions in I_{Aux}, we can again always create a sequence S of user decisions on GUI actions in I_C such that $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus Rules_{I_C}$ and therefore the result holds. $\square$

6.5 Discussion and Related Work

In the design of IMPL we so far just considered bridge rule modifications for repairing the system. An interesting issue for further research is to drop this convention. A promising way to proceed in this direction is to integrate IMPL with recent work on managed MCSs [BEFW11], where bridge rules are extended such that they can arbitrarily modify the knowledge base of a context and even its semantics. Accordingly, IMPL could be extended with the possibility of using management operations on contexts in system modifications.

Realize IMPL in ACTHEX requires a usable and working ACTHEX implementation. No full-fledged ACTHEX implementation exists at the moment, therefore we are working on such an implementation and on improvements of ACTHEX which support a realization of IMPL using the rewriting technique described in Section 6.3.2. In particular, we work on realizing the generalization of taking into account the environment in external atom evaluation, and possibilities for explicitly implementing model and execution schedule selection functions in an ACTHEX plugin.

A notable feature of IMPL is a user interface for selecting or editing modifications. An interesting aspect for future research is the usability of that interface, and the possibility of reducing their number by grouping them according to nonground bridge rules. We conjecture that this could lead to a considerable improvement of usability.

6.5.1 Related Work

Related to IMPL is the action language *IMPACT* [SBD$^+$00], which is a declarative formalism for actions in distributed and heterogeneous multi-agent systems. *IMPACT* is a very rich general purpose formalism, which however is more difficult to manage compared to the special purpose language IMPL. Furthermore, user interaction as in IMPL is not directly supported in *IMPACT*; nevertheless most parts of IMPL could be embedded in *IMPACT*.

In the fields of access control, e.g., surveyed in [BCOS09], and privacy restrictions [DHS07], policy languages have also been studied in detail. As a notable example, $\mathcal{PDL}$ [CLN00] is a declarative policy language based on logic programming which maps events in a system to actions. $\mathcal{PDL}$ is richer than IMPL concerning action inter-dependencies, whereas actions in IMPL have a richer internal structure than $\mathcal{PDL}$ actions. Moreover, actions in IMPL depend on the content of a policy answer set. Similarly, inconsistency analysis input in IMPL has a deeper structure than events in $\mathcal{PDL}$.

In the context of relational databases, logic programs have been used for specifying repairs for databases that are inconsistent wrt. a set of integrity constraints [GGZ03, EFGL08, MB10]. These approaches may be considered fixed policies without user interaction, like an IMPL policy simply applying diagnoses in a homogeneous MCS. Note however, that an important motivation for developing IMPL is the fact that automatic repair approaches are not always a viable option for dealing with inconsistency in a MCS.

Active integrity constraints (AICs) [CGZ09, CT08a, CT08b] and *inconsistency management policies (IMPs)* [MPP^{+}08] have been proposed for specifying repair strategies for inconsistent databases in a flexible way. AICs extend integrity constraints by introducing update actions, for inserting and deleting tuples, to be performed if the constraint is not satisfied. On the other hand, an IMP is a function which is defined wrt. a set of functional dependencies mapping a given relation R to a 'modified' relation R' obeying some basic axioms.

Although suitable IMPL policy encodings can mimic database repair programs—AICs and (certain) IMPs—for specific classes of integrity constraints, there are fundamental conceptual differences between IMPL and the above approaches to database repair. Most notably, IMPL policies aim at restoring consistency by modifying bridge rules, which leaves knowledge bases unchanged; opposed to that, IMPs and AICs consider a set of fixed constraints and repair the database. Another difference is that IMPL policies are able to operate on heterogeneous knowledge bases and may involve user interaction.

7 Summary and Conclusion

The recurring topic of this thesis is inconsistency in multi-context systems, which are a formalism for building distributed knowledge-based applications by interlinking smaller existing knowledge-based systems. Pursuing research in the topic of multi-context systems led us to closely investigate the related area of HEX programs, which are a formalism for integrating declarative reasoning with external computations in procedural languages. We conducted work on a theoretical level by formal analysis of mathematical properties, and on an empirical level by implementing research software and experimentally evaluating the efficiency of our algorithms.

At the beginning of this work, not even the notion of inconsistency in MCSs was clearly defined. We hence looked at various ways a MCS can become inconsistent. As a result of that research, we were able to define the formal concepts of *diagnosis* and *inconsistency explanation*, which cleanly characterize inconsistencies in MCSs in terms of bridge rules.

We investigated properties of these notions, in particular the relationship between diagnoses and explanations and the respective minimal notions, and the computational complexity of computing these notions. An important result of this research was the insight that diagnoses and explanations are in a duality relationship, and that our notions do not exhibit unreasonably high computational complexity.

We also developed a method of computing diagnoses and inconsistency explanations: first we theoretically described a rewriting which uses the HEX formalism as underlying knowledge representation framework, then we implemented this rewriting in the dlvhex research prototype and conducted empirical experiments with the resulting tool, called MCS-IE.

The experience gained with MCS-IE showed a major scalability problem, and after an investigation we identified the dlvhex engine and its approach of evaluating HEX programs as culprit. This prompted further research, this time not into MCS but into the related HEX formalism, and into new methods for evaluating HEX programs.

Evaluating HEX is an interleaved computation which consists of (i) grounding HEX program fragments, (ii) evaluating semantics of normal answer set programs, and (iii) evaluating semantics of external atoms. Our research on an improved HEX evaluation yielded a novel evaluation framework, which changes the way a HEX program is decomposed for evaluation, i.e., it allows for dividing and conquering steps (i)-(iii) mentioned above more efficiently than previous methods for evaluating HEX. We described this novel HEX evaluation framework, mathematically showed its correctness, implemented it in a new version of the dlvhex research prototype, and conducted empirical experiments that show the superior performance of the new framework over the previous state of the art.

Finally we introduced the policy language IMPL for managing inconsistency in MCSs; we defined syntax and semantics of this language, discussed possible scenarios for applying IMPL in practical applications, and showed how IMPL can be realized by rewriting it to the ACTHEX formalism (which is an extension of the HEX formalism).

Our novel notions for analyzing inconsistency in MCSs and the policy language IMPL have the potential to make future knowledge-based applications more robust. By making inconsistency more manageable, and avoiding that systems become unresponsive in case of inconsistency, this could also extend the usage of knowledge-based systems in applications. The work on improving efficiency of HEX evaluation makes the HEX formalism applicable to a broader range of practical applications. Moreover this work furthered our understanding of the HEX formalism by making more of its intrinsic properties explicit, thereby improving its foundations for further research.

Outlook

Our approach of analyzing and managing inconsistency in MCSs has the potential to improve robustness of current and future knowledge-based systems. However these methods have not yet been applied to a real-world application. Therefore such an application would be the next logical step which would strengthen our approach beyond theoretical considerations and synthetically generated benchmark instances.

For practical applicability, it might be necessary to realize inconsistency analysis and management in a distributed algorithm. A particular suggestive way to do this would be to integrate our method into the DMCS [BDTE$^+$10b] algorithm and solver [DMC10] which already computes MCS semantics in a distributed fashion. Such an integration is currently investigated in a masters thesis project, co-supervised by the author of this thesis.

The novel HEX evaluation framework introduced in this thesis is a step towards making HEX evaluation more efficient, however it is only the first step: our framework provides the possibility to efficiently decompose a HEX program, but we did not yet formally investigate how an ideal evaluation heuristics should look like, i.e., how the framework must be configured to yield optimal evaluation efficiency. However it is likely that no universal solution exists to such a heuristics, therefore real use cases where HEX is applied in practice will provide worthwhile opportunities to investigate evaluation heuristics.

Orthogonal to the framework introduced in this thesis are conflict-driven approaches for HEX evaluation [EFKR12]. These approaches integrate clause learning into HEX evaluation and promise to improve efficiency of HEX even more in the future (see [EFK$^+$12b]).

In real applications, it will be necessary to combine good evaluation heuristics and conflict-driven approaches to make HEX an efficient tool for reasoning with external computations.

In summary, both for inconsistency management in MCSs and for HEX programs there are many possibilities to extend the work of this thesis in the future. Such future work, application-centered or theoretical, can have a substantial impact on applicability, performance, and usage of the MCS and HEX formalism in practical applications. This in turn will influence the acceptance of the MCS and HEX formalisms within the research community and their uptake as generic tools for reasoning in other disciplines.

Bibliography

[ABC03] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming*, 3(4-5):393–424, 2003.

[ADS08] Vincent Armant, Philippe Dague, and Laurent Simon. Distributed consistency-based diagnosis. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 5330 of *Lecture Notes in Computer Science*, pages 113–127, 2008.

[AGM85] Carlos A. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[AM05] Eyal Amir and Sheila A. McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162(1-2):49–88, 2005.

[Ang88] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[BA08] Antonis Bikakis and Grigoris Antoniou. *Ambient Intelligence*, chapter Distributed Defeasible Contextual Reasoning in Ambient Computing, pages 308–325. Springer, 2008.

[BA10] Antonis Bikakis and Grigoris Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1492–1506, 2010.

[BAH11] Antonis Bikakis, Grigoris Antoniou, and Panayiotis Hassapis. Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowledge and Information Systems*, 27(1):45–84, 2011.

[BC03] Leopoldo E. 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, 2003.

[BCM$^+$03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, Cambridge, 2003.

[BCOS09] Piero A. Bonatti, Juri Luca De Coi, Daniel Olmedilla, and Luigi Sauro. Rule-based policy representations and reasoning. In François Bry and Jan Maluszynski, editors, *Semantic Techniques for the Web, The REWERSE Perspective*, volume 5500 of *Lecture Notes in Computer Science*, pages 201–232. Springer, 2009.

[BDTE^{+}10a] Seif El-Din Bairakdar, Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Decomposition of Distributed Nonmonotonic Multi-Context Systems. In Tommie Meyer and Eugenia Ternovska, editors, *International Workshop on Non-Monotonic Reasoning (NMR)*, CEUR Workshop Proceedings, pages 24–37. CEUR-WS.org, May 2010.

[BDTE^{+}10b] 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, *European Conference on Logics in Artificial Intelligence (JELIA)*, volume 6341 of *Lecture Notes in Artificial Intelligence*, pages 352–355. Springer, 2010.

[BE07] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 385–390. AAAI Press, 2007.

[BEF11] Gerhard Brewka, Thomas Eiter, and Michael Fink. Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 233–258. Springer, 2011.

[BEFI10] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. HEX programs with action atoms. In Manuel V. Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming (ICLP) 2010*, pages 24–33, 2010.

[BEFS10] Markus Bögl, Thomas Eiter, Michael Fink, and Peter Schüller. The MCS-IE system for explaining inconsistency in multi-context systems. In Tomi Janhunen and Ilkka Niemelä, editors, *European Conference on Logics in Artificial Intelligence (JELIA)*, Lecture Notes in Artificial Intelligence, pages 356–359. Springer, 2010.

[BEFW11] Gerhard Brewka, Thomas Eiter, Michael Fink, and Antonius Weinzierl. Managed multi-context systems. In Toby Walsh, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, Lecture Notes in Artificial Intelligence, pages 786–791. Springer, 2011.

[Ber89] Claude Berge. *Hypergraphs*. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1989.

[Ber11] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[BGP^{+}07] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging ASP programs by means of ASP. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 31–43, 2007.

[BLR97] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in disjunctive datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1997.

[BM08] Arnold Binas and Sheila A. McIlraith. Peer-to-peer query answering with inconsistent knowledge. In Gerhard Brewka and Jérôme Lang, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 329–339, Sydney, Australia, September 16–19 2008.

[BN08] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Computing Surveys*, 41(1):1–41, 2008.

[BO07] Piero A. Bonatti and Daniel Olmedilla. Rule-based policy representation and reasoning for the semantic web. In Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola Henze, Paula-Lavinia Patranjan, and Robert Tolksdorf, editors, *Reasoning Web Summer School*, Lecture Notes in Computer Science, pages 240–268, 2007.

[BRMO03] Mark Brodie, Irina Rish, Sheng Ma, and Natalia Odintsova. Active probing strategies for problem diagnosis in distributed systems. In Georg Gottlob and Toby Walsh, editors, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1337–1338. Morgan Kaufmann, 2003.

[BRS07] 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.

[BS89] Howard A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68(2):135–154, 1989.

[BS03] A. Borgida and L. Serafini. Distributed description logics: Assimilating information from peer sources. *Journal on Data Semantics I*, pages 153–184, 2003.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BV05] Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In Marina De Vos and Alessandro Provetti, editors, *International Workshop on Answer Set Programming*, 2005.

[CCIL08] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *International Conference on Logic Programming (ICLP)*, Lecture Notes in Computer Science, pages 407–424. Springer, 2008.

[CDT89] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. A theory of diagnosis for incomplete causal models. In N. S. Sridharan, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1311–1317, 1989.

[CGL$^+$08] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Inconsistency tolerance in P2P data integration: An epistemic logic approach. *Information Systems*, 33(4-5):360–384, 2008.

[CGZ09] Luciano Caroprese, Sergio Greco, and Ester Zumpano. Active integrity constraints for database consistency maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 21(7):1042–1058, 2009.

[CHI88] Yves Crama, Peter L. Hammer, and Toshihide Ibaraki. Cause-effect relationships and partially defined Boolean functions. *Annals of Operations Research*, 16:299–326, 1988.

[CLN00] Jan Chomicki, Jorge Lobo, and Shamim A. Naqvi. A logic programming approach to conflict resolution in policy management. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 121–132, 2000.

[CPD07] Luca Console, Claudia Picardi, and Daniele Theseider Dupré. A framework for decentralized qualitative model-based diagnosis. In Manuela M. Veloso, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 286–291, 2007.

[CSH06] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Record*, 35:34–41, September 2006.

[CT08a] Luciano Caroprese and Miroslaw Truszczynski. Declarative semantics for active integrity constraints. In Maria Garcia de la Banda and Enrico Pontelli, editors, *International Conference on Logic Programming (ICLP)*, volume 5366 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[CT08b] Luciano Caroprese and Miroslaw Truszczynski. Declarative semantics for revision programming and connections to active integrity constraints. In Steffen Hölldobler, Carsten Lutz, and Heinrich Wansing, editors, *European Conference on Logics in Artificial Intelligence (JELIA)*, volume 5293 of *Lecture Notes in Computer Science*, pages 100–112. Springer, 2008.

[dAP07] Sandra de Amo and Mônica Sakuray Pais. A paraconsistent logic programming approach for querying inconsistent databases. *International Journal of Approximate Reasoning*, 46(2):366–386, 2007.

[DCC86] Newton Da Costa and Walter Carnielli. On paraconsistent deontic logic. *Philosophia*, 16:293–305, 1986.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[DHS07] Claudiu Duma, Almut Herzog, and Nahid Shahmehri. Privacy in the semantic web: What policy languages have to offer. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 109–118, 2007.

[DHX12] dlvhex, 2012. `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`.

[dK91] Johan de Kleer. Focusing on probable diagnoses. In Thomas L. Dean and Kathleen McKeown, editors, *AAAI Conference on Artificial Intelligence (AAAI)*, pages 842–848. AAAI Press, 1991.

[dKK03] Johan de Kleer and James Kurien. Fundamentals of model-based diagnosis. In *IFAC Fault Detection, Supervision and Safety of Technical Processes (SafeProcess)*, 2003.

[DLV12] DLVSYSTEM s.r.l., 2012. `http://www.dlvsystem.com/`.

[DMC10] DMCS - The DMCS solver, 2010. `http://www.kr.tuwien.ac.at/research/systems/dmcs/`.

[DMC11] DMCS Experiments, 2011. `http://www.kr.tuwien.ac.at/research/systems/dmcs/experiments.html`.

[DTEFK09] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular Nonmonotonic Logic Programming Revisited. In Patricia M. Hill and David S. Warren, editors, *International Conference on Logic Programming (ICLP)*, volume 5649 of *Lecture Notes in Computer Science*, pages 145–159. Springer, July 2009.

[DTEFK10] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Distributed Nonmonotonic Multi-Context Systems. In Fangzhen Lin and Uli Sattler, editors, *International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 60–70. AAAI Press, May 2010.

[EBDT$^+$09] Thomas Eiter, Gerhard Brewka, Minh Dao-Tran, Michael Fink, Giovambattista Ianni, and Thomas Krennwallner. Combining nonmonotonic knowledge bases with external sources. In Silvio Ghilardi and Roberto Sebastiani, editors, *International Symposium on Frontiers of Combining Systems (FroCoS)*, volume 5749 of *Lecture Notes in Computer Science*, pages 18–42. Springer, 2009.

[ED08] Faezeh Ensan and Weichang Du. Aspects of inconsistency resolution in modular ontologies. In Sabine Bergler, editor, *Canadian Conference on AI*, volume 5032 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008.

[EFGL08] Thomas Eiter, Michael Fink, Gianluigi Greco, and Domenico Lembo. Repair localization for query answering from inconsistent databases. *ACM Transactions on Database Systems*, 33(2), 2008.

[EFI$^+$11] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, and Peter Schüller. Pushing efficient evaluation of HEX programs by modular decomposition. In James Delgrande and Wolfgang Faber, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Lecture Notes in Artificial Intelligence, pages 93–106, May 2011.

[EFIS11] Thomas Eiter, Michael Fink, Giovambattista Ianni, and Peter Schüller. Towards a policy language for managing inconsistency in multi-context systems. In Alessandra Mileo and Michael Fink, editors, *International Workshop on Logic-based Interpretation of Context: Modelling and Applications (Log-IC)*, volume 738 of *CEUR Workshop Proceedings*, pages 23–35. CEUR-WS.org, May 2011.

[EFIS12a] Thomas Eiter, Michael Fink, Giovambattista Ianni, and Peter Schüller. The IMPL policy language for managing inconsistency in multi-context systems. In Alessandra Mileo and Michael Fink, editors, *Postproceedings of the International Conference on Applications of Declarative Programming and Knowledge Management (INAP) and the Workshop on Logic Programming (WLP)*, Lecture Notes in Artificial Intelligence. Springer, 2012.

[EFIS12b] Thomas Eiter, Michael Fink, Giovambattista Ianni, and Peter Schüller. Managing inconsistency in multi-context systems using the IMPL policy language. Technical Report INFSYS RR-1843-12-05, Vienna University of Technology, Institute for Information Systems, 2012.

[EFK09] Thomas Eiter, Michael Fink, and Thomas Krennwallner. Decomposition of Declarative Knowledge Bases with External Functions. In Craig Boutilier, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 752–758. AAAI Press, July 2009.

[EFK$^+$12a] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Eliminating unfounded set checking for HEX-programs. In Michael Fink and Yuliya Lierler, editors, *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*, 2012. To appear.

[EFK$^+$12b] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Exploiting unfounded sets for HEX-program evaluation. In *European Conference on Logics in Artificial Intelligence (JELIA)*, 2012. To appear.

[EFK$^+$12c] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Improving hex-program evaluation based on unfounded sets. Technical Report INFSYS RR-1843-12-08, Vienna University of Technology, Institute for Information Systems, 2012. To appear.

[EFKR12] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. In *Technical Communications of the International Conference on Logic Programming (ICLP)*, 2012. To appear.

[EFS10] Thomas Eiter, Michael Fink, and Peter Schüller. Approximations for explanations of inconsistency in partially known multi-context systems. In Gerhard Brewka, Viktor Marek, and Mirek Truszczynski, editors, *Thirty Years of Nonmonotonic Reasoning*, Lecture Notes in Artificial Intelligence, October 2010. 15 pages.

[EFS11] Thomas Eiter, Michael Fink, and Peter Schüller. Approximations for explanations of inconsistency in partially known multi-context systems. In James Delgrande and Wolfgang Faber, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Lecture Notes in Artificial Intelligence, pages 107–119, May 2011.

[EFSW09] Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Towards diagnosing inconsistency in nonmonotonic multi-context systems. In Alessandra Mileo and James P. Delgrande, editors, *International Workshop on Logic-based Interpretation of Context: Modelling and Applications (Log-IC)*, volume 550 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009. 4 pages (no page numbers).

[EFSW10] Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Finding explanations of inconsistency in nonmonotonic multi-context systems. In Fangzhen Lin and Uli Sattler, editors, *International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 329–339. AAAI Press, 2010.

[EFW10] Thomas Eiter, Michael Fink, and Antonius Weinzierl. Preference-based inconsistency assessment in multi-context systems. In Tomi Janhunen and Ilkka Niemelä, editors, *European Conference on Logics in Artificial Intelligence (JELIA)*, Lecture Notes in Artificial Intelligence, pages 143–155, 2010.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web Summer School*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, September 2009.

[EIL$^+$08] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[EIP$^+$06] Thomas Eiter, Giovambattista Ianni, Axel Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with rules and ontologies. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web Summer School*, pages 93–127, 2006.

[EIST04] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Nonmonotonic description logic programs: Implementation and experiments. In Franz Baader and Andrei Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Computer Science*, pages 511–527. Springer, 2004.

[EIST05] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In L. P. Kaelbling and A. Saffiotti, editors, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 90–96, Denver, USA, 2005. Professional Book Center.

[EIST06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In York Sure and John Domingue, editors, *European Semantic Web Conference (ESWC)*, Lecture Notes in Computer Science, pages 273–287, 2006.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[FPL11] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

[Gab93] Dov M. Gabbay. Labelled deductive systems: A position paper. In J. Oikkonen and J. Väänänen, editors, *Logic Colloquium '90*, volume 2 of *Lecture Notes in Logic*, pages 66–88. Springer, 1993.

[Gec08] Gecode: An open constraint solving library. In *Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP)*, Paris, France, May 2008. Presentation (40 slides).

[GGZ03] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1389–1408, 2003.

[GH91] Dov Gabbay and Anthony Hunter. Making inconsistency respectable 1: A logical framework for inconsistency in reasoning. In *Foundations of Artificial Intelligence Research*, volume 535 of *Lecture Notes in Computer Science*, pages 19–32, 1991.

[GH93] Dov M. Gabbay and Anthony Hunter. Making inconsistency respectable: Part 2 - Meta-level handling of inconsistency. In Michael Clarke, Rudolf Kruse, and Serafín Moral, editors, *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU)*, volume 747 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 1993.

[GKKS11] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in gringo series 3. In James Delgrande and Wolfgang Faber, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected Boolean search problems. In Willem Jan van Hoeve and John N. Hooker, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2009.

[GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187–188(0):52–89, 2012.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *International Conference and Symposium on Logic Programming (ICLP)*, pages 1070–1080, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[GOS09] Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In Patricia M. Hill and David Scott Warren, editors, *International Conference on Logic Programming (ICLP)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2009.

[Got92] Georg Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2:397–425, 1992.

[GRP09] Dov M. Gabbay, Odinaldo Rodrigues, and Gabriella Pigozzi. Connections between belief revision, belief merging and social choice. *Journal of Logic and Computation*, 19(3):445–446, 2009.

[GS94] Fausto Giunchiglia and Luciano Serafini. Multilanguage hierarchical logics, or: How we can do without modal logics. *Artificial Intelligence*, 65(1):29–70, 1994.

[GST07] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo: A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John S.

Schlipf, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[GW96] Paul W. P. J. Grefen and Jennifer Widom. Integrity constraint checking in federated databases. In *IFCIS International Conference on Cooperative Information Systems (CoopIS)*, pages 38–47, 1996.

[HM85] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Information Systems*, 3(3):253–278, 1985.

[HM92] Joseph Y. Halpern and Yoram Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54(3):319–379, 1992.

[HPRW96] Lisa Hellerstein, Krishnan Pillaipakkamnatt, Vijay Raghavan, and Dawn Wilkins. How many queries are needed to learn? *Journal of the ACM*, 43(5):840–862, 1996.

[HRO06] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *International Conference on Very Large Data Bases (VLDB)*, pages 9–16. ACM, 2006.

[IS95] Katsumi Inoue and Chiaki Sakama. Abductive framework for nonmonotonic theory change. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 204–210, 1995.

[JOTW09] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity Aspects of Disjunctive Stable Models. *Journal of Artificial Intelligence Research*, 35:813–857, 2009.

[KL92] Michael Kifer and Eliezer L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, October 1992.

[KP05] 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.

[LGI$^+$05] Nicola Leone, Gianluigi Greco, Giovambattista Ianni, Vincenzino Lio, Giorgio Terracina, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Riccardo Rosati, Domenico Lembo, Maurizio Lenzerini, Marco Ruzzi, Edyta Kalka, Bartosz Nowicki, and Witold Staniszkis. The INFOMIX system for advanced integration of incomplete and inconsistent data. In Fatma Özcan, editor, *ACM SIGMOD International Conference on Management of Data*, pages 915–917. ACM, 2005.

[LR07] 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, 2007.

[LS11] Yuliya Lierler and Peter Schüller. Parsing Combinatory Categorial Grammar with Answer Set Programming: Preliminary report. In *Workshop on Logic Programming (WLP)*, September 2011. CoRR 1108.5567, 12 pages (no page numbers).

[LS12] Yuliya Lierler and Peter Schüller. Parsing combinatory categorial grammar via planning in answer set programming. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, pages 436–453. Springer, 2012.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *International Conference on Logic Programming (ICLP)*, pages 23–38, Santa Margherita Ligure, Italy, June 1994. MIT-Press.

[MB10] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data & Knowledge Engineering*, 69(6):545–572, 2010.

[McC87] John McCarthy. Generality in artificial intelligence. *Communications of the ACM*, 30(12):1030–1035, 1987.

[McC93] John McCarthy. Notes on formalizing context. In Ruzena Bajcsy, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 555–562, 1993.

[MIE12a] MCS-IE: Multi-context systems inconsistency explainer, 2012. `http://www.kr.tuwien.ac.at/research/systems/mcsie/`.

[MIE12b] MCS-IE Example Workbench, 2012. `http://www.kr.tuwien.ac.at/research/systems/mcsie/tut/`.

[Mit05] David G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[MPP$^+$08] Maria Vanina Martinez, Francesco Parisi, Andrea Pugliese, Gerardo I. Simari, and V. S. Subrahmanian. Inconsistency management policies. In Gerhard Brewka and Jérôme Lang, editors, *International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 367–377. AAAI Press, 2008.

[Nie08] Ilkka Niemelä. Stable models and difference logic. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008. 5 pages (no page numbers).

[OJ08] Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theory and Practice of Logic Programming*, 8(5-6):717–761, 2008.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PAS12] Potassco, the Potsdam Answer Set Solving Collection, 2012. `http://potassco.sourceforge.net/`.

[PC88] Luís Moniz Pereira and Miguel Calejo. A framework for prolog debugging. In Robert A. Kowalski and Kenneth A. Bowen, editors, *International Conference and Symposium on Logic Programming (ICLP)*, pages 481–495, 1988.

[PEB94] Chris Preist, Kave Eshghi, and Bruno Bertolino. Consistency-based and abductive diagnoses as generalized stable models. *Annals of Mathematics and Artificial Intelligence*, 11(1-4):51–74, 1994.

[Pep08] Pavlos Peppas. Belief revision. In *Handbook of Knowledge Representation*, pages 317–360. Elsevier, 2008.

[PRS10] Simona Perri, Francesco Ricca, and Marco Sirianni. A parallel ASP instantiator based on DLV. In Leaf Petersen and Enrico Pontelli, editors, *Declarative Aspects of Multicore Programming (DAMP)*, Lecture Notes in Computer Science, pages 73–82. Springer, 2010.

[Prz88] Theodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufman, Washington DC, 1988.

[Prz91] T.C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3):401–424, 1991.

[Rei78] Raymond Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum, New York / London, 1978.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[Ros94] K.A. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of the ACM*, 41(6):1216–1267, 1994.

[RS05] Floris Roelofsen and Luciano Serafini. Minimal and absent information in contexts. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 558–563. Professional Book Center, 2005.

[SBD$^+$00] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.

[SC95] Marco Schaerf and Marco Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74(2):249–310, 1995.

[Sch06] Roman Schindlauer. *Answer-Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, 2006.

[Sch08] Simon Schenk. On the semantics of trust and caching in the semantic web. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science, pages 533–549. Springer, 2008.

[Sch10] Peter Schüller. Methods and algorithms for managing inconsistency in multi-context systems. International Conference on the Principles of Knowledge Representation and Reasoning, Doctoral Consortium (KR-DC), May 2010. Poster.

[SEF10] Peter Schüller, Thomas Eiter, and Michael Fink. Towards approximating output-projected equilibria in partially known multi-context systems. In Hans K. Kaiser and Raimund Kirner, editors, *Proceedings of the Junior Scientist Conference 2010*, pages 315–316. Vienna University of Technology, April 2010.

[SK96] B. Selman and H. Kautz. Knowledge Compilation and Theory Approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SW11] Peter Schüller and Antonius Weinzierl. Semantic reasoning with SPARQL in heterogeneous multi-context systems. In Camille Salinesi and Oscar Pastor, editors, *Advanced Information Systems Engineering Workshops (CAiSE)*, volume 83 of *Lecture Notes in Business Information Processing*, pages 575–585. Springer, June 2011. (1st International Workshop on Semantic Search (SSW)).

[Syr06] Tommi Syrjänen. Debugging inconsistent answer set programs. In *International Workshop on Nonmonotonic Reasoning (NMR)*, pages 77–83, 2006.

[tTvH96] Annette ten Teije and Frank van Harmelen. Computing approximate diagnoses by using approximate entailment. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, pages 256–265, 1996.

[Val84] L. G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[Wei10] Antonius Weinzierl. Comparing inconsistency resolutions in multi-context systems. In Marija Slavkovik, editor, *Proceedings of the* 15^{th} *Student Session of the European Summer School for Logic, Language and Information (ESSLLI)*, pages 17–24, August 2010.