Abstraction for Reasoning about Agent Behavior with Answer Set Programming

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Zeynep Gözen Sarıbatur
Matrikelnummer 01428912

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Zweitbetreuung: Assoc. Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

Diese Dissertation haben begutachtet:

João Leite

Tran Cao Son

Wien, 8. August 2019

Zeynep Gözen Sarıbatur

Abstraction for Reasoning about Agent Behavior with Answer Set Programming

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Zeynep Gözen Sarıbatur
Registration Number 01428912

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Second advisor: Assoc. Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

The dissertation has been reviewed by:

João Leite

Tran Cao Son

Vienna, 8th August, 2019

Zeynep Gözen Sarıbatur

Erklärung zur Verfassung der Arbeit

Zeynep Gözen Sarıbatur
Obere Amtshausgasse 46/4/53, 1050, 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.

Wien, 8. August 2019

Zeynep Gözen Sarıbatur

Acknowledgements

First and foremost, I would like to thank my supervisor Thomas Eiter, for his continuous support, guidance and patience throughout my PhD studies. He never failed to surprise me with his ability to always find time and energy (sometimes even more than me) for our meetings and to give scrupulous attention to the details. It was truly inspirational to see him always find a charm in what he does, so that it worths to put an effort and to use valuable time. I would also like to thank my co-supervisor Georg Weissenbacher for providing insight and a different perspective to the problems we were tackling.

A special thanks goes to DK LogiCS for giving me the opportunity to continue my academic journey in the lovely city of Vienna and in a friendly atmosphere. This would not have been possible without the passion of Anna Prianichnikova to make us feel like a family, and, of course, the founder Helmut Veith, who we miss dearly. Also big thanks to Eva Nedoma and Beatrix Buhl, for always being there to help out.

My PhD journey would not have been so smooth without having Peter Schüller come on board in the last two years. He will always have a special place as my non-official co-advisor with his invaluable support, witty remarks, inspiring comments, and realistic perspective to research that has helped me keep my feet on the ground. I am also very thankful to Stefan Woltran for our pleasant conversations and for his inspiring remarks to my work, and to Magdalena Ortiz for the friendliness and for being a great example of balancing family life and work. Also thanks to Johannes Wallner for his help and advice in the last years.

Warm thanks to Chitta Baral for hosting me at Arizona State University, which helped in expanding my horizons in research and life, and to Torsten Schaub for hosting me at University of Potsdam in a great team environment and for always making me feel more than welcome.

Throughout my studies I was lucky to meet inspiring colleagues and friends, who were always open to spend quality time together. Special thanks to Shqiponja Ahmetaj, Harald Beck, Gerald Berger, Wolfgang Dvořák, Adrian Haret, Tobias Kaminski, Thomas Linsbichler, Neha Lodha, Anna Lukina, Jan Maly, Matthias Schlaipfer, and Ilina Stoilkovska, for our coffee breaks, lunch meet-ups, hiking trips, excursions, and all other non-scientific activities. A warm hug goes to Giray Havur for being an amusing comrade during this Viennese life.

Last but most importantly, I am deeply grateful to my family: my mother Ayşe for her constant love, support and encouragement throughout my life, for always wishing me the best even though that may mean spending less time together, and for her heart-warming enthusiasm for even the tiniest accomplishments in this journey, my uncle Mehmet and aunt Inga-lill, for their loving support and for always reminding me the possibility to just leave everything and visit my second home in Sweden, and, finally, Erdem for being my solid rock throughout this journey.

Kurzfassung

Fortschritte im Bereich der Künstlichen Intelligenz (KI) machen es immer wichtiger, das Verhalten von künstlichen Agenten zu verstehen und wesentliche Elemente von deren Modellen strukturiert beschreiben zu können, insbesondere jene die deren Verhalten beeinflussen. Ein logikbasierter Ansatz für den Entwurf von KI Agenten beschreibt deren Wissen über die Welt als mathematisches Modell in einer expressiven Repräsentationssprache mit präzise definierter Syntax und Semantik. Solch ein Modell beschreibt die Fähigkeiten des Agenten und erlaubt es, Schlussfolgerungen über zukünftige Aktionen durchzuführen. Der Wunsch solch eine Repräsentationssprache zu entwickeln ufert in Herausforderungen, Weltwissen und die nichtmonotone Natur menschlichen Schlussfolgerns zu repräsentieren. Dies führte zur Entwicklung vieler Formalismen im Bereich der Wissensrepräsentation und des Automatischen Schlussfolgerns. Bis heute ist das Verstehen von Schlüsseleigenschaften im Verhalten von Agenten die auf logischen Formalismen aufbauen, ein herausforderndes Problem.

Antwortmengenprogrammierung (ASP) is solch ein Formalismus, der breite Anwendung findet, nicht zuletzt dank seiner hohen Ausdrucksstärke und der Verfügbarkeit von effizienten Softwaretools. Ein besonderes Anwendungsgebiet von ASP ist die Darstellung des Verhaltens von Agenten, da es eine ausdrucksstarke Umgebung zur Repräsentation von Handlungen und Veränderungen in der Welt. Dies macht ASP zu einem wünschenswerten Instrument zum Schlussfolgern über das Verhalten eines künstlichen Agenten. Existierende Arbeiten zum Verständnis von ASP-Programmen existieren, aber sie liefern oft Erklärungen mit zu vielen Details, welche verhinern, die entscheidenden Teile des Verhaltens eines Programmes zu erkennen.

Abstraktion ist ein Prozess, der von Menschen permanent zur Problemlösung und Ermittlung der wesentlichen Elemente einer Fragestellung verwendet wird. Dies führte dazu, die Verwendung von Abstraktion in der KI zu untersuchen, insbesondere um Probleme yu vereinfachen und beim Entwurf intelligenter Agenten und für die automatisierter Problemlösung. Durch das Weglassen von Details werden Szenarien auf diejenigen reduziert, die einfacher zu handhaben und zu verstehen sind. Details werden nur dann wieder zum Szenario hinzugefügt, wenn dies erforderlich ist. Überraschenderweise wurde Abstraktion, abgesehen von einigen Vereinfachungsmethoden, im Kontext der nichtmonotonen Wissensrepräsentation bisher nicht berücksichtigt, und insbesondere nicht im Bereich von ASP.

Diese Dissertation befasst sich mit der Herausforderung, die Schlüsselelemente im Verhalten eines Agenten mithilfe von Abstraktion und in einer ASP-Perspektive zu verstehen. Wir nähern uns dieser Herausforderung aus zwei Richtungen. Zunächst untersuchen wir eine Abstraktionsmethode, die das Verhalten des ursprünglichen Programms beibehält und gleichzeitig Details, die irrelevant für das Verhalten sind, aus der Problembeschreibung beseitigt. Zu diesem Zweck beschreiben wir eine formale Semantik für Agenten deren Verhalten auf einer vordefinierten Policy basiert mithilfe von Abstraktion über nicht-unterscheidbare Details. Zweitens machen wir den ersten Schritt um Abstraktion auch im Kontext von ASP einsetzen zu können. Wir konzentrieren uns auf zwei Ansätze für Abstraktion: (1) Abstraktion durch Auslassung und (2) Domänenabstraktion. Für beide Ansätze beschreiben wir eine neuartige Methode zur Erstellung einer Zusammenfassung des Originalprogrammes basierend auf einem reduzierten Symbolinventar. Unsere Methode stellt sicher, dass die Antwortmengen des Originalprogramms durch das abstrakte Programm überschätzt werden und so keine Lösungen verloren gehen können. Wir beschreiben eine Methodik, die es ermöglicht mit einer anfänglichen Abstraktion zu beginnen und automatisch eine verfeinerte Abstraktion zu finden, die eine konkrete Lösung des Originalproblems erzielt. Eine Evaluierung der implementierten Werkzeuge und der Methodik zeigen das Potenzial dieses neuartigen Ansatzes für die Problemanalyse, wo es darauf ankommt sich auf die wesentlichen Teile eines Programms zu konzentrieren die ein Problem unerfüllbar machen, sowie für die Berechnung von konkreten Antwortmengen eines ASP-Programms die nur relevante Details der Problembeschreibung widerspiegeln.

Abstract

Recent advances in the field of AI aggravate the challenge of understanding the behavior of the designed agent and distinguishing the core elements in the designed model that plays a role in the behavior. A logic-based approach for designing AI agents is about representing their knowledge of the world as a mathematical model through a powerful representation language with precisely defined syntax and semantics, that describes the capabilities of the agent and allows it to reason about the next course of actions. The desire to develop such a representation language emerges the challenges of representing commonsense knowledge and the nonmonotonic nature of human reasoning, which has led to the invention of many formalisms in the field of Knowledge Representation and Reasoning. However, understanding the key elements in the behavior of an agent that is designed using such formalisms remains to be a challenging problem.

Answer Set Programming (ASP) is one such formalism which is recognized as a knowledge representation and reasoning paradigm, currently widely used in problem solving thanks to its expressive power and the availability of efficient solvers. One particular application area of ASP is the representation of agent behavior, as it offers an expressive setting which is convenient for representing and reasoning about actions and change. This makes ASP a desirable tool for reasoning about the behavior of the agent. There have been investigations on understanding how ASP programs work, which however may lead to obtaining explanations with too many details that prevent one from seeing the crucial parts of the behavior.

Abstraction is a process that is unwittingly used by humans for problem solving and figuring out the key elements. This observation has led to investigations of using abstraction in Computer Science and AI to simplify problems, especially, in the design of intelligent agents and automated problem solving. By omitting details, scenarios are reduced to ones that are easier to deal with and to understand, by adding back further details only when necessary. Surprisingly, other than some simplification methods, abstraction has not been considered in the context of nonmonotonic knowledge representation and reasoning, and specifically not in ASP.

This thesis tackles the challenge of understanding the key elements in the behavior of an agent with an ASP perspective, by employing abstraction. We approach the challenge from two directions. First, we investigate the representation of an abstraction that preserves the behavior of the original program while getting rid of details irrelevant to the behavior. For this, we introduce a formal semantics for describing agent behavior following a designed policy by abstracting over the indistinguishable details, while preserving the behavior. Second, we make the initial step for employing abstraction in the context of Answer Set Programming, to be able to abstract over the irrelevant details of answer set programs. We focus on two approaches of abstraction: (1) abstraction by omission, and (2) domain abstraction, and introduce a method to construct an abstract program with a smaller vocabulary, by ensuring that the original program is over-approximated. We introduce an abstraction-&-refinement methodology that makes it possible to start with an initial abstraction and automatically achieve an abstraction with a concrete solution. The evaluations of the implemented tools of the methodology reveal the potential of the approach for problem analysis by focusing on the parts of the program that cause the unsatisfiability and by achieving concrete abstract answer sets that reflect relevant details only.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Background

There is an increasing need in understanding the behavior of an AI agent that is designed to perform a certain task. These autonomous agents are capable of reasoning about their actions and acting in the environment in order to satisfy their design objectives. One of the very first papers in AI [McC59] suggested a logic-based approach for designing such agents by representing their knowledge of the world as a mathematical model through a collection of formulas, which is viewed as the knowledge base (KB). Such a KB is to be constructed using a powerful representation language with precisely defined syntax and semantics, that allows to describe the capabilities of the agent, represent facts about the world and help the agent in reasoning about its course of actions. Starting with this groundbreaking view on the problem, the field of Knowledge Representation and Reasoning (KR) [VHLP08] has come a long way in addressing the challenges of representing knowledge and empowering the agents with reasoning capabilities. However, designing an agent that is capable of performing a particular activity then introduces the challenge of reasoning about the behavior of the agent in the environment in order to check whether it progresses as desired and to understand the core elements of the described model that plays a role in the behavior, which is the challenge that is tackled in this thesis.

One key issue in building a KB with a representation of the world has been to represent commonsense knowledge. The common tools that are used for formalizing mathematical reasoning fails in representing such knowledge, due to the need to represent the non-monotonic nature of human reasoning. Nonmonotonic reasoning [BMT11] refers to being able to withdraw previous conclusions about the world when receiving new information. For example, you know that standing in the rain gets you wet, but having an umbrella will keep you dry. If however you find out that the umbrella has holes, you can no longer conclude that you will stay dry. Expressing commonsense reasoning and nonmonotonicity has been the challenge that is tackled for many years which has led to the invention of many formalisms. The most famous of these are circumscription [McC81a, McC86], default logic [Rei80], and nonmonotonic modal logics [MD80, McD82, Moo85]. Bringing ideas for representing nonmonotonic reasoning into logic programming has led to the development of nonmonotonic logic programming semantics, and in particular answer sets [GL91, Bar03].

Answer Set Programming (ASP) [Lif08b, BET11] is a knowledge representation and reasoning paradigm widely used in problem solving thanks to its expressive power and the availability of efficient solvers. ASP has been applied in many areas of AI [EGL16], such as to represent knowledge (e.g., mathematical models of problems, behaviour of dynamic systems, beliefs and actions of agents) and to solve combinatorial search problems (e.g., planning, diagnosis) and knowledge-intensive problems (e.g., query answering, explanation generation). The idea is to declaratively represent a problem as a "program" whose models (called "answer sets" [GL91]) correspond to the solutions of the problem. One particular application area of ASP is the representation of agent behavior. ASP can be used to formalize actions, planning, and agent policies, in an expressive setting (e.g. direct and indirect action effects) [Lif99b, Bar03, EGL16], and has led to dedicated action languages [Lif99a, GL98a], which are a useful tool for defining actions and reasoning about them, by modeling dynamic systems as transition systems. The declarative setup of action languages helps in describing dynamic systems in an understandable, concise language. They also address the problems encountered when reasoning about actions, e.g., the frame problem [MH69], ramifications [GS88], and the qualification problem [McC81b]. As action languages are closely related to classical logic and ASP, they can be translated into logic programs and queried for computation.

The expressivity and representation power makes ASP a convenient tool for problem solving and for investigating ways to help in understanding the problem with its key elements. There have been studies in understanding how ASP programs find a solution (or none) to a problem which mainly focus on debugging answer sets [BGP$^+$07, GPST08, OPT10] or finding justifications [PSE09, ST13, CFF14].These approaches could also be used in understanding the problem at hand. However, as noted in [FS19], the obtained explanations may contain too many details which prevent one from seeing the crucial parts. This is where some notion of abstraction would come in handy.

Abstraction is a process that is exploited in human reasoning and understanding, by reasoning over the models of the world that are built mentally [Cra52, JL83]. Although the word itself comes from the meaning of "to draw away", there is no precise definition that is able to cover all meanings that it gains depending on its utilizations. Various meanings of abstraction are interpreted in different disciplines such as Philosophy, Cognitive Science, Art, Mathematics and Artificial Intelligence, with the shared consensus of the aim to "distill the essential" [SZ13]. Among the several interpretations on the meaning of abstraction, one that comes up is the capability of *abstract thinking*. This is achieved by removing the irrelevant details and identifying the "essence" of the problem [Kra07]. The notion of *relevance* is especially important in problem solving, since the problem at hand may become too complex to solve if every detail is taken into account. A good strategy to solve a complex problem is to start with a coarse solution and then refine it by adding back more details. For example, when planning a trip, first the destination is picked and a coarse travel plan is determined. Thinking about the precise details of the travel, such as taking the subway to the airport, comes later. This gives a *hierarchy* of levels of abstraction, with the lowest level containing all of the details. Another view on abstraction is the *generalization* aspect, which is the process of distinguishing the common properties among the objects. For example, the physical attributes of the plane, e.g., color and size, and their possible differences are irrelevant to the travel plan. We are (mostly) only interested in the fact that there is a plane that takes us from Vienna to Naples. Overall, the general aim of abstraction is to simplify the problem at hand to one that is easier to deal with and to understand.

For solving a problem and figuring out the key elements, humans unwittingly make use of abstraction. In AI and related combinatorial problems, such problems vary from moving blocks in a certain order to achieve a final configuration to finding a plan for an agent in an environment under constraints, and to constraint problems such as coloring the nodes of a given graph. Human problem solving typically relies on a more high-level/abstract view and focuses on certain details of a problem only when necessary. In graph coloring, for instance, isolated nodes can be viewed as one node and colored the same without thinking about the specific details. If a given graph is non-colorable, then we may try to find some subgraph (e.g., a clique) of it which causes the unsolvability, and would not care about other nodes in the graph. Similarly with blocks, if, for example, the labels of the blocks are not of importance, we would not take them into account when figuring out the actions. If from the given initial layout of the blocks the final configuration can not be achieved, we would try to find the particular blocks that cause this. As for figuring out the agent's behavior in an environment, we can look at the given constraints to predict what will happen. For example, if we know that the agent will traverse the environment by always moving to the farthest point it can observe, we can then deduce the path the agent will take by omitting the rest of the details in the environment and conclude whether the agent will manage to achieve its goal.

As it is commonly agreed that abstraction plays a key role in representing knowledge and in reasoning, the usage of the concept has also been investigated in the design of intelligent agents and automated problem solving. From the early days of AI research, abstraction has been a useful heuristic for problem solving. First the problem is solved in an abstracted space, and then the abstract solution is used to guide the search for a solution in the original space [NS72, Sac74, Kno94]. This idea has been used by the Planning community especially for computing heuristic functions to guide the plan search in the state space. Several abstraction methods were introduced towards this direction, especially to automatically compute abstractions that give a good heuristic [Ede01, HHH$^+$07, SH13]. However, it is well known that the success in solving a problem relies on how "good" the abstraction is. For this, theoretical approaches for defining abstractions with desired properties have been investigated [Hob90, GW92, NL95]. Apart from gaining efficiency which is shown to be not always the case [BJ95, HSD06], abstraction forms a basis to obtain high-level explanations and an understanding of the problem at hand.

The use of abstraction is of interest in other areas of Computer Science as well. Particularly interesting research has been taking place for Model Checking, an automated verification technique where the desired behavioral properties are checked over the model of a program [BK08, CHVB18]. This technique is applied for checking the behavior of reactive systems which interact with the environment according to their designed program and that usually display nondeterministic behavior. Such a behavior is modeled by a transition system that reflects all possible behaviors of the program, which then gives rise to the infamous *state explosion problem*. One way to tackle this is to define property preserving abstractions, such that if the desired property holds in the constructed abstract model then it also holds in the original model [CGL94, LGS+95, DGG97]. The seminal counterexample guided abstraction refinement (CEGAR) method [CGJ+03] is on automatically generating such abstractions. The method starts with an initial abstraction that over-approximates the behavior and then refines the abstraction upon encountering spurious violations, i.e., counterexamples, of the desired property. The process continues by achieving more precise abstractions until the property is proved or disproved by a concrete counterexample.

Surprisingly, abstraction has not been considered much in the context of nonmonotonic knowledge representation and reasoning. Simplification methods such as equivalence-based rewriting [GKK+08, Pea04], partial evaluation [BD97, JNS+06], or forgetting [Lei17], have been extensively studied. However, they strive for preserving the semantics, while abstraction may change it and lead to an over-approximation of the models (answer sets) of a program, in a modified language. The notion of abstraction could be useful for focusing on the relevant details in an answer set program for finding an answer set, or for realizing that no answer set exists, and disregarding the irrelevant ones. Once a notion of over-approximation is introduced, a CEGAR-style approach can be used to start with a highly coarse abstraction and automatically search for such an abstraction. This thesis makes the initial step of employing the notion of abstraction in the context of Answer Set Programming.

1.2 Contributions

The dissertation focuses on two forms of abstraction. First, we investigate the representation of an abstraction that preserves the behavior of the original program while getting rid of details irrelevant to the behavior. For this, we take an ASP-based perspective to the representation of reactive agents, by describing an iterative behavior that perceives the current state of the world, figures out the next actions, executes them and observes the outcomes. When employed in real world domains, these agents are expected to handle the changes in the environment or to fill the gaps in their incomplete knowledge while interacting with the environment, and still be able to achieve their goals. Usually these agents are provided with a policy that guides them in making decisions about their next actions. As these agents behave according to the policy, the behavior of the agent depends entirely on the information that the policy uses in the state and the successor state reached after executing the determined actions. Depending on the policy, not every information in the state may be relevant to the agent's behavior or the states that are passed through while executing the actions. We address the shortage of representations that are capable of modeling reactive policies which distinguishes relevant details of the states and the transitions.

We introduce a formal semantics for describing policies that express a reactive behavior for an agent, and connect our framework with the representation power of ASP-based action languages. In this framework, we combine components that are efficient for describing reactivity such as target establishment and (online) planning. Our representation allows one to analyze the flow of executing the given reactive policy, and lays foundations for checking properties of policies. Additionally, we recognize the issue of keeping irrelevant information in the state which the policy does not use, as having to represent such information adds to the state explosion problem when reasoning about the agent's behavior. For this, we consider a state clustering through the notion of *indistinguishability* that abstracts over such information, while preserving the behavior of the policy. This helps in checking the properties over the policy with a guarantee that the result also holds in the original system. The flexibility of the representation opens a range of possibilities for designing behaviors.

Second, we consider abstraction as an over-approximation, and employ the notion of abstraction in the context of Answer Set Programming, to be able to abstract over the irrelevant details of answer set programs. We introduce an abstraction from a program by constructing an abstract program with a smaller vocabulary, and ensuring that the original program is over-approximated, i.e., every original answer set can be mapped to some abstract answer set. We focus on two approaches of abstraction: (1) abstraction by omission, and (2) domain abstraction. The former is about omitting atoms from a program, i.e., to cluster them into $\top$ (truth), and consider the abstract program which is over the remaining atoms, while the latter investigates abstraction over non-ground ASP programs given a mapping over their domain (i.e., the Herbrand universe) that singles out domain elements. We study the properties of the introduced abstractions, and provide complexity results.

As spurious answer sets can be introduced by the over-approximation, one may need to go over all abstract answer sets until a concrete one is found. If the original program has no answer sets, all encountered abstract answer sets will be spurious. Eliminating the spurious answer sets is possible by refining the abstraction. However, checking the concreteness of an abstract answer set returns unsatisfiability, if it is spurious, without any reason on why this is the case. In order to obtain hints on spuriousness, we make use of ASP debugging approaches [BGP+07, OPT10], by altering them to be able to debug checking the existence of an original answer set that can be mapped to the abstract answer set. The refinement decisions are then made using the debugging hints obtained from the checking.

Inspired from CEGAR [CGJ+03], we introduce an abstraction-&-refinement methodology that starts with an initial abstraction and refines it repeatedly using hints that are obtained from checking the abstract answer sets, until a concrete solution (or unsatisfiability) is encountered. This makes it possible to automatically achieve an abstraction with a concrete solution. We have implemented the approach in the tools ASPARO, for omission-based abstraction, and DASPAR, for domain abstraction. Given an answer set program and an initial abstraction, these tools are able to automatically achieve an abstraction mapping that creates an abstract program where either a concrete answer set is encountered, or unsatisfiability is achieved.

We have conducted experimental evaluations to observe the achievement of abstract solutions to the problems. We evaluate the omission-based abstraction approach for unsatisfiable problems to observe its use in finding the unsatisfiability reason of a problem by keeping the atoms relevant for the unsatisfiability. For this, we introduce the notion of blocker sets as sets of atoms such that abstraction to them preserves unsatisfiability of a program. A minimal program is then a minimal cause of unsatisfiability. For unsatisfiable ASP programs, we observe that automatic abstraction refinement is able to catch the unsatisfiability without refining back to the original program. The evaluation of the domain abstraction approach focuses on the achieved abstractions on which a concrete solution to the problem is encountered. We compare the effects of different variations of the methodology with respect to the abstract answer sets picked, different forms of concreteness determination and the refinement decisions. The experiments show the potential of the approach to aid (in) program analysis as it allows for problem solving over abstract notions, by achieving concrete abstract answer sets that reflect relevant details only.

In order to handle problems that involve multi-dimensional structures, such as grid-cells, a differentiated view of an abstraction is needed for being able to provide insight that is similar to humans, by focusing on certain areas and abstracting away the rest. For this, we empower the domain abstraction to handle such a hierarchical view of abstraction that automatically adjusts the granularity towards the relevant details of the problem.

We have implemented the tool mDASPAR, which is an extension of DASPAR that handles multi-dimensional abstractions. We evaluate the approach in detecting the unsolvability of benchmarks problems involving grid-cells. In that we observe the capability of zooming in to the area which shows the reason for unsolvability of a problem instance. This becomes the machine's way of explanaining unsolvability, which is then compared with how humans provide explanations. A user experiment is conducted to compare the resulting abstractions with human explanations, which shows that such a hierarchic abstraction can provide intuitive and "to the point" explanations of unsolvability. The user experiment on human explanations also reveals the implicit abstraction capabilities of humans and the acknowledged need for studying the meaning of explanation.

Furthermore, we utilize mDASPAR for the problem of refuting policies in grid-cells. We observe that for a given instance, an abstraction is obtained that focuses on the area which shows a counterexample path that refutes the policy or which proves that the policy works.

1.3 Thesis Structure

The thesis is structured as follows.

- In Chapter 2, we provide an overview on representing agents and their actions (Section 2.1), Answer Set Programming (Section 2.2), the notion of abstraction (Section 2.3), and computational complexity (Section 2.4);
- in Chapter 3, we describe a formal semantics for describing policies that express a reactive behavior for an agent:
  - In Section 3.1, we introduce the general framework for modeling policies, where we consider components for describing reactivity such as target establishment and planning. We apply an indistinguishability notion on the states w.r.t. the behavior of the policy, and show the properties necessary to guarantee the preservation of the behavior in the abstract system;
  - in Section 3.2, we show the relation with ASP-based action languages;
  - Section 3.3 shows an extension of the framework to dynamic environments by employing the notion of maintenance.
- in Chapter 4, we make the first step towards employing the concept of abstraction in ASP as an over-approximation that achieves abstraction over the irrelevant aspects of answer set programs:
  - In Section 4.1, we introduce the abstraction notion to ASP, describe possible approaches and the CEGAR-style methodology that is considered;
  - Section 4.2 introduces abstraction by omitting atoms from the program and constructing over-approximations. For an application in unsatisfiable ASP programs, we introduce the notion of blocker sets as sets of atoms such that abstraction to them preserves unsatisfiability of a program;
  - Section 4.3 introduces abstraction over the domain of the program;
  - in order to handle the unavoidably introduced spurious abstract answer sets, in Section 4.4, we propose a method for determining refinements for the abstractions by employing ASP debugging methods.
  - the overall abstraction and refinement methodology is described in Section 4.5;
  - Section 4.6 points out the possibility to extend the domain abstraction to consider a multi-dimensionality in order to describe more sophisticated abstractions;
  - in Section 4.7, we address related work in ASP on simplification methods.
- Chapter 5 investigates possible applications of abstraction in understanding the key elements of problems, by abstracting away as many irrelevant details as possible that may be traced before finding a solution of a problem or realizing that it is unsolvable:

  - In Section 5.1, we describe the implementations for the abstraction and refinement methods;
  - Section 5.2 shows the evaluation results by focusing on the achievement of abstract solutions to the problems. Section 5.2.1 shows the results of using omission abstraction in finding satisfiability blockers of programs, and Section 5.2.2 reports about the achieved non-trivial domain abstractions and the results of having variations in the methodology;
  - in Section 5.3, we discuss the use of domain abstraction in understanding planning problems expressed in ASP.

- in Chapter 6, we focus on problems involving grid-cell structures to observe the use of a two-dimensional abstraction over the grids in focusing on the essential parts of the problem and achieving abstract solutions;
  - we begin by describing the problem types that we focus on in Section 6.1;
  - in Section 6.2, we define the 2-dimensional abstraction on grid-cells based on quad-trees;
  - Section 6.3 describes the implementation mDASPAR;
  - in Section 6.4, we evaluate the approach on unsatisfiable problems and conduct a user experiment for comparing the obtained abstract explanations;
  - in Section 6.5, we discuss the application of abstraction to the problem of policy refutation.
- We conclude in Chapter 7, by summarizing our contributions. We discuss related work and possible future directions.

1.3.1 Publications

Parts of the results in this thesis have been published. In the following we list the relevant publications and indicate which chapters or sections contain the corresponding contributions.

[SE16a] Zeynep G. Saribatur and Thomas Eiter. *Reactive policies with planning for action languages.* In Loizos Michael and Antonis Kakas, editors, *Proceedings of the 15th European Conference On Logics In Artificial Intelligence (JELIA)*, vol. 10021 of *Lecture Notes in Computer Science*, pages 463–480. Springer, 2016. Sections 3.1 and 3.2

[SBE17] Zeynep G. Saribatur, Chitta Baral, and Thomas Eiter. *Reactive maintenance policies over equalized states in dynamic environments.* In Eugénio Oliveira, Joao Gama, Zita Vale, and Henrique Lopes Cardoso, editors, *Proceedings of the 18th EPIA Conference on Artificial Intelligence*, vol. 10423 of *Lecture Notes in Computer Science*, pages 709–723. Springer, 2017. Section 3.3

[SE18a] Zeynep G. Saribatur and Thomas Eiter. *Omission-based abstraction for answer set programs.* In *Proceeding of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 42–51. AAAI Press, 2018.
Chapters 4 and 5

[SSE19] Zeynep G. Saribatur, Peter Schüller, and Thomas Eiter. *Abstraction for non-ground answer set programs.* In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA)*, vol. 11468 of *Lecture Notes in Computer Science*, pages 576–592. Springer, 2019. Chapters 4 and 5

[ESS19] Thomas Eiter, Zeynep G. Saribatur, and Peter Schüller. *Abstraction for zooming-in to unsolvability reasons of grid-cell problems.* In *Proceedings of the Workshop on Explainable Artificial Intelligence (XAI)*, 2019. To appear.
Section 4.6 and Chapter 6

Preliminary versions of [SE16a, SE18a, SSE19] have been published in the following venues.

[SE16b] Zeynep G. Saribatur, Thomas Eiter. *Reactive policies with planning for action languages.* In *Proceedings of the 16th International Workshop on Non-Monotonic Reasoning (NMR)*. 2016.

[SE18c] Zeynep G. Saribatur, Thomas Eiter. *Towards abstraction in ASP with an application on reasoning about agent policies.* In *Proceedings of the 12th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*, 2018.

A longer version of [SE18a] is published as a technical report [SE18b] and has been invited for submission to Theory and Practice of Logic Programming as a result of KR 2018 paper evaluations.

CHAPTER 2

Background

In this chapter, we give an overview of the concepts and notions that the thesis is built upon.

Outline In Section 2.1 we provide some background information on representing agents and their actions, and the challenges of reasoning about action and change. Section 2.2 presents the syntax and semantics of Answer Set Programming, and its use in representing actions. In Section 2.3 we give an overview on the notion of abstraction as used in the model checking community. Finally, we briefly recall the concepts of computational complexity in Section 2.4.

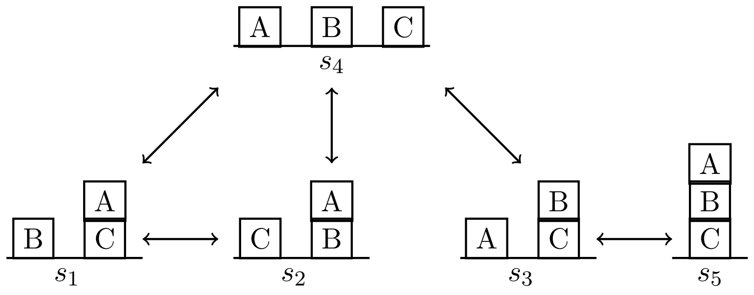
2.1 Agents, Actions and Change

We give overview on representing agents and their actions; for further details, see [RN03, GNT04].

An *agent* is an autonomous system the observes its environments through its sensors and acts upon that environment through its actuators. The agent's choice of actions for an observation relies on an *agent function* that maps any given observation to an action, which defines the agent's behavior. AI is about designing the agent program that implements the agent function mapping. A *model-based agent* maintains an internal state which is built and updated from the observations and the knowledge that is described in the agent program. This knowledge needs to contain some information of how the world evolves independently of the agent, and how the agent's actions affect the world, which is referred to as the *model* of the world. This way the agent uses the given model of the world to decide on its actions with the new observations.

There can be different properties of the environment in which the agent acts. The environment is *fully observable* if the agent can observe the complete state of the

Figure 2.1: Blocksworld



environment; it is *partially observable* if parts of the state remains unobserved. If the next state of the environment can be completely determined by the current state and the action executed by the agent, then the environment is *deterministic*. The environment can also change independently from the agent's actions, which may cause nondeterminism of the actions' outcome or the need to always check the current state of the environment while deciding on an action. Such environments are called to be *dynamic*. *Static* environments are the ones where the environment keeps still.

The behavior of an agent is usually represented by a state transition system.

Definition 2.1. *A* transition system *is a tuple* $\mathcal{T} = \langle S, \mathcal{A}, \Phi \rangle$ *where*

- *S is the finite set of states,*
- *$\mathcal{A}$ is the finite set of possible actions, and*
- *$\Phi : S \times \mathcal{A} \to 2^S$ is the transition function, which returns the set of possible successor states after applying a possible action in the current state.*

An action a is *applicable* in a state s if there is at least one state s' such that $s' \in \Phi(s, a)$ and a is *deterministic* at state s, if there is at most one such state. A transition system is *deterministic* if there is only one initial state and all actions are deterministic.

For any states $s, s' \in S$, we say that there is a *trajectory* between s and s', denoted by $s \to^{\sigma} s'$ for some action sequence $\sigma = \langle a_1, \ldots, a_n \rangle$ where $n \geq 0$, referred to as a *plan*, if there exist $s_0, \ldots, s_n \in S$ such that $s = s_0, s' = s_n$ and $s_{i+1} \in \Phi(s_i, a_{i+1})$ for all $0 \leq i < n$. Both the sequence $\langle s_0, \ldots, s_n \rangle$ of states and the sequence $\langle s_0, a_0, \ldots, a_{n-1}, s_n \rangle$ of alternating states and actions are referred as a trajectory.

Example 2.1 (Blocksworld Domain)**.** The blocksworld domain consists of 3 blocks, A,B,C, where the agent can move the blocks on top of each other (by respecting the alphabetical order, i.e., B can not be on top of A) and move them to the table. Figure 2.1 shows the transition system of this problem.

For (classical) planning problems, additionally an initial state and a (finite) set of goal states are given. Then the aim is to find a sequence of actions in a transition system that leads from the initial state to a given goal state.

Definition 2.2. *A* (classical) planning problem *is a tuple $P = \langle \mathcal{T}, s_0, G \rangle$ where*

- *$\mathcal{T}$ is a deterministic transition system,*
- *s_0 is the initial state, and*
- *G is the finite set of goal states.*

A solution *to P is a plan $\langle a_1, \ldots, a_n \rangle$ such that there exist a trajectory $s_0, a_1, s_1, \ldots, a_n, s_n$ where $s_n \in G$.*

An overview of the classical representation

The classical representation of the transition system and the planning problem uses first-order predicate logic. Each state of the world is described using a set of logical atoms that are true or false under some interpretation, and the actions are represented in terms of changes to the truth values of these atoms.

Let $\mathcal{L}$ denote a first-order language with finitely many predicate symbols and constant symbols, without function symbols. A *state* is a set of ground atoms of $\mathcal{L}$. An atom p holds in s iff $p \in s$. For a set g of literals, s *satisfies* g, denoted $s \models g$, when there is a substitution θ such that every positive literal of $\theta(g)$ is in s and no negated literal of $\theta(g)$ is in s.

Example 2.2 (ctd)**.** The blocksworld domain can be described using three blocks a, b, c and the predicates $on(X, Y), onTable(X), free(X)$. Concrete representations of states s_1 and s_5 in Figure 2.1 are

$$s_1 = \{on(a, c), onTable(c), onTable(b), free(a), free(b)\}$$
$$s_5 = \{on(a, b), on(b, c), onTable(c), free(a)\}.$$

The predicate symbols of which the truth value of the atoms vary from state to state are referred to as *fluents*. For example, the blocksworld domain has fluents on, $onTable$. Notice that the atoms that do not hold in the states are not explicitly specified. This is due to the *closed-world assumption* which assumes that any atom that is not mentioned at a state is assumed to be false.

An *action* of the transition system is represented in terms of the *preconditions* that must hold before it can be executed and the *effects* that occur after it has been executed. Actions are also referred to as *operators*.

Example 2.3 (ctd)**.** The actions of the blocksworld domain are described as below.

```
moveBTB(b1,b2)
  ;; move block b1 from the table on to block b2
  precond: onTable(b1), free(b1), free(b2)
  effect: on(b1,b2), -onTable(b1), -free(b2)

moveBBB(b1,b2,b3)
```

```
  ;; move block b1 from block b2 to block b3
  precond: on(b1,b2), free(b1), free(b3)
  effect: on(b1,b3), free(b2), -on(b1,b2), -free(b3)

moveBBT(b1,b2)
  ;; move block b1 from block b2 to the table
  precond: on(b1,b2), free(b1)
  effect: onTable(b1), -on(b1,b2)
```

An action is *applicable* in any state that satisfies its precondition. The *result* of executing an applicable action a in a state s is a state s' which the same as s except that any positive literal p in the effect of a is added to s' and any negative literal $\neg p$ is removed from s'. A plan is then a sequence of actions, when executed in the initial state, results in a state that satisfies the goal.

Example 2.4 (ctd)**.** For the transition system described as in Figure 2.1, a plan from s_1 to $G = \{s_5\}$ is the action sequence $\langle$`moveBBT(a,c)`,`moveBTB(b,c)`,`moveBTB(a,b)`$\rangle$.

The classical representation can also be extended to talk about disjunctive preconditions, quantified expressions which are considered in the planning language PDDL. However, the representation only works for problems with restrictive assumptions, such as fully observability, determinism, and having time implicitly defined.

Beyond classical planning

In real-world scenarios, it is rarely the case such a restricted representation is sufficient. Partial observability in an environment leads to having multiple initial states, over which a classical planning problem can not be defined. The nondeterminism in the effects of actions is another challenge of applications in real-world scenarios.

The problem of planning for nondeterministic and partially observable domains gave rise to different classes of planning problems. One interesting planning problem is *conformant planning*, which is on finding a plan that achieves the goal from all possible initial states and through all possible transitions caused by nondeterminism.

Definition 2.3. *A* conformant planning problem *is a tuple* $P = \langle \mathcal{T}, S_0, G\rangle$ *where*

- $\mathcal{T}$ *is a transition system,*
- S_0 *is the finite set of initial states, and*
- G *is the finite set of goal states.*

A solution *to* P *is a plan* $\langle a_1, \dots, a_n\rangle$ *such that for all trajectories* $s_0, a_1, s_1, \dots, a_n, s_n$ *with* $s_0 = s$ *for some* $s \in S_0$*, it holds that* $s_n \in G$*.*

Because of the uncertainty, all possible trajectories have to be taken into account, which makes the search much more difficult than classical planning. Different approaches have been proposed to tackle this problem; [SW98] proposed to develop separate plan graphs, extending the ideas of the classical planner Graphplan [BF97], for each possible world and search all graphs simultaneously, [Rin99] extended the approach of expressing planning problems as satisfiability problems [KS96] to QBFs, and [BG00b, HB06] proposed to encode conformant planning as heuristic search. Representing the problem using symbolic model checking techniques has also been considered [CR00, CRB04]. As an alternative, an approximation-based approach was introduced [STGM05, TSGM11] that is based on theory of action and change, and a technique to compile the problem into classical planning was proposed [PG09].

Conditional planning is another type planning problem which searches for a plan that conditions over the possible observations that can be made under the partial observability. A famous approach is to model this problem in Partially Observable Markov Decision Processes (POMDPs) [KLC98] which allows to assign probabilities to transitions. *Strong cyclic plans* [CPRT03] extend such a notion of finding "safe plans" with an iterative trial-and-error strategy that has the possibility of terminating and, once it has terminated, is guaranteed to achieve the goal. The problem of strong planning [BCRT06] is on generating conditional plans that are guaranteed to achieve the goal in spite of the nondeterminism and partial observability of the domain, which can be solved by turning the problem into a non-deterministic search problem in the belief space [HB05, BKS06], or alternatively, by compiling into a non-deterministic problem in the state space [APG09].

In order tackle these problems efficiently, the focus has been mainly on formalisms with limited expressive power.

Need for more expressiveness

Expressing reasoning about actions and change in a logical formalism gives rise to various challenges. The most famous one is the *frame problem* [MH69] which is the problem of representing what does not change in the world when an action is made. For example, moving a block will not change the positions of all of the remaining blocks. A closely related challenge is the *ramification problem* [GS88] which is about representing all the consequences of an action execution. For example, moving the table causes all of the blocks on top to move along with it. Depending on the level of details represented in the state, expressing all such indirect effects can become a difficult task. The *qualification problem* [McC81b] is the problem of specifying when an action qualifies for execution. For example, to be able to succesfully move the table, the table should not be too heavy, it should not be glued to the floor, it should be in one piece, etc. Fully representing the conditions for a successful execution of an action may require to consider many possible scenarios.

Humans approach such problems by disregarding the unlikely situations unless there are hints to their presence. For example, as it is unlikely that tables get glued to the floor such a scenario does not come to mind when deciding to lift the table. However, when such situations do occur, then some of the previous conclusions about the world will get retracted, since the new information shows the change. When representing change, this is referred to as the *commonsense of inertia*; the state of the object remains the same unless there is evidence that it changed.

The above problems showed that expressing the human reasoning view over the changes in the world requires *nonmonotonicity*. Motivated by this observation, nonmonotonic formalisms have been developed, the most famous ones being circumscription [McC81a, McC86], default logic [Rei80], and nonmonotonic modal logics [MD80, McD82, Moo85]. In the meantime, a separate research direction have been focusing on developing declarative programming languages, particularly logic programming [Llo87, CR96], with the idea of combining logical knowledge representation with the theory of automated deduction. Investigating ways to extend this language with nonmonotonic features resulted in the proposal of stable-model semantics [GL88] and well-founded semantics [VGRS91], then led to the proposal of an extended logic programming language [GL91] which is now known as Answer Set Programming.

2.2 Answer Set Programming

Answer Set Programming is a declarative problem solving paradigm oriented towards difficult search problems. The basic idea of ASP is to encode the problem through a non-monotonic logic program in a declarative manner so that rather than specifying a concrete algorithm that solves the problem, one describes the solutions in terms of rules and constraints. Then, an ASP solver is used to compute the models (i.e., answer sets or stable models) of this problem encoding, which show the solutions to the problem. The success of ASP in many practical applications has been encouraged by the availability of efficient ASP solvers, such as DLV [LPF$^+$06], Smodels [SNS02], and Clasp [GKNS07].

Next, we present the formal syntax and semantics of ASP, and remark about further notions. We then show how ASP can be used in representing actions and change.

Syntax

We have a function-free first order vocabulary $\mathcal{L} = (\mathcal{P}, \mathcal{C})$ consisting of non-empty finite sets $\mathcal{P}$ of predicates and $\mathcal{C}$ of constants. Let $\mathcal{V}$ represent the (finite) set of variable symbols. A *term* is either a constant from $\mathcal{C}$ or a variable from $\mathcal{V}$. An *atom* is defined as $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$, each t_i is a term, and k is called the *arity* of p. Atoms are called *propositional* if $k = 0$ and *ground* if they do not contain variables. A *literal* l is an atom a or a strongly ("classically") negated atom $\neg a$, and a *default-negated* literal is a literal of the form *not* l, which evaluates to *true* if the truth of l cannot be proven.

The intuitive meaning of *not* a evaluating to *true* is that "a cannot be proved (derived) using rules and is false by default (or believed to be false)". This is different from proving that a is false, which is expressed by $\neg a$.

A *rule* r is an expression of the form

$$\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_m, \mathit{not}\ \alpha_{m+1}, \ldots, \mathit{not}\ \alpha_n, \quad 0 \leq m \leq n, \tag{2.1}$$

where each α_i is a literal. We refer to α_0 as the *head* of r, and $\alpha_1, \ldots, \alpha_m, \mathit{not}\ \alpha_{m+1}, \ldots,$ $\mathit{not}\ \alpha_n$ as the *body* or r. A rule is called a *constraint* if α_0 is falsity ($\perp$, then omitted) and a *fact*, if $n = 0$.

We also write r as $\alpha_0 \leftarrow B(r)$, such that $H(r) = \alpha_0$ denotes the head and $B(r)$ denotes the set of all the body literals $B^+(r) \cup B^-(r)$, where $B^+(r) = \{\alpha_1, \ldots, \alpha_m\}$ is the *positive body* and $B^-(r) = \{\alpha_{m+1}, \ \ldots, \alpha_n\}$ is the *negative body* of r; thus, we have $H(r) \leftarrow B^+(r), \mathit{not}\ B^-(r)$. Furthermore, we let $B^\pm(r) = B^+(r) \cup B^-(r)$. We occasionally omit r from $B^\pm(r)$, $B(r)$ etc. if r is understood.

A *program* Π is a finite set of rules. A program Π is a called *positive program* if for all $r \in \Pi$, $n = m$, i.e., $B^-(r) = \emptyset$. If, additionally, no classical negation occurs in Π, then Π is called a *normal program.* A rule is *ground* if all literals in $H(r) \cup B(r)$ are ground, and a program is *ground* if all its rules are ground.

Semantics

The answer set semantics is defined via ground programs. For a program Π, we define its *ground instantiation* as follows.

Given a program Π, its *Herbrand universe*, denoted by HU_Π, is the set of all constant symbols $C \subseteq \mathcal{C}$ appearing in Π; in case there is no constant symbol, then $HU_\Pi = \{c\}$ for some arbitraty constant symbol. The *Herbrand base* of a program Π, denoted by HB_Π, is the set of all ground literals constructed using predicates from $\mathcal{P}$ and constants from $\mathcal{C}$.

The *ground instances of a rule* $r \in \Pi$, denoted by $grd(r)$, is obtained by replacing all variables in r with constant symbols in HU_Π. The *grounding* of a program Π then becomes $grd(\Pi) = \bigcup_{r\in\Pi} grd(r)$. To group the rules in $grd(\Pi)$ with the same head q, we use $\mathit{def}(q, \Pi) = \{r \in \Pi \mid H(r) = q\}$.

Let Π be a ground program. A set $L \subseteq HB_\Pi$ of literals is *consistent*, if $p, \neg p \nsubseteq L$ for every atom $p \in HB_\Pi$. An *interpretation* I is a consistent subset of HB_Π. An interpretation I *satisfies* a rule $r \in \Pi$, denoted by $I \models r$, if $H(r) \subseteq I$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. An interpretaion is a *model* of Π, denoted by $I \models \Pi$, if $I \models r$ for all $r \in \Pi$. A model I is *minimal*, if there is no model J of Π such that $J \subset I$.

Example 2.5. Consider the program Π below and the interpretation $I = \{a, b, d\}$.

$$\begin{aligned} c &\leftarrow \mathit{not}\ d. \\ d &\leftarrow \mathit{not}\ c. \\ a &\leftarrow \mathit{not}\ b, c. \\ b &\leftarrow d. \end{aligned}$$

I is a model of Π, but it is not minimal, since the interpretation $I' = \{b, d\}$ is also a model of Π.

Definition 2.4 (GL-reduct)**.** *The* Gelfond-Lifschitz *(GL-)*reduct *of a program* Π *relative to an interpretation* $I \subseteq HB_\Pi$*, denoted by* Π^I*, is the ground positive program obtained from* $grd(\Pi)$ *when each rule* $H(r) \leftarrow B^+(r), not\ B^-(r)$

(i) with $B^-(r) \cap I \neq \emptyset$ *is deleted, and*
(ii) is replaced by $H(r) \leftarrow B^+(r)$*, otherwise.*

Informally, the first step is to remove the rules where I contradicts a default negated literal, and from the remaining rules, the second step removes their negative body. An interpretation I is an *answer set* of Π, if it is the minimal model of the GL-reduct Π^I.

Apart from the GL-reduct which is considered to be the standard definition for stable models (i.e., answer sets), a collection of other definitions can be found in [Lif08a].

An alternative formulation introduced in [FLP04] considers a different definition of the reduct.

Definition 2.5 (FLP-reduct)**.** *The* FLP-reduct *of a program* Π *relative to an interpretation* I*, denoted by* $f\Pi^I$*, is the program*

$$f\Pi^I = \{r \in grd(\Pi) \mid I \models B(r)\}$$

which is obtained by removing the rules whose body is not satisfied by I*.*

An interpretation I is an answer set of Π, if it is the minimal model of the FPL-reduct $f\Pi^I$. The answer sets according to both definitions coincide for the programs that we consider in this work.

The set of answer sets of a program Π is denoted as $AS(\Pi)$. A program Π is *unsatisfiable*, if $AS(\Pi) = \emptyset$.

Example 2.6 (ctd)**.** Π has two answer sets, viz. $I_1 = \{c, a\}$ and $I_2 = \{d, b\}$; indeed,

- $\Pi^{I_1} = \{c \leftarrow not\ d.;\ a \leftarrow not\ b, c.\}$ and I_1 is a minimal model of Π^{I_1}; similarly,
- $\Pi^{I_2} = \{d \leftarrow not\ c.;\ b \leftarrow d.\}$ has I_2 among its minimal models.

The main reasoning tasks related with programs under the answer set semantics are as follows:

- *Answer set existence*: decide whether a given program Π has an answer set, i.e., decide if $AS(\Pi) = \emptyset$;
- *Answer set checking*: decide whether a given interpretation I is an answer set of a given program Π, i.e., decide if $I \in AS(\Pi)$;
- *Brave (resp. Cautious) reasoning*: decide whether a given ground formula φ holds in some (resp. every) answer set of a given Π.

Further notions and extensions

The *dependency graph* of a program Π, denoted G_Π is a directed graph (V, E), where the vertices V equals HB_Π, and the edges $E = E^+ \cup E^-$ consist of positive edges E^+ from any $q = H(r)$ to any $p_1 \in B^+(r)$ and negative edges E^- from any $q = H(r)$ to any $p_2 \in B^-(r)$, for all $r \in grd(\Pi)$.

Example 2.7 (ctd)**.** G_Π has positive edges $a \to c$ and $b \to d$ and negative edges $c \to d$, $d \to c$ and $a \to b$.

A non-empty set L of ground literals describes an *odd loop* of Π if for each pair $p, q \in L$ there is a path τ from p to q in G_Π with odd number of negative edges. Constraints are viewed as simple odd loops. As well-known, Π is satisfiable, if it contains no odd loop.

Example 2.8 (ctd)**.** The program Π has no odd loop, and thus has some answer set.

The *positive dependency graph* is the dependency graph containing only the positive edges, denoted by G^+_Π. A program Π is *tight* if G^+_Π is acyclic. A non-empty set L of ground literals describes a *positive loop* of Π if for each pair $p, q \in L$ there is a path τ from p to q in G^+_Π such that each literal in τ is in L.

An alternative characterization of answer sets is defined in [Lee05], by using a notion of externally supportedness as follows. A set L of ground literals is *externally supported by* Π *w.r.t. an interpretation* I if there is a rule $r \in grd(\Pi)$ such that (i) $H(r) \cap L \neq \emptyset$, (ii) $I \models B^+(r)$ and $B^-(r) \cap I = \emptyset$ and (iii) $B^+(r) \cap L = \emptyset$. The third condition ensures that the support for $H(r)$ in L comes from *outside* of L.

Proposition 2.1. *I is an answer set of Π iff $I \models \Pi$ and every loop L of Π such that $L \subseteq I$ is externally supported by Π w.r.t. I.*

This characterization corresponds to one by Leone et al. [LRS97] in terms of *unfounded sets* where a set L of ground literals is *unfounded* w.r.t. an interpretation I iff L is not externally supported by Π w.r.t. I, i.e., literals in L only have support by themselves. A literal q is *unsupported* by an interpretation I if for each $r \in def(q, \Pi)$, $B^+(r) \nsubseteq I$ or $B^-(r) \cap I \neq \emptyset$ [VGRS91].

Stratified programs have the property that an ordering for the evaluation of the rules in the program can be found, through which the value of negative literals can be predetermined. More formally, a program Π is *stratified*, if a set Σ can be defined as a partitioning $\Sigma_1, \dots, \Sigma_k$ of ground atoms in Π, such that for each $p \in \Sigma_i$ and $q \in \Sigma_j$ (i) if there exists a rule r such that $q \in H(r)$ and $p \in B^+(r)$ then $j \geq i$, and (ii) if there exists a rule r such that $q \in H(r)$ and $p \in B^+(r)$ then $j < i$. Such a partitioning Σ is called a *stratification* of Π. The stratification specifies an evaluation order by which one can evaluate the program using an iterative minimal model computation along the partitions. This sequential evaluation can be used to compute a stable model; notably, a stratified program has a unique stable model. A program is stratified iff its dependency graph contains no cycles with a negative edge [ABW88].

Choice rules are a syntactic extension that are of the form $\{\alpha\} \leftarrow B$, which stands for the rules $\alpha \leftarrow B, \mathit{not}\ \overline{\alpha}$ and $\overline{\alpha} \leftarrow B, \mathit{not}\, \alpha$, where $\overline{\alpha}$ is a new atom. Cardinality constraints and conditional literals are further common syntactic extensions [SNS02]; in particular, $i_\ell\{\, a(X) : b(X)\,\}i_u$ is true whenever at least i_ℓ and at most i_u instances of $a(X)$ subject to $b(X)$ are true.

Example 2.9. Consider the following rules.

$$1\{a(X,Y,Z) : b(X), c(Y)\}1 \leftarrow d(Z). \tag{2.2}$$

$$\{a(X) : b(X); c(X,Y) : b(X), d(Y)\}1. \tag{2.3}$$

Rule (2.2) states that exactly one instance of $a(X,Y,Z)$ subject to $b(X)$ and $c(Y)$ has to be true, where the value of Z depends on the instance of $d(Z)$ that is true, and rule (2.3) states that at most one instance of either $a(X)$ subject to $b(X)$ or $c(X,Y)$ subject to $b(X)$ and $d(Y)$ has to be true.

A *weak constraint* [LPF$^+$06] is of the form

$$:\sim \alpha_1, \dots, \alpha_m, \mathit{not}\ \alpha_{m+1}, \dots, \mathit{not}\ \alpha_n.[w : l]$$

where w (the *weight*) and l (the *level*) are positive integer constants or variables. This is a constraint that can be violated with a cost w. When assigning a cost to an answer set, the costs of all violated (instances of) weak constraints (grouped by levels of priorities l) are added up. Among all answer sets, those whose cost vector is lexicographically smallest are chosen as *optimal answer sets*. Using weak constraints is a convenient way of performing optimizations.

ASP solvers first generate a grounding of the given program, and then a search for an answer set is conducted over the ground program. In order to help with efficient grounding, some syntactic restrictions are imposed on the rules of the input program. *Rule safety* [LPF$^+$06] is obtained by having every variable in a rule occur in some positive body literal, and *domain-restriction* [SN01] is on ensuring that every variable in a rule occurs in a positive *domain predicate*, which are predicates not defined via negative recursion or using choice rules.

The syntax of the input program for the ASP solvers contains `:-` instead of $\leftarrow$ and `-` for strong negation $\neg$. The constraint $\perp \leftarrow B(r)$ is represented by omitting $\perp$ in the corresponding rule. Each rule must be terminated with a dot.

2.2.1 Actions and ASP

Representing actions and change in ASP is achieved by adding a time variable to the atoms, and introducing action atoms that cause changes over time, where the actions are defined by their preconditions and effects over the atoms. ASP can then be used to the describe the dynamic domain by a "history program" [Lif99b] whose answer sets represent possible evolutions of the system over a time interval.

For illustration, the following rule describes a *direct effect* of the action $goTo(X, Y)$ over the robot's location $rAt(X, Y)$.

$$rAt(X, Y, T{+}1) \leftarrow goTo(X, Y, T). \tag{2.4}$$

Actions can also have *indirect effects* over the state (rules not mentioning actions); e.g., the robot location is visited:

$$visited(X, Y, T) \leftarrow rAt(X, Y, T). \tag{2.5}$$

Inertia laws (unaffectedness) can be elegantly expressed, e.g.

$$rAt(X,\ Y,\ T{+}1) \ \leftarrow\ rAt(X, Y, T), not\, \neg rAt(X, Y, T{+}1).$$

says that the robot location remains by default the same. One can also give further restrictions on the state, e.g., the robot and an obstacle can never be in the same cell. place:

$$\leftarrow rAt(X, Y, T), obsAt(X, Y, T). \tag{2.6}$$

Example 2.10 (ctd)**.** An ASP encoding for the blocksworld problem is shown in Figure 2.2. The planning problem is then expressed by defining the initial state as a set of facts

$$\begin{aligned}&on(a, c, 0).\\&on(c, table, 0).\\&on(b, table, 0).\end{aligned}$$

and the goal state through additional constraints

$$\begin{aligned}&\bot \leftarrow not\ on(a, b, t_{max}).\\&\bot \leftarrow not\ on(b, c, t_{max}).\\&\bot \leftarrow not\ on(c, table, t_{max}).\end{aligned}$$

that the answer set should satify. The answer set for setting $t_{max} = 3$ contains the action atoms $\{move(a, table, 0), move(b, c, 1), move(a, b, 2)\}$ which describes the solution plan.

Notice that the encoding shown in Example 2.10 can be used to find plans of different lengths and/or of different number of blocks by only modifying the facts and the goal constraints. The description of the planning problem can easily be modified by adjusting the constraints and/or adding new ones if necessary e.g., no more than two blocks can be on the table at the same time.

Its elegant way of expressing direct/indirect effects of actions, and addressing the frame and ramification problems makes ASP a suitable representation model for problems regarding dynamic domains [BG00a, GK14]. The notions behind the expressive power of ASP has also led to dedicated *action languages* [GL98a] which are higher-level languages designed for specifying state-action-state transition diagrams.

Figure 2.2: Blocksworld for one gripper (slightly modified from [Lif02])

$time(0 \ldots t_{max}).timea(0 \ldots t_{max}-1).block(a).block(b).block(c).loc(table).$
$loc(B) \leftarrow block(B).$
% guess
$\{move(B,L,T) : block(B), loc(L)\} \leftarrow timea(T).$
% effect of moving a block
$on(B,L,T+1) \leftarrow move(B,L,T), block(B), loc(L), timea(T).$
% inertia
$on(B,L,T+1) \leftarrow on(B,L,T), not\ \neg on(B,L,T+1), loc(L), block(B), timea(T).$
% uniqueness of location
$\neg on(B,L_1,T) \leftarrow on(B,L,T), L \neq L_1, block(B), loc(L), loc(L_1), time(T).$
% two blocks cannot be on top of the same block
$\bot \leftarrow 2\{on(B_1,B,T) : block(B_1)\}, block(B), time(T).$
% a block can't be moved unless it is clear
$\bot \leftarrow move(B,L,T), on(B_1,B,T), block(B), block(B_1), loc(L), timea(T).$
% no concurrent actions
$\bot \leftarrow move(B,L,T), move(B_1,L_1,T), B \neq B_1.$
$\bot \leftarrow move(B,L,T), move(B_1,L_1,T), L \neq L_1.$

Action languages

The aim to describe actions and their effects with a formal language that is inspired from the use of natural language has led to dedicated high-level *action languages*. This line of research started with the investigation of using ASP for representing actions and their effects by introducing the action language $\mathcal{A}$ [GL93]. Later, considering further extensions and properties has led to the invention of the action languages $\mathcal{B}$ (having indirect effects) [GL98a], $\mathcal{C}$ (a view of causality) [GL98b], $\mathcal{K}$ [EFL$^+$03] (incomplete knowledge) and many more.

These languages are based on formalizing actions by describing a particular type of transition systems based on action signatures. An *action signature* consists of a set **V** of value names, a set **F** of fluent names and a set **A** of action names.

Definition 2.6. *A transition system $\langle S, V, \Phi\rangle$ of an action signature $\langle \boldsymbol{V}, \boldsymbol{F}, \boldsymbol{A}\rangle$ consists of*

- *a set S of states,*
- *a function $V : \boldsymbol{F} \times S \rightarrow \boldsymbol{V}$, and*
- *a subset Φ of $\subseteq S \times \boldsymbol{A} \times S$*

Any fluent P has a *value* $V(P,s)$ in any *state* s *of the world*. The states s' such that $\langle s,a,s'\rangle \in \Phi$ are the possible *results of the execution* of the action a in the state s. An action a is *executable* at a state s, if there is at least one state s' such that $\langle s,a,s'\rangle \in \Phi$ and a is *deterministic* at state s, if there is at most one such state.

A transition system can be thought as a labeled directed graph, where a state s is represented by a vertex labeled with $P \to V(P,s)$, that gives the value of the fluents. Every triple $\langle s,a,s'\rangle \in \Phi$ is represented by an edge leading from a state s to s' and labeled by a.

Concurrent execution of actions can be defined by considering transitions $\langle s,A,s'\rangle$ with a set $A \subseteq \mathbf{A}$ of actions, where each action $a \in A$ is executable at s. Here we confine to *propositional* action signatures, which have truth values as value names, $\mathbf{V}=\{\mathsf{f},\mathsf{t}\}$.

The action language $\mathcal{C}$ [GL98b] is based on *causality*, where one distinguishes the cases that a fact "holds" and that it is "caused". Its syntax consists of static and dynamic laws of the form

$$\begin{array}{ll}\textbf{caused } F & \textbf{if } G,\\ \textbf{caused } F & \textbf{if } G \textbf{ after } U\end{array}$$

respectively, where F and G are formulas of fluents, and U is a formula containing fluents and elementary actions.

A translation of $\mathcal{C}$ into ASP has been shown in [LT99], and an implementation using satisfiability solvers led to the causal calculator (CCALC) [MT98, McC99]. In this thesis, we focus on a fragment of the language C where the heads of the static and dynamic laws only consist of literals. This restriction on the laws reduces the cost of evaluating the transitions to polynomial time.

2.3 Abstraction

In this section, we give an overview of the notion of abstraction as over-approximation, commonly used in model checking (for further details, see [CGL94, CGJ$^+$03]).

We extend the transition system in Definition 2.1 to also contain a set $S_0 \subseteq S$ of initial states, i.e., $\mathcal{T} = \langle S, S_0, \mathcal{A}, \Phi\rangle$. An *abstraction mapping* is a surjection $h : S \to \hat{S}$ that induces an equivalence relation $\sim\ \subseteq S \times S$ by

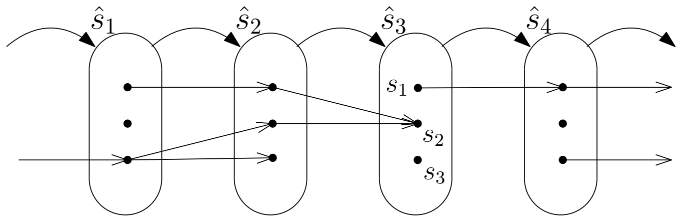
$$d \sim e \text{ iff } h(d) = h(e).$$

Definition 2.7. *Let $\mathcal{T} = \langle S, S_0, \mathcal{A}, \Phi\rangle$ and $\hat{\mathcal{T}} = \langle \hat{S}, \hat{S}_0, \mathcal{A}, \hat{\Phi}\rangle$ be transition systems, and h be an abstraction mapping $h : S \to \hat{S}$. $\hat{\mathcal{T}}$* over-approximates *$\mathcal{T}$ if*

(1) for all $d \in S$ such that $d \in S_0$, $h(d) \in \hat{S}_0$ holds; and
(2) for all $d, e \in S$ such that $e \in \Phi(d,a)$, for some $a \in \mathcal{A}$, $h(e) \in \hat{\Phi}(h(d),a)$ holds.

In [CGL94], the over-approximation is denoted by $\mathcal{T} \sqsubseteq_h \hat{\mathcal{T}}$. By definition, a state $\hat{s}$ of $\hat{\mathcal{T}}$ represents all those states s of $\mathcal{T}$ for which $h(s) = \hat{s}$, and it *simulates* each such s, so if s

Figure 2.3: A spurious abstract trajectory



has a transition to some s', then $\hat{s}$ will have a transition to $\hat{s}' = h(s')$. Similarly, if $\mathcal{T}$ has s as an initial state, then $\hat{\mathcal{T}}$ also has $\hat{s}$ as an initial state. Thus, we obtain the following result.

Proposition 2.2 (Lemma 5.5 in [CGL94]). *Assume $\mathcal{T} \sqsubseteq_h \hat{\mathcal{T}}$. If $\tau = \langle s_0, \ldots, s_n \rangle$ is a trajectory of $\mathcal{T}$, then there exists a trajectory $\hat{\tau} = \langle \hat{s}_0, \ldots, \hat{s}_n \rangle$ in $\hat{\mathcal{T}}$, where $h(s_i) = \hat{s}_i, 0 \leq i \leq n$.*

Knowing that each trajectory is preserved in the over-approximation, we get the following result.

Corollary 2.3. *If $\hat{\mathcal{T}}$ contains no trajectory from $\hat{S}$ to a set $\hat{S}'$ of states, then for all $s \in S$ and s' such that $h(s') \in \hat{S}'$, $\mathcal{T}$ does not contain a trajectory from s to s'.*

The result in Corollary 2.3 is used for checking *universal* properties over the abstract model. These properties are expected to hold *for all paths starting from* S_0. If the abstract model satisfies the property, i.e., no trajectory that refutes the property exists, then by Corollary 2.3, we can conclude that the original model also satisfies the property. The reverse of these results however is not guaranteed, unless $\hat{\mathcal{T}}$ is an *exact approximation* of $\mathcal{T}$ [CGL94]. The over-approximation may cause in obtaining spurious trajectories in $\hat{\mathcal{T}}$, which do not have a corresponding original trajectory in $\mathcal{T}$.

Definition 2.8. *Let $\hat{\mathcal{T}}$ be an over-approximation of $\mathcal{T}$. A trajectory $\hat{\tau} = \langle \hat{s}_0, \ldots, \hat{s}_n \rangle$ of $\hat{\mathcal{T}}$ is* spurious *if there exists no trajectory $\tau = \langle s_0, \ldots, s_n \rangle$ such that $h(s_i) = \hat{s}_i, 0 \leq i \leq n$.*

The reason for spuriousness is not being able to trace the abstract trajectory in the original model.

Example 2.11. Figure 2.3 shows a part $\langle \ldots, \hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4, \ldots \rangle$ of an abstract trajectory, which is spurious due to the state $\hat{s}_3$, since no original trajectory can be found that matches the transition from $\hat{s}_3$ to $\hat{s}_4$ once some state in $\hat{s}_3$ is reached from a state in $\hat{s}_2$. Here, s_2 is a dead-end state which is reachable from the initial state of the trajectory but has no outgoing transitions to match the abstract trajectory, and s_1 is a bad state which is not reachable but has an outgoing transition that caused to have the spurious trajectory.

Eliminating spurious trajectories can be done by *refining* the abstract mapping h to a more fine-grained abstraction. The refinement methodoloy introduced by [CGJ+03] is based on refining the abstraction so that the dead-end states and the bad-states are no longer mapped to the same abstract state.

The well-known counterexample guided abstraction refinement (CEGAR) method [CGJ+03] starts with an initial abstraction on a given program and checks the desired property over the abstract program. Upon encountering spurious solutions, the abstraction is refined by removing the spurious transitions observed through the solution, so that the spurious solution is eliminated from the abstraction. This iteration continues until a concrete solution is found.

2.4 Computational Complexity

We assume familiarity with basic concepts of complexity theory, e.g., Turing machines, complexity classes and reductions. For comprehensive details we refer to [Pap03] and [AB09]. In the following, we briefly recall some notions needed in this work.

We denote by P (resp. NP) the classes of *decision problems* (i.e., computational problems with yes/no answer) which can be solved in polynomial time by deterministic (resp. nondeterministic) Turing machines. For a complexity class C, class co-C denotes all the decision problems whose complement is in C. Given a complexity class C, a C-oracle models computations with calls to subroutines and decides a given problem from C in a single step. For a complexity class C and an oracle A, C^A denotes a class of problems that can be decided by a Turing machine within the time bound of C, by invoking an A-oracle. The *polynomial hierarchy* consists of complexity classes $\Sigma_k^{\mathsf{P}}, \Pi_k^{\mathsf{P}}$ and Δ_k^{P} defined by

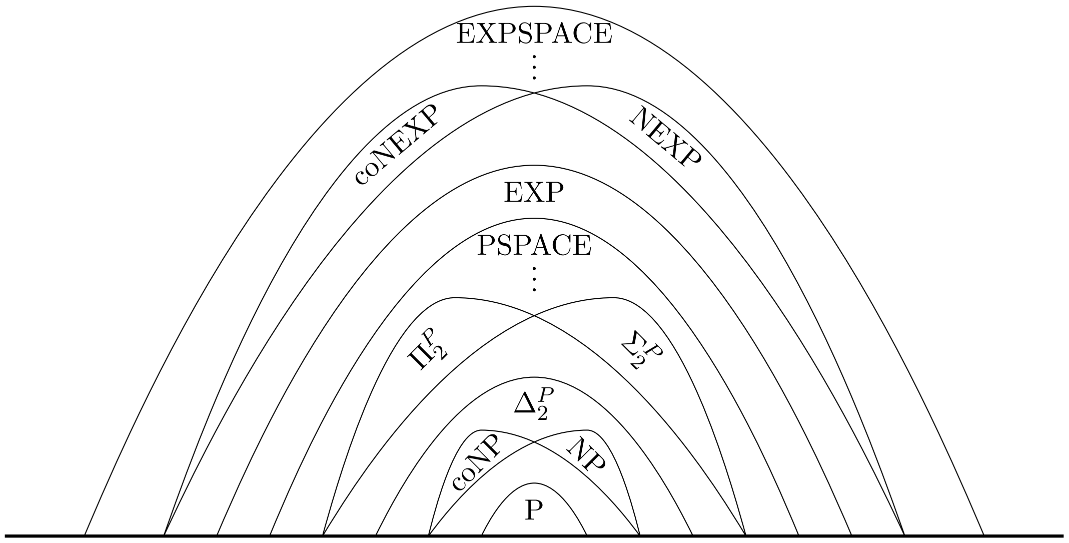
$$\begin{aligned}
\Sigma_0^{\mathsf{P}} = \Pi_0^{\mathsf{P}} = \Delta_0^{\mathsf{P}} &= \mathsf{P}\\
\Delta_{k+1}^{\mathsf{P}} &= \mathsf{P}^{\Sigma_k^{\mathsf{P}}}\\
\Sigma_{k+1}^{\mathsf{P}} &= \mathsf{NP}^{\Sigma_k^{\mathsf{P}}}\\
\Pi_{k+1}^{\mathsf{P}} &= \mathrm{co} - \Sigma_{k+1}^{\mathsf{P}}
\end{aligned}$$

for all $k \geq 0$. Note that $\Sigma_1^{\mathsf{P}} = \mathsf{NP}, \Pi_1^{\mathsf{P}} = \mathrm{co}\mathsf{NP}$ and $\Delta_1^{\mathsf{P}} = \mathsf{P}$.

PSPACE (resp. NPSPACE) is the class of decision problems solvable by deterministic (resp. nondeterministic) Turing machines in polynomial space. The complexity class EXP (resp. NEXP) consists of decision problems which can be solved in exponential time by deterministic (resp. nondeterministic) Turing machines. Figure 2.4 illustrates the relationships between the complexity classes used in our results.

Planning. Plan existence in a propositional domain with deterministic actions (i.e., deterministic transition relation) and fully observable initial states is PSPACE-complete

Figure 2.4: Relationships between complexity classes (according to current complexity hypotheses)



[Byl94]. Conformant planning in an unobservable propositional domain is EXPSPACE-complete [HJ99] and in partially-observable domains it is 2EXPSPACE-complete [Bon10]. For polynomial-length plans, with partial information on the initial state, an unobservable (resp. fully observable) domain leads to Σ_2^P(resp. Π_2^P)-completeness [BKT00].

ASP. The complexity of various problems in Answer Set Programming has been extensively studied. Consistency checking (i.e., answer set existence) for ground normal programs is NP-complete [DEGV01] and for the non-ground case it is NEXP-complete, informally, due to the need to ground the program. However, if the arities of the predices are bounded, then the complexity gets reduced to Σ_2^P-complete [EFFW07].

Part I

Behavior-Preserving Abstraction

CHAPTER 3

Semantics for Reactive Agent Policies using Abstraction

In this chapter, we describe a formal semantics for describing policies that express a reactive behavior for an agent, by using the representation power of transition systems. We combine components that are efficient for describing reactivity such as target establishment and (online) planning.

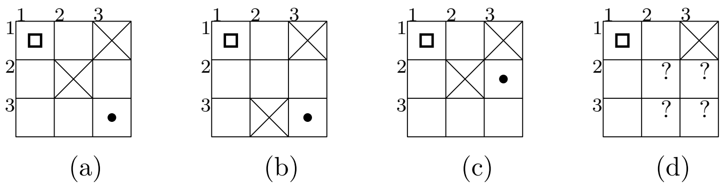
One concern of representing an agent's behavior for a given policy is the issue of keeping irrelevant information in the state which the policy does not use; having to represent such information does not help when reasoning over the policy's behavior, as many details are considered. For this, we consider a state clustering as a form of abstraction that omits such information, while ensuring that the behavior of the policy is preserved and no additional features are introduced. This helps in checking the properties over the policy with a guarantee that the result also holds in the original system.

Outline In Section 3.1, we introduce the general framework for modeling policies. Then, in Section 3.2, we show the relation with action languages by considering (a fragment of) the action language $\mathcal{C}$ as a particular application. We then extend the framework to dynamic environments in Section 3.3 with the notion of maintenance. We conclude in Section 3.4 with some discussion.

3.1 Modeling Policies in Transition Systems

We consider policies that have a main goal μ in mind, and guide the agent with action sequences that are computed according to the knowledge base KB, which is the formal representation of the world's model with a transition system view.

Figure 3.1: (a),(b),(c): Possible instances, (d): Agent's observation, □:agent, •:person, ×:obstacle, ?:unknown



Definition 3.1 (Policy)**.** *Given a system $A = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{A}, \Phi \rangle$, a goal μ, and a set Σ of plans with actions $\mathcal{A}$, a* policy *is a function $P_{\mu,KB} : \mathcal{S} \rightarrow 2^{\Sigma}$ s.t. $P_{\mu,KB}(s) \subseteq \Sigma(s)$.*

The policy guides the agent by setting up targets and determining the course of actions to bring about these targets. The determination of targets for a given state is done by a *target component*, while the (higher level) transition between states is determined by the course of actions computed by some *(online) planner component*.

For illustration, below describes the running example.

Example 3.1. Consider a memoryless agent that can sense horizontally and vertically, in an unknown $n \times n$ grid cell environment with obstacles, where a missing person needs to be found. Figures 3.1(a-c) show some instances for n=3. Suppose we are given a policy of "always go to the farthest reachable point in visible distance (until a person is found)".

Target determination at the states according to the given policy can be done using a logic program as shown below.

$$targetCell(X_1, Y_1) \leftarrow farthest(X, Y, X_1, Y_1), rAt(X, Y), not\ personDetected. \quad (3.1)$$

$$personDetected \leftarrow personDetected(X, Y). \quad (3.2)$$

$$targetPerson(X, Y) \leftarrow personDetected(X, Y). \quad (3.3)$$

$$personFound \leftarrow personDetected(X, Y), rAt(X, Y). \quad (3.4)$$

The farthest reachable point in these states is (3,1), which is determined as the *target*. Then the policy computes the course of actions to reach this target. Clearly, in Figure 3.1a the person will be found when moved to (3,1). However, in Figure 3.1b after reaching (3,1), the agent/policy will decide to move to (1,1) again, which results in a loop. Also, in Figure 3.1c, after reaching (3,1), the agent/policy can either choose to move to (3,3) (which results in seeing the person), or to move back to (1,1). So there is a possibility for the agent to go in a loop. Hence, the policy does not work for the last two instances.

Notice that these initial states provide the same observations for the agent, which is shown in Figure 3.1d, since it can only observe horizontally and vertically. In these states, the agent only sees that the first column is clear of obstacles, and the first row has one obstacle. Since the rest of the environment can not be observed, these states are *indistinguishable* to the agent, and the policy determines in all these initial states the same target and plan.

Our aim is to define a transition system that shows the policy execution, while also employing the notion of *indistinguishability* to do state clustering. Having a classification on states and defining higher level transitions helps in abstraction and allows one to emulate a modular hierarchic approach, in which a higher level (macro) action, expressed by a target, is realized by a sequence of (micro) actions that is compiled by the external planner, which may use different ways (planning on the fly, using scripts etc.)

3.1.1 State Profiles According to the Policy

Large state spaces are a major issue for the (original) transition system when dealing with large environments. However, depending on the agent's designed behavior, and its determination of its course of actions at a state, some information in the state may not be necessary, relevant or even observable. In this sense, the states that contain different facts about such information can be seen as *indistinguishable* to the agent. Such indistinguishable states can be clustered into one with respect to the *profiles* they provide and only the relevant information to the agent/policy can be kept.

Definition 3.2. *A* profile scheme *is a tuple* $p = \langle a_1, .., a_n \rangle$ *of attributes* a_i *that can take values from a set* V_i*; a* (concrete) profile *is a tuple* $\langle v_1, ..., v_n \rangle$ *of values.*

A profile at a state consists of values of attributes that are partitioned as *currently relevant*, *irrelevant* and *not yet observed*, depending on the observability of the environment and the policy. Currently relevant attributes at a state can be regarded as the *active profile*.

Example 3.2 (ctd)**.** Reconsider Figure 3.1. Due to partial observability, the agent is unable to distinguish its state, and the policy does not consider the unobservable parts. The agent's observation, "$\mathit{rAt}(1,1)$, $\mathit{obstacleAt}(1,3)$, $\mathit{reachable}(1,2)$, $\mathit{reachable}(2,1)$, $\mathit{reachable}$ $(3,1)$" that is *currently relevant* and the rest of the environment that is *not yet observed*, is viewed as a profile, and the states with this profile can be clustered in one group (Figure 3.1d).

The profile of a state is determined by evaluating a set of formulas that yield the attribute values. We consider a *classification function*, $h : S \rightarrow \Omega_h$, where Ω_h is the set of possible state clusters with respect to the profiles. For partially observable environments, the information relevant to the policy can correspond to the observations at the state, thus same observations can yield the same profiles. However, in fully observable environments, observability is not of concern. One needs to check the policy to determine profiles.

Example 3.3 (ctd)**.** The policy uses the information of the robot's location $\mathit{rAt}(X, Y)$, the farthest points from the robot's location $\mathit{farthest}(X, Y, X_1, Y_1)$ and whether or not the person is detected. Additionally, the computation of the farthest points requires the use of the observed points from X, Y.

Definition 3.3 (Equalized state)**.** *An* equalized state *relative to the classification function h is a state $\hat{s} \in \Omega_h$.*

The term *equalized* comes from the fact that the states in the same cluster are considered as the same, i.e., equal. We abuse the notation $s \in \hat{s}$ when talking about a state s that is clustered into an equalized state $\hat{s}$, and identify $\hat{s}$ with its pre-image (i.e., the set of states that are mapped to $\hat{s}$ according to h).

3.1.2 Components of the Policy

We consider two components for describing reactivity by the policy such as target establishment and (online) planning. The behavior of the policy over an equalized state is defined as follows.

Definition 3.4 (Policy behavior)**.** *Given a set of equalized states $\widehat{S}$, for an equalized state $\hat{s} \in \widehat{S}$, the policy $P_{\mu,KB}$ uses a* target function $\mathcal{B}(\hat{s})$ *to determine a target g_B from a* set of possible targets, G_B*, and then an* outsourced planner $Reach(\hat{s}, g_B)$ *to compute a plan to reach the target from the current equalized state, i.e.,*

$$P_{\mu,KB} = P^{\mathcal{B},Reach}_{\mu,KB}(\hat{s}) = \{\sigma \mid \sigma \in Reach(\hat{s}, g_B), g_B \in \mathcal{B}(\hat{s})\}.$$

The target function $\mathcal{B}(\hat{s})$ gets the equalized state as input and produces the possible targets to achieve. These targets may be expressed as formulas over the states (in particular, of states that are represented by fluents or state variables), or in some other representation.

Definition 3.5 (Appropriateness)**.** *The clustering function h is* appropriate *for $\mathcal{B}$, if for each state $s \in S$, it holds that for all $s_1, s_2 \in h(s)$ and $g_B \in \mathcal{B}(h(s))$, $s_1 \models g_B \Leftrightarrow s_2 \models g_B$.*

If a clustering is appropriate, then for any equalized state $\hat{s}$, we have

$$\hat{s} \models g_B \Leftrightarrow \forall s \in \hat{s} : s \models g_B. \tag{3.5}$$

This makes it possible to talk about targets over the equalized states and define the behavior of the policy over the equalized system. Here, we focus on appropriate clusterings.

The aim of the policy is to intend to reach a state that satisfies the conditions of the target. For this we make use of an outsourced planner $Reach(\hat{s}, g_B)$ component.

Definition 3.6 ($Reach$ and Res)**.** *$Reach$ is an outsourced function that returns a set of plans needed to reach a state that meets the target condition g_B from the current equalized state $\hat{s} \in \widehat{S}$:*

$$Reach(\hat{s}, g_B) \subseteq \{\sigma \mid \forall \hat{s}' \in Res(\hat{s}, \sigma) : \hat{s}' \models g_B\}$$

and Res gives the resulting states of executing a sequence of actions at a state $\hat{s}$*:* $Res(\hat{s}, \langle\rangle) = \{\hat{s}\}$*, and*

$$Res(\hat{s}, \langle a_1, \ldots, a_n\rangle) \;=\begin{cases} \bigcup_{\hat{s}' \in \hat{\Phi}(\hat{s}, a_1)} Res(\hat{s}', \langle a_2, \ldots, a_n\rangle) & \hat{\Phi}(\hat{s}, a_1) \neq \emptyset \\ \{\hat{s}_{err}\} & \hat{\Phi}(\hat{s}, a_1) = \emptyset \end{cases}$$

for $n \geq 1$*. Here* $\hat{s}_{err}$ *is an artifact state that does not satisfy any target, and* $\hat{\Phi}$ *is a transition relation of executing an action at a state* $\hat{s}$*:*

$$\hat{\Phi}(\hat{s}, a) = \{\hat{s}' \mid \exists s' \in \hat{s}'\ \exists s \in \hat{s} : s' \in \Phi(s, a)\}. \tag{3.6}$$

Example 3.4 (ctd)**.** For the equalized state in Figure 3.1d, say $\hat{s}$, the target is determined as $targetCell(3, 1)$. Thus, the policy needs to find a plan that reaches a state to achieve $g_B = rAt(3, 1)$. The resulting state $\hat{s}' \in Res(\hat{s}, \sigma)$ for sequence of actions $\sigma = \langle goTo(2, 1), goTo(3, 1)\rangle$ satisfies the condition $\hat{s}' \models g_B$.

3.1.3 Transition Systems According to the Policy

We now define the notion of a transition system that is able to represent the evaluation of the policy on the state clusters.

Equalized transition system The transition system that represents the policy evaluation is defined over the original transition system by taking into account the classification function and the policy.

Definition 3.7 (Equalized TS)**.** *An* equalized (higher level) transition system $\mathcal{T}_{h,P_{\mu,KB}}$*, with respect to the classification function* h *and the policy* $P_{\mu,KB}$*, is defined as* $\mathcal{T}_{h,P_{\mu,KB}} = \langle \widehat{S}, \widehat{S}_0, \Sigma, G_B, \mathcal{B}, \Phi_{\mathcal{B}}\rangle$*, where*

- $\widehat{S}$ *is the finite set of equalized states;*
- $\widehat{S}_0 \subseteq \widehat{S}$ *is the finite set of initial equalized states, where* $\hat{s} \in \widehat{S}_0$ *if there is some* $s_i \in \hat{s}$ *such that* $s_i \in S_0$ *holds;*
- Σ *is the set of possible plans* $\sigma = \langle a_1, a_2, \ldots, a_n\rangle$ *where* $a_i \in \mathcal{A}$*, for all* $i, 1 \leq i \leq n$*.*
- G_B *is the finite set of possible* targets *relative to the behavior, where a target can be satisfied by more than one equalized state;*
- $\mathcal{B} : \widehat{S} \to 2^{G_B}$*, is the* target function *that returns the possible targets to achieve from the current equalized state, according to the policy;*
- $\Phi_{\mathcal{B}} : \widehat{S} \times \Sigma \to 2^{\widehat{S}}$ *is the transition function according to the policy, called the* policy execution function*, where*
  $$\Phi_B(\hat{s}, \sigma) = \{\hat{s}' \mid \hat{s}' \in Res(\hat{s}, \sigma), \sigma \in Reach(\hat{s}, g_B), g_B \in \mathcal{B}(\hat{s})\};$$
  it returns the possible resulting equalized states after applying the plan determined by the policy in the current equalized state.

Figure 3.2: A transition in the equalized transition system

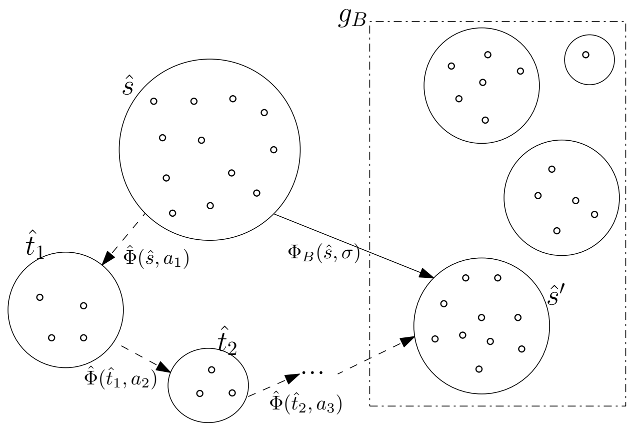
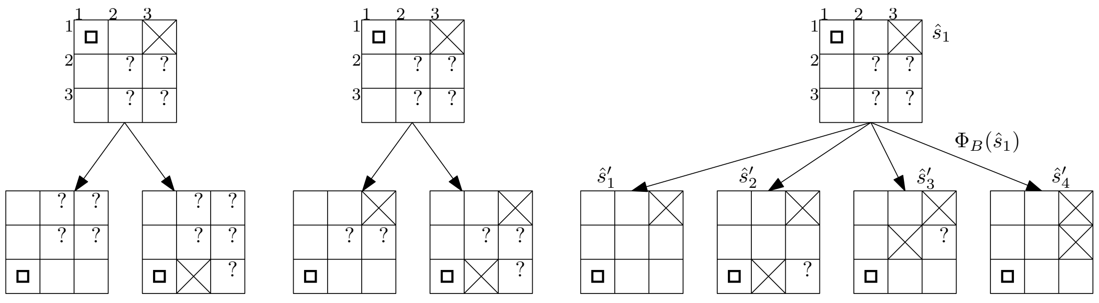

g_B
$\hat{s}$
$\hat{t}_1$
$\hat{\Phi}(\hat{s}, a_1)$
$\Phi_B(\hat{s}, \sigma)$
$\hat{s}'$
$\hat{t}_2$
$\hat{\Phi}(\hat{t}_1, a_2)$
$\hat{\Phi}(\hat{t}_2, a_3)$
...

Figure 3.3: Parts of an equalized transition system w.r.t. different state classifications



(a) w/o knowledge gain (b) w/ knowledge gain (c) observation while moving

The equalized transition system $\langle \widehat{S}, \widehat{S}_0, \Sigma, \Phi_B \rangle$ can be viewed as a transition system $\langle S, S_0, \mathcal{A}, \Phi \rangle$ with an infinite set of actions. Additionally, it contains auxiliary definitions $\langle G_B, \mathcal{B} \rangle$ that are used in defining the policy.

Figure 3.2 demonstrates a transition in the equalized transition system. Depending on the current state, $\hat{s}$, a plan σ can be executed if it is returned by *Reach* to reach the target g_B that is determined by the policy. There may be more than one equalized state satisfying g_B, and the policy execution function $\Phi_B(\hat{s}, \sigma)$ executes σ and finds a transition into one of these states, $\hat{s}'$. In our case, the actions taken in the transitions do not matter Therefore, we project away the knowledge of the executed action sequences, and only consider $\Phi_B : \widehat{S} \rightarrow 2^{\widehat{S}}$. Thus, the transition Φ_B becomes a big jump between states, where the actions taken and the states passed in between are omitted.

Example 3.5. Figure 3.3 shows a part of the equalized transition system constructed according to the policy w.r.t. different state classification h. In all three versions the indistinguishable states due to partial observability are clustered into one. The policy is applied according to current observations (targeting the farthest reachable point, which

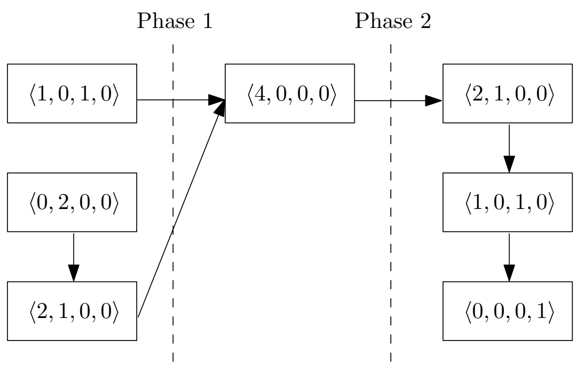


Figure 3.4: Equalized transition system of blocksworld ($n = 4$)

for $\hat{s}_1$ is $(3, 1)$), and the possible successor states are shown. There are several possibilities for the resulting state that satisfy the target $g_B = rAt(3, 1)$.

Figure 3.3a shows a classification that only takes into account the current observations in the state, while the classification in Figure 3.3b also distinguishes the knowledge of the observations made in the previous states. Figure 3.3c shows a classification that distinguishes the knowledge gained while moving in the environment and observing the surroundings.

Example 3.6. Let us consider a simple blocksworld example where a policy (of two phases) is defined as follows:

- if in phase 1 and not all the blocks are on the table, move one free block on a stack with highest number of blocks to the table.
- if all the blocks are on the table, move to phase 2.
- if in phase 2 and not all the blocks are on top of each other, move one of the free blocks on the table on top of the stack with more than one block (if exists any, otherwise move the block on top of some block).

Since the policy does not take labels of the blocks into consideration, a classification can be of the following form: We introduce an n-tuple $\langle b_1, \dots, b_n \rangle$ to denote equalized states such that for $i \leq n$, b_i would represent the number of stacks that have i blocks. For example, for 4 blocks, a state $\langle 1, 0, 1, 0 \rangle$ where $b_1 = 1, b_2 = 0, b_3 = 1, b_4 = 0$ would represent all the states in the original transition system with the profile "contains a stack of 1 block and a stack of 3 blocks". Notice that in the original transition system for four labeled blocks, there are 24 possible states that have this profile and if the blocks need to be in order, then there are four possible states.

Figure 3.4 demonstrates the corresponding equalized transition system for the case of four blocks. The equalized transition system for this example is in the following form:

- $\widehat{S}$ is the set of equalized states according to the abstraction as described above.

- $\widehat{S}_0 \in \widehat{S}$ is the initial equalized states (all elements of $\widehat{S}$ except $\langle 0, \dots, 0, 1 \rangle$).
- $G_B = \widehat{S}$, since the policy is related with all the blocks, it can determine targets as the whole states.
- $\mathcal{B} : \widehat{S} \to \widehat{S}$ is the target function.
- $\Phi_B : \widehat{S} \to \widehat{S}$ is the policy execution function, returning the resulting successor state after applying one action desired by the behavior, shown as in Figure 3.4.

Notice that we assume that the outsourced *Reach* function is able to return conformant plans that guarantee to reach a state that satisfies the determined targets. For practical reasons, we consider *Reach* to be able to return a subset of all conformant plans. The maximal possible *Reach*, where we have equality, is denoted with $Reach_0$.

Consider the case of uncertainty, where the agent requires to do some action, e.g., *checkDoor*, to gain further knowledge about its state. The target function can be modified to return dummy fluents as targets to ensure that the action is made, e.g., *doorIsChecked*, and given this target, the *Reach* function can return the desired action as the plan. The nondeterminism of the environment is modeled through the possible outcomes of *Res*.

Our generic definition allows for the possibility of representing well-known concepts like purely reactive systems or conformant planning. Reactive systems can be represented with the policy "pick some action", which models systems that immediately react to the environment without reasoning. As for conformant planning, one can set the target as the main goal. Then, *Reach* would have the difficult task of finding a plan that quarantees reaching the main goal. If however, such a plan is available, then we have the following.

Proposition 3.1. *Let $P = \langle a_1, \dots, a_n \rangle$,$n \geq 1$, be a conformant plan that reaches a goal state g from the initial states $s_{01}, \dots, s_{0r}$ in the original transition system. The plan P can be polynomially expressed in an equalized transition system.*

One can mimic the plan by modifying the targets G_B and the target function $\mathcal{B}$ in a way that at each point in time the next action in the plan is returned by *Reach*, and the corresponding transition is made. For that, one needs to record information in the states and keep track of the targets.

Proof. For the conformant plan sequence $S_0, a_1, S_1, \dots, a_n, S_n$, where $S_i = \{s_{i,1}, \dots, s_{i,r_i}\}$, $0 \leq i \leq n$, the classification function h is defined as $h(s_{i,j}) = \hat{s}_i$ for $1 \leq j \leq r_i$. We show that for the policy $P_{\mu,KB}$ defined as follows we obtain $P_{\mu,KB}(\hat{s}_{i-1}) = a_i, 1 \leq i \leq n$. We set μ as the goal condition g reached after applying P, i.e., $s \models \mu, \forall s \in S_n$.

Now let A_P be the set of actions in P. For each action $a \in A_P$ the set G_B of targets consists of dummy fluents $made(a)$, and the target function $\mathcal{B}$ decides on a target $g_B \in G_B$

to pick depending on which dummy target is satisfied in the current state, i.e.,

$$\mathcal{B}(\hat{s}) = \begin{cases} \mathit{made}(a_1) \text{ if } \hat{s} = S_0 \\ \mathit{made}(a_i) \text{ if } \hat{s} \models \mathit{made}(a_{i-1}), i > 1 \end{cases}$$

Additionally, each state contains the information $\mathit{reached}(s', a)$ for $s' \in S, a \in A_P$ denoting from which state it can be reached with which action, i.e., $s \models \mathit{reached}(s', a)$ if $s \in \Phi(s', a)$. Then whether or not a state s satisfies a target is defined as follows.

$$\begin{aligned} &s \models \mathit{made}(a_1) \text{ if } s \models \mathit{reached}(s', a_1) \wedge s' \in S_0 \\ &s \models \mathit{made}(a_i) \text{ if } s \models \mathit{reached}(s', a_i) \wedge s' \models \mathit{made}(a_{i-1}), i > 1 \end{aligned}$$

Due to the definition of h, we have $\hat{s}_i \models \mathit{made}(a_i) \iff \forall s \in \hat{s}_i : s \models \mathit{made}(a_i)$. Thus, we obtain $a_i \in \mathit{Reach}(\hat{s}_{i-1}, \mathit{made}(a_i)), 1 \leq i \leq n$. □

3.1.4 Complexity Issues

As the function *Reach* is outsourced, we rely on an implementation that returns conformant plans to achieve transitions in the equalized transition systems. This raises the issue whether a given such implementation is suitable, and leads to the question of soundness (only correct plans are output) and completeness (some plan will be output, if one exists). We next assess how expensive it is to test this, under some assumptions about the representation and computational properties of (equalized) transition systems, which will then also be used for assessing the cost of policy checking.

Assumptions We assume that given a state $s \in S$, which is implicitly given using a binary encoding, the cost of evaluating the classification $h(s)$, the (original) transition $\Phi(s, a)$ for some action a, and recognizing the initial state, say with $\Phi_{init}(s)$, is feasible in polynomial time. The cost could also be in NP, if projective (i.e., existentially quantified) variables are allowed. Furthermore, we assume that the size of the representation of a "target" in G_B is polynomial in the size of the state, so that given a string, one can check in polynomial time whether it is a correct target description g_B. This test can also be relaxed to be in NP by allowing projective variables.

Given these assumptions, we have the following two results on the cost of checking whether a given implementation of *Reach* is sound and complete; we assume here that testing whether $\sigma \in \mathit{Reach}(\hat{s}, g_B)$ is feasible in Π_2^p (i.e., it is no worse than a naive guess and check algorithm that verifies conformant plans).

Theorem 3.2 (Checking soundness of *Reach*)**.** *Let* $\mathcal{T}_h = \langle \widehat{S}, \widehat{S}_0, G_B, \mathcal{B}, \Phi_\mathcal{B} \rangle$ *be a transition system w.r.t. a classification function h. Checking whether every transition found by the policy execution function* $\Phi_\mathcal{B}$ *induced by a given implementation Reach is correct is in* Π_3^p.

Proof. According to Definition 3.21, every transition from a state $\hat{s}$ to some state $\hat{s}'$ corresponds to some plan σ returned by $\mathit{Reach}(\hat{s}, g_B)$. Thus first one needs to check whether each plan $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ returned by *Reach* given some $\hat{s}$ and g_B is correct. For that, we need to check two conditions on the corresponding trajectories of the plan: (i) for all partial trajectories $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{i-1}$ it holds that for the upcoming action a_i from the plan σ, $\hat{\Phi}(\hat{s}_{i-1}, a_i) \neq \emptyset$ (i.e., the action is applicable). (ii) for all trajectories $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$, $\hat{s}_n \models g_B$. Condition (i) can be checked by guessing a trajectory $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{i-1}$ and checking if $\hat{\Phi}(\hat{s}_{i-1}, a_i) = \emptyset$, which can be done with a coNP oracle that checks if for all $s \in \hat{s}_{i-1}$, $\Phi(s, a_i) = \emptyset$ holds; thus condition (i) checking is in Π^p_2. Condition (ii) can be checked by guessing a trajectory $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ and checking if $\hat{s}_n \not\models g_B$, which can be done with a NP oracle that checks if there exists $s \in \hat{s}_n$ such that $s \not\models g_B$; thus condition (ii) checking is in Π^p_2.

Now, to decide whether for some state $\hat{s}$ and target g_B the function $\Phi_B(\hat{s}, g_B)$ does not work correctly, we can guess $\hat{s}$ (resp. $s \in \hat{s}$), g_B, a plan σ, and verify that $\sigma \in Reach(\hat{s}, g_B)$ and that σ is not correct. As we can do the verification with an oracle for Σ^p_2 in polynomial time, correctness can be refuted in Σ^p_3; thus the problem is in Π^p_3. □

The result for soundness of *Reach* is complemented with another result for completeness with respect to short (polynomial size) conformant plans that it returns.

Theorem 3.3 (Checking completeness of *Reach*). *Let $\mathcal{T}_h = \langle \widehat{S}, \widehat{S}_0, G_B, \mathcal{B}, \Phi_{\mathcal{B}} \rangle$ be a transition system w.r.t. a classification function h. Deciding whether for a given implementation Reach, Φ_B fulfills $\hat{s}' \in \Phi_B(\hat{s})$ whenever a short conformant plan from $\hat{s}$ to some $g_B \in \mathcal{B}(\hat{s})$ exists and $\hat{s}'$ is the resulting state after the execution of the plan in T_h, is in Π^p_4.*

Proof. For a counterexample, we can guess some $\hat{s}$ and $\hat{s}'$ (resp. $s \in \hat{s}$, $s' \in \hat{s}'$) and some short plan $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ and verify that (i) σ is a valid conformant plan in $\mathcal{T}_h$ to reach $\hat{s}'$ from $\hat{s}$, and (ii) that a target g_B exists such that $Reach(\hat{s}, g_B)$ produces some output.

In order to verify (i) we can guess a sequence $\hat{s}_0, a_1, \hat{s}_1, \ldots, a_n, \hat{s}_n$ and check if it is a valid plan sequence, which we can check with an Π^p_2 oracle (condition (i) of the proof of Theorem 3.2). To verify (ii) we need to check if for all guesses of targets g_B and short plans σ', either (ii-a) g_B is not a target for $\hat{s}$ or (ii-b) σ' is not produced by $Reach(\hat{s}, g_B)$. Condition (ii-a) is a simple checking of $g_B \notin \mathcal{B}(\hat{s})$. As for (ii-b) we need to check if for all plans σ' produced by $Reach(\hat{s}, g_B)$ we have $\sigma' \neq \sigma$. For this we can guess some σ' and check if $\sigma' \in Reach(\hat{s}, g_B)$ and $\sigma' = \sigma$. By assumption, checking (ii-b) can be done using an Π^p_2 oracle. Thus, checking (ii) can be done using an Π^p_3 oracle. This establishes membership in Π^p_4. □

The complexities drop if checking the output of *Reach* is lower (e.g., it drops to Π^p_2 for soundness and to Π^p_3 for completeness, if output checking is in co-NP).

In this work, we assume that *Reach* is complete. We also restrict the plans σ that are returned by *Reach* to have polynomial size. This constraint would not allow for exponentially long conformant plans (even if they exist). Thus, the agent is forced to develop targets that it can reach in polynomially many steps. This does not limit the capability of the agent in general. The “long” conformant plans can be split into short plans with a modified policy and by encoding specific targets into the states, such that at each state, one chooses the next action with respect to the conformant plan. The targets can be encoded to give the stage of the plan execution so that the respective action is taken, or they can be encoded to assign the latest action in the conformant plan that is done from the current state.

The following proposition states that the original transitions according to the policy are preserved in the equalized transition system. For a plan $\sigma = \langle a_1, \dots, a_n \rangle$, we use the notation $\Phi(s,\sigma)$ as a shorthand for $\Phi(\dots\Phi(\Phi(s,a_1),a_2)\dots,a_n)$.

Proposition 3.4. *Given a state $s \in S$, we have*

$$\{h(s') \mid s' \in \Phi(s,\sigma), \sigma \in P(s)\} \subseteq \{\hat{s}' \mid \hat{s}' \in \Phi_B(h(s),\sigma), \sigma \in \Sigma\}.$$

Proof. Towards a contradiction, assume there exists a transition $\langle s,\sigma,s'\rangle$ in $\mathcal{T}$ where $\hat{s}' \notin \Phi_B(h(s),\sigma')$ for all $\sigma' \in \Sigma$. This means that no σ' was determined from $P_{\mu,KB}(h(s))$. From the definition of the policy components, we know that if the policy P chooses to move to s', then this behavior can be defined through some target g_B where $s' \models g_B$, i.e., $\hat{s}' \models g_B$, that the policy is aiming for, i.e., $g_B \in \mathcal{B}(h(s))$. Our assumption yields that there exists no plan $\sigma' \in \Sigma$ such that $\sigma' \in \mathit{Reach}(h(s),g_B)$. This however brings us to a contradiction due to the assumption that *Reach* is complete, because it is clear that the plan σ achieves a transition from s to s'. □

This brings us the result that the behavior of the policy in the original transition system is preserved in the equalized transition system.

Proposition 3.5. *A trajectory $s_0, s_1, \dots, s_n$ in $\mathcal{T}$ where $s_0 \in S_0$ and $s_{i+1} \in \Phi(s_i,\sigma)$ for some $\sigma \in P(s_i)$, $i > 1$, has a corresponding trajectory $\hat{s}_0, \hat{s}_1, \dots, \hat{s}_n$ in $\mathcal{T}_{h,P}$ where $\hat{s}_0 \in \widehat{S}_0$ and $\hat{s}_{i+1} \in \Phi_B(\hat{s}_i)$, $i > 1$.*

Thus, for appropriate clusterings we achieve the following corollary. The main goal μ that the policy is aiming for can be expressed as a formula that should be satisfied at a state.

Corollary 3.6. *If all trajectories in $\mathcal{T}_{h,P}$ achieve μ, then all trajectories that follow the policy in $\mathcal{T}$ achieve μ.*

This result allows us to define the working of the policy over the equalized transition system. Note that the policy could be easily modified to stop or to loop in any state $\hat{s}$ that satisfies the goal.

Definition 3.8. *The policy* works *w.r.t. the main goal μ, if for each run $\hat{s}_0, \hat{s}_1, \dots$ such that $\hat{s}_0 \in \widehat{S}_0$ and $\hat{s}_{i+1} \in \Phi_B(\hat{s}_i)$, for all $i \geq 0$, there is some $j \geq 0$ such that $\hat{s}_j \models \mu$.*

One can also make use of temporal operators, and define μ by a temporal formula (e.g., $\mathbf{AF}(\mathit{personFound})$) and then check whether the initial states in $\widehat{S}_0$ satisfy the formula.

Under the assumptions from above, we obtain the following.

Theorem 3.7. *Given a policy P for a goal μ, the main problem of determining whether the policy works is in* PSPACE*.*

Proof. One needs to look at all runs $\hat{s}_0, \hat{s}_1, \ldots$ from every initial state $\hat{s}_0$ in the equalized transition system and check whether each such run has some state $\hat{s}_j$ that satisfies the main goal μ. Given that states have a representation in terms of fluent or state variables, there are at most exponentially many different states.

To find a counterexample, it is sufficient to build a run of at most exponential length in which μ is not satisfied, since like mentioned in the PSPACE membership proof of classical planning [Byl94] any plan beyond that length must have loops. Such a run can be nondeterministically built by picking a first state $\hat{s}_0$ among all possible initial states in $\hat{S}_0$, then picking a second state $\hat{s}_1$ among all possible ones in $\Phi_B(\hat{s}_0)$, then picking a third state $\hat{s}_2$ among all possible ones in $\Phi_B(\hat{s}_1)$, etc. until either some picked state $\hat{s}_i$ has been picked before (which is guessed as well), which shows that there is a possibility to loop, or a trajectory of exponential length is built where the final state $\hat{s}_n \not\models \mu$. As NPSPACE = PSPACE, the result follows. □

Note that in this formulation, we have tacitly assumed that the main goal can be established in the original system, thus at least some trajectory from some initial state to a state fulfilling the goal exists. In a more refined version, we could define the working of a policy relative to the fact that some abstract plan would exist that makes μ true; naturally, this may impact the complexity of the policy checking.

3.1.5 Constraining Equalization

Until now, we focused on over-approximating the behavior of the policy to ensure that all possible executions of the policy are preserved. However, our aim is not to introduce new features with an over-approximation, but to keep the structure of the original transition system and discard the unnecessary parts with respect to the policy. In this section, we discuss the conditionthat the classification function h needs to satisfy in order to achieve this.

For deciding on a plan σ with $\mathit{Reach}(\hat{s}, g_B)$ from a state $\hat{s}$ for some target g_B, the existence of some plan that is able to achieve some state $s' \models g_B$ from $s \in \hat{s}$ is enough. However, it is not guaranteed that any state mapped to $h(s')$ can be reached by σ. The definition of $\hat{\Phi}$ (3.6) allows for certain transitions that do not have corresponding concrete transitions in the original transition system.

In order to have the capability of backtracking the plans determined in the equalized transition system, the classification function should satisfy the following condition.

- A transition between equalized states $\hat{s}$ and $\hat{s}'$, i.e., $\hat{s}' \in \hat{\Phi}(\hat{s}, a)$ should have a corresponding original transition from any state mapped to $\hat{s}'$.

$$\forall s_1', s_2' \in \hat{s}' : \exists s_1 \in \hat{s}, s_1' \in \Phi(s_1, a) \Leftrightarrow \exists s_2 \in \hat{s}, s_2' \in \Phi(s_2, a) \tag{3.7}$$

When the classification function h satisfies (3.7), then we can obtain the following property.

$$\hat{s}' \in \hat{\Phi}(\hat{s}, a) \Leftrightarrow \forall s' \in \hat{s}',\ \exists s \in \hat{s} :\ s' \in \Phi(s, a) \tag{3.8}$$

The clustering h is called *proper* if condition (3.8) is satisfied.

Theorem 3.8. *Let $\mathcal{T}_h$=$\langle \widehat{S}, \widehat{S}_0, G_B, \mathcal{B}, \Phi_{\mathcal{B}} \rangle$ be a transition system w.r.t. a classification function h. Let $\hat{\Phi}$ be the transition function that the policy execution function $\Phi_{\mathcal{B}}$ is based on. The problem of checking whether $\hat{\Phi}$ is proper is in Π_2^p.*

Proof. As a counterexample, one needs to guess $\hat{s}, a$, $\hat{s}' \in \hat{\Phi}(\hat{s}, a)$ and $s' \in \hat{s}'$ such that for all $s \in \hat{s}$ has $s' \notin \Phi(s, a)$, which can be checked with a coNP oracle. Thus, the properness checking is in Π_2^p. □

For the next property we need the notion of reachability among the equalized states. A state $\hat{s}$ is *reachable* from an initial state in the equalized transition system if and only if $s \in \mathcal{R}_i$ for some $i \in \mathbb{N}$ where $\mathcal{R}_i$ is defined as follows.

$$\mathcal{R}_0 = \widehat{S}_0,\ \ \mathcal{R}_{i+1} = \textstyle\bigcup_{\hat{s} \in \mathcal{R}_i} \Phi_B(\hat{s}),\ \ i \geq 1,\ \ \text{and}\ \ \mathcal{R}^{\infty} = \textstyle\bigcup_{i \geq 0} \mathcal{R}_i.$$

Under the assumptions that apply to the previous results, we can state the following.

Theorem 3.9. *The problem of determining whether a state in an equalized transition system is reachable is in* PSPACE.

Proof. For some state s, we can nondeterministically build a run, similar to the proof of Theorem 3.7, to check whether s can be reached. □

The notions of soundness and completeness of an outsourced planning function *Reach* could be restricted to reachable states; however, this, would not change the worst case cost of testing these properties in general (assuming that $\hat{s} \in \mathcal{R}$ is decidable with sufficiently low complexity).

The following proposition shows that the policy execution function is sound.

Proposition 3.10 (soundness). *Let $\mathcal{T}_h$=$\langle \widehat{S}, \widehat{S}_0, G_B, \mathcal{B}, \Phi_{\mathcal{B}} \rangle$ be a transition system w.r.t. a classification function h that satisfies* (3.7)*. Let $\hat{s}_1, \hat{s}_2 \in \widehat{S}$ be equalized states that are reachable from some initial states, and $\hat{s}_2 \in \Phi_B(\hat{s}_1)$. For any concrete state $s_2 \in \hat{s}_2$, there is a concrete state $s_1 \in \hat{s}_1$ such that $s_1 \to^{\sigma} s_2$ for some action sequence $\sigma \in P(s_1)$.*

Proof of Proposition 3.10 is based on the possibility of backward tracking with any of the plans σ executed to reach $\hat{s}_2$ from $\hat{s}_1$ and knowing that the tracking can begin with any $s \in \hat{s}_2$ as $s \models g_B$ holds.

Proof. For equalized states $\hat{s}_1, \hat{s}_2$, having $\hat{s}_2 \in \Phi_B(\hat{s}_1)$ means that $\hat{s}_2$ satisfies a target condition that is determined at $\hat{s}_1$, and is reachable via executing some plan σ. Let $s_2 \in \hat{s}_2$, since (3.8) holds, we can apply backwards tracking from s_2 following the transitions Φ corresponding to the actions in the plan σ backwards. By (3.5), we know that s_2 satisfies the required target condition, and since it reaches some $s_1 \in \hat{s}_1$ with σ, we can see that $\sigma \in P(s_1)$ would hold. In the end, we can find a concrete state $s_1 \in \hat{s}_1$ from which one can reach the state $s_2 \in \hat{s}_2$ by applying the plan $\sigma \in P(s_1)$ in the original transition system. □

Thus, we obtain the following corollary, with the requirement of only having initial states clustered into the equalized initial states (i.e., no “non-initial” state is mapped to an initial equalized state). Technically, it should hold that $\forall s \in S_0 : h^{-1}(h(s)) \subseteq S_0$.

Corollary 3.11. *If there is a trajectory in the equalized transition system with initial state clustering from an equalized initial state $\hat{s}_0$ to some $\hat{s}_n$, then for any $s \in \hat{s}_n$ a trajectory that follows the policy can be found in the original transition system from some concrete initial state $s_0 \in \hat{s}_0$.*

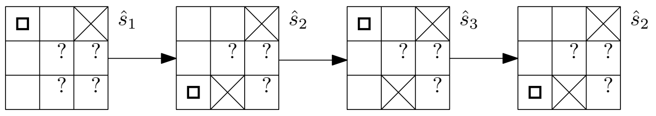
Proof. For the trajectory $\hat{s}_0, \dots, \hat{s}_n$ we can backtrack from $\hat{s}_i$ to $\hat{s}_{i-1}$, for $1 < i \leq n$, knowing that by Proposition 3.10 we have a concrete transition to any $s \in \hat{s}_i$ from some $s' \in \hat{s}_{i-1}$. We eventually reach some initial state $s_0 \in \hat{s}_0$. □

Our aim is to analyze the reactive policy through the equalized transition system. If the policy does not work as expected, there will be trajectories showing the failure. Knowing that any such trajectory found in the equalized transition system exists in the original transition system is enough to conclude that the policy indeed does not work in the original system.

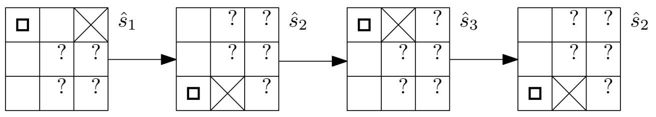
Example 3.7. The trajectory $\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_2$ of $T_{h,P}$ in Figure 3.5a shows a path in which the policy does not reach the goal condition and a loop occurs. Notice that the abstract transitions satisfy the properness condition. This trajectory can also be mapped to a trajectory in $\mathcal{T}$ involving $s_1 \in \hat{s}_1$ shown in Figure 3.1b.

Observe that the properness condition forces to keep the information gain in the state to hold in the successor state in order to ensure that any state mapped to the equalized state can be reached from the predecessor equalized state. This can also be seen in Figure 3.5. The trajectory in Figure 3.5b does not satisfy the condition and although it describes a loop over the equalized transition system, this loop can not be mapped back to the original transition system.

Figure 3.5: A loop trajectory in the equalized transition system



(a) w/ properness property



(b) w/o properness property

The assumptions so far can not avoid the case where a plan σ returned by *Reach* from some state $\hat{s}$ on the equalized transition system does not have a corresponding trajectory from an original state $s \in \hat{s}$ in the original transition system, since the main focus was to ensure the existence of at least one such state with a corresponding trajectory. Thus, a trajectory found in the equalized transition system from an initial state $\hat{s}_0$ is not guaranteed to exist from all initial states $s \in \hat{s}_0$. In order to achieve this the transition $\hat{\Phi}$ needs to satisfy an additional condition such as

$$\hat{s}' \in \hat{\Phi}(\hat{s}, a) \Leftrightarrow \forall s \in \hat{s},\ \exists s' \in \hat{s}' : s' \in \Phi(s, a). \tag{3.9}$$

Under this condition, every plan returned by *Reach* from some state $\hat{s}$ can be successfully executed from any state $s \in \hat{s}$ in the original transition system $\mathcal{T}$.

However, still we may lose trajectories of $\mathcal{T}$ as clustering the states might restrain conformant plans; for this, also stronger conditions like exact approximation [CGL94], $\hat{s}' \in \hat{\Phi}(\hat{s}, a) \Leftrightarrow \forall s \in \hat{s}, \forall s' \in \hat{s}' : s' \in \Phi(s, a)$, is not enough. One would need to modify the target determination, i.e., the set of targets $G_{\mathcal{B}}$ and the function $\mathcal{B}$.

3.2 Bridging to Action Languages

We now describe how our representation of the behavior of the policy can fit into action languages. Given a domain description defined by an action language and its respective (original) transition system, modeling a reactive policy and constructing the corresponding equalized transition system can be done as follows.

Classifying the State Space The approach to classify the (original) state space relies on defining a function that classifies the states. There are at least two kinds of such classification; one can classify the states depending on the observed values of the fluents, or introduce a new set of fluents and classify the states depending on their values:

Type 1: Extend the set of truth values by $\mathbf{V}' = \mathbf{V} \cup \{\mathsf{u}\}$, where u denotes the value to be *unknown*. Consider an *observability relation* $\mathcal{O} : \mathbf{F} \times S \to \mathbf{V}'$ which returns how the fluents' values are observed at the states. Then, consider a set of clusters, $\widehat{S}$, where a cluster $\hat{s}_i \in \widehat{S}$ contains all the states $s \in S$ that have the same observed values, i.e., $\widehat{S} = \{\hat{s} \mid \forall d, e \in S,\ d, e \in \hat{s} \Longleftrightarrow \forall p \in \mathbf{F} : \mathcal{O}(p,d){=}\mathcal{O}(p,e)\ \}$. The value function for the clusters is $\widehat{V} : \mathbf{F} \times \widehat{S} \rightarrow \mathbf{V}'$.

Type 2: Consider a set of (auxiliary) fluent names $\mathbf{F}_a$, where each fluent $p \in \mathbf{F}_a$ is *related* with some fluents of $\mathbf{F}$. The relation can be shown with a mapping $\Delta : 2^{\mathbf{F} \times \mathbf{V}} \rightarrow \mathbf{F}_a \times \mathbf{V}$. Then, consider a new set of clusters, $\widehat{S}$, where a cluster $\hat{s}_i \in \widehat{S}$ contains all the states $s \in S$ that give the same values for all $p \in \mathbf{F}_a$, i.e., $\widehat{S} = \{\hat{s} \mid \forall d, e \in S,\ d, e \in \hat{s} \Longleftrightarrow \forall p \in \mathbf{F}_a : V(p,d){=}V(p,e)\ \}$. The value function for the clusters is $\widehat{V} : \mathbf{F}_a \times \widehat{S} \rightarrow \mathbf{V}$.

We can consider the states in the same classification to have the same *profile*, and the classification function h as a membership function that assigns the states to groups.

Remarks: (1) In Type 1, introducing the value *unknown* allows for describing sensing actions and knowing a fluent's true value later. Also, one needs to impose constraints; e.g., a fluent related to a grid cell can not be unknown while the robot can observe it. (2) In Type 2, one needs to modify the action descriptions according to the newly defined fluents and define *abstract actions*. However, this is not necessary in Type 1, assuming that the action descriptions only use fluents that have *known* values.

Example 3.8. In the action language $\mathcal{C}$, we introduce unknown values by auxiliary fluents as follows.

$$\textbf{caused } \mathit{uReachable}(X,Y) \textbf{ if } \mathit{not\ reachable}(X,Y)\ \wedge\ \mathit{not}\ \neg \mathit{reachable}(X,Y).$$

i.e. if it is not known that a grid cell is reachable or not, then the fluent *uReachable* becomes true. Additional rules are added to express that it becomes false otherwise.

Defining a Target Language A policy is defined through a language which figures out the targets and helps in determining the course of actions.

Definition 3.9. *A* target language *of a policy is a tuple* $\langle \widehat{\boldsymbol{F}}, \mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}}), \mathcal{F}_{G_B}(\widehat{\boldsymbol{F}}) \rangle$ *where*

- $\widehat{\boldsymbol{F}}$ *is the set of fluents that the equalized transition system is built upon,*
- $\mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}})$ *is the set of target determination formulas constructed over* $\widehat{\boldsymbol{F}}$*, and*
- $\mathcal{F}_{G_B}(\widehat{\boldsymbol{F}})$ *is the set of possible targets determined via the evaluation of* $\mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}})$*.*

Example 3.9 (ctd)**.** $\mathcal{F}_{\mathcal{B}}(\widehat{\mathbf{F}})$ corresponds to the set of causal laws in (3.1)-(3.4) and $\mathcal{F}_{G_B}(\widehat{\mathbf{F}})$ consists of all atoms $\mathit{targetCell}(X,Y)$ and $\mathit{targetPerson}(X,Y)$ for $1 \le X \le n$, $1 \le Y \le n$.

Notice that the separation of formulas $\mathcal{F}_{\mathcal{B}}(\widehat{\mathbf{F}})$ and the targets $\mathcal{F}_{G_B}(\widehat{\mathbf{F}})$ is to allow for outsourced planners that understand simple target formulas. These planners need no knowledge to find plans. However, if one is able to use planners that are powerful enough, then the target language can be given as input to the planner, so that the planner determines the target and finds the corresponding plan.

Transition Between States The transitions in the (projected) equalized transition system can be denoted with $\widehat{R} \subseteq \widehat{S} \times \widehat{S}$, where $\widehat{R}$ corresponds to the projection of the policy execution function Φ_B that uses

(a) the target language to determine targets,
(b) an outsourced planner (corresponding to the function *Reach*) to find conformant plans and
(c) the computation of executing the plans (corresponding to the function *Res*).

Thus, $\widehat{R}$ shows the resulting states after applying the policy.

Equalized Transition System over Action Language $\mathcal{C}$ An equalized transition system that describes the behavior of a given policy can be defined over the action language $\mathcal{C}$ as follows.

Definition 3.10. *Given a policy with a target language $\langle \widehat{\boldsymbol{F}}, \mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}}), \mathcal{F}_{G_B}(\widehat{\boldsymbol{F}}) \rangle$, an equalized transition system $\langle \widehat{S}, \widehat{V}, \widehat{R} \rangle$ that describes the policy consists of:*

(i) a set $\widehat{S}$ of all interpretations of $\widehat{\boldsymbol{F}}$ such that, $\hat{s}$ satisfies every static law in $\mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}})$.

(ii) a function $\widehat{V}(P, \hat{s}) = \hat{s}(P)$, where $P \in \widehat{\boldsymbol{F}}$,

(iii) a transition relation $\widehat{R} \subseteq \widehat{S} \times \widehat{S}$ which consists of $\langle \hat{s}, \hat{s}' \rangle$ such that

a) for every $s' \in \hat{s}'$ there is a trajectory from some $s \in \hat{s}$ of the form $s, A_1, s_1, \dots, A_n, s'$ in the original transition system;

b) for static laws $f_1, f_2, \dots, f_m \in \mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}})$ for which $\hat{s}$ satisfies the body, it holds that $\hat{s}' \models g$ for some $g \in \mathcal{M}(f_1, \dots, f_m)$, where $\mathcal{M}$ is a mapping $\mathcal{M} : 2^{\mathcal{F}_{\mathcal{B}}(\widehat{\boldsymbol{F}})} \rightarrow 2^{\mathcal{F}_{G_B}(\widehat{\boldsymbol{F}})}$, that gives the relation between the formulas and the targets.

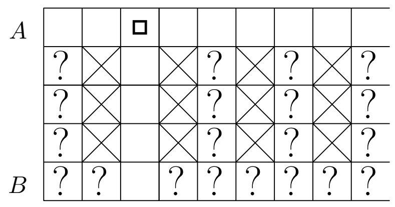
Notice that $\widehat{R}$ in (iii) has no prescription of (a) how a trajectory is computed or (b) how a target is determined. This makes the implementation of these components flexible.

By focusing on a fragment of $\mathcal{C}$, we match the above conditions on complexity. Furthermore, by results on the complexity of action language $\mathcal{C}$ [Tur02, EFL$^+$04], the results in Theorems 3.2-3.8 can be turned into completeness results already for this fragment. Other languages can be similarly used to describe the equalized transition system, as long as they are powerful enough to express the concepts in the previous section.

3.3 Reactive Maintenance Policies

The notions introduced in the previous sections are for agents acting in static environments, or in environments that do not interfere with the agent's actions. However, an interesting

Figure 3.6: Supermarket example



extension to the notions would be to consider the dynamicity of the environment, as dynamic environments may change the state of the world and interfere with the behavior of a reactive agent that follows a given policy.

In this section, we investigate the representation of reactive policies in such dynamic environments. During the execution of a plan given by the policy, a state change may require the agent to stop and examine the current situation, to determine the next steps. In such cases, rather than "achieving" certain conditions of a main goal, the focus is more on "maintaining" the conditions. Baral et al. [BEBN08] introduced maintenance given a *window of opportunity*, a respite from the environment actions. This notion enables us to distinguish the agent following a policy and doing its best to maintain the goal, if the environment does not interfere during a time period.

Different from [BEBN08], we are interested in policies that yield sequences of actions, which requires awareness of environment actions that may concurrently be made. Furthermore, while [BEBN08] considers explicit states, our focus is on implicit state representations, which allows for the use of logical formalisms to represent transitions.

We consider the below example, where the environment is playing a role in the agent's achievement to the goal condition when following the policy.

Example 3.10 (Supermarket example)**.** Consider an agent that is looking for a person in a supermarket with a layout shown in Figure 3.6. Although the agent knows the layout, it does not know where the person might be, and is given the below policy to follow.

(1) If at row A: walk towards right to the next aisle.
(2) If reached the end of row A: walk towards row B.
(3) If at row B: walk towards left to the next aisle.
(4) If reached the end of row B: walk towards row A.
(5) If observed the person: move towards the person.

If the agent observes the person at any time step, then it stops and moves towards the person. If the environment is static, i.e., the person does not move, then the agent's behavior following the policy can be represented as in Section 3.1. However, our focus is on the dynamic nature of the environment; the person may also be moving, while the agent executes its actions. Thus, the environment actions play a role in the agent's behavior and need to be distinguished.

3.3.1 Behavior of a Policy in Dynamic Environments

We first define a system that represents dynamic environments and extend the definition of a policy by also including environment actions. We then describe a system that represents possible outcomes of executing a policy plan in a dynamic environment, and define the maintenance by the policy.

Definition 3.11 (Dynamic system)**.** *A* dynamic system *is a quadruple* $A{=}\langle \mathcal{S}, \mathcal{S}_0, \mathcal{A}, \Phi_c\rangle$*, where*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{A} = \mathcal{A}_a \cup \mathcal{A}_e$ *is the finite set of agent* ($\mathcal{A}_a$) *and environment* ($\mathcal{A}_e$) *actions;*
- $\Phi_c : \mathcal{S} \times \mathcal{A}_a \times \mathcal{A}_e \rightarrow 2^{\mathcal{S}}$ *is a non-deterministic transition function.*

We assume that the idle action a_{nop} (resp. e_{nop}) is included in $\mathcal{A}_a$ (resp. $\mathcal{A}_e$) and Φ_c considers concurrent actions, where for all $s \in \mathcal{S}$, $\Phi_c(s, a_{nop}, e_{nop}) = \{s\}$. A sequence $\overline{a} = a_1, a_2, \ldots, a_n$ of agent actions is executable if

$$\exists s_0, \ldots, s_n : \forall i < n, s_{i+1} \in \Phi_c(s_i, a_{i+1}, e_{nop}) \;\wedge\; a_{i+1} \neq a_{nop}.$$

We denote such *(potential) plans* by Σ_a, and by $\Sigma_a(s) \subseteq \Sigma_a$ those that are executable from s. We use the notation $\Sigma'_a = \Sigma_a \cup \{a_{nop}\}$ to also consider the idle agent action.

A policy is defined similarly as in Definition 3.1 over the possible plans of the agent at a state.

Definition 3.12 (Policy)**.** *Given a system* $A = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{A}, \Phi_c\rangle$ *and a set* Σ_a *of plans with actions of* $\mathcal{A}_a{\subseteq}\mathcal{A}$*, a* policy *is a function* $P_{\mu,KB} : \mathcal{S}{\rightarrow}2^{\Sigma'_a}$ *s.t.* $P_{\mu,KB}(s) \subseteq \Sigma_a(s) \cup \{a_{nop}\}$*.*

For any state s, $\{a_{nop}\} \subseteq P_{\mu,KB}(s)$ should hold, for the cases of a moving environment while the agent is idle. We say that $P_{\mu,KB}$ is undefined for a state s, if $P_{\mu,KB}(s) = \{a_{nop}\}$. For readability, we omit subscripts of P, as they are considered to be fixed.

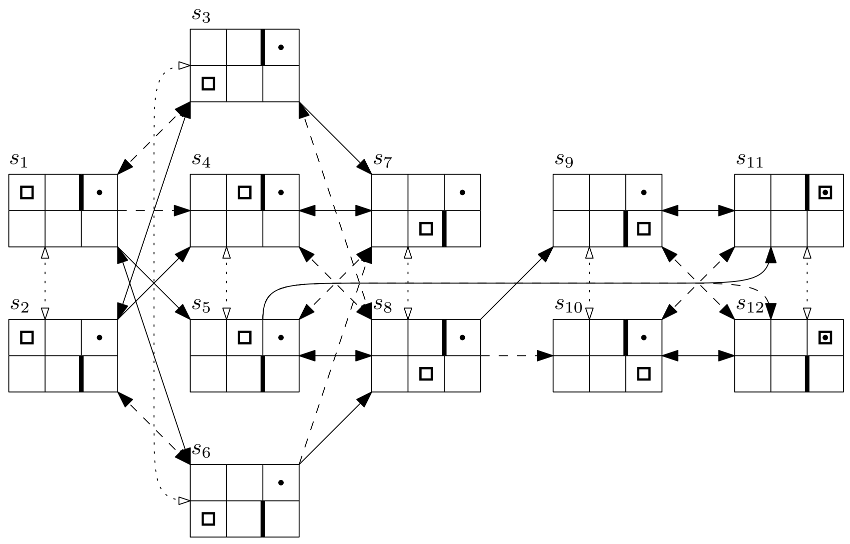
Notice that a plan given by the policy might become inexecutable if the environment acts. In order to consider environments that may interfere with the agent's plan execution, we will express possible outcomes of the desire towards executing the *policy plans.*

Transitions as action sequences We extend the definition of the system $A = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{A}, \Phi_c\rangle$ to represent execution of action sequences.

Definition 3.13 (Dynamic system over action sequences)**.** *Given a dynamic system* $A = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{A}, \Phi_c\rangle$ *with the set* Σ'_a *of potential plans for the agent and the set* $\Sigma_e = \mathcal{A}_e^*$ *of sequences of environment actions, the system* A *is extended with action sequences as* $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma\rangle$ *where*

- $\Sigma = \Sigma'_a \cup \Sigma_e$*, and*

Figure 3.7: Dynamic system of the sliding door example



- $\Phi_\Sigma : \mathcal{S}\times\Sigma'_a\times\Sigma_e \to 2^{\mathcal{S}}$ *is the transition function, where for* $\overline{a} = \langle a_1,\ldots,a_n\rangle$ *and* $\overline{e} = \langle e_1,\ldots,e_n\rangle$*, if* $|\overline{a}| = |\overline{e}|$,

  $\Phi_\Sigma(s,\overline{a},\overline{e}) = \{s' \mid \exists s_0,\ldots,s_n : \forall i < n, s_{i+1} \in \Phi_c(s_i,a_{i+1},e_i) \ \wedge \ s_0 = s \wedge s_n = s'\}$*;*

  and undefined, otherwise.

The transition function Φ_Σ yields the states from executing concurrent action sequences. We use $\Phi_\Sigma(s,P(s),\overline{e})$ as a shorthand for $\bigcup_{\overline{a}\in P(s)} \Phi_\Sigma(s,\overline{a},\overline{e})$.

The evolution of the world described by the system is characterized by trajectories and the closure of a system is defined using these trajectories as follows.

Definition 3.14 (Trajectory and Closure)**.** *In a system* $A_\Sigma = \langle\mathcal{S},\mathcal{S}_0,\Sigma,\Phi_\Sigma\rangle$*, an alternating sequence of states and action sequences* $s_0,\sigma_1,s_1,\ldots,\sigma_n,s_n$ *is a* trajectory *if* $s_i \in \Phi_\Sigma(s_{i-1},\overline{a}_i,\overline{e}_i), i \geq 0$*, for* $\sigma_i = (\overline{a}_i,\overline{e}_i) \in \Sigma$*. The* closure *w.r.t. a set* $S \subseteq \mathcal{S}$ *is*

$$Cl_\Sigma(S,A_\Sigma) = \bigcup_{s\in S}\{s_n \mid \textit{some trajectory } s_0,\sigma_1,s_1,\ldots,\sigma_n,s_n \in A_\Sigma, n \geq 0, \textit{exists s.t. } s_0 = s\}.$$

For illustration of the concepts we use a basic scenario, since even a simplified (yet still interesting) supermarket example has quite a number of states and is difficult to visualize.

Example 3.11 (Sliding door example)**.** Consider a sliding door scenario, where an agent, initially located at (0,0), can move right (r), down (d) or up (u), and a sliding door, located between columns 1-2, can move up (dU), down (dD) or remain still (dN). The goal is to reach (0,2). The dynamic system is shown in Figure 3.7, where $\mathcal{S}_0 = \{s_1,s_2\}$. The action labels are omitted for clarity. The dashed arrows represent the transitions in which the agent moves and the environment (i.e., the door) remains idle, the dotted arrows are the transitions in which the agent is idle, and the lines represent the transitions in which both the agent and the environment moves.

The closure w.r.t. $\mathcal{S}_0$ contains each state in the system, as each state can be reached by a trajectory from the initial states. For example, $s_9 \in Cl_\Sigma(\mathcal{S}_0, A_\Sigma)$ since there is a trajectory $\langle s_1, (r, dD), s_5, (d, dU), s_8, (r, dD), s_9 \rangle$ from $s_1 \in \mathcal{S}_0$.

Following the policy in a dynamic environment

We consider three outcomes of executing a policy plan in the dynamic environment:

(1) The environment's actions may not interfere with the execution of the plan, and the agent can execute the whole plan and reach the state that the policy was aiming for.

(2) The environment may act in a way that a state is reached, from which the remainder of the plan becomes non executable (even if the environment does no longer move).

(3) The agent may reach a state that has a possibility to reach the main goal, so that, instead of executing the remaining plan, a new plan can be determined towards the goal.

We describe a transition function that yields the states by executing some plan returned by P as follows.

Definition 3.15 (Policy transition). *Given a dynamic system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$ and a policy P, the* policy transition function *is defined as $\Psi_{P,\Sigma_e} : \mathcal{S} \times \Sigma_a' \rightarrow 2^{\mathcal{S}}$ with $\Psi_{P,\Sigma_e}(s, \overline{a}) = S_1 \cup S_2 \cup S_3$, where*

$$S_1 = \{s' \mid \overline{a} \in P(s), \exists \overline{e} \in \Sigma_e : s' \in \Phi_\Sigma(s, \overline{a}, \overline{e})\}\} \tag{3.10}$$

$$S_2 = \bigcup_{\overline{a}=\overline{a}'\overline{a}'' \in P(s)} \{s' \mid \exists \overline{e} \in \Sigma_e, s' \in \Phi_\Sigma(s, \overline{a}', \overline{e}) \wedge \Phi_\Sigma(s', \overline{a}'', \overline{e}_{nop}) \models \bot\} \tag{3.11}$$

$$S_3 = \bigcup_{\overline{a}=\overline{a}'\overline{a}'' \in P(s)} \{s' \mid \exists \overline{e} \in \Sigma_e, s' \in \Phi_\Sigma(s, \overline{a}', \overline{e}) \wedge \exists s'' \in \Phi_\Sigma(s', P(s'), \overline{e}_{nop}) : s'' \models \mu\} \tag{3.12}$$

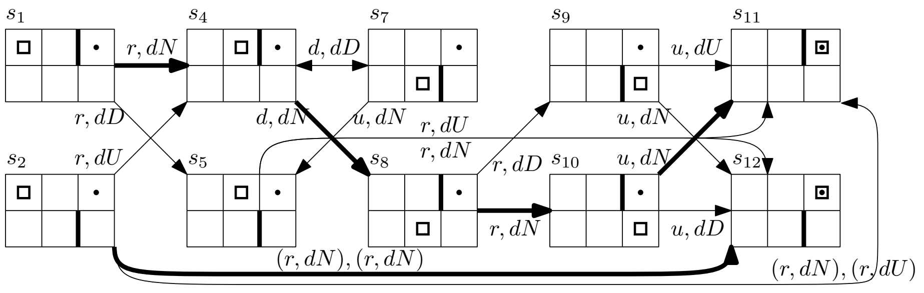
where for a set S of states, $S \models \alpha \Leftrightarrow \forall s \in S : s \models \alpha$ and $\overline{e}_{nop} \in \{e_{nop}\}^*$. The notation $\overline{a} = \overline{a}'\overline{a}''$ is used to distinguish a prefix $\overline{a}'$ of $\overline{a}$, and the remaining action sequence $\overline{a}''$.

The states reachable from s if all of the plan $\overline{a}$ can be executed are computed in (3.10). In (3.11) and (3.12), we focus on prefixes $\overline{a}'$ of $\overline{a}$ to compute the (middle) states reached while executing $\overline{a}$. The states in (3.11) are those reached due to some environment actions $\overline{e}$ during the execution of $\overline{a}$, where the remaining plan $\overline{a}''$ is no longer executable, even if the environment is idle after this point. From the middle states in (3.12), the main goal μ can be reached with a new policy plan (if the environment remains idle).

We represent the case when the environment remains idle with $\Psi_{P,\overline{e}_{nop}}$, where

$$\Psi_{P,\overline{e}_{nop}}(s, \overline{a}) \quad = \quad \{s' \mid \overline{a} \in P(s), s' \in \Phi_\Sigma(s, \overline{a}, \overline{e}_{nop})\} \cup$$

Figure 3.8: Closure w.r.t. the initial states in the sliding door example



$$\bigcup_{\overline{a}=\overline{a}'\overline{a}''\in P(s)} \{s' \in \Phi_\Sigma(s, \overline{a}', \overline{e}_{nop}) \mid \exists s'' \in \Phi_\Sigma(s', P(s'), \overline{e}_{nop}) : s'' \models \mu\}.$$

Informally, the agent will either execute the plan determined by the policy or if it realizes, while executing the plan, that there is a possibility to reach the main goal, it determine a new plan. As there is no interference from the environment, we do not need to take into account possible environment actions, and the case (3.11) will not occur. Notice that $\Psi_{P,\overline{e}_{nop}}(s, \overline{a}) \subseteq \Psi_{P,\Sigma_e}(s, \overline{a})$.

From a state s, a state s' reached after trying to execute a plan $\overline{a}$, i.e., $s' \in \Psi_{P,\Sigma_e}(s, \overline{a})$, is referred as a *checkpoint state* from s, where the agent determines its next policy actions. A trajectory that follows the policy passes through these states. The set of all checkpoint states from a set S of states is similar to Definition 3.14 when the closure Cl_{P,Σ_e} is defined over the trajectories of Ψ_{P,Σ_e}.

Definition 3.16 (Policy Trajectory and Policy Closure)**.** *Given a system* $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$, *and a policy* P *with a policy transition function* Ψ_{P,Σ_e}, *a* policy trajectory *is a trajectory* $s_0, \sigma_1, s_1, \ldots, \sigma_n, s_n$ *in* A_Σ *where* $s_i \in \Psi_{P,\Sigma_e}(s_{i-1}, \overline{a}_i), i \geq 0$, *for* $\sigma_i = (\overline{a}_i, \overline{e}_i) \in \Sigma$ *and* $\overline{a}_i \in P(s_{i-1})$. *The* policy closure *w.r.t. a set* $S \subseteq \mathcal{S}$ *is*

$$Cl_{P,\Sigma_e}(S, A_\Sigma) = \bigcup_{s\in S} \{s_n \mid \begin{array}{l} \textit{some policy trajectory } s_0, \sigma_1, s_1, \ldots, \sigma_n, s_n \in A_\Sigma, n \geq 0, \\ \textit{exists s.t. } s_0 = s \end{array} \}.$$

We sometimes represent policy trajectories with $s_0, s_1, \ldots, s_n$ instead of $s_0, \sigma_1, s_1, \ldots, \sigma_n, s_n$ by projecting away the action sequences, in order to emphasize the behavior of the policy.

Example 3.12 (Sliding door example ctd)**.** Consider a policy P that tells to move right whenever possible and if not possible then to move up/down (depending on which one is executable). In case (0,2) is observable (say, the agent can detect that the door is located in its row), the policy returns the plan to reach that cell. Once the agent reaches (0,2), no more action is taken.

Figure 3.8 shows possible policy trajectories from the initial states s_1, s_2. We have $\Psi_{P,\Sigma_e}(s_1) = \{s_3, s_4, s_5\}$ and $\Psi_{P,\Sigma_e}(s_2) = \{s_4, s_{11}, s_{12}\}$. Notice that in state s_2 the cell (0,2) is observable, as the door is not at the same row. Thus, $P(s_2) = \langle r, r \rangle$. If the door remains still in the first step, i.e., $\langle dN, dN\rangle$ or $\langle dN, dU\rangle$, the policy plan can be executed without interference (3.10) and the goal position can be reached. However, if the door moves up in the first step, then this interferes with the agent's plan (3.11) as it can no longer move right in the second step due to the door (i.e., the remainder of the plan is not executable). Thus, the policy transition ends up in state s_4, where a new policy plan is decided.

All shown states in Figure 3.8 constitute the policy closure w.r.t. the initial states.

Maintenance

The idea is to keep track of the checkpoint states when the policy is followed, and define the maintenance over them. First, the notion of unfolding a policy is defined as a sequence of states the system may go through if it follows the policy, while the environment remains idle, for at most k steps, where k is a constant.

Definition 3.17 (Unfold)**.** *For a system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma\rangle$ and $s \in \mathcal{S}$, Unfold$_k(s, A_\Sigma, P)$ is the set of all sequences $\bar{s} = s_0, \dots, s_l$ where $l \leq k$ and $s_0 = s$ s.t. $P(s_j)$ is defined for all $j < l$, $s_{j+1} \in \Psi_{P,\bar{e}_{nop}}(s_j, P(s_j) \setminus \{a_{nop}\})$, and if $l < k$, then $P(s_j)$ is undefined.*

Example 3.13 (Sliding door example ctd)**.** In Figure 3.8, the trajectories in which the environment remains still give the unfolding trajectories from the initial states (shown with thick arrows).

Based on this, we define the k-maintainability.

Definition 3.18 (k-Maintainability)**.** *For a system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma\rangle$, the policy P k-maintains $S \subseteq \mathcal{S}$ w.r.t. a goal condition μ, if for each state $s \in Cl_{P,\Sigma_e}(S, A_\Sigma)$ and sequence $s_0, s_1 \dots, s_l$ in Unfold$_k(s, A_\Sigma, P)$ some $j \leq l$ exists such that $s_j \models \mu$.*

We say that the original system A_Σ is k-maintained by policy P w.r.t. μ, if P k-maintains the initial states S_0 w.r.t. μ.

Example 3.14 (Sliding door example ctd)**.** As seen in Figure 3.8, the system is 4-maintained by the policy, since from any state in the closure w.r.t. the initial states, the agent reaches the goal position in at most 4 steps if the environment remains idle.

The door scenario is simplistic, and as one adds new properties of the agent, the environment, or a more involved policy, the state space immediately gets larger. Furthermore, as in the supermarket example, the state may contain information irrelevant to the agent's behavior, which leads to a large number of states with unnecessary information.

Modeling alternating execution In the original definition of maintenance [BEBN08], the system is considered to have alternating execution of agent and environment actions, in the following form.

Definition 3.19 ([BEBN08])**.** *A* system *is a quadruple $A^0 = \langle \mathcal{S}, \mathcal{A}, \Phi, poss \rangle$, where*

- *$\mathcal{S}$ is a finite set of system states;*
- *$\mathcal{A} = \mathcal{A}_{ag} \cup \mathcal{A}_{env}$ is a finite set of agent ($\mathcal{A}_{ag}$) and environment ($\mathcal{A}_{env}$) actions;*
- *$\Phi : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ is a non-deterministic transition function;*
- *$poss : \mathcal{S} \to 2^{\mathcal{A}}$ is a function representing possible actions.*

They define a *control function* (i.e. *policy*) as $P : \mathcal{S} \to \mathcal{A}_{ag}$ such that $P(s) \in poss(s)$ whenever $P(s)$ is defined.

Note that if we restrict the concurrent action transition function Φ_c to only allow for execution of (a, e_{nop}) and (a_{nop}, e), this can model an alternating execution of agent and environment actions. The transition (a_{nop}, e_{nop}) then corresponds to having no possible actions for the agent or the environment at a state. Thus, we have the following proposition when only policies with 1-step plans are considered.

Proposition 3.12 (Connection to Baral et al. [BEBN08])**.** *A system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$, where Φ_Σ is built over a restricted Φ_c, is k-maintained by a 1-step policy P w.r.t. μ iff the corresponding system A^0 defined as in Definition 3.19 is k-maintainable for the set $\mathcal{S}_0$ of initial states, due to existence of P, w.r.t. the set $E \subseteq \mathcal{S}$ of states where $E = \{s \mid s \in \mathcal{S}, s \models \mu\}$.*

Proof. For 1-step policies, the transition function Ψ_{P,Σ_e} gets simplified to considering only one-step transitions (3.10), which then makes it possible to compare with the transitions in A^0. ($\Rightarrow$) By definition, we have for each state $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$ and sequence $s_0, s_1 \dots, s_l$ in $Unfold_k(s, A_\Sigma, P)$ that some $j \leq l$ exists such that $s_j \models \mu$, i.e., $\{s_0, s_1 \dots, s_l\} \cap E \neq \emptyset$. This achieves k-maintainability of the set $\mathcal{S}_0$ of states in the system A^0.

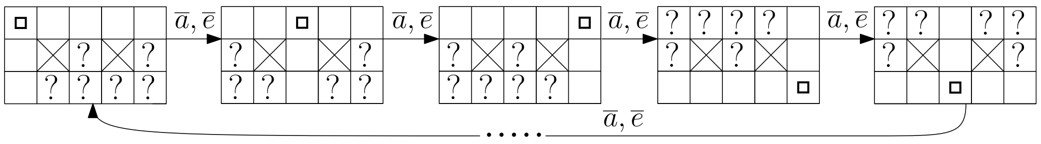
($\Leftarrow$) The control function P witnesses the k-maintainability of the set $\mathcal{S}_0$ of states in the system A^0. Given that the set E of desired states is defined according to a desired condition μ, P can be used as a 1-step policy that k-maintains A_Σ. $\square$

3.3.2 Omitting Unnecessary Information

In Section 3.1, we considered state clustering by getting rid of irrelevant information w.r.t. the policy or the observability of the environment with a focus on static environments. However, in case of dynamic environments, such a clustering idea is unable to distinguish the environment's movement in a state's irrelevant/unobserved part.

Figure 3.9 shows some part of the system when the notion in Section 3.1 is applied to the supermarket example; the states where the agent observes the person are omitted for simplicity. The agent only knows that the environment did some actions $\overline{e}$. Therefore, whenever the person is not observed, the unobserved part is considered to be unknown, because the dynamic nature can not guarantee that some information gained holds in the next state.

Figure 3.9: Equalization is unable to distinguish the unobserved environment movements


$\overline{a}, \overline{e}$
$\overline{a}, \overline{e}$
$\overline{a}, \overline{e}$
$\overline{a}, \overline{e}$
$\overline{a}, \overline{e}$
?
.....

To define maintenance, we must be able to distinguish the transitions where the environment does not move (especially, in the unobserved parts) and represent how this affects the knowledge about the state clusters. To this end, we use equalized d-states.

Definition 3.20. *An* equalized dynamic (d-) state *is a pair* $\langle \hat{s}, \hat{\theta} \rangle$*, where*

(1) the equalized state, $\hat{s}$*, contains the indistinguishable states w.r.t. the policy, and*

(2) the inferred state, $\hat{\theta}$*, contains the states which are inferred to possibly hold by using the knowledge of the environment's movements.*

The state $\hat{s}$ contains the information relevant to the policy or the observability of the environment, while the state $\hat{\theta}$ makes further inferences to represent the effect of the environment's movements; in particular, whether the environment moved or not. Thus, there can be multiple pairs with identical equalized states, but different inferred states.

Building the clusters We consider two *classification functions* described by surjections:

- $h : S \rightarrow \Omega$, where Ω is the set of possible equalized states, and
- $h_r : S \rightarrow 2^{\Theta}$, where Θ is the set of possible inferred states.

Necessarily, h and h_r should satisfy that for every $\langle \hat{s}, \hat{\theta} \rangle$, we have $h_r^{-1}(\hat{\theta}) \subseteq h^{-1}(\hat{s})$. A state may be mapped to more than one inferred state cluster, as these clusters depend on the previous states and the movement of the environment.

The classification function h is based on the notion of indistinguishability, as in Section 3.1. The state clustering is done only to omit the irrelevant information w.r.t. the policy, so that P returns the same output for the cluster. Without going into the details of the policy's components, we assume for simplicity that the clustering satisfies the condition

$$\forall s \in \mathcal{S}, P(s) = P(h(s)) \tag{3.13}$$

which makes sure that for states that are mapped to the same cluster, the policy returns the same plans, i.e., $\forall d, e \in \mathcal{S}, h(d) = h(e) \Rightarrow P(d) = P(e)$.

As said, inferred states depend on the previous states and the taken environment actions. In detail,

- the initial set of inferred states is $\Theta_0 = \{h(s) \mid s \in S_0\}$, and

- the clustering satisfies the following constraint: for all $d, e \in \mathcal{S}$ such that $h_r(d) = h_r(e)$ it holds that
  - $h(d) = h(e)$ and
  - $\exists d', e' \colon d \in \Psi_{P,\Sigma_e \setminus \overline{e}_{nop}}(d', P(d')), e \in \Psi_{P,\Sigma_e \setminus \overline{e}_{nop}}(e', P(e'))$ with $h_r(d') = h_r(e')$.

In other words, only the states that can be reached from a previous state, due to some sequence of environment actions, are mapped into the inferred states. This is similarly done for the case of $\overline{e}_{nop}$, to distinguish the states reached only by $\overline{e}_{nop}$.

It was shown in Section 3.2, how to do state clustering h for action languages such that the set $\hat{S}$ of state clusters is achieved. Inferred state clustering h_r is possible along the same lines. For this, we need to take into account the transitions $\Phi_{P,\Sigma'_e} \subseteq S \times \Sigma'_a \times \Sigma_e \times S$ restricted to the policy P and the environment actions $\Sigma'_e = \Sigma_e \setminus \overline{e}_{nop}$, defined as

$$\Phi_{P,\Sigma'_e} = \{(s, \overline{a}, \overline{e}, s') \mid (s, \overline{a}, \overline{e}, s') \in \Phi_\Sigma, \overline{a} \in P(s), \overline{e} \in \Sigma'_e\}.$$

The clustering can then be defined as follows:

$$\hat{S}_r = \{\hat{s}_r \mid \forall d_1, d_2 \in S, d_1, d_2 \in \hat{s}_r \iff (\exists \hat{s} \in \hat{S}, d_1, d_2 \in \hat{s}) \wedge \tag{3.14}$$

$$(\exists d'_1, d'_2 : (d'_1, P(d'_1), \overline{e}_1, d_1), (d'_2, P(d'_2), \overline{e}_2, d_2) \in \Phi_{P,\Sigma'_e})\} \tag{3.15}$$

In order to map two states to the same inferred state, first both of these states should be mapped to the same equalized state (3.14) and these states should be reachable from a previous state when the policy is followed (3.15). The clustering definition is similar for $\overline{e}_{nop}$.

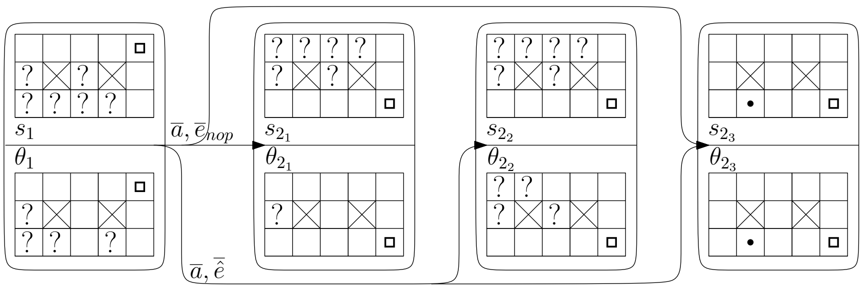
Abstract environment actions Clustering the states by omitting the information about the irrelevant part of the state leads to abstracting the irrelevant environment actions. Since the main aim is to represent whether or not the environment concurrently moved, we distinguish between the actions e_{nop} and $\hat{e}$ which is an abstraction of all other environment actions; we let $\widehat{\mathcal{A}}_e = \{e_{nop}, \hat{e}\}$ and consider the mapping $\phi_h : \mathcal{A}_e \to \widehat{\mathcal{A}}_e$.

Such an abstraction can be seen as the coarsest one possible, as it only distinguishes whether the environment moved or not. It is sufficient for defining maintenance, since the focus is on cases in which the environment does not move.

Example 3.15 (Supermarket example ctd)**.** Figure 3.10 shows an example of equalized d-states. As expected, the inferred states have less possible locations for the person than the equalized states, since the possible locations are inferred more precisely depending on whether he/she moved or not.

Furthermore, whether or not the person concurrently moves results in different inferred states. From state $\langle s_1, \theta_1 \rangle$, the action $\overline{e}_{nop}$ causes the cells observed at θ_1 to remain the same in θ_{2_1}, although to the agent's view, s_{2_1}, they become unknown. If the person executes some actions $\overline{\hat{e}}$, his possible locations in θ_{2_2} are inferred from θ_1. Notice that θ_{2_2} is not the same as s_{2_2}, as it shows the locations the person can move to in the same time steps as the agent. A transition may also lead to a state where the agent observes the person; then $s_{2_3} = \theta_{2_3}$ holds, since the person is obviously not in the unobserved parts.

Figure 3.10: Pair states with inferred states that distinguish if the environment moved or not



Equalized Dynamic Systems and Maintenance

A system that represents the policy execution in a dynamic environment is defined over the original system by taking the classification functions and the policy into account.

Definition 3.21. *Given a dynamic system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$, a policy P with the policy transition function Ψ_{P,Σ_e}, an* equalized dynamic system A_P^{h,h_r}*, w.r.t. the classification functions h, h_r and the policy P, is defined as $A_P^{h,h_r} = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \Sigma'_a, \widehat{\Sigma}_e, \widehat{\Psi}_{P,\Sigma_e} \rangle$, where*

- *$\widehat{\mathcal{S}}$ is the finite set of equalized d-states;*
- *$\widehat{\mathcal{S}}_0 \subseteq \widehat{\mathcal{S}}$ is the set of initial equalized d-states, if $h^{-1}(\widehat{\theta}) \cap \mathcal{S}_0 \neq \emptyset$;*
- *$\Sigma'_a = \Sigma_a \cup \{a_{nop}\}$ is the possible plans with agent actions;*
- *$\widehat{\Sigma}_e = \widehat{A}_e^*$ is the set of sequences of abstract environment actions (for $\widehat{A}_e = \{e_{nop}, \hat{e}\}$);*
- *$\widehat{\Psi}_{P,\widehat{\Sigma}_e} : \widehat{\mathcal{S}} \times \Sigma'_a \to 2^{\widehat{\mathcal{S}}}$ is the policy transition function in the dynamic environment, i.e.,*

$$\widehat{\Psi}_{P,\widehat{\Sigma}_e}(\langle \hat{s}, \hat{\theta} \rangle, \overline{a}) = \{\langle h(s'), h_r(s') \rangle \mid \ \overline{a} \in P(\hat{s}), \exists s \in h_r^{-1}(\hat{\theta}), s' \in \Psi_{P,\Sigma_e}(s, \overline{a})\}.$$

The transitions $\widehat{\Psi}_{P,\overline{e}_{nop}}$ show an idle environment. For simplicity, $\langle \hat{s}, \hat{\theta} \rangle$ is denoted as $\hat{s}\hat{\theta}$.

The equalized dynamic system is defined over the state clusters of the checkpoint states in the original system. A small set of states and transitions help to focus on the details important for the policy, without losing any property of the behavior.

Proposition 3.13. *A policy trajectory $s_0, s_1, \ldots, s_n$ in A_Σ where $s_0 \in S_0$ and $s_{i+1} \in \Psi_{P,\Sigma_e}(s_i, \overline{a})$ for some $\overline{a} \in P(s_i)$, $i > 1$, has a corresponding trajectory $\hat{s}\hat{\theta}_0, \hat{s}\hat{\theta}_1, \ldots, \hat{s}\hat{\theta}_n$ in A_P^{h,h_r} where $\hat{s}\hat{\theta}_0 \in \widehat{S}_0$ and $\hat{s}\hat{\theta}_{i+1} \in \widehat{\Psi}_{P,\widehat{\Sigma}_e}(\hat{s}\hat{\theta}_i, \overline{a})$ for some $\overline{a} \in P(\hat{s}_i)$, $i > 1$.*

Proof. Towards a contradiction assume that there is a trajectory $s_0, s_1, \ldots, s_n$ in A_Σ which does not have a corresponding trajectory in A_P^{h,h_r}. This means that some transition s_i, s_{i+1}, $i < n$ is not represented in A_P^{h,h_r}. So, there exists no $\overline{a} \in P(h(s_i))$ such that $h(s_{i+1})h_r(s_{i+1}) \in \widehat{\Psi}_{P,\widehat{\Sigma}_e}(h(s_i)h_r(s_i), \overline{a})$. In other words, it holds that for every $\overline{a} \in P(h(s_i))$, for all $s \in h_r^{-1}(h_r(s_i))$ there does not exist some $s' \in \Psi_{P,\Sigma_e}(s, \overline{a})$ such that $s' \in h_r^{-1}(h_r(s_{i+1}))$.

However, with the assumption on the clustering (3.13), we know that $P(h(s_i)) = P(s_i)$ and since $s_i \in h_r^{-1}(h_r(s_i))$ and $s_{i+1} \in h_r^{-1}(h_r(s_{i+1}))$, we obtain that a transition among s_i, s_{i+1} is not possible in A_Σ, which is a contradiction. $\square$

Knowing that every policy trajectory in the original system is preserved in the equalized dynamic system, we can move on to define the maintenance over the equalized dynamic system. For that, we first need to lift the previous definitions.

The closure $\widehat{Cl}(\widehat{S}, A_P^{h,h_r})$ is defined akin to Definition 3.14, using the trajectories of $\widehat{\Psi}_{P,\widehat{\Sigma}_e}$.

Definition 3.22 (Closure over equalized d-states). *The* closure *of an equalized dynamic system A_P^{h,h_r} w.r.t. a set $S \subseteq \widehat{\mathcal{S}}$ is*

$$\widehat{Cl}(S, A_P^{h,h_r}) = \bigcup_{\hat{s}\hat{\theta} \in S} \{\hat{s}\hat{\theta}_n \mid \begin{array}{l} \textit{some trajectory } \hat{s}\hat{\theta}_0, \hat{s}\hat{\theta}_1, \ldots, \hat{s}\hat{\theta}_n \in A_P^{h,h_r}, n \geq 0, \\ \textit{exists s.t. } \hat{s}\hat{\theta}_0 = \hat{s} \end{array} \}.$$

Proposition 3.13 gives us the following result.

Lemma 3.14. *For a given set S of states, any state s in $Cl_{P,\Sigma_e}(S, A_\Sigma)$ has a corresponding state $\hat{s}\hat{\theta}$ in $\widehat{Cl}(\widehat{S}, A_P^{h,h_r})$, where $\hat{s} = h(s)$ and $\hat{\theta} = h_r(s)$.*

The result holds since for every pair of successor states in A_Σ, there is a corresponding pair of equalized d-states in A_P^{h,h_r}. So any sequence of states considered in the closure of A_Σ w.r.t. a set S, has a corresponding sequence in the closure of A_P^{h,h_r} w.r.t. $\widehat{S}$.

Maintenance over the equalized dynamic system.

We define the unfolding of the policy over the equalized d-states, similar to Definition 3.23, by considering $\widehat{\Psi}_{P,\overline{e}_{nop}}$.

Definition 3.23 (Unfolding over equalized d-states). *Let A_P^{h,h_r} be an equalized dynamic system. For $\hat{s} \in \widehat{\mathcal{S}}$, $Unfold_k(\hat{s}, A_P^{h,h_r})$ is the set of all sequences $\overline{\hat{s}} = \hat{s}_0, \ldots, \hat{s}_l$ where $\hat{s}_0 = \hat{s}$ and $l \leq k$ s.t.*

(i) *$P(\hat{s}_j)$ is defined for all $j < l$,*
(ii) *$\hat{s}_{j+1} \in \widehat{\Psi}_{P,\overline{e}_{nop}}(\hat{s}_j, P(\hat{s}_j) \setminus \{a_{nop}\})$, and*
(iii) *if $l < k$, then $P(\hat{s}_j)$ is undefined.*

The clustering condition (3.13) ensures that no transitions will be introduced different from how the policy behaves in the original system. By Proposition 3.13, we get the following result.

Lemma 3.15. *For some k, for each sequence $s_0, \ldots, s_l$ in $Unfold_k(s, A_\Sigma, P)$, some sequence $\hat{s}\hat{\theta}_0, \ldots, \hat{s}\hat{\theta}_l$ in $\widehat{Unfold}_k(\hat{s}\hat{\theta}, A_P^{h,h_r})$ exists with $\hat{s}_i{=}h(s_i)$ and $\hat{\theta}_i{=}h_r(s_i), 0 \leq i \leq l$.*

Similar to Section 3.1, we focus on *appropriate* clusterings (Definition 3.5) of the states which ensures that we have $\hat{s} \models \mu \iff \forall s \in h^{-1}(\hat{s}) : s \models \mu$ for the equalized states. For equalized d-states, let $\hat{s}\hat{\theta} \models \mu$ denote $\hat{s} \models \mu$.

We define the k-maintainability of the equalized dynamic system as follows.

Definition 3.24 (Equalized k-Maintainability)**.** *An equalized system $A_P^{h,h_r} = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \Sigma'_a, \widehat{\Sigma}_e, \widehat{\Psi}_{P,\Sigma_e} \rangle$ is* k-maintainable, *if the policy P k-maintains $\widehat{\mathcal{S}}_0$ w.r.t. μ: For each state $\hat{s}\hat{\theta}$ in $\widehat{Cl}(\widehat{S}_0, A_P^{h,h_r})$ and sequence $\hat{s}\hat{\theta}_0, \hat{s}\hat{\theta}_1, \ldots, \hat{s}\hat{\theta}_l$ in $\widehat{Unfold}_k(\hat{s}\hat{\theta}, A_P^{h,h_r})$ some $j \leq l$ exists s.t. $\hat{s}\hat{\theta}_j \models \mu$.*

Example 3.16 (Supermarket example ctd)**.** In the supermarket example for environment size 5×9 (Figure 3.6), the given policy 9-maintains the set of initial states. At any state in the closure, the person may be located at the farthest possible point, and if he does not move for at least 9 steps, then the agent will eventually observe the person and catch him.

The following theorem shows that the clustering does not introduce false positives.

Theorem 3.16 (Soundness)**.** *If A_P^{h,h_r} is k-maintainable, then A_Σ is k-maintained by P.*

Proof. Assume that A_Σ is not k-maintained by P. Let $s \in Cl_{P,\Sigma_e}(S_0, A_\Sigma)$ be a state and $\tau{=}s_0, s_1 \ldots, s_l$ in $Unfold_k(s, A_\Sigma, P)$ such that $s_l \not\models \mu$. By Lemma 3.14-3.15, we know that some $\hat{s}\hat{\theta} \in \widehat{Cl}(\widehat{S}_0, A_P^{h,h_r})$ exists with $\hat{s} = h(s)$ and that some $\hat{\tau}{=}\hat{s}\hat{\theta}_0, \hat{s}\hat{\theta}_1 \ldots, \hat{s}\hat{\theta}_l$ in $\widehat{Unfold}_k(\hat{s}\hat{\theta}, A_P^{h,h_r})$ exists with $\hat{s}_i{=}h(s_i)$, $0 \leq i \leq l$. By assumption on s_l, $\hat{s}\hat{\theta}_l \not\models \mu$. Hence, A_P^{h,h_r} is not k-maintainable. □

In order to have completeness, we need further restrictions on the state clustering h, to avoid introducing spurious trajectories. We consider the properness condition (similar to condition (3.8) from Section 3.1.5):

$$\hat{s}\hat{\theta}' \in \widehat{\Psi}_{P,\widehat{\Sigma}_e}(\hat{s}\hat{\theta}, \overline{a}) \iff \forall s' \in h^{-1}(\hat{s}'), \exists s \in h^{-1}(\hat{s}) : \Psi_{P,\Sigma_e}(s, \overline{a}) \tag{3.16}$$

This condition ensures that if A_P^{h,h_r} has a transition from $\hat{s}\hat{\theta}_1$ to $\hat{s}\hat{\theta}_2$, then any state mapped to $\hat{s}\hat{\theta}_2$ has a transition from some state mapped to $\hat{s}\hat{\theta}_1$. This allows for the possibility of backtracking any trajectory found in A_P^{h,h_r} and map it back to A_Σ.

We then obtain the completeness result with the requirement that no "non-initial" states are mapped to initial equalized d-states, i.e., it should hold that $\forall s \in S_0, h^{-1}(h(s)) \subseteq S_0$.

Theorem 3.17 (Completeness)**.** *If A_Σ is k-maintained by P and h is proper, then A_P^{h,h_r} is k-maintainable.*

Proof. Towards a contradiction assume A_P^{h,h_r} is not k-maintainable, i.e., there exists a state $\hat{s}\hat{\theta} \in \widehat{Cl}(\widehat{S}_0, A_P^{h,h_r})$ and a sequence $\hat{s}\hat{\theta}_0, \hat{s}\hat{\theta}_1, \ldots, \hat{s}\hat{\theta}_l$ in $\widehat{Unfold}_k(\hat{s}\hat{\theta}, A_P^{h,h_r})$ with $\hat{s}\hat{\theta}_0 = \hat{s}\hat{\theta}$ such that for all $j \leq l$ we have $\hat{s}\hat{\theta}_j \not\models \mu$. Let $s_l \in h^{-1}(\hat{s}_l)$. By condition (3.16), we can go from s_l through the sequence back to some $s \in h^{-1}(\hat{s})$. We claim that $s \in Cl_{P,\Sigma_e}(S_0, A_\Sigma)$; it follows then that A_Σ is not k-maintained by P, which reaches a contradiction.

Proof of claim: $\hat{s}\hat{\theta} \in \widehat{Cl}(\widehat{S}_0, A_P^{h,h_r})$ means that there is a trajectory in A_P^{h,h_r} from some $\hat{s}\hat{\theta}_0 \in \widehat{S}_0$. So, by condition (3.16), we can go from s to some $s_0 \in h^{-1}(\hat{s}_0)$. Since $\forall s \in S_0, h^{-1}(h(s)) \subseteq S_0$ holds, we have $s_0 \in S_0$. Hence, $s \in Cl_{P,\Sigma_e} S_0, A_\Sigma$. □

The equalized dynamic system can represent static environments by only allowing e_{nop} and can be related to the equalized static system (Section 3.1). If the actions are reversible (as in the supermarket example) and the agent's observations during a plan execution contribute to the decision making in the next state, we obtain the following.

Corollary 3.18. *If the equalized dynamic system A_P^{h,h_r} is k-maintainable, then the policy P works in at most k steps in the equalized static system $A_{h,P}$.*

Proof. Assume for a contradiction that A_P^{h,h_r} is k-maintainable, but the policy P does not work within k steps in $A_{h,P}$. This means that in $A_{h,P}$ there is a trajectory $\hat{\sigma} = \hat{s}_0, \ldots, \hat{s}_n, n \leq k$ with $\hat{s}_0 \in \hat{S}_0$ where for any $j \leq n$, $\hat{s}_j \not\models \mu$. We will show that both forms of transitions (3.10) and (3.12) can be emulated for the $\overline{e}_{nop}$ case, thus causing a contradiction with the assumed behavior of A_P^{h,h_r}. Notice that the transition (3.11) is not considered, as it requires to have a non-idle environment action, which does not occur in the static case.

The transitions of form (3.10) reach the states that the policy aims for without interruption, which is the same as the definition of Φ_B in $A_{h,P}$. Now consider the case of making an observation during execution of the transitions in $\hat{\sigma}$ of a possibility to achieve the goal. Formally, this means that there exists a plan $\overline{a} \in P(\hat{s}_i)$ with $\hat{s}_{i+1} \in \Phi_B(\hat{s}_i, \overline{a})$ for some $i < n$ that contains a prefix $\overline{a}' = a_1, \ldots, a_i$ where there exists $\hat{s}' \in \hat{\Phi}(\hat{s}_i, \overline{a}')$ with $\hat{s}'' \in \Phi_B(\hat{s}', P(\hat{s}'))$ such that $\hat{s}'' \models \mu$. Remember that we assumed that the actions in $A_{h,P}$ are reversible and the agent's observations during a plan execution can contribute to the policy's decision making in the next state. Thus, from $\hat{s}_{i+1}$ the policy P can decide on a plan that consists of the reversed action sequence of $\overline{a}''$ for $\overline{a} = \overline{a}'\overline{a}''$ and a plan from $P(\hat{s}')$ to reach $\hat{s}''$ with $\hat{s}'' \models \mu$ in the next state. This way (3.12) can be emulated. □

The result follows, since the assumptions ensure that if the agent reaches a state while also observing the main goal on the way, then the policy will have the agent reach the main goal in the next state. However, the reverse of the corollary may not hold, as the dynamic nature of the environment may have the agent end up in a state that was not considered in the static environment.

3.3.3 Computational Complexity

In this section, we consider the computational complexity of k-maintainability.

Assumptions. We assume that given states $s, s' \in \mathcal{S}$, which are given in a binary encoding, and $(a, e) \in \Sigma'_a \times \Sigma_e$, deciding $\Phi_c(s, (a, e), s')$ is in Σ^p_i, for some $i \geq 0$; this reflects theory-based specification by action theories, logic programs, or QBFs possibly with projective auxiliary variables.

Proposition 3.19. *Checking executability of any sequences $\bar{a}$, $\bar{e}$ on $\mathcal{A}^*_a$, resp. $\mathcal{A}^*_e$, at a state s is in Σ^p_j, where $j = \max(1, i)$.*

Proof. For sequences $\bar{a} = a_1, a_2, \ldots, a_n$ and $\bar{e} = e_1, e_2, \ldots, e_n$, we need to guess a sequence of states $s_1, \ldots, s_n$ and check whether $s_1 = s$ and $\Phi_c(s_{t-1}, (a_t, e_t), s_t)$ holds for all $t > 1$. □

Consequently, deciding whether a sequence $\bar{a} \in \mathcal{A}^*_a$ may occur in $P(s)$ such that a successor state exists for a suitable $\bar{e}$ has the same complexity as deciding executability of $\bar{a}, \bar{e}$ at s. We (reasonably) assume that $P(s)$ selects among those $\bar{a}$ only polynomially many and of polynomial length (in the state size), called *p-jump* plans. Furthermore, we assume that recognizing plans $\bar{a}$ in $P(s)$ and deciding $P(s) \neq \{a_{nop}\}$ is feasible in polynomial time (this holds e.g. if $P(s)$ is computable in logspace), and that the goal test $s \models \mu$ is also in polynomial time. Then we obtain:

Lemma 3.20. *Deciding (i) given $s, \bar{a}$ and s' whether $\Psi_{P,\Sigma_e}(s, \bar{a}, s')$ holds is in Σ^p_{j+1} and (ii) given a sequence $\bar{s}$ whether $\bar{s} \in Unfold_k(s, A_\Sigma, P)$ is in Σ^p_j.*

Proof. For deciding (i), we need to check if $\Psi_{P,\Sigma_e}(s, \bar{a}, s')$ satisfies one of the cases (3.10)-(3.12). For checking (3.10), we can guess an environment action sequence $\bar{e}$. For (3.11), we additionally guess a prefix $\bar{a}'$ of $\bar{a}$, and for (3.12) an additional guess of a state s'' to check for $s'' \in \Phi_\Sigma(s', P(s'), \bar{e}_{nop})$ is needed. For deciding (ii), we need to guess an action sequence $\bar{a}$ to check if the policy P decides these actions and satisfy the conditions in Definition 3.23. □

The lemma also holds for goal tests in Π^p_i. If the initial state check $s \in \mathcal{S}_0$ is in Π^p_j, we obtain the following result.

Theorem 3.21 (k-Maintaining Check)**.** *Deciding whether a system $A_\Sigma = \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$ is k-maintained, $k \geq 0$, by a given policy P w.r.t. a goal μ is* PSPACE*-complete.*

Proof. By Lemma 3.20(i), deciding $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$ is in NPSPACE. We can check the existence of a counterexample to k-maintenance by guessing a state $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$ and checking if the condition of Definition 3.18 is violated, by guessing the sequence $\bar{s} = s_0, s_1, \ldots, s_l$ in $\bar{s} \in Unfold_k(s, A_\Sigma, P)$ stepwise. As NPSPACE = PSPACE, this yields the upper bound. On the other hand, the problem is PSPACE-hard already in plain settings, with deterministic actions and simple policies, due to the PSPACE-completeness of succinct graph reachability [Bal96]. □

The complexity is lowered, if we assume that for deciding $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$? an oracle in some class of the Polynomial Hierarchy is available and that k is polynomially bounded; in particular, we have the following.

Theorem 3.22 (Bounded k-Maintaining Check)**.** *If deciding whether $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$ is in Σ_i^p and $k \geq 0$ is polynomially bounded, then deciding whether a system $A_\Sigma {=} \langle \mathcal{S}, \mathcal{S}_0, \Sigma, \Phi_\Sigma \rangle$ is k-maintained, $k \geq 0$, by a given policy P w.r.t. a goal μ is Π_j^p-complete.*

Proof. In order to check the existence of a counterexample to k-maintenance, for a guessed state $s \in Cl_{P,\Sigma_e}(\mathcal{S}_0, A_\Sigma)$, we guess a sequence $\bar{s} = s_0, s_1, \dots, s_l, l \leq k$ in $\bar{s} \in Unfold_k(s, A_\Sigma, P)$. By assumption and Lemma 3.20, both checks are feasible in Σ_j. Furthermore, testing $s_l \models \mu$ is polynomial. Thus, we get Π_j^p-membership. The matching lower bound Π_j is easily established by a transition relation with Σ_j complexity. □

3.4 Discussion

In this chapter, we described a formal semantics to represent the behavior of a reactive agent that follows a given policy. The notions of profiles and state clustering help in omitting irrelevant information. This also comes in handy when dealing with partial observability, since it omits the unobservable information that is irrelevant to the policy. In the equalized transition system, the trajectories from the initial states correspond to the policy execution, where one can check and verify properties of the policy. The properness condition ensures that any counterexample found in the equalized transition system stating a failure of the policy has a concrete trajectory in the original transition system. This way, the shortcomings of the policy can be detected, and thus improved.

For target language definitions, we can use other formalisms with different expressiveness capabilities, e.g., answer set programming. Target descriptions can be made more complex by considering formulas. In particular, target formulas with disjunctions would express nondeterminism in the environment that affects the target determination. Handling this within the framework requires further study.

It is also possible to use other plans, e.g., short conditional plans, in the planner component. Furthermore, this component can be extended by considering a plan library of precomputed plans. This offline planning component can provide the frequently used plans and reduce the calls to the online planner.

The policies described in the static setting determine targets with some target function and call a planner for a conformant plan that guarantees reaching the target. In the dynamic setting, plans need not be conformant, as no targets are considered and all states reached by trying to execute the plan occur in the closure, and thus matter for defining the maintenance.

ASP and action languages with incomplete knowledge

Representing incomplete information about the states and formalizing sensing actions have been investigated in order to compute conformant [EFL$^+$03, STB04, TSGM11] or conditional plans [SB01]. The notion of combined states introduced in [SB01] consists of the real state and the state in which the agent thinks it may be in. In our case the dynamic state is defined over two types of state clusters; one showing what can be distinguished over the state and the other showing what is possible to hold. The notions of approximation considered in [TSGM11, SB01] are sound and thus makes it possible to search for a solution of a planning problem over the approximation. In [TSGM11] they also study further conditions to obtain completeness over the approximation, applicable to action theories whose static causal laws are of a special form. Such approximation notions could be useful in distinguishing the relevant details to the policy behavior to construct the equalizations.

Having a three-valued semantics helps in reflecting undefinedness of the truth values of literals. Such a semantics has been investigated in ASP, where the notion of *partial stable models* was introduced [Prz90]. In [JNS$^+$06], a translation to a disjunctive program was shown that preserves the partial stable models semantics. This allows to compute the partial stable models of a program by computing the stable models of the translated program. Such a notion can be handy in defining the *unknown* values when describing sensing actions, and singling out the details relevant to the policy in the ASP program. Reasoning about the behavior of the policy can then be done over the partial models of the program.

Further investigations in this direction are not conducted in this thesis, as the focus is shifted to the unexplored area of considering abstraction as an over-approximation that reduces the vocabulary of the program and the problem size.

Part II

Exploiting Over-Approximation

CHAPTER 4

Abstraction for Answer Set Programs

In this chapter, we make the first step towards employing the concept of abstraction in ASP, with the aim of computing over-approximations and abstracting over the irrelevant aspects of answer set programs.

We focus on abstraction from a program by constructing an (abstract) program with a smaller vocabulary and ensuring that the original program is over-approximated, i.e., every original answer set can be mapped to some abstract answer set. The smaller vocabulary simplifies the search for some answer set in the abstract program, but then the computed abstract answer set needs to be checked for concreteness. As spurious answer sets can be introduced, one may need to go over all abstract answer sets until a concrete one is found. If the original program has no answer set, all encountered abstract answer sets will be spurious. We introduce a CEGAR [CGJ$^+$03] inspired methodology that starts with an initial abstraction and upon encountering spurious answer sets, refines the abstraction until a concrete solution is found. Deciding on a refinement is done by finding a cause of the spuriousness using ASP debugging approaches (e.g., [BGP$^+$07]).

Outline In Section 4.1, we introduce the abstraction notion to ASP, describe possible approaches and the CEGAR-style methodology that is considered. We then describe two main abstraction approaches for answer set programs. Section 4.2 introduces abstraction by omitting atoms from the program and constructing over-approximations, while Section 4.3 introduces abstraction over the domain of the program. In order to handle the unavoidably introduced spurious abstract answer sets, in Section 4.4 we propose a method for determining refinements for the abstractions by employing ASP debugging methods. We then describe the overall abstraction and refinement methodology in Section 4.5. Before we conclude, Section 4.6 points out the possibility to extend the domain abstraction to consider a multi-dimensional abstraction mapping in order to describe more sophisticated abstractions. We conclude in Section 4.7 with a discussion and address related work in ASP on simplification methods.

4.1 Introducing Abstraction in ASP

The well-known notion of abstraction is about intentionaly discarding some of the details of a problem in order to take a more high-level view in finding a solution. Abstracting over the details can be done in a way that the problem is over-approximated, which means all original solutions have a corresponding solution in the abstraction. This way of abstraction also makes it possible to encounter spurious abstract solutions, which do not have a corresponding concrete solution at the original level.

Our aim is introduce such a notion of abstraction in ASP, which means to introduce a method that over-approximates a given program Π, with vocabulary $\mathcal{A}$, through constructing a simpler program Π' with a vocabulary reduced to $\mathcal{A}'$, i.e., $|\mathcal{A}| \geq |\mathcal{A}'|$, and ensuring that the results of reasoning on the original program are not lost, at the cost of obtaining spurious answer sets. With this aim, we propose the following definition for abstraction of answer set programs.

Definition 4.1. *Given two (ground or non-ground) programs* Π *and* Π' *on atoms* $\mathcal{A} \subseteq HB_{\Pi}$ *and* $\mathcal{A}' \subseteq HB_{\Pi'}$*, respectively, with* $|\mathcal{A}| \geq |\mathcal{A}'|$, Π' *is an* abstraction *of* Π *if there exists a mapping* $m : \mathcal{A} \rightarrow \mathcal{A}' \cup \{\top\}$ *such that for any answer set* I *of* Π*,* $I' = \{m(\alpha) \mid \alpha \in I\}$ *is an answer set of* Π'.

We refer to m as an *abstraction mapping*. This notion of abstraction gives us the possibility to do clustering over atoms of the program. This way, an abstract program with a smaller vocabulary can be used to compute some abstract answer set. The reduced vocabulary simplifies the search for an abstract answer set I' at the abstract level, while an additional check has to be done to see whether an original answer set can be computed in Π that maps to I'.

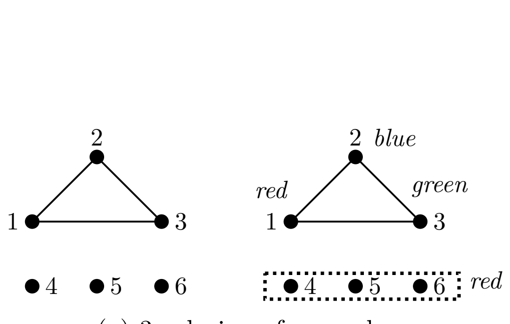
In the thesis, we introduce approaches for two kinds of abstraction mappings: (1) abstraction by omission, and (2) domain abstraction. The former is about omitting atoms from a program, i.e., clustering them into $\top$, and considering an abstract program over the remaining atoms, while the latter investigates abstraction over non-ground ASP programs given a mapping over their domain (i.e., the Herbrand universe) that singles out the domain elements.

Next example illustrates the abstraction types.

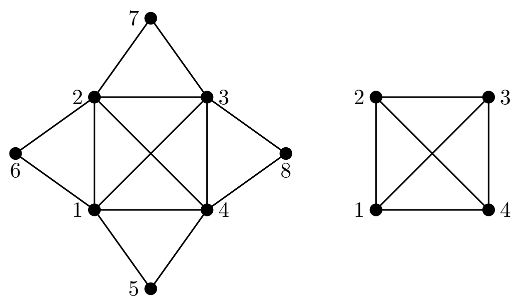
Example 4.1. Consider the program that describes the graph 3-coloring problem below (adapted from the coloring encoding in the ASP Competition 2013) and the graphs shown in Figure 4.1.

$$color(red).\quad color(green).\quad color(blue).$$

Figure 4.1: Graph coloring instances



(a) 3-coloring of a graph



(b) Non-3-colorable graph

$$
\begin{aligned}
\{chosenColor(N,C)\} &\leftarrow node(N), color(C).\\
colored(N) &\leftarrow chosenColor(N,C).\\
\bot &\leftarrow not\ colored(N), node(N).\\
\bot &\leftarrow chosenColor(N,C_1), chosenColor(N,C_2), C_1 \neq C_2.\\
\bot &\leftarrow chosenColor(N_1,C), chosenColor(N_2,C), edge(N_1,N_2).
\end{aligned}
\tag{4.1}
$$

The graph instance in Figure 4.1a has 162 possible 3-colorings of the nodes, while there are 6 possible colorings of nodes 1-2-3. The graph instance in Figure 4.1b is not 3-colorable due to the clique 1-2-3-4.

We consider two possible abstractions over the nodes in the problem. One possibility is to omit the details of certain nodes in the instance, and to focus on the coloring of the remaining nodes. The other possibility is to cluster certain nodes into one node and to consider the coloring of the abstracted instance.

For Figure 4.1b, omitting all the nodes except 1-2-3-4 would still give the uncolorability result. As for Figure 4.1a clustering the nodes 4-5-6 into one abstract node, say $\hat{4}$, would result in 3-coloring the nodes 1-2-3 and assigning some colors to the cluster node $\hat{4}$, e.g., $chosenColor(1, red)$, $chosenColor(2, blue)$, $chosenColor(3, green)$, $chosenColor(\hat{4}, red)$.

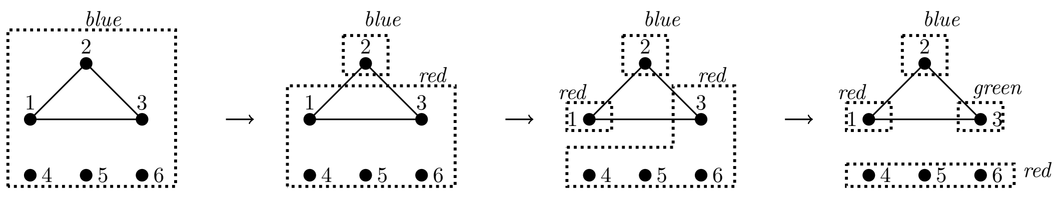
In the following sections, we describe methods to construct an abstract ASP program for a given ASP program and an abstraction mapping of the above two kinds, by ensuring that the original program is over-approximated.

Due to its definition, an over-approximation of a program gives us the following property.

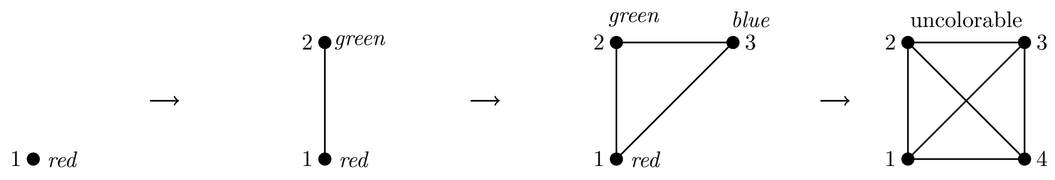
Proposition 4.1. *Let Π' be an abstraction of Π. If $AS(\Pi') = \emptyset$, then we have $AS(\Pi) = \emptyset$.*

In Example 4.1, the abstract program constructed from the encoding (4.1) with the instance shown in Figure 4.1b by omitting all the nodes except 1-2-3-4 should return no abstract answer set.

Figure 4.2: Abstraction refinement upon spurious graph colorings



(a) Dividing abstract node clusters



(b) Adding back omitted nodes

The over-approximation can also cause to encounter abstract answer sets that do not have a corresponding original answer set.

Definition 4.2 (Spurious & concrete answer sets)**.** *Let* Π' *be an abstraction of* Π *for the mapping* m*. The answer set* $I' \in AS(\Pi')$ *is* concrete *if there exists an answer set* $I \in AS(\Pi)$ *such that* $m(I) = I'$*; otherwise, it is* spurious.

In Example 4.1, the abstract program constructed from the encoding (4.1) with the instance shown in Figure 4.1a by clustering the nodes 4-5-6 into one abstract node should only consist of concrete abstract answer sets, since any coloring of the cluster node will be concrete.

In order to get rid of spurious abstract answer sets, the abstraction mapping m needs to be *refined* to a more fine-grained abstraction. In case of omission abstraction, the refinement would be to add back some of the omitted atoms, while in domain abstraction it would be to divide the clusters.

Abstraction Refinement Methodology We consider a CEGAR-style abstraction refinement approach, by starting with an initial abstraction and then refining the abstraction until a concrete solution is achieved. Before describing the general methodology, we first illustrate the idea with the graph coloring example.

Example 4.2 (ctd)**.** Consider the original graph instances shown in Figure 4.1 and a coarse abstraction shown in the left-most parts of Figure 4.2. Figure 4.2a shows an abstraction that clusters all the nodes into one abstract node to decide on a coloring. When there is only one node, then an assignment of one color, say *blue*, can be decided. However, in the original graph, not all nodes mapped to the abstract node can be colored to the same color due to the existence of edges between some nodes. A refinement would be to divide the abstract clusters into a finer-grained domain. Until an abstraction is achieved where the nodes with edges between them are distinguished, a spurious graph coloring is always possible to occur.

Figure 4.2b shows the abstraction of omitting the knowledge of the nodes in the graph except one. Then it is again easy to decide on a color for the node. However, in the original domain, there is no coloring that can match node 1 being colored to *red*, since originally the graph is uncolorable. The refinement of this abstraction would be to add back some of the nodes and the knowledge about the edges. Until an abstraction is achieved where the four nodes causing the uncolorability is distinguished, a spurious graph coloring always occurs.

We now describe the methodology illustrated in Figure 4.3.

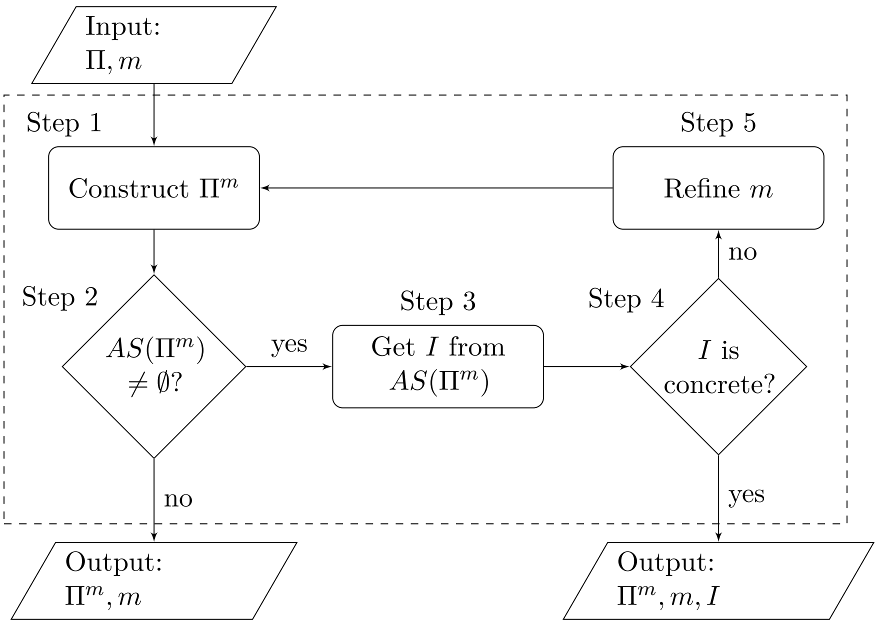
For a program Π, we start with an initial abstraction mapping m to construct an abstract program $\widehat{\Pi}_m$ (Step 1) that over-approximates the original program Π with the methods introduced in Sections 4.2-4.3 and then compute the abstract answer sets. The over-approximation condition guarantees that in case Π has a answer set I then it will have a corresponding abstract answer set $m(I)$ in the abstract program $\widehat{\Pi}_m$. However, there can also be some abstract answer sets which are spurious. If however the abstract program $\widehat{\Pi}_m$ does not have an answer set (Step 2), by Proposition 4.1 this shows that Π is unsatisfiable, then the abstract program $\widehat{\Pi}_m$ and the mapping m that achieved the unsatisfiability are returned. When we pick an abstract answer set $I' \in AS(\widehat{\Pi}_m)$ (Step 3), we check for its concreteness (Step 4). If I is concrete, then it shows a solution to Π. Thus, the abstract program $\widehat{\Pi}_m$, the mapping m and the concrete abstract answer set I are returned. If I' is spurious, then we decide to refine the abstraction mapping m to m' (Step 5) in order to construct a more fine-grained abstract program $\widehat{\Pi}_{m'}$ according to the updated mapping. We then compute the answer sets of the updated abstract program $\widehat{\Pi}_{m'}$ and continue as above. The abstraction refinement loop continues until either a picked abstract answer set is concrete, or the abstract program returns no answer sets. The procedure eventually stops as once the mapping m is refined back to the trivial mapping, i.e., all elements of the original domain are mapped to themselves, $\widehat{\Pi}_m$ will be the same as Π. Thus, if Π is unsatisfiable then the procedure will stop at Step 2, otherwise any answer set picked in Step 3 will be concrete, thus the procedure will stop at Step 4.

The abstraction methods used in Step 1 are introduced in Section 4.2 and Section 4.3 for omission abstraction and domain abstraction, respectively. The correctness checking of an abstract answer set (Step 4) and then deciding on a refinement (Step 5) is done using a debugging approach which is introduced in Section4.4.

4.2 Omission-based Abstraction

Abstraction by omission is on omitting a set of atoms from a program to obtain an over-approximation. The abstraction mapping that describes omission is as follows.

Figure 4.3: Abstraction & Refinement Methodology



Definition 4.3. *Given a set $A \subseteq \mathcal{A}$ of atoms, an* omission (abstraction) mapping *is $m_A : \mathcal{A} \to \mathcal{A} \cup \{\top\}$ such that $m_A(\alpha) = \top$ if $\alpha \in A$ and $m_A(\alpha) = \alpha$ otherwise.*

An omission mapping removes the set A of atoms from the vocabulary and keeps the rest. We refer to A as the *omitted atoms*.

Example 4.3. Consider the programs Π_1, Π_2 and Π_3 below and let the set A of atoms to be omitted to be $\{b\}$.

	Π_1	Π_2	Π_3
	$c \leftarrow not\ d.$	$c \leftarrow not\ d.$	$\{a\}.$
	$d \leftarrow not\ c.$	$d \leftarrow not\ c.$	$\{c\} \leftarrow a.$
	$a \leftarrow not\ b, c.$	$\{a\} \leftarrow c.$	$d \leftarrow not\ a.$
	$b \leftarrow d.$		
AS	$\{c,a\},\{d,b\}$	$\{c,a\},\{d\},\{c\}$	$\{c,a\},\{d\},\{a\}$

Observe that for $I'_1 = \{m_A(c), m_A(a)\} = \{c, a\}$ we have $I'_1 \in AS(\Pi_2)$ and $I'_1 \in AS(\Pi_3)$ and for $I'_2 = \{m_A(d), m_A(b)\} = \{d\}$ we have $I'_2 \in AS(\Pi_2)$ and $I'_2 \in AS(\Pi_3)$. Thus, according to Definition 4.1, both of the programs Π_2 and Π_3 are an abstraction of Π_1. Moreover, they are over-approximations, as they have answer sets $\{c\}$ and $\{a\}$, respectively, which cannot be mapped back to the answer sets of Π_1.

Although both Π_2 and Π_3 are abstractions, notice that the structure of Π_2 is more similar to Π_1, while Π_3 has an entirely different structure of rules.

Next we show a systematic way of building, given an ASP program and a set A of atoms, an abstraction of Π by omitting the atoms in A that we denote by $\mathit{omit}(\Pi, A)$. The aim is to ensure that every original answer set of Π is mapped to some abstract answer set of $\mathit{omit}(\Pi, A)$, while (unavoidably) some spurious abstract answer sets may be introduced. Thus, an over-approximation of the original program Π is achieved.

4.2.1 Program Abstraction

The basic method is to project the rules to the non-omitted atoms and introduce choice when an atom is omitted from a rule body, in order to make sure that the behavior of the original rule is preserved.

We build from Π an abstract program $\mathit{omit}(\Pi, A)$ according to the abstraction m_A. For every rule $r : \alpha \leftarrow B(r)$ in Π,

$$\mathit{omit}(r, A) = \begin{cases} r & \text{if } \; A \cap B^{\pm} = \emptyset \wedge \alpha \notin A, & (a) \\ \{\alpha\} \leftarrow B^{+}(r) \setminus A, \mathit{not}\ (B^{-}(r) \setminus A) & \text{if } \; A \cap B^{\pm} \neq \emptyset \wedge \alpha \notin A \cup \{\bot\}, & (b) \\ \emptyset & \text{otherwise.} & (c) \end{cases}$$

In (a), we keep the rule as it is, if it does not contain any omitted atom. Item (b) is for the case when the rule is not a constraint and the rule head is not in A. Then the body of the rule is projected onto the remaining atoms, and a choice is introduced to the head. Note that we treat default negated atoms, $B^{-}(r)$, similarly, i.e., if some $\alpha \in B^{-}(r) \cap A$, then we omit $\mathit{not}\ \alpha$ from $B(r)$. As for the remaining cases (either the rule head is in A or the rule is a constraint containing some atom from A), the rule is omitted by item (c). We use $\emptyset$ as a symbol for picking no rule.

We sometimes denote $\mathit{omit}(\Pi, A)$ as $\widehat{\Pi}_{\overline{A}}$, where $\overline{A} = \mathcal{A} \setminus A$, to emphasize that it is an abstract program constructed with the remaining atoms $\overline{A}$. For an interpretation I and respectively a set S of atoms, $I|_{\overline{A}}$ and $S|_{\overline{A}}$ denotes the projection to the atoms in $\overline{A}$. For a rule r, we use $m_A(B(r))$ as a shorthand for $B(\mathit{omit}(r, A))$ to emphasize that the mapping m_A is applied to each atom in the body.

Example 4.4. Consider a program Π and its abstraction $\widehat{\Pi}_{\overline{A}}$ for $A = \{b, d\}$, according to the above steps.

	Π	$\widehat{\Pi}_{\overline{A}}$
	$c \leftarrow \mathit{not}\ d.$	$\{c\}.$
	$d \leftarrow \mathit{not}\ c.$	
	$a \leftarrow \mathit{not}\ b, c.$	$\{a\} \leftarrow c.$
	$b \leftarrow d.$	
AS	$\{c, a\}, \{d, b\}$	$\{\}, \{c\}, \{c, a\}$

For $I'_1 = \{m_A(c), m_A(a)\} = \{c, a\}$ we have $I'_1 \in AS(\widehat{\Pi}_{\overline{A}})$ and for $I'_2 = \{m_A(d), m_A(b)\} = \{\}$ we have $I'_2 \in AS(\widehat{\Pi}_{\overline{A}})$. Thus, every answer set of Π can be mapped to some answer set of $\widehat{\Pi}_{\overline{A}}$, when the omitted atoms are projected away, i.e., $AS(\Pi)|_{\overline{A}} = \{\{c,a\},\{\}\} \subseteq \{\{c,a\},\{\},\{c\}\} = AS(\widehat{\Pi}_{\overline{A}})$.

Notice that in $\widehat{\Pi}_{\overline{A}}$, constraints are omitted if the body contains an omitted atom (item (c)). If instead the constraint gets shrunk by just omitting the atom from the body, then for some interpretation $\hat{I}$, the body may be satisfied, causing $\hat{I} \notin AS(\widehat{\Pi}_{\overline{A}})$, while this was not the case in Π for any $I \in AS(\Pi)$ with $I|_{\overline{A}} = \hat{I}$. Thus I cannot be mapped to an abstract answer set of $\widehat{\Pi}_{\overline{A}}$, i.e., $\widehat{\Pi}_{\overline{A}}$ is not an over-approximation of Π. The next example illustrates this.

Example 4.5 (Example 4.4 ctd)**.** Consider an additional rule $\{\leftarrow c, b.\}$ in Π, which does not change its answer sets. If however in the abstraction $\widehat{\Pi}_{\overline{A}}$ this constraint only gets shrunk to $\{\leftarrow c.\}$, by omitting b from its body, we get $AS(\widehat{\Pi}_{\overline{A}}) = \{\}$. This causes $\widehat{\Pi}_{\overline{A}}$ to have no abstract answer set to which the original answer set $\{c,a\}$ can be mapped to. Omitting the constraint from $\widehat{\Pi}_{\overline{A}}$ as described above avoids such cases of losing the original answer sets in the abstraction.

Abstracting choice rules We focused above on rules of the form $\alpha \leftarrow B$ only. However, the same principle is applicable to choice rules $r : \{\alpha\} \leftarrow B(r)$. When building $omit(r,A)$, item (a) keeps the rule as it is, item (b) removes the omitted atom from $B(r)$ and keeps the choice in the head, and item (c) omits the rule. This would be syntactically different from considering the expanded version (1) $\alpha \leftarrow B(r), not\ \overline{\alpha}$. (2) $\overline{\alpha} \leftarrow B(r), not\ \alpha$. where $\overline{\alpha}$ is an auxiliary atom. If α is omitted, the rule (2) turns into a guessing rule, but it is irrelevant as $\overline{\alpha}$ occurs nowhere else. If α is not omitted but some atom in B, both rules are turned into guessing rules and the same answer set combinations are achieved as with keeping r as a choice rule in item (b). However, the number of auxiliary atoms would increase, in contrast to treating choice rules r genuinely.

Over-Approximation

The following result shows that $omit(\Pi, A)$ can be seen as an over-approximation of Π.

Theorem 4.2. *For every answer set $I \in AS(\Pi)$ and atoms $A \subseteq \mathcal{A}$, it holds that $I|_{\overline{A}} \in AS(omit(\Pi, A))$.*

Proof. Towards a contradiction, assume I is an answer set of Π, but $I|_{\overline{A}}$ is not an answer set of $omit(\Pi, A)$. This can occur because either (i) $I|_{\overline{A}}$ is not a model of $\Pi' = omit(\Pi, A)^{I|_{\overline{A}}}$ or (ii) $I|_{\overline{A}}$ is not a minimal model of Π'.

(i) If $I|_{\overline{A}}$ is not a model of Π', then there exists some rule $r \in \Pi'$ such that $I|_{\overline{A}} \models B(r)$ and $I|_{\overline{A}} \not\models H(r)$. By the construction of $omit(\Pi, A)$, r is not obtained by case (b), i.e., by modifying some original rule to get rid of A, because then r would be a choice rule with head $H(r) = \{\alpha\}$, and r would be satisfied. Consequently, r is a rule from case (a), and thus $r \in \Pi$. We note that $I|_{\overline{A}}$ and I coincide on all atoms that occur in r. Thus,

$I|_{\overline{A}} \models B(r)$ implies that $I \models B(r)$, and as $I \models r$, it follows $I \models H(r)$, which then means $I|_{\overline{A}} \models H(r)$; this is a contradiction.

(ii) Suppose $I' \subset I|_{\overline{A}}$ is a model of Π'. We claim that then $J = I' \cup (I \cap A) \subset I$ is a model of Π^I, which would contradict that $I \in AS(\Pi)$. Assume that $J \not\models \Pi^I$. Then J does not satisfy some rule $r : \alpha \leftarrow B(r)$ in Π^I, i.e., $J \models B(r)$ but $J \not\models \alpha$, i.e., $\alpha \notin J$. The rule r can either be (a) a rule which is not changed for Π', (b) a rule that was changed to $\{\alpha\} \leftarrow \widehat{B}$ in Π', or (c) a rule that was omitted, i.e., $\alpha \in A$. In each case (a)-(c), we arrive at a contradiction:

(a) Since $r \in \Pi^I$ and r involves no atom in A, we have $r \in \Pi'$. As $I|_{\overline{A}} \models r$ and $J|_{\overline{A}}$ coincides with $I'|_{\overline{A}}$, we have that $J|_{\overline{A}} \models r$, and thus $J \models r$; this contradicts $J \not\models \alpha$.

(b) By definition of J, we have $\alpha \in I|_{\overline{A}} \setminus I'$. Since $J \models B(r)$, it follows that $J|_{\overline{A}} \models \widehat{B}$ and since $I' = J|_{\overline{A}}$ that $I' \models \widehat{B}$. As I' is a model of Π', we have that I' satisfies the choice atom $\{\alpha\}$ in the head of the rewritten rule, i.e., either (1) $\alpha \in I'$ or (2) $\alpha \notin I'$; but (1) contradicts $\alpha \in I|_{\overline{A}} \setminus I'$, while (2) means that I' is not a smaller model of Π' than $I|_{\overline{A}}$, as then $\alpha' \in I' \setminus I|_{\overline{A}}$ would hold, which is again a contradiction.

(c) As r is in Π^I, we have $I \models B(r)$ and since I is an answer set of Π, that $I \models \alpha$. As $\alpha \notin J$, by construction of J it follows that $\alpha \notin I$, which contradicts $I \models \alpha$.

□

By introducing choice rules for any rule that contains an omitted atom, all possible cases that would be achieved by having the omitted atom in the rule are covered. Thus, the abstract answer sets cover the original answer sets. On the other hand, not every abstract answer set may cover some original answer set, which motivates the following notion.

Definition 4.4 (cf. Definition 4.2)**.** *Given a program* Π *and a set* A *of atoms, an answer set* $\hat{I}$ *of* $omit(\Pi, A)$ *is* concrete*, if* $\hat{I} \in AS(\Pi)|_{\overline{A}}$ *holds, and* spurious *otherwise.*

In other words, a spurious abstract answer set $\hat{I}$ cannot be completed to any original answer set, i.e., no extension $I = \hat{I} \cup X$ of $\hat{I}$ to all atoms (where $X \subseteq A$) is an answer set of Π.

Example 4.6 (Example 4.4 ctd)**.** The program $\widehat{\Pi}_{\overline{A}}$ constructed for $\overline{A} = \{a, c\}$ has the answer sets $AS(\widehat{\Pi}_{\overline{A}}) = \{\{\}, \{c\}, \{c, a\}\}$. The abstract answer sets $\hat{I}_1 = \{\}$ and $\hat{I}_2 = \{c, a\}$ are concrete since they can be extended to the answers sets $I_1 = \{d, b\}$ and $I_2 = \{c, a\}$ of Π, as $I_1|_{\overline{A}} = \hat{I}_1$ and $I_2|_{\overline{A}} = \hat{I}_2$, respectively. On the other hand, the abstract answer set $\hat{I} = \{c\}$ is spurious: as a is false in $\hat{I}$, it must be false in Π, but c being true in turn affects that b and d must be false in Π as well; this violates rule $a \leftarrow \mathit{not}\ b, c.$ in Π.

In Section 4.4, we show an alternative way of checking the spuriousness of an abstract answer set, which we then use in determining a refinement of the abstraction mapping.

Refining abstractions

Upon encountering a spurious answer set, one can either continue checking other abstract answer sets until a concrete one is found, or *refine* the abstraction in order to reach an abstract program with less spurious answer sets. Formally, refinements are defined as follows.

Definition 4.5. *Given a omission mapping* $m_A = \mathcal{A} \rightarrow \mathcal{A} \cup \{\top\}$*, a mapping* $m_{A'} = \mathcal{A} \rightarrow \mathcal{A} \cup \{\top\}$ *is a* refinement *of* m_A *if* $A' \subseteq A$*.*

Intuitively, a refinement is made by adding some of the omitted atoms back.

Example 4.7 (Example 4.4 ctd)**.** A mapping that omits the set $A' = \{b\}$ is a refinement of the mapping that omits $A = \{b, d\}$, as d is added back. This affects that in the abstraction program the choice rule $\{c\}$. is turned back to $c \leftarrow \mathit{not}\ d$. and the rule $d \leftarrow \mathit{not}\ c$. is undeleted, i.e., $\mathit{omit}(\Pi, A') = \{c \leftarrow \mathit{not}\ d.; d \leftarrow \mathit{not}\ c.; \{a\} \leftarrow c\}$, which has the abstract answer sets $\hat{J}_1 = \{d\}$, $\hat{J}_2 = \{c, a\}$ and $\hat{J}_3 = \{c\}$. Note that while $\hat{J}_1$ and $\hat{J}_2$ are concrete, $\hat{J}_3$ is spurious; intuitively, adding d back does not eliminate the spurious answer set $\{c\}$ of $\mathit{omit}(\Pi, A)$.

The previous example motivates us to introduce a notion for sets of omitted atoms that need to be added back in order to get rid of a spurious answer set.

Definition 4.6 (Put-back set)**.** *Let* $\hat{I} \in AS(\mathit{omit}(\Pi, A))$ *be any spurious abstract answer set of a program* Π *for omitted atoms* A*. A* put-back set *for* $\hat{I}$ *is any set* $PB \subseteq A$ *of atoms such that no abstract answer set* $\hat{J}$ *of* $\mathit{omit}(\Pi, A')$ *where* $A' = A \setminus PB$ *exists with* $\hat{J}|_{\overline{A}} = \hat{I}$*.*

That is, re-introducing the put-back atoms in the abstraction, the spurious answer set $\hat{I}$ is *eliminated* in the modified abstract program. Notice that multiple put-back sets (even incomparable ones) are possible, and the existence of some put-back set is guaranteed, as putting all atoms back, i.e., setting $PB = A$, eliminates the spurious answer set.

Example 4.8 (Example 4.4 ctd)**.** The discussion in Example 4.7 shows that $\{d\}$ is not a put-back set for the spurious answer set $\hat{I} = \{c\} \in \widehat{\Pi}_{\overline{A}}$, and neither $\{b\}$ is a put-back set: the abstract program for $A' = A \setminus \{b\} = \{d\}$ is $\mathit{omit}(\Pi, A') = \{\{c\}.; a \leftarrow \mathit{not}\, b, c.; \{b\}.\}$, which has $\{b, c\}$ with $\{b, c\}_{\overline{A}} = \hat{I}$ among its abstract answer sets. Thus, $\hat{I}$ has only the trivial put-back set $\{b, d\}$.

In practice, small put-back sets are intuitively preferable to large ones as they keep higher abstraction; we consider such preference in [SE18b].

4.2.2 Properties of Omission Abstraction

We now consider some basic but useful semantic properties of our formulation of program abstraction. Notably, it amounts to the original program in the extremal case and reflects the inconsistency of it in properties of spurious answer sets.

Proposition 4.3. *For any program* Π*,*

(i) $omit(\Pi, \emptyset) = \Pi$ *and* $omit(\Pi, \mathcal{A}) = \emptyset$*.*

(ii) $AS(\Pi) = \emptyset$ *iff* $I = \{\}$ *is spurious w.r.t.* $A = \mathcal{A}$*.*

(iii) $AS(omit(\Pi, A)) = \emptyset$ *implies* $AS(\Pi) = \emptyset$*.*

(iv) $AS(\Pi) = \emptyset$ *iff some* $A \subseteq \mathcal{A}$ *has only spurious answer sets iff every* $omit(\Pi, A)$*,* $A \subseteq \mathcal{A}$*, has only spurious answer sets.*

Proof. (i) Omitting the set $\emptyset$ from Π causes no change in the rules, while omitting the set $\mathcal{A}$ causes all the rules to be omitted.

(ii) Since $\hat{I} = \{\}$ and $A = \mathcal{A}$, we have $Q_{\hat{I}}^{\overline{A}} = \{\}$. Thus, by the alternative definition, $I = \{\}$ is spurious w.r.t. $A = \mathcal{A}$ iff $AS(\Pi \cup Q_{\hat{I}}^{\overline{A}}) = \emptyset$ iff $AS(\Pi) = \emptyset$.

(iii) Corollary of Theorem 4.2.

(iv) If $AS(\Pi) = \emptyset$, then no $\hat{I} \in AS(omit(\Pi, A))$ for any $A \subseteq \mathcal{A}$ can be extended to an answer set of Π; thus, all abstract answer sets of $omit(\Pi, A)$ are spurious. This in turn trivially implies that $omit(\Pi, A)$ has for some $A \subseteq \mathcal{A}$ only spurious answer sets. Finally, assume the latter holds but $AS(\Pi) \neq \emptyset$; then Π has some answer set I, and by Theorem 4.2, $I|_{\overline{A}} \in AS(omit(\Pi, A))$, which would contradict that $omit(\Pi, A)$ has only spurious answer sets.

□

The abstract program is built by a syntactic transformation, given the set A of atoms to be omitted. It turns out that we can omit the atoms sequentially, and the order does not matter.

Lemma 4.4. *For any program* Π *and atoms* $a_1, a_2 \in \mathcal{A}$*,*

$$omit(omit(\Pi, \{a_1\}), \{a_2\}) = omit(omit(\Pi, \{a_2\}), \{a_1\}).$$

Proof. The rules of Π that do not contain a_1 or a_2 remain unchanged, and the rules that contain one of a_1 or a_2 will be updated at the respective abstraction steps. The rules that contain both a_1 and a_2 are treated as follows:

- Consider a rule $a_1 \leftarrow B$ with $a_2 \in B$ (w.l.o.g.). Omitting first a_2 from the rule causes to have $\{a_1\} \leftarrow B \setminus \{a_2\}$, and omitting then a_1 results in omission of the rule. Omitting first a_1 from the rule causes the omission of the rule at the first abstraction step.

- Consider a rule $\alpha \leftarrow B$, with $a_1, a_2 \in B$ and $\alpha \neq a_1, a_2$. Omitting first a_2 from the rule causes to have $\{a\} \leftarrow B \setminus \{a_2\}$, and omitting then a_1 causes to have $\{a\} \leftarrow B \setminus \{a_1, a_2\}$. The same rule is obtained when omitting first a_1 and then a_2. $\square$

An easy induction argument shows then the property mentioned above.

Proposition 4.5. *For any program* Π *and set* $A = \{a_1, \ldots, a_n\}$ *of atoms,*

$$omit(\Pi, A) = omit(omit(\cdots(omit(\Pi, \{a_{\pi(1)}\}), \cdots \{a_{\pi(n-1)}\}), \{a_{\pi(n)}\})$$

where π *is any permutation of* $\{1, \ldots, n\}$.

Thus, the abstraction can be done one atom at a time.

Omitting atoms in a program means projecting them away from the answer sets. Thus, for a mapping m_A, the concrete answer sets in $omit(\Pi, A)$ always have corresponding answer sets in the programs computed for refinements of m_A.

Proposition 4.6. *Suppose* $\hat{I}$ *is a concrete answer set of* $omit(\Pi, A)$ *for a program* Π *and a set* A *of atoms. Then for every* $A' \subseteq A$ *some answer set* $\hat{I}' \in AS(omit(\Pi, A'))$ *exists such that* $\hat{I}'|_{\overline{A}} = \hat{I}$.

Proof. By Definition 4.4, $\hat{I} \in AS(\Pi)|_{\overline{A}}$, i.e. there exists some $I \in AS(\Pi)$ s.t. $I|_{\overline{A}} = \hat{I}$. By Theorem 4.2, for every $B \subseteq \mathcal{A}$, $I|_{\overline{B}} \in AS(omit(\Pi, B))$ holds, and in particular for $B \subseteq A$; we thus obtain $(I|_{\overline{B}})|_{\overline{A}} = I|_{\overline{A}} = \hat{I}$. $\square$

The next property is convexity of spurious answer sets.

Proposition 4.7. *Suppose* $\hat{I} \in AS(omit(\Pi, A))$ *is spurious and that* $omit(\Pi, A')$, *where* $A' \subseteq A$, *has some answer set* $\hat{I}'$ *such that* $\hat{I}'|_{\overline{A}} = \hat{I}$. *Then for every* A'' *such that* $A' \subseteq A'' \subseteq A$, *it holds that* $\hat{I}'|_{\overline{A''}} \in AS(omit(\Pi, A''))$ *and* $\hat{I}'|_{\overline{A''}}$ *spurious.*

Proof. We first note that $\hat{I}'$ is spurious as well: if not, some $I \in AS(\Pi)$ exists such that $I|_{\overline{A'}} = \hat{I}'$; but then $I|_{\overline{A}} = (I|_{\overline{A'}})|_{\overline{A}} = \hat{I}'|_{\overline{A}} = \hat{I}$, which contradicts that $\hat{I}$ is spurious. Applying Theorem 4.2 to $omit(\Pi, A')$ and A'', we obtain that $\widehat{I'}|_{\overline{A''}}$ is an answer set of $omit(omit(\Pi, A'), A'')$, which by Proposition 4.5 coincides with $omit(\Pi, A'')$. Moreover, $\hat{I}'|_{\overline{A''}}$ is spurious, since otherwise $\hat{I}$ would not be spurious either, which would be a contradiction. $\square$

The next proposition intuitively shows that once a spurious answer set is eliminated by adding back some of the omitted atoms, no extension of this answer set will show up when further omitted atoms are added back.

Proposition 4.8. *Suppose that $\hat{I} \in AS(omit(\Pi, A))$ is a spurious answer set and $PB \subseteq A$ is a put-back set for $\hat{I}$. Then for every $A' \subseteq A \setminus PB$ and answer set $\hat{I}' \in AS(omit(\Pi, A \setminus (PB \cup A'))$ it holds that $\hat{I}'|_{\overline{A}} \neq \hat{I}$.*

Proof. Towards a contradiction, assume that for some $A' \subseteq A \setminus PB$ and answer set $\hat{I}' \in AS(omit(\Pi, A \setminus (PB \cup A'))$ it holds that $\hat{I}'|_{\overline{A}} = \hat{I}$. By Proposition 4.7, we obtain that $\hat{I}'$ is spurious and moreover that $\hat{I}'|_{\overline{A \setminus PB}} \in AS(omit(\Pi, A \setminus PB))$ is spurious. However, as $(\hat{I}'|_{\overline{A \setminus PB}})|_{\overline{A}} = \hat{I}|_{\overline{A}}$, this contradicts that PB is a put-back set for $\hat{I}$. ☐

Faithful Abstractions

Ideally, abstraction simplifies a program but does not change its semantics. Our next notion serves to describe such abstractions.

Definition 4.7. *An abstraction $omit(\Pi, A)$ is* faithful*, if it has no spurious answer sets.*

Faithful abstractions are a syntactic representation of projected answer sets, since we obtain $AS(omit(\Pi, A)) = AS(\Pi)|_{\overline{A}}$. They fully preserve the information contained in the answer sets, and allow for reasoning (both brave and cautious) that is sound and complete over the projected answer sets.

Example 4.9 (Example 4.4 ctd)**.** Consider omitting the set $A = \{a, c\}$ from Π. The resulting $\widehat{\Pi}_{\overline{A}}$ is faithful, since its answer sets $\{\{\}, \{b, d\}\}$ are the ones obtained from projecting $\{a, c\}$ away from $AS(\Pi)$.

	Π	$\widehat{\Pi}_{\overline{A}}$
	$c \leftarrow not\ d.$	
	$d \leftarrow not\ c.$	$\{d\}.$
	$a \leftarrow not\ b, c.$	
	$b \leftarrow d.$	$b \leftarrow d.$
AS	$\{c, a\}, \{d, b\}$	$\{\}, \{d, b\}$

However, while an abstraction may be faithful, by adding back omitted atoms the faithfulness might get lost. In particular, if the program Π is satisfiable, then $A = \mathcal{A}$ is a faithful abstraction; by adding back atoms, spurious answer sets might arise. This motivates the following notion.

Definition 4.8. *A faithful abstraction $omit(\Pi, A)$ of a program Π w.r.t a set A of atoms is* refinement-safe*, if for all $A' \subseteq A$, the abstract program $omit(\Pi, A')$ has no spurious answer sets.*

In a sense, a refinement-safe abstraction allows us to zoom in details without losing already established relationships between atoms, as they appear in the abstract answer sets, and no spuriousness check is needed. In particular, this applies to programs that are unsatisfiable. By Proposition 4.3-(iii), unsatisfiability of an abstraction $omit(\Pi, A)$ implies that the original program is unsatisfiable, and hence the abstraction is faithful. Moreover, we obtain:

Proposition 4.9. *Given a program* Π *and a set* A *of atoms, if* $omit(\Pi, A)$ *is unsatisfiable, then it is refinement-safe faithful.*

Proof. Assume that A is refined to some $A' \subset A$, where some atoms are added back in the abstraction, and the constructed $omit(\Pi, A')$ is not unsatisfiable, i.e., $AS(omit(\Pi, A')) \neq \emptyset$. By Theorem 4.2, it must hold that $AS(omit(\Pi, A'))|_{\overline{A}} \subseteq AS(omit(\Pi, A))$, which contradicts to the fact that $omit(\Pi, A)$ is unsatisfiable. □

4.2.3 Computational Complexity

In this section, we turn to the computational complexity of reasoning tasks that are associated with program abstraction. We start with noting that constructing the abstract program and model checking on it is tractable.

Lemma 4.10. *Given* Π *and* A*, (i) the program* $omit(\Pi, A)$ *is constructible in logarithmic space, and (ii) checking whether* $I \in AS(omit(\Pi, A))$ *holds for a given* I *is feasible in polynomial time.*

As for item (i), the abstract program $omit(\Pi, A)$ is easily constructed in a linear scan of the rules in Π; item (ii) reduces then to answer set checking of an ordinary normal logic program, which is well-known to be feasible in polynomial time (and in fact P-complete).

However, tractability of abstract answer set checking is lost if we ask in addition for concreteness or spuriousness.

Proposition 4.11. *Given a program* Π*, a set* A *of atoms, and an interpretation* I*, deciding whether* $I|_{\overline{A}}$*, is a concrete (resp., spurious) abstract answer set of* $omit(\Pi, A)$ *is* NP*-complete (resp. co*NP*-complete).*

Proof. Indeed, we can guess an interpretation J of Π such that (a) $J_{\overline{A}} = I_{\overline{A}}$, (b) $J_{\overline{A}} \in AS(omit(\Pi, A))$, and (c) $J \in AS(\Pi)$. By Lemma 4.10, (b) and (c) are feasible in polynomial time, and thus deciding whether $I_{\overline{A}}$ is a concrete abstract answer set is in NP. Similarly, $I_{\overline{A}}$ is not a spurious abstract answer set iff for some J condition (a) holds and either (b) fails or (c) holds; this implies coNP membership.

The NP-hardness (resp. coNP-hardness) is immediate from Proposition 4.3 and the NP-completeness of deciding answer set existence. □

Thus, determining whether a particular abstract answer set causes a loss of information is intractable in general. If we do not have a candidate answer set at hand, but want to know whether the abstraction causes a loss of information with respect to all answer sets of the original program, then the complexity increases.

Theorem 4.12. *Given a program* Π *and a set* A *of atoms, deciding whether some answer set* $\hat{I} \in AS(omit(\Pi, A))$ *exists that is spurious is* Σ_2^p*-complete.*

Proof. As for membership in Σ_2^p, some answer set $\hat{I} \in omit(\Pi, A)$ that is spurious can be guessed and checked by Proposition 4.11 with a coNP oracle in polynomial time. The Σ_2^p-hardness is shown by a reduction from evaluating a QBF $\exists X \forall Y\, E(X, Y)$, where $E(X, Y) = \bigvee_{i=1}^{k} D_i$ is a DNF of conjunctions $D_i = l_{i_1} \wedge \cdots \wedge l_{i_{n_i}}$ over atoms $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ where without loss of generality in each D_i some atom from Y occurs.

We construct a program Π as follows;

$$x_i \leftarrow not\, \overline{x_i}. \tag{4.2}$$
$$\overline{x_i} \leftarrow not\, x_i. \quad \text{for all } x_i \in X \tag{4.3}$$
$$y_j \leftarrow not\, \overline{y_j}, not\, sat. \tag{4.4}$$
$$\overline{y_j} \leftarrow not\, y_j, not\, sat. \quad \text{for all } y_j \in Y \tag{4.5}$$
$$sat \leftarrow l_{i_1}^*, \ldots l_{i_{n_i}}^*. \tag{4.6}$$

where $\overline{X} = \{\overline{x}_1, \ldots \overline{x_n}\}$ and $\overline{Y} = \{\overline{y}_1, \ldots \overline{y_m}\}$ are sets of fresh atoms and for each atom $a \in X \cup Y$, we let $a* = a$ and $(\neg a)^* = \overline{a}$. Furthermore, we set $A = Y \cup \overline{Y} \cup \{sat\}$.

Intuitively, the answer sets $\hat{I}$ of $omit(\Pi, A)$, which consists of all rules (4.2)-(4.3), correspond 1-1 to the truth assignments σ of X. A particular such $\hat{I} = \hat{I}_\sigma = \{x_i \in X \mid \sigma(x_i) = true\} \cup \{\overline{x_i} \mid x_i \in X, \sigma(x_i) = false\}$ is spurious, iff it can not be extended after putting back all omitted atoms to an answer set J of Π. Any such J must not include sat, as otherwise the rules (4.4) and (4.5) would not be applicable w.r.t. J, which means that all y_j and $\overline{Y_j}$ would be false in J; but then sat could not be derived from Π and J, as no rule (4.6) is applicable w.r.t. J by the assumption on the D_i.

Now if $\hat{I}_\sigma$ is not spurious, then some answer set J of Π as described exists. As $sat \notin J$, the rules (4.4) and (4.5) imply that exactly one of y_j and $\overline{y}_j$ is in J, for each y_j, and thus J induces an assignment μ to Y. As no rule (4.6) is applicable w.r.t. J, it follows that $E(\sigma(X), \mu(Y))$ evaluates to false, and thus $\forall Y E(\sigma(X), Y)$ does not evaluate to true. Conversely, if $\forall Y E(\sigma(X), Y)$ does not evaluate to true, then some answer set J of Π that coincides with $\hat{I}$_σ on $X \cup \overline{X}$ exists, and hence $\hat{I}$_σ is not spurious. In conclusion, it follows that $omit(\Pi, A)$ has some spurious answer set iff $\exists X \forall Y E(X, Y)$ evaluates to true. $\square$

An immediate consequence of the previous theorem is that checking whether an abstraction $omit(\Pi, A)$ is faithful has complementary complexity.

Corollary 4.13. *Given a program* Π *and a set* $A \subseteq \mathcal{A}$ *of atoms, deciding whether* $omit(\Pi, A)$ *is faithful is* Π_2^p*-complete.*

Further complexity results are presented in Appendix B. Computing a minimal put-back set and some refinement-safe faithful abstraction, that does not remove a given set A_0 of atoms, each shown to be in $\mathbf{FP}^{\Sigma_2^P}[\mathit{log}, \mathit{wit}]$-complete. Here, the class $\mathbf{FP}^{\Sigma_k^P}[\mathit{log}, \mathit{wit}]$, for $k \geq 1$, contains all search problems that can be solved in polynomial time with a witness oracle for Σ_k^p [BKT93]; a *witness* oracle for Σ_k^p returns in case of a yes-answer to an instance a polynomial size witness string that can be checked with a Σ_{k-1}^p oracle in polynomial time.

4.2.4 Extensions

In this section, we discuss possible extensions of the approach to more expressive programs, in particular to non-ground programs and to disjunctive logic programs.

Note that lifting the framework to programs with strong negation is easily possible, where as usual negative literals $\neg p(\vec{t})$ are viewed as atoms of a positive predicate $\neg p$ and with an additional constraint $\leftarrow p(\vec{t}), \neg p(\vec{t})$.

Non-Ground Case

In case of omitting atoms from non-ground programs, a simple extension of the method described above is to remove all non-ground atoms from the program that involve a predicate p that should be omitted. This, however, may require to introduce domain variables in order to avoid the derivation of spurious atoms. Specifically, if in a rule $r : \alpha \leftarrow B(r)$ a non-ground atom $p(V_1, \ldots, V_n)$ that is omitted from the body shares some arguments, V_i, with the head α, then α is conditioned for V_i with a domain atom $\mathit{dom}(V_i)$ in the constructed rule, so that all values of V_i are considered.

Example 4.10. Consider the following program Π with domain predicate *int* for an integer domain $\{1, ..., 5\}$:

$$a(X_1, X_2) \leftarrow c(X_1), b(X_2). \tag{4.7}$$
$$d(X_1, X_2) \leftarrow a(X_1, X_2), X_1 \leq X_2. \tag{4.8}$$

In omitting $c(X)$, while rule (4.8) remains the same, rule (4.7) changes to

$$\{a(X_1, X_2) : \mathit{int}(X_1)\} \leftarrow b(X_2).$$

From Π and the facts $c(1), b(2)$, we get the answer set $\{c(1), b(2), a(1,2), d(1,2)\}$, and with $c(2), b(2)$ we get $\{c(2), b(2), a(2,2), d(2,2)\}$. After omitting $c(X)$, the abstract program with fact $b(2)$ has 32 answer sets. Among them are $\{b(2), a(1,2), d(1,2)\}$ and $\{b(2), a(2,2), d(2,2)\}$, which cover the original answer sets, i.e., each original answer set can be mapped to some abstract one.

For a more fine-grained omission, let the set A consist of the atoms $\alpha = p(c_1, \ldots, c_k)$ and let $A_p \subseteq A$ denote the set of ground atoms with predicate p that we want to omit. Consider a k-ary predicate θ_p such that for any $c_1, \ldots, c_k$, we have $\theta_p(c_1, \ldots, c_k) = \mathit{true}$ iff

$p(c_1,\dots,c_k) \in A_p$; for a (possibly non-ground) atom $\alpha = p(t_1,\dots,t_k)$, we write $\theta(\alpha)$ for $\theta_p(t_1,\dots,t_k)$. We can then build from a non-ground program Π an abstract non-ground program $omit(\Pi, A)$ according to the abstraction m_A, by mapping every rule $r : \alpha \leftarrow B$ in Π to a set $omit(r, A)$ of rules such that

$$omit(r,A) \text{ includes } \begin{cases} r & \text{if } \; A_{pred(\beta)} = \emptyset \text{ for all } \beta \in \{\alpha\} \cup B^{\pm}, \\ \alpha \leftarrow B, not\ \theta(\beta) & \text{if } \; A_{pred(\beta)} \neq \emptyset \ \wedge \ \beta \in \{\alpha\} \cup B^{\pm}, \\ \{\alpha\} \leftarrow B \setminus \{\beta\}, \theta(\beta) & \text{if } \; \beta \in B \ \wedge \ \alpha \neq \bot \ \wedge \ \theta(\beta) \text{ is satisfiable}, \\ \emptyset & \quad \text{otherwise}, \end{cases}$$

and no other rules. The steps above assume that in a rule a most one predicate to omit occurs in a single atom β. However, the steps can be readily lifted to consider omitting a set $\{\beta_1,\dots,\beta_n\}$ of atoms with multiple predicates from the rules. For this, $\alpha \leftarrow B, not\ \theta(\beta)$ will be converted into $\alpha \leftarrow B, not\ \theta(\beta_1), \dots, not\ \theta(\beta_n)$ and $\{\alpha\} \leftarrow B \setminus \{\beta\}, \theta(\beta)$ gets converted into a set of rules $\{\alpha\} \leftarrow B \setminus \{\beta_1,\dots,\beta_n\}, \theta(\beta_1); \dots; \{\alpha\} \leftarrow B \setminus \{\beta_1,\dots,\beta_n\}, \theta(\beta_n)$.

Example 4.11 (Example 4.10 ctd)**.** Suppose we want to omit $c(X)$ for $X{<}3$, i.e., $A = \{c(1), c(2)\} = A_c$. We have $\theta(c(1)) = \theta(c(2)) = true$ and $\theta(c(X)) = false$, for $X \in \{3, ..., 5\}$. The abstract non-ground program $omit(\Pi, A)$ is

$$\begin{aligned} a(X_1,X_2) &\leftarrow c(X_1), b(X_2), not\ \theta(c(X_1)). \\ \{a(X_1,X_2)\} &\leftarrow b(X_2), \theta(c(X_1)). \\ d(X_1,X_2) &\leftarrow a(X_1,X_2), X_1{\le}X_2. \end{aligned}$$

The abstract answer sets with facts $b(2), \theta(c(1)), \theta(c(2))$ are $\{\{b(2)\}, \{b(2), a(2,2), d(2,2)\}, \{b(2), a(1,2), d(1,2)\}$, and $\{b(2), a(1,2), a(2,2), d(1,2), d(2,2)\}\}$. The program $omit(\Pi, A)$ is over-approximating Π while not introducing that many abstract answer sets as in the coarser abstraction in Example 4.10.

Disjunctive Programs

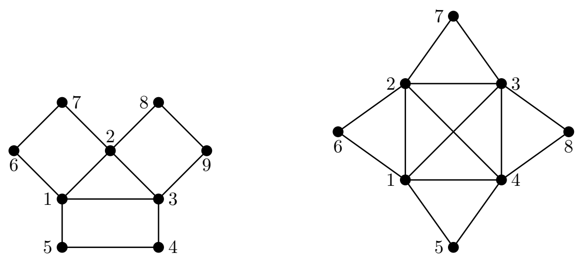
For disjunctive programs, splitting the disjunctive rules yields an over-approximation.

Proposition 4.14. *For a program Π' constructed from a given Π by splitting rules of form $\alpha_{0_1} \vee \dots \vee \alpha_{0_k} \leftarrow B(r)$ into $\{\alpha_{0_1}\} \leftarrow B(r); \dots; \{\alpha_{0_k}\} \leftarrow B(r)$, we have $AS(\Pi) \subseteq AS(\Pi')$.*

The current abstraction method can then be applied over Π'. However, it is possible that for an unsatisfiable Π the constructed Π' becomes satisfiable; the reason for unsatisfiability of Π can then not be grasped.

The approach from above can be extended to disjunctive programs Π, by injecting auxiliary atoms to disjunctive heads in order to cover the case where the body does not fire in the original program. To obtain with a given set A of atoms an abstract disjunctive

Figure 4.4: Graph coloring instances



(a) A non 2-colorable graph (b) A non 3-colorable graph

program $omit(\Pi, A)$, we define abstraction of disjunctive rules $r : \alpha_1 \vee \cdots \vee \alpha_n \leftarrow B$ in Π, where $n \geq 2$ and all $\alpha_i \neq \bot$ are pairwise distinct, as follows.

$$omit(r, A) = \begin{cases} r & \text{if } A \cap B^{\pm} = \emptyset \ \wedge \ A \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset, \\ \alpha_1 \vee \cdots \vee \alpha_k \vee x \leftarrow m_A(B) & \text{if } A \cap \{\alpha_1, \ldots, \alpha_n\} = \{\alpha_{k+1}, \ldots, \alpha_n\} \ \wedge \\ & \quad k \geq 1, \\ \alpha_1 \vee \cdots \vee \alpha_n \vee x \leftarrow m_A(B) & \text{if } A \cap B^{\pm} \neq \emptyset \ \wedge \ A \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

where x is a fresh auxiliary atom. Further development of the approach for disjunctive programs in a syntax preserving manner remains as future work.

4.2.5 Satisfiability Blockers of Programs

An interesting application area for an omission-based abstraction in ASP is finding an *explanation* for unsatisfiability of programs. We approach the unsatisfiability of an ASP program with the aim to obtain a projection of the program that shows the cause of the unsatisfiability. For example, consider the graphs shown in Figure 4.4. The one in Figure 4.4a is not 2-colorable due to the subgraph induced by the nodes 1-2-3, while the one in Figure 4.4b is not 3-colorable due to the subgraph of the nodes 1-2-3-4. From the original programs that encode this problem, abstracting away the rules that assigns colors to the nodes not involved in these subgraphs should still keep the unsatisfiability, thus showing the actual reason of non-colorability of the graphs.

To obtain an explanation of unsatisfiability, we introduce the notion of *blocker sets* as sets of atoms such that abstraction to them preserves unsatisfiability of a program. After describing the implementation, we report about our experiments where the aim was to observe the use of abstraction and refinement for achieving an over-approximation of a program that is still unsatisfiable and to compute the $\subseteq$-minimal blockers of the programs, which projects away the part that is unnecessary for the unsatisfiability.

Blocker Sets If a program Π has no answer sets, we can obtain by omitting sufficiently many atoms from it an abstract program that has some abstract answer set. By

Proposition 4.3-(iv), any such answer set will be spurious. On the other hand, as long as the abstracted program has no answer sets, by Proposition 4.3-(iii) also the original program Π has no answer set. This motivates us to use omission abstraction in order to catch a "real" cause of inconsistency in a program. To this end, we introduce the following notion.

Definition 4.9. *A set $C \subseteq \mathcal{A}$ of atoms is an* (answer set) blocker set *of Π, if $AS(omit(\Pi, \mathcal{A} \setminus C)) = \emptyset$.*

In other words, when we keep the set C of atoms and omit the rest from Π to obtain the abstract program Π', then the latter is still unsatisfiable. This means that the atoms in C are *blocking* the occurrence of answer sets: no answer set is possible as long as all these atoms are present in the program, regardless of how the omitted atoms will be evaluated in building an answer set.

Example 4.12 (Example 4.4 ctd)**.** Modify Π by changing the last rule to $b \leftarrow not\ b.$, in order to have a program Π' which is unsatisfiable. Omitting the set $A = \{d\}$ from Π' creates the abstract program $\widehat{\Pi}'_{\overline{\{d\}}}$ which is still unsatisfiable. Thus, the set $C = \mathcal{A} \setminus A = \{a, b, c\}$ is a blocker set of Π'. This is similar for omitting the set $A = \{a, c\}$, which then causes to have $C = \{d, b\}$ as a blocker set of Π'.

Π'	$\widehat{\Pi}'_{\overline{\{d\}}}$	$\widehat{\Pi}'_{\overline{\{a,c\}}}$
$c \leftarrow not\ d.$	$\{c\}.$	
$d \leftarrow not\ c.$		$\{d\}$
$a \leftarrow not\ b, c.$	$a \leftarrow not\ b, c.$	
$b \leftarrow not\ b.$	$b \leftarrow not\ b.$	$b \leftarrow not\ b.$
unsatisfiable	unsatisfiable	unsatisfiable

Notice that $C = \mathcal{A}$, i.e., no atom is omitted, is trivially a blocker set if Π is unsatisfiable, while $C = \emptyset$, i.e., all atoms are omitted, is never a blocker set since $AS(omit(\Pi, \mathcal{A})) = \{\emptyset\}$.

We can view a blocker set as an *explanation* of unsatisfiability; by applying Occam's razor, simpler explanations are preferred, which in pure logical terms motivates the following notion.

Definition 4.10. *A blocker set $C \subseteq \mathcal{A}$ is* $\subset$-minimal, *if for all $C' \subset C$, $AS(omit(\Pi, \mathcal{A} \setminus C')) \neq \emptyset$.*

By Proposition 4.9, in order to test whether a blocker set C is minimal, we only need to check whether for no $C' = C \setminus \{c\}$, for $c \in C$, the abstraction $omit(\Pi, \mathcal{A} \setminus C')$ has an answer set. That is, for a minimal blocker set C, we have that $\mathcal{A} \setminus C$ is a *maximal unsatisfiable abstraction*, i.e., a maximal set of atoms that can be omitted while keeping the unsatisfiability of Π.

Figure 4.5: Blocker rule set for 2-colorability of Figure 4.4a

$$\begin{array}{ll}
\{chosenColor(1, red)\}. & \bot \leftarrow not\ colored(1).\\
\{chosenColor(2, red)\}. & \bot \leftarrow not\ colored(2).\\
\{chosenColor(3, red)\}. & \bot \leftarrow not\ colored(3).\\
\{chosenColor(1, green)\}. & \bot \leftarrow chosenColor(1, red), chosenColor(1, green).\\
\{chosenColor(2, green)\}. & \bot \leftarrow chosenColor(2, red), chosenColor(2, green).\\
\{chosenColor(3, green)\}. & \bot \leftarrow chosenColor(3, red), chosenColor(3, green).\\
colored(1) \leftarrow chosenColor(1, red). & \bot \leftarrow chosenColor(2, red), chosenColor(1, red).\\
colored(2) \leftarrow chosenColor(2, red). & \bot \leftarrow chosenColor(3, red), chosenColor(1, red).\\
colored(3) \leftarrow chosenColor(3, red). & \bot \leftarrow chosenColor(3, red), chosenColor(2, red).\\
colored(1) \leftarrow chosenColor(1, green). & \bot \leftarrow chosenColor(2, green), chosenColor(1, green).\\
colored(2) \leftarrow chosenColor(2, green). & \bot \leftarrow chosenColor(3, green), chosenColor(1, green).\\
colored(3) \leftarrow chosenColor(3, green). & \bot \leftarrow chosenColor(3, green), chosenColor(2, green).
\end{array}$$

Example 4.13 (Example 4.12 ctd)**.** The program Π' has the single minimal blocker set $C = \{b\}$. Indeed, the rule $b \leftarrow not\ b$ does not admit an answer set. Thus, every blocker set must contain b, and C is the smallest such set.

We remark that the atoms occurring in the blocker sets are intuitively the ones responsible for the unsatisfiability of the program. In order to observe the reason of unsatisfiability, one has to look at the remaining abstract program. For this, we consider the notion of *blocker rule set* associated with a blocker set C, which are the rules that remain in $omit(\Pi, \mathcal{A} \setminus C)$. For example, the programs $\Pi', \widehat{\Pi}'_{\overline{\{d\}}}$ and $\widehat{\Pi}'_{\overline{\{a,c\}}}$ in Example 4.12 contain the blocker rule sets associated with $\{a, b, c, d\}, \{a, b, c\}$ and $\{b, d\}$, respectively. Here, the abstract programs contain choice rules due to the omission in the body, and the unsatisfiability of the programs shows that the evaluation of the respective rule does not make a difference for unsatisfiability. In other words, whether these rules are projected to the original rules by removing the choice, e.g. $\{c\}$. in $\widehat{\Pi}'_{\overline{\{d\}}}$ gets changed to c., or whether they are converted into constraints, e.g. $\leftarrow not\ c$, the program will still be unsatisfiable.

Example 4.12 illustrated a simple reason for unsatisfiability. However, the introduced notion is also able to capture more complex reasons of unsatisfiability that involve multiple rules related with each other, which is illustrated in the next example.

Example 4.14 (Graph coloring)**.** Consider coloring the graph shown in Figure 4.4 with two colors green and red. Due to the clique formed by the nodes 1, 2, 3, it is not 2-colorable. The 3-coloring encoding (4.1) of Example 4.1 is altered to 2-colorability by omitting the fact $color(green)$. For the given graph, by grounding and elimination of facts, this encoding reduces to the following rules, where $n \in \{1, \dots, 9\}$, and $c, c_1, c_2 \in \{red, green\}$:

$$\{chosenColor(n, c)\}.$$
$$colored(n) \leftarrow chosenColor(n, c).$$
$$\bot \leftarrow not\ colored(n).$$

$$\bot \leftarrow \mathit{chosenColor}(n, c_1), \mathit{chosenColor}(n, c_2), c_1{\neq}c_2.$$
$$\bot \leftarrow \mathit{chosenColor}(n_1, c), \mathit{chosenColor}(n_2, c). \qquad \text{nodes } n_1, n_2 \text{ are adjacent}$$

Omitting a node n in the graph means to omit all ground atoms related to n; omitting all nodes except $1, 2, 3$ gives us a blocker set with the corresponding blocker rule set shown in Figure 4.5. This abstract program is unsatisfiable and omitting further atoms in the abstraction yields spurious satisfiability. The set of atoms that remain in the program is actually the minimal blocker set for this program. We can also observe the property of unsatisfiable programs being refinement-safe faithful (Proposition 4.9), as refining the shown abstraction by adding back atoms relevant with the other nodes will still yield unsatisfiable programs.

4.3 Domain Abstraction

The approach we presented in the previous section is propositional in nature and does not account for the fact that in ASP, non-ground rules talk about a domain of discourse, where for the (non)existence of an answer set, the precise set of elements may not matter, but rather how certain elements are related. For example, the graph coloring problem encoding (4.1) expresses that each node should be colored to a color that its neighbor does not have. The names of the neighbor nodes are not relevant to the color determination, rather the relation of having a neighbor with a certain chosen color.

In this section, we tackle the issue of automatically constructing and evaluating a suitable abstract program Π' for a given non-ground ASP program Π with an abstraction over its domain.

To illustrate the abstraction and its various challenges, we use the following example.

Example 4.15 (Running example)**.** Consider the following example program Π with domain predicate $\mathit{int}/1$ for an integer domain $D = \{0, \dots, 5\}$.

$$c(X) \leftarrow \mathit{not}\ d(X), X < 5, \mathit{int}(X). \tag{4.9}$$
$$d(X) \leftarrow \mathit{not}\ c(X), \mathit{int}(X). \tag{4.10}$$
$$b(X, Y) \leftarrow a(X), d(Y), \mathit{int}(X), \mathit{int}(Y). \tag{4.11}$$
$$e(X) \leftarrow c(X), a(Y), X \leq Y, \mathit{int}(X), \mathit{int}(Y). \tag{4.12}$$
$$\bot \leftarrow b(X, Y), e(X), \mathit{int}(X), \mathit{int}(Y). \tag{4.13}$$

We furthermore have facts $a(1)$, $a(3)$, $\mathit{int}(0), \dots, \mathit{int}(5)$.

We take a first-order view in which $\mathcal{A}$ is the Herbrand base of Π, which results from the available predicate symbols and the constant symbols (the domain D of discourse, i.e., the Herbrand universe), which are by default those occurring in Π. *Domain abstraction* induces abstraction mappings in which constants are merged.

Definition 4.11. *Given a domain D of* Π, *a* domain (abstraction) mapping *is a function* $m : D \rightarrow \widehat{D}$ *for a set* $\widehat{D}$ *(the* abstracted domain*) with* $|\widehat{D}| \leq |D|$.

Thus, a domain mapping divides D into *clusters* of elements $\{d \in D \mid m(d) = \hat{d}\}$, where $\hat{d} \in \widehat{D}$, seen as equal; if unambiguous, we also write $\hat{d}$ for its cluster $m^{-1}(\hat{d})$.

Example 4.16 (ctd)**.** A possible abstraction mapping for Π with $\hat{D}_1 = \{k_1, k_2, k_3\}$ clusters $1, 2, 3$ to the element k_1 and 4 and 5 to singleton clusters, i.e., $m_1 = \{\{1, 2, 3\}/k_1, \{4\}/k_2, \{5\}/k_3\}$. A naive mapping is $m_2 = \{\{1, .., 5\}/k\}$ with $\hat{D}_2 = \{k\}$.

Each domain mapping m naturally extends to ground atoms $a = p(v_1, \dots, v_n)$ by

$$m(a) = p(m(v_1), \dots, m(v_n)).$$

To obtain for a program Π and a Herbrand base $\mathcal{A}$ an induced abstraction mapping $m : \mathcal{A} \rightarrow \mathcal{A}'$ where $\mathcal{A}' = m(\mathcal{A}) = \{m(a) \mid a \in \mathcal{A}\}$, we need an abstract program Π' as in Definition 4.1. However, simply applying m to Π does not work. Moreover, we want domain abstraction for non-ground Π that results in a non-ground Π'. Building a suitable Π' turns out to be challenging and needs to solve several issues, which we gradually address in the next section.

4.3.1 Towards an Abstract Program

For a program Π, given a mapping m that describes an abstraction over its domain, we start with the intuition of applying m to each rule, i.e., each atom in the rule is modified according to m, in order to obtain a non-ground program Π' which is an abstraction of Π. By taking a look at the cases where simply applying this intuition fails to achieve the desired outcome, we gradually present the approach that addresses these challenges.

Standardizing apart. Firstly, the common use of a variable in the rule as arguments of different atoms in the positive body has to be treated before an abstraction is applied.

Example 4.17 (ctd)**.** The constraint (4.13)

$$\perp \leftarrow b(X, Y), e(X), \mathit{int}(X), \mathit{int}(Y)$$

contains the common use of the variable X. If this rule is lifted with no change by following the intuition, then $b(k, k)$ and $e(k)$ would never occur in the abstract answer sets, while in the original program, answer sets can contain $b(x_1, y)$ and $c(x_2)$ as long as $x_1 \neq x_2$. If the variables in the rule are standardized apart as

$$\perp \leftarrow b(X, Y), e(X_1), X = X_1, \mathit{int}(X), \mathit{int}(X_1), \mathit{int}(Y),$$

then the focus of the abstraction can be directed towards the relation, i.e., $X = X_1$, in the rule, which is covered next.

Furthermore, the occurrence of constants in the arguments of atom needs to be represented using variables: If a literal $l(t_1,\dots,t_n)$ in a rule has a constant as one of its arguments, i.e., $t_i = c$, then the rule is modified by having $l(t_1,\dots,t_{i-1},X,t_{i+1},\dots,t_n), X = c$.

Example 4.18 (ctd)**.** If the constraint (4.13) is of form $\perp \leftarrow b(2,3), e(1).$, then it needs to be changed to the form

$$\perp \leftarrow b(X,X_1), e(X_2), X = 2, X_1 = 3, X_2 = 1, \mathit{int}(X), \mathit{int}(X_1), \mathit{int}(X_2).$$

This way, the effect of the abstraction over the domain can be treated more easily through the relations in the rules.

Handling built-ins and (in)equalities. Original rules may rely on certain *built-in relations* involving variables, such as $<, \leq$ in (4.9) and (4.12), or $=$ and $\neq$. Simply lifting the rule in the abstraction by also lifting these relations prevents achieving an over-approximation due to the behavior of the relations over the abstract clusters. Thus, dealing with the uncertainty caused by the domain clustering by the mapping becomes necessary.

Example 4.19 (ctd)**.** We abstract from Π using m_2. The rule (4.11) has no built-in relation and it does not cause a trouble to lift it with no change:

$$b(X,Y) \leftarrow a(X), d(Y), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y);$$

however, lifting rule (4.12) simply to

$$e(X) \leftarrow c(X), a(Y), X \leq Y, \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y).$$

does not work, as $X \leq Y$ behaves differently over the cluster k. As $k \leq k$ under m_2, whenever $c(k)$ and $a(k)$ holds the lifted rule derives $e(k)$. This applies, e.g., to the abstraction of $I = \{a(1), a(3), c(4), d(0), \dots, d(3)\}$, where (4.12) derives no e-atom as $4 \not\leq 3$ and $4 \not\leq 1$. However, I is an answer set of Π and must not be lost in the abstraction. Thus, when a cluster causes uncertainties over built-ins, we permit $e(k)$ to be false even if $c(k)$ and $a(k)$ holds by creating instead the following rule:

$$\{e(X)\} \leftarrow c(X), a(Y), X \leq Y, \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y).$$

Negation. A naive abstraction approach then could be to turn all rule heads into choices. However, negative literals or certain built-ins (e.g., $\neq, <$) may cause a loss of original answer sets in the abstraction.

Example 4.20 (ctd)**.** We change in (4.12) the symbol $\leq$ to $\neq$ and consider

$$\{e(X)\} \leftarrow c(X), a(Y), X \neq Y, \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y).$$

As $k = k$, the abstract body is never satisfied and $e(k)$ is never derived. However, Π has answer sets containing $c(2)$, $a(3)$, and thus also $e(2)$, as $2 \neq 3$; they are all lost. Adding a choice rule with a flipped relation, $X = Y$, catches such cases.

Similarly, let us change $a(Y)$ in (4.12) to *not* $a(Y)$. When the rule is lifted to

$$\{e(X)\} \leftarrow c(X), not\ a(Y), X \leq Y, \widehat{int}(X), \widehat{int}(Y),$$

$e(k)$ is not derived as $a(k)$ holds and originally a holds only for 1 and 3. Thus, original answer sets I may contain $e(2)$ or $e(4)$ but they are lost in the abstraction. Such cases are caught by additional rules with reversed negation for $a(Y)$:

$$\{e(X)\} \leftarrow c(X), a(Y), X \leq Y, \widehat{int}(X), \widehat{int}(Y).$$

Constraints. Naively lifting the constraints to the abstract rules results in losing answer sets for the non-singleton domain clusters.

Example 4.21 (ctd)**.** After standardizing apart, if the constraint (4.13) is lifted with no change with the lifted relation, this would again result in losing original answer sets in which $b(x_1, y)$ and $c(x_2)$ occur together for $x_1 \neq x_2$.

In conclusion, only creating choices is not enough to preserve all original answer sets; we need a fine-grained systematic approach to deal with uncertainties.

4.3.2 Lifted Built-in Relations

As shown before, for the aim of lifting the original rules in the abstraction to be applied over the abstract domain, the built-in relations in the rules need special treatment, and so do multiple usages of a variable in the rules. To unify both issues, we focus on rules of form

$$r : l \leftarrow B(r), \Gamma_{rel}(r)$$

where the variables in $B(r)$ are standardized apart and Γ_{rel} consists of built-in atoms that constrain the variables in $B(r)$.

Example 4.22 (ctd)**.** The rule (4.11) has $\Gamma_{rel}(r) = \top$ while the rule (4.13) must be standardized apart into $\bot \leftarrow b(X, Y), e(X_1), \Gamma_{rel}$ with $\Gamma_{rel} = (X = X_1)$.

The uncertainty that arises during the abstraction is caused by relation restrictions over non-singleton clusters (i.e., $|\widehat{d}| > 1$) or by negative literals mapped to non-singleton abstract literals. In order to address the uncertainty due to relation restrictions in the rules, we consider a notion of *relation types* with respect to the abstraction. For simplicity, we first focus on binary built-ins, e.g., $=, <, \leq, \neq$, and a $\Gamma_{rel}(r)$ of the form $rel(X, c)$ or $rel(X, Y)$. Later in Section 4.3.4, we show how other forms of relations can be addressed.

Table 4.1: Cases for lifting a binary relation *rel*

$$
\begin{aligned}
\tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)&: \quad rel(\hat{d}_1, \hat{d}_2) \wedge \forall x_1 \in \hat{d}_1, \forall x_2 \in \hat{d}_2.\, rel(x_1, x_2)\\
\tau_{\mathrm{II}}^{rel}(\hat{d}_1, \hat{d}_2)&: \neg rel(\hat{d}_1, \hat{d}_2) \wedge \forall x_1 \in \hat{d}_1, \forall x_2 \in \hat{d}_2.\, \neg rel(x_1, x_2)\\
\tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)&: \quad rel(\hat{d}_1, \hat{d}_2) \wedge \exists x_1 \in \hat{d}_1, \exists x_2 \in \hat{d}_2.\, \neg rel(x_1, x_2)\\
\tau_{\mathrm{IV}}^{rel}(\hat{d}_1, \hat{d}_2)&: \neg rel(\hat{d}_1, \hat{d}_2) \wedge \exists x_1 \in \hat{d}_1, \exists x_2 \in \hat{d}_2.\, rel(x_1, x_2)
\end{aligned}
$$

Lifted relation types When the relation *rel* is lifted to the abstract domain, the cases τ_{I}–τ_{IV} for $rel(\hat{d}_1, \hat{d}_2)$ occur in a mapping as shown in Table 4.1.

If $rel(\hat{d}_1, \hat{d}_2)$ holds for some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$, type III is more common in domain abstractions with clusters, while type I occurs for singleton mappings (i.e., $|\hat{d}_1| = |\hat{d}_2| = 1$) or for relations such as $\neq$ or $<$ when $\hat{d}_1 \neq \hat{d}_2$.

Example 4.23. Consider a mapping $m = \{\{1\}/k_1,\ \{2,3\}/k_2,\ \{4,5\}/k_3\}$. For the relation "$=$", $k_1 = k_1$ holds and for any $x_1, x_2 \in k_1 = \{1\}$, $x_1 = x_2$ holds and type I applies. In contrast, $k_2 = k_2$ holds while $2, 3 \in k_2$ and $2 \neq 3$; so type III applies. Further, $k_2 < k_3$ holds and for any $x \in k_2 = \{2,3\}$ and $y \in k_3 = \{4,5\}$, we have $x < y$ and so type I applies.

If $rel(\hat{d}_1, \hat{d}_2)$ does not hold for some $\hat{d}_1, \hat{d}_2 \in \widehat{D}$, type II is common, e.g., $=, \leq$, whereas type IV may occur for $\neq$ or $<$.

Example 4.24 (ctd)**.** Reconsider m. Then $k_2 \neq k_2$ does not hold while $k_2 = \{2,3\}$ has different elements $2 \neq 3$ (type IV). Moreover, $k_1 = k_2$ does not hold in $\widehat{D}$ nor does $x = y$ for every $x \in k_1 = \{1\}$ and $y \in k_2 = \{2,3\}$ (type II).

Note that for an abstraction m, we let $\mathcal{T}_m$ be the set of all atoms $\tau_{\iota}^{rel}(\hat{d}_1, \hat{d}_2)$ where $\iota \in \{\mathrm{I}, \ldots, \mathrm{IV}\}$ is the type of the built-in instance $rel(\hat{d}_1, \hat{d}_2)$ for m; note that $\mathcal{T}_m$ is easily computed.

Respecting the order relation Notice that if the original domain D contains an order relation among its elements, i.e., $rel(x_1, x_2)$ where $rel \in \{<, \leq\}$, then in order to be able to talk about the relation $rel(\hat{d}_1, \hat{d}_2)$ for the abstract elements $\hat{d}_1, \hat{d}_2$, in the abstract domain $\widehat{D}$ the relation *rel* should be defined. Furthermore, the abstraction mapping should respect the order relation among the elements to avoid unnecessary uncertainty.

Example 4.25 (ctd)**.** In Example 4.23, the abstract elements k_1, k_2, k_3 were assumed to have the order relation $k_1 < k_2 < k_3$. If the mapping m was done to arbitrary abstract elements a, b, c the relation types could not have been determined due to $i < j$ for $i, j \in \{a, b, c\}$ being undefined.

Now consider the mapping $m' = \{\{4\}/k_1,\ \{1,5\}/k_2,\ \{2,3\}/k_3\}$, which does not respect the order relation of the elements in D for the abstraction. The relation types could still be defined for m', however for the relation $<$ the relation types will mostly be of type III and IV, resulting in many uncertainties.

4.3.3 Abstract Program Construction

We next formally define how to construct an abstract program.

By our analysis shown in Section 4.3.1, the basic idea to construct an abstract program for a program Π with a domain mapping m is as follows. We either just abstract each atom in a rule, or in case of uncertainty due to domain abstraction, we guess rule heads to catch possible cases, or we treat negated literals by shifting their polarity depending on the abstract domain clusters.

For a ground literal l, we say that l is *mapped to a non-singleton cluster* if $|m^{-1}(m(l))| > 1$, and it is *mapped to a singleton cluster* otherwise. We use auxiliary facts $isCluster(\hat{d})$ (resp. $isSingleton(\hat{d})$) for the abstract domain elements $\hat{d} \in \widehat{D}$ to denote $|m^{-1}(\hat{d})| > 1$ (and $|m^{-1}(\hat{d})| = 1$). These atoms can also be used to represent whether an abstract literal is a singleton or non-singleton cluster. For the abstract literal $m(l)$, if there exists some term $t \in arg(m(l))$ for which $isCluster(t)$ holds, then this means that $m(l)$ is a non-singleton cluster. Otherwise, it is a singleton cluster.

Example 4.26 (Example 4.15 ctd)**.** Consider the domain mapping $m = \{\{1\}/k_1, \{2,3\}/k_2, \{4,5\}/k_3\}$. For the abstract domain, we have $isSingleton(k_1), isCluster(k_2), isCluster(k_3)$. For the literals, the singleton clusters are $a(k_1), c(k_1), d(k_1), e(k_1)$ and $b(k_1,k_1)$, while the remaining literals are non-singletons.

We remark that due to their definition, if either $\tau^{rel}_{\text{III}}(\hat{d}_1,\hat{d}_2)$ or $\tau^{rel}_{\text{IV}}(\hat{d}_1,\hat{d}_2)$ holds true for some $\hat{d}_1,\hat{d}_2 \in \widehat{D}$, this means that either $\hat{d}_1$ or $\hat{d}_2$ is mapped to a cluster, i.e., $isCluster(\hat{d}_i)$ for some $i \in \{1,2\}$.

As we consider non-ground program, we need to take care of cyclic dependendencies of literals at the non-ground level. A *negative dependency cycle* of length $n \geq 2$ is of the form

$$a_1(\overline{x}_1) \rightarrow a_2(\overline{x}_2) \rightarrow \ldots \rightarrow a_n(\overline{x}_n) \rightarrow a_{n+1}(\overline{x}_{n+1})$$

with $a_{n+1}(\overline{x}_{n+1}) = a_1(\overline{x}_1)$, where $a_i(\overline{x}_i) \rightarrow a_{i+1}(\overline{x}_{i+1})$ denotes that $a_i(\overline{x}_i)$ is in the head of a rule r_i, and r_i has in its negative body some literal $a_{i+1}(\overline{x}'_{i+1})$ that unifies with $a_{i+1}(\overline{x}_{i+1})$. For example, a choice rule consists of a negative cyclic dependency chain of length 2, and a rule of form $p(X) \leftarrow not\ p(X), q(X)$ contains a cycle of length 1.

Definition 4.12 (ctd)**.** *The rules* (4.9)- (4.10) *describe a cyclic dependency of* $d(X) \rightarrow c(X) \rightarrow d(X)$.

In this work, we focus on the predicates of the literals to determine the dependency, thus consider negative cyclic dependency chains of form $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow a_{n+1}$ where $a_{n+1} = a_1$. Later, when we handle a cyclic dependency of the program Π, we will consider a set L_c of literals whose predicates are involved in a cyclic dependency, i.e., for each pair $l_1, l_2 \in L_c$ there exists a chain $pred(l_1) \rightarrow \ldots \rightarrow pred(l_2)$.

Note that determining cyclicity through the predicates is an over-approximated view of cyclic dependency. The cyclic dependency determination can be made more fine grained by also taking into account the arguments and applying unification. Further information on the variables and their restrictions by the relation atoms in the rules can also be employed in finding cyclic dependencies.

Restricted Case For ease of presentation, we first consider programs Π with rules having

(i) at most one negative body literal,

(ii) a single, binary built-in literal, and

(iii) no cyclic dependencies between non-ground literals.

Example 4.27 (ctd)**.** The program Π only adheres to the restrictions (i)-(ii). However the non-ground atoms $d(X)$ and $c(X)$ negatively depend on one another.

Definition 4.13 (rule abstraction)**.** *Given a rule $r: \ l \leftarrow B(r), rel(t_1,t_2)$ as above and a domain mapping m, the set r^m contains the following rules:*

(a) $m(l) \leftarrow m(B(r)), rel(\hat{t}_1,\hat{t}_2), \tau_{\mathrm{I}}^{rel}(\hat{t}_1,\hat{t}_2).$

(b) $\{m(l)\} \leftarrow m(B(r)), rel(\hat{t}_1,\hat{t}_2), \tau_{\mathrm{III}}^{rel}(\hat{t}_1,\hat{t}_2).$

(c) $\{m(l)\} \leftarrow m(B(r)), \overline{rel}(\hat{t}_1,\hat{t}_2), \tau_{\mathrm{IV}}^{rel}(\hat{t}_1,\hat{t}_2).$

(d) For $l_i \in B^-(r)$*:*

(i) $\bigcup_{j \in arg(l_i)} \Big\{ \{m(l)\} \leftarrow m(B^{sh}_{l_i}(r)), rel(\hat{t}_1,\hat{t}_2), isCluster(\hat{j}). \Big\}$

(ii) $\bigcup_{j \in arg(l_i)} \Big\{ \{m(l)\} \leftarrow m(B^{sh}_{l_i}(r)), \overline{rel}(\hat{t}_1,\hat{t}_2), \tau_{\mathrm{IV}}^{rel}(\hat{t}_1,\hat{t}_2), isCluster(\hat{j}). \Big\}$

where $B^{sh}_{l_i}(r){=}B^+(r) \cup \{l_i\}, not\ B^-(r) \backslash \{l_i\}$*,* $\overline{rel}$ *denotes the complement of rel, and for* $k \in \{1,2\}$*, if* t_k *is a constant then* $\hat{t}_k = m(t_k)$*, else* $\hat{t}_k = t_k$*, i.e., variables are not mapped. Similarly, if* $j \in arg(l_i)$ *is a constant then* $\hat{j} = m(\hat{j})$*, else* $\hat{j} = j$*.*

In step (a), the case of having no uncertainty due to abstraction is applied. Steps (b) and (c) are for the cases of uncertainty. The head becomes a choice, and for case IV, we flip the relation, $\overline{rel}$, to catch the case of the relation holding true (which is causing the uncertainty). No rules are added for case II, since the body of the rule will never be satisfied due to the relation not holding true in the abstract domain (similar as in the original domain). Constraints (e.g., (4.13)) are omitted in the cases with uncertainty (i.e., all steps except (a)), since converting them into choice rules becomes ineffective.

Example 4.28 (ctd)**.** Consider the domain mapping $m = \{\{1\}/k_1, \{2,3\}/k_2, \{4,5\}/k_3\}$. We have $\tau_{\mathrm{I}}^{\leq}(x,y)$ true for $(x,y) \in \{(k_1,k_1), (k_1,k_2), (k_1,k_3), (k_2,k_3)\}$, and $\tau_{\mathrm{III}}^{\leq}(x,y)$ true for $(x,y) \in \{(k_2,k_2), (k_3,k_3)\}$, and only type II for all other tuples (x,y). The abstract rules for (4.12) are:

$$e(X) \leftarrow c(X), a(Y), X \leq Y, \tau_{\mathrm{I}}^{\leq}(X,Y), \widehat{int}(X), \widehat{int}(Y).$$
$$\{e(X)\} \leftarrow c(X), a(Y), X \leq Y, \tau_{\mathrm{III}}^{\leq}(X,Y), \widehat{int}(X), \widehat{int}(Y).$$

In step (d) of Definition 4.13, we grasp the uncertainty arising from negation by adding rules that shift the negative literal only if it shares arguments mapped to a non-singleton cluster.

Example 4.29 (ctd)**.** Rule (4.9) has a negative literal, *not* $d(X)$, and the relation $X<5$ with shared argument X. When it is lifted to $X<k_3$, it has $\tau^{<}_{\mathrm{II}}(a,b)$ true for $(a,b)\in\{(k_3,k_1),\ (k_3,k_2)\}$, $\tau^{<}_{\mathrm{IV}}(k_3,k_3)$, and type I for all other tuples (a,b).

By case (a), it is abstracted without change for τ_{I} abstract values, and by case (c) the relation is flipped for τ_{IV}. Furthermore, a shift on the polarity of the negative literal is made:

$$\begin{aligned}
c(X) &\leftarrow \mathit{not}\ d(X), X < k_3, \tau^{<}_{\mathrm{I}}(X,k_3), \widehat{\mathit{int}}(X).\\
\{c(X)\} &\leftarrow \mathit{not}\ d(X), X \geq k_3, \tau^{<}_{\mathrm{IV}}(X,k_3), \widehat{\mathit{int}}(X).\\
\{c(X)\} &\leftarrow d(X), X < k_3, \mathit{isCluster}(X), \widehat{\mathit{int}}(X).\\
\{c(X)\} &\leftarrow d(X), X \geq k_3, \tau^{<}_{\mathrm{IV}}(X,k_3), \mathit{isCluster}(X), \widehat{\mathit{int}}(X).
\end{aligned}$$

Notice that the case of having $\mathit{rel}{=}\top$ is covered by step (a). For this case, the lifted relation will only have type I, thus the rules from the remaining steps need not be added.

Example 4.30 (ctd)**.** In Example 4.15, for any mapping m, the rule (4.11) gets lifted to the abstract rule

$$b(X,Y) \leftarrow a(X), d(Y), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y).$$

Semantically, the rules added in steps (a)-(b) are to ensure that $m(I)$ is a model of Π^m, as either the original rule is kept or it is changed to a choice rule. Steps (c)-(d) serve to catch the cases that may violate the minimality of the model due to a negative literal or a relation over non-singleton clusters. The abstract program is now as follows.

Definition 4.14 (Abstract program Π^m)**.** *Given a program Π as above and a domain abstraction m, the abstract program for m consists of the rules*

$$\begin{aligned}
\Pi^m = &\{r^m \mid r:\ l \leftarrow B(r), \mathit{rel}(t_1,t_2) \in \Pi\}\\
&\cup\ \{x. \mid x \in \mathcal{T}_m\}\\
&\cup\ \{m(p(\vec{c})). \mid p(\vec{c}). \in \Pi\}
\end{aligned}$$

Notably, the construction of Π^m is modular, rule by rule.

Theorem 4.15. *Let m be a domain mapping of a program Π under the above assumptions (i)–(iii). Then for every $I \in AS(\Pi)$, $m(I) \cup \mathcal{T}_m \in AS(\Pi^m)$.*

Proof. Let $\widehat{I}$ and $\widehat{\Pi}$ denote $m(I)$ and Π^m, respectively. Towards a contradiction, assume that there exists some $I \in AS(\Pi)$ s.t. $\widehat{I} \cup \mathcal{T}_m \notin AS(\widehat{\Pi})$. This can occur either because (i) $\widehat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\widehat{I}\cup\mathcal{T}_m}$ or (ii) $\widehat{I} \cup \mathcal{T}_m$ is not a minimal model of $\widehat{\Pi}^{\widehat{I}\cup\mathcal{T}_m}$.

(i) Suppose $\widehat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$. Then there exists some rule $\hat{r} \in \widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$ where $\widehat{I} \cup \mathcal{T}_m \models B(\hat{r})$ and $\widehat{I} \cup \mathcal{T}_m \not\models H(\hat{r})$. By construction of Π, $\hat{r}$ is only obtained by step (a), otherwise $\hat{r}$ would be a choice rule with head $H(\hat{r}) = \{m(l)\}$, and $\hat{r}$ would be satisfied. Consequently $\hat{r}$ is a rule from step (a) for r in Π.

Since $\widehat{I} \cup \mathcal{T}_m \models m(B(r)), rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$, we have $\widehat{I} \cup \mathcal{T}_m \models m(B(r))$. If we have $p(\hat{e}_1, \dots, \hat{e}_n) \in m(B^+(r))$, some $e_i \in \hat{e}_i$ exists such that $p(e_1, \dots, e_n) \in I$ as all variables are standardized apart, $I \models B^+(r)$ for this choice. As for $p(\hat{e}_1, \dots, \hat{e}_n) \in m(B^-(r))$, then $p(e_1, \dots, e_n) \notin I$ for all $e_i \in \hat{e}_i$. So we can instantiate the abstract body $m(B(r))$ to some original body $B(r)$ where $I \models B(r)$. Also having $\widehat{I} \models rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$ means $I \models rel(d_1, d_2)$ for all $d_i \in \hat{d}_i$, thus we have $I \models B(r), rel(d_1, d_2)$. So $r : l \leftarrow B(r), rel(d_1, d_2)$ is in Π^I. As I is a model, it follows that $I \models l$, which then means $\widehat{I} \models m(l)$; this is a contradiction.

(ii) Suppose there exists some $\widehat{J} \subset \widehat{I}$ such that $\widehat{J} \cup \mathcal{T}_m$ is a model of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$. We claim that $J = m^{-1}(\widehat{J}) \cap I$ is a model of Π^I; as $J \subset I$ holds, this would contradict that $I \in AS(\Pi)$. Assume $J \not\models \Pi^I$. Then J does not satisfy some rule $r : l \leftarrow B(r), rel(d_1, d_2)$ in Π^I, i.e., $J \models B(r), rel(d_1, d_2)$ but $J \not\models l$. As $J \subset I$ and I is a model of Π^I, we have $I \models l$, thus, $l \in I \setminus J$.

Now, we look at the cases for applying the mapping m to r, by considering the abstractions $m(B(r))$ and $rel(\hat{d}_1, \hat{d}_2)$, and show that a contradiction is always achieved.

First, assume that $\hat{I} \models m(B(r))$. There are the following cases for $m(J)$: (1-1) $m(J) \models m(B(r))$, or (1-2) $m(J) \not\models m(B(r))$.

(1-1) As $m(J) \models m(B(r))$, we look at $rel(\hat{d}_1, \hat{d}_2)$. We know that $J \models rel(d_1, d_2)$.

- If $rel(\hat{d}_1, \hat{d}_2)$ has the relation type $\tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$, this means that we have $m(J) \models rel(\hat{d}_1, \hat{d}_2)$, and thus $m(J) \cup \mathcal{T}_m \models rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$. As $\hat{J} = m(J)$ and $\hat{J} \subset \hat{I}$, we also get $\hat{I} \cup \mathcal{T}_m \models rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$, thus the non-ground rule created by step (a) has an instantiation $m(l) \leftarrow m(B(r)), rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{I}}(\hat{d}_1, \hat{d}_2)$ in $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$. As $\hat{J}$ and $\hat{I}$ are models of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$, we have $\hat{J} \models m(l)$ and $\hat{I} \models m(l)$. Thus, $l \in m^{-1}(\hat{J})$ and $l \in I$; by definition of J, we get $l \in J$ thus $J \models l$, which is a contradiction.
- If $rel(\hat{d}_1, \hat{d}_2)$ has the relation type $\tau^{rel}_{\mathrm{III}}(\hat{d}_1, \hat{d}_2)$, this again means that we have $m(J) \models rel(\hat{d}_1, \hat{d}_2)$, and thus $m(J) \cup \mathcal{T}_m \models rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{III}}(\hat{d}_1, \hat{d}_2)$ and $\hat{I} \cup \mathcal{T}_m \models rel(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{III}}(\hat{d}_1, \hat{d}_2)$. Thus, as $m(l) \in \hat{I}$ the non-ground choice rule created by step (b) amounts to $m(l) \leftarrow m(B(r)), rel(\hat{t}_1, \hat{t}_2), \tau^{rel}_{\mathrm{III}}(\hat{t}_1, \hat{t}_2)$ in $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$, which again achieves $\hat{J} \models m(l)$, thus $J \models l$, a contradiction.
- If $rel(\hat{d}_1, \hat{d}_2)$ has the relation type $\tau^{rel}_{\mathrm{IV}}(\hat{d}_1, \hat{d}_2)$, then we have $m(J) \not\models rel(\hat{d}_1, \hat{d}_2)$, i.e., $m(J) \models \overline{rel}(\hat{d}_1, \hat{d}_2)$, and thus $m(J) \cup \mathcal{T}_m \models \overline{rel}(\hat{d}_1, \hat{d}_2), \tau^{rel}_{\mathrm{IV}}(\hat{d}_1, \hat{d}_2)$. With similar reasoning as above, we find an instantiation of the non-ground rule created by step (c) and achieve $J \models l$, a contradiction.

(1-2) We show that there is no possibility to have $\hat{J} \not\models m(B(r))$, for $\hat{J} = m(J)$, while $\hat{I} \models m(B(r))$. In order to have $\hat{J} \not\models m(B(r))$, some positive literal $\hat{l}_i \in m(B^+(r))$ must occur in $\hat{I} \backslash \hat{J}$ so that $\hat{J} \not\models m(B^+(r))$. However, this contradicts with $J \models B^+(r)$.

Now, assume that $\hat{I} \not\models m(B(r))$. As $I \models B(r)$, we know that $\hat{I} \models m(B^+(r))$ holds. So we have the rule r in the form $l \leftarrow B^+(r), not\ l_i, rel(d_1, d_2)$ (according to restriction (i) on having at most one negative literal) where $l_i \neq l$ and $\hat{I} \not\models m(B(r))$ means $\hat{I} \models m(l_i)$ for $l_i \in B^-(r)$ while $I \not\models l_i$, i.e., $l_i \notin I$. So we get $\hat{I} \models m(B^{sh}_{l_i}(r))$. Then there are the following cases for $m(J)$: (2-1) $m(J) \models m(B^{sh}_{l_i}(r))$, or (2-2) $m(J) \not\models m(B^{sh}_{l_i}(r))$.

(2-1) As we have $m(J) \models m(B^{sh}_{l_i}(r))$, we look at $rel(\hat{d}_1, \hat{d}_2)$. We know that $J \models rel(d_1, d_2)$.

- For cases $\tau^{rel}_{\text{I}}(\hat{d}_1, \hat{d}_2)$ and $\tau^{rel}_{\text{III}}(\hat{d}_1, \hat{d}_2)$, as we have $J \models rel(d_1, d_2)$, we get $\hat{J} \models rel(\hat{d}_1, \hat{d}_2)$ and $\hat{I} \models rel(\hat{d}_1, \hat{d}_2)$. Notice that since $m(l_i) \in \hat{I}$, there must be some $l'_i \in I$ such that $m(l_i) = m(l'_i)$, thus l_i is mapped to a non-singleton cluster $m(l_i)$. So the atom $isCluster(\hat{j})$ holds true in $\hat{J}$ and $\hat{I}$ for some $j \in arg(l_i)$ for which $|m^{-1}(m(j))| > 1$. Thus in $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$ we get an instantiation $m(l) \leftarrow m(B^{sh}_{l_i}(r)),\ rel(\hat{d}_1, \hat{d}_2), isCluster(\hat{j})$ of the non-ground rule created by (step d-i), and again achieve $J \models l$, which is a contradiction.
- For the case $\tau^{rel}_{\text{IV}}(\hat{d}_1, \hat{d}_2)$, with similar reasoning as in (1-1), we find instantiations of the non-ground rules created by (step d-ii) and achieve $J \models l$, which is a contradiction.

(2-2) We show that there is no possibility to have $m(J) \not\models m(B^{sh}_{l_i}(r))$, while $\hat{I} \models m(B^{sh}_{l_i}(r))$. As $J \models B(r)$, we know that $m(J) \models m(B^+(r))$ holds. So $m(J) \not\models m(B^{sh}_{l_i}(r))$ means $m(J) \not\models m(l_i)$ while $\hat{I} \models m(l_i)$. Now, we take a look at Π^I. As there must be some $l'_i \in I$ (such that $m(l_i) = m(l'_i)$), this means that there is some rule $r' : l'_i \leftarrow B(r'), rel(d'_1, d'_2)$ in Π^I. We then take a look at the abstraction of r'. By doing the same case analysis of (1-1), (1-2) and (2-1), we achieve $m(J) \models m(l'_i)$, i.e., $m(J) \models m(l_i)$, which yields a contradiction. As for (2-2), this means the rule r' is of form $r' : l'_i \leftarrow B^+(r'), not\ l_{i_2}, rel(d'_1, d'_2)$, where we want to claim $m(J) \not\models m(l_{i_2})$. For this, we take a look at another rule r'' in Π^I of form $r'' : l'_{i_2} \leftarrow B(r''), rel(d''_1, d''_2)$ with $m(l'_{i_2}) = m(l_{i_2})$. By restriction (iii) on no negative cyclic dependency among the literals, this recursive process eventually ends, say, after n steps, at some rule $r'^n : l'_{i_n} \leftarrow B(r'^n), rel(d'^n_1, d'^n_2)$ where case (2-2) is not applicable, and $m(J) \models l'_{i_n}$ is achieved. Then by tracing the rules back to r we get $m(J) \models m(l_i)$. Thus $m(J) \not\models m(B^{sh}_{l_i}(r))$ is not possible.

□

Abstract Program (General Case)

We now describe how to remove the restrictions (i)–(iii) on programs from above.

(G-i) Multiple negative literals. If rule r has $|B^-(r)|>1$, we shift each negative literal that shares arguments mapped to a non-singleton cluster. Thus, instead of shifting one literal $l_i \in B^-(r)$, we consider the shifting of multiple literals $L \subseteq B^-(r)$ at a time and all combinations of (non-)shifting of the literals in $B^-(r)$.

Definition 4.15 (Treating multiple negative literals)**.** *Step (d) of Definition 4.13 is modified as*

(d) For $\emptyset \subset L \subseteq B^-(r)$:

(i) $\bigcup_{j \in arg(l_i), l_i \in L} \Big\{ \{m(l)\} \leftarrow m(B^{sh}_L(r)), rel(\hat{t}_1, \hat{t}_2), isCluster(\hat{j}). \Big\}$

(ii) $\bigcup_{j \in arg(l_i), l_i \in L} \Big\{ \{m(l)\} \leftarrow m(B^{sh}_L(r)), \overline{rel}(\hat{t}_1, \hat{t}_2), \tau^{rel}_{\mathrm{IV}}(\hat{t}_1, \hat{t}_2), isCluster(\hat{j}). \Big\}$

where $B^{sh}_L(r) = B^+(r) \cup L, not\ B^-(r) \setminus L$.

This definition allows us to discard the restriction that all negative literals must share a variable with the relation atom, by shifting them to ensure that the case of having a non-singleton mapping is considered. Step (d-i) coincides to steps (d-i) and (d-iii) of Definition 4.13 and step (d-ii) coincides to step (d-ii) of Definition 4.13.

Example 4.31. Consider the rule (4.10) modified as

$$d(X) \leftarrow not\ c(X), not\ a(X), int(X).$$

The constructed non-ground abstract rules by following step (d-i) of Definition 4.15 will be

$$\{d(X)\} \leftarrow c(X), not\ a(X), isCluster(X), \widehat{int}(X), \widehat{int}(X_1).$$
$$\{d(X)\} \leftarrow not\ c(X), a(X), isCluster(X), \widehat{int}(X), \widehat{int}(X_1).$$
$$\{d(X)\} \leftarrow c(X), a(X), isCluster(X), \widehat{int}(X), \widehat{int}(X_1).$$

Step (d-ii) is similarly applied.

(G-ii) Multiple relation literals. A simple approach to handle a built-in part

$$\Gamma_{rel} = rel_1(t_{1,1}, t_{2,1}), .., rel_k(t_{1,k}, t_{2,k}), k > 1,$$

is to view it as literal of an $2k$-ary built-in $rel'(X_{1,1}, X_{2,1}, .., X_{1,k}, X_{2,k})$. The abstract version of such rel' and the cases I-IV are lifted from x_1, x_2 to $x_1, .., x_n$ as in Table 4.2.

Example 4.32. For $\Gamma_{rel} = (X_1{=}X_2, X_3{=}X_4)$, we use a new relation $rel'(X_1, X_2, X_3, X_4)$. For abstract values $\hat{d}_1, .., \hat{d}_4$ such that $\hat{d}_1 = \hat{d}_2 \wedge \hat{d}_3 = \hat{d}_4$ holds. We have type τ_{I} if all $\hat{d}_i$ are singleton clusters and τ_{III} if some $\hat{d}_i$ is non-singleton; otherwise (i.e., $\overline{rel}'(\hat{d}_1, \hat{d}_2, \hat{d}_3, \hat{d}_4)$ holds) type τ_{II} applies.

(G-iii) Cyclic dependencies. Rules which are involved in a negative cyclic dependency need special consideration.

Table 4.2: Cases for lifting an n-ary relation *rel'*

$\tau_{\mathrm{I}}^{rel'}(\hat{d}_1,\dots,\hat{d}_n)$:	$rel'(\hat{d}_1,\dots,\hat{d}_n) \wedge \forall x_1 \in \hat{d}_1,\dots,\forall x_n \in \hat{d}_n.\, rel'(x_1,\dots,x_n)$
$\tau_{\mathrm{II}}^{rel'}(\hat{d}_1,\dots,\hat{d}_n)$:	$\neg rel'(\hat{d}_1,\dots,\hat{d}_n) \wedge \forall x_1 \in \hat{d}_1,\dots,\forall x_n \in \hat{d}_n.\, \neg rel'(x_1,\dots,x_n)$
$\tau_{\mathrm{III}}^{rel'}(\hat{d}_1,\dots,\hat{d}_n)$:	$rel'(\hat{d}_1,\dots,\hat{d}_n) \wedge \exists x_1 \in \hat{d}_1,\dots,\exists x_n \in \hat{d}_n.\, \neg rel'(x_1,\dots,x_n)$
$\tau_{\mathrm{IV}}^{rel'}(\hat{d}_1,\dots,\hat{d}_n)$:	$\neg rel'(\hat{d}_1,\dots,\hat{d}_n) \wedge \exists x_1 \in \hat{d}_1,\dots,\exists x_n \in \hat{d}_n.\, rel'(x_1,\dots,x_n)$

Example 4.33 (ctd)**.** Consider the rules (4.9)-(4.10) (Example 4.15) and the mapping $\{\{1,\dots,5\}\ /k\}$. The abstract rules for them are

$$\{c(X)\} \leftarrow not\ d(X), X \geq k, \tau_{\mathrm{IV}}^{<}(X,k), \widehat{int}(X). \tag{4.14}$$

$$\{c(X)\} \leftarrow d(X), X < k, isCluster(X), \widehat{int}(X). \tag{4.15}$$

$$\{c(X)\} \leftarrow d(X), X \geq k, \tau_{\mathrm{IV}}^{<}(X,k), isCluster(X), \widehat{int}(X). \tag{4.16}$$

$$\{d(X)\} \leftarrow c(X), isCluster(X), \widehat{int}(X). \tag{4.17}$$

in addition to the abstracted rules due to step (a). Consider the answer set $I = \{c(0), d(1), c(2), d(3), c(4), d(5)\}$ of Π. We have $\hat{I} = m(I) = \{c(k), d(k)\}$. Although $\hat{I}$ is a model of $(\Pi^m)^I$, either $c(k)$ or $d(k)$ is unfounded, thus $\hat{I}$ is not minimal, i.e., not an answer set of Π^m. The negative cyclic dependency (i.e., "choice") of c- and d-atoms does not occur for $c(k)$ and $d(k)$ in the constructed Π^m.

To resolve this, the literals of Π that are involved in a negative loop need to be specially treated.

Definition 4.16 (Treating cyclic dependency)**.** *Given a set L_c of literals involved in a negative cyclic dependency, Definition 4.13 is modified by redefining $B^{sh}_{l_i}(r)$ as*

$$B^{sh}_{l_i,L_c}(r) = \begin{cases} B^+(r) \cup \{l_i\}, not\ B^-(r) \backslash \{l_i\} & l_i \notin L_c \\ B^+(r), not\ B^-(r) \backslash \{l_i\} & l_i \in L_c \end{cases}$$

In step (d) of Definition 4.13, the newly defined $B^{sh}_{l_i,L_c}(r)$ eliminates the literals l_i that are involved in a loop from the body instead of shifting their polarity.

Example 4.34 (ctd)**.** For the program Π in (4.9)-(4.13) with the mapping $m = \{\{1,\dots,5\}/k\}$, the constructed program Π^m becomes as below.

$$c(X) \leftarrow not\ d(X), X < k, \tau_{\mathrm{I}}^{<}(X,k), \widehat{int}(X). \tag{4.18}$$

$$\{c(X)\} \leftarrow not\ d(X), X \geq k, \tau_{\mathrm{IV}}^{<}(X,k), \widehat{int}(X). \tag{4.19}$$

$$\{c(X)\} \leftarrow X \geq k, \tau_{\mathrm{IV}}^{<}(X,k), isCluster(X), \widehat{int}(X). \tag{4.20}$$

$$\{c(X)\} \leftarrow X < k, isCluster(X), \widehat{int}(X). \tag{4.21}$$

$$d(X) \leftarrow not\ c(X), \widehat{int}(X). \tag{4.22}$$

$$\{d(X)\} \leftarrow \mathit{isCluster}(X), \widehat{\mathit{int}}(X). \tag{4.23}$$
$$b(X,Y) \leftarrow a(X), d(Y), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y). \tag{4.24}$$
$$e(X) \leftarrow c(X), a(Y), X \leq Y, \tau_{\mathrm{I}}^{\leq}(X,Y), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y). \tag{4.25}$$
$$\{e(X)\} \leftarrow c(X), a(Y), X \leq Y, \tau_{\mathrm{III}}^{\leq}(X,Y), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(Y). \tag{4.26}$$
$$\perp \leftarrow b(X,Y), e(X_1), X = X_1, \tau_{\mathrm{I}}^{=}(X,X_1), \widehat{\mathit{int}}(X), \widehat{\mathit{int}}(X_1), \widehat{\mathit{int}}(Y). \tag{4.27}$$

with $\mathcal{T}_m = \{\tau_{\mathrm{III}}^{\leq}(k,k), \tau_{\mathrm{III}}^{=}(k,k), \tau_{\mathrm{IV}}^{<}(k,k)\}$ and abstract facts $\{a(k), \widehat{\mathit{int}}(k)\}$.

Notice that when the rules are grounded to the relation type facts $\mathcal{T}_m$, only the rules (4.19)-(4.24) and (4.26) remain to be used for the answer set computation.

Treating cyclic dependency with multiple negative literals can then be done by modifying the shifting procedure $B_L^{sh}(r)$ in Definition 4.15 with $B_{l_i,L_c}^{sh}(r)$ of Definition 4.16 as

$$B_{L,L_c}^{sh}(r) = B^+(r) \cup (L \setminus L_c), \mathit{not}\ B^-(r) \setminus L \tag{4.28}$$

where the negative literals in L are omitted from the negative body, and those that do not occur in L_c get their polarity shifted.

Handling multiple cycles $L_{c_1}, \ldots, L_{c_n}$ can be done by defining the set L_c in Definition 4.16 as the union of all the cycles, i.e., $L_c = \bigcup L_{c_i}$. This way $B_{l_i,L_c}^{sh}(r)$ will eliminate all the literals involved in some loop.

Let Π^m denote the program obtained from a general program Π with the generalized abstraction procedure. Then:

Theorem 4.16. *Let m be a domain mapping of a program Π. Then for every $I \in AS(\Pi)$, the abstract interpretation $\widehat{I} = m(I) \cup \mathcal{T}_m$ is an answer set of Π^m.*

Proof. Similar to proof of Theorem 4.15, we assume towards a contradiction that there exists some $I \in AS(\Pi)$ such that $\widehat{I} \cup \mathcal{T}_m \notin AS(\widehat{\Pi})$. This can occur either because (i) $\widehat{I} \cup \mathcal{T}_m$ is not a model of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$ or (ii) $\widehat{I} \cup \mathcal{T}_m$ is not a minimal model.

(i) Let $\widehat{I} \cup \mathcal{T}_m$ be not a model of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$, then there exists some rule $\hat{r} \in \widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$ where $\widehat{I} \cup \mathcal{T}_m \models B(\hat{r}) \wedge \widehat{I} \cup \mathcal{T}_m \not\models H(\hat{r})$. For cases (G-i) and (G-iii), the contradiction is achieved similar to the proof of Theorem 4.15, since $\hat{r}$ is a rule from step (a). As for case (G-ii), we will have $\widehat{I} \cup \mathcal{T}_m \models m(B(r)), \mathit{rel}'(\hat{d}), \tau_{\mathrm{I}}^{\mathit{rel}}(\hat{d})$, where $\hat{d}$ is a shorthand for $\hat{d}_{1,1}, \hat{d}_{2,1}, \ldots, \hat{d}_{1,k}, \hat{d}_{2,k}$; then by definition of rel' this means $I \models B(r), \mathit{rel}_1(d_{1,1}, d_{2,1}), \ldots, \mathit{rel}_k(d_{1,k}, d_{2,k})$ of r in Π^I. This reaches a contradiction as I is a model and $I \models l$, which means $\hat{I} \models m(l)$.

(ii) Now let there be $\widehat{J} \subset \widehat{I}$ such that $\widehat{J} \cup \mathcal{T}_m$ is a model of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$. We claim that $J = m^{-1}(\widehat{J}) \cap I \subset I$ is a model of Π^I; which would contradict that $I \in AS(\Pi)$. Assume $J \not\models \Pi^I$. J does not satisfy some rule $r : l \leftarrow B(r), \mathit{rel}(d_1, d_2)$ in Π^I, i.e., $J \models B(r), \mathit{rel}_1(d_{1,1}, d_{2,1}), \ldots, \mathit{rel}_k(d_{1,k}, d_{2,k})$ but $J \not\models l$, i.e., $l \notin J$. As $J \subset I$ and I is a model of Π^I, we have $I \models l$, i.e., $l \in I \setminus J$. We consider the abstractions $m(B(r))$ and $rel_1(\hat{d}_{1,1}, \hat{d}_{2,1}), \dots, rel_k(\hat{d}_{1,k}, \hat{d}_{2,k})$.

First, assume $\hat{I} \models m(B(r))$. There are the following cases for $m(J)$: (1-1) $m(J) \models m(B(r))$, or $m(J) \not\models m(B(r))$.

(1-1) As $m(J) \models m(B(r))$, we look at $rel_1(\hat{d}_{1,1}, \hat{d}_{2,1}), \dots, rel_k(\hat{d}_{1,k}, \hat{d}_{2,k})$. We know that $J \models rel_1(d_{1,1}, d_{2,1}), \dots, rel_k(d_{1,k}, d_{2,k})$.

(1-1-a) If all $rel_i(\hat{d}_{1,i}, \hat{d}_{2,i})$ have the relation type $\tau_{\mathrm{I}}^{rel_i}(\hat{d}_{1,i}, \hat{d}_{2,i})$, this means that we have $m(J) \models rel_1(\hat{d}_{1,1}, \hat{d}_{2,1}), \dots, rel_k(\hat{d}_{1,k}, \hat{d}_{2,k})$, and thus

$$m(J) \cup \mathcal{T}_m \models rel'(\hat{d}), \tau_{\mathrm{I}}^{rel'}(\hat{d}). \tag{4.29}$$

As $\hat{J} = m(J)$ and $\hat{J} \subset \hat{I}$, we also get $\hat{I} \cup \mathcal{T}_m \models rel'(\hat{d}), \tau_{\mathrm{I}}^{rel'}(\hat{d})$, thus the non-ground rule created by step (a) has an instantiation $m(l) \leftarrow m(B(r)), rel'(\hat{d}), \tau_{\mathrm{I}}^{rel'}(\hat{d})$ in $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$. As $\hat{J}$ and $\hat{I}$ are models of $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$, we have $\hat{J} \models m(l)$ and $\hat{I} \models m(l)$. Thus, by definition of J, we get $J \models l$, which is a contradiction.

(1-1-b) If at least one $rel_i(\hat{d}_{1,i}, \hat{d}_{2,i})$ has the relation type $\tau_{\mathrm{III}}^{rel_i}(\hat{d}_{1,i}, \hat{d}_{2,i})$, while no $rel(\hat{d}_{1,j}, \hat{d}_{2,j})$ has the relation type $\tau_{\mathrm{IV}}^{rel}(\hat{d}_{1,j}, \hat{d}_{2,j})$, above (4.29) is achieved for $\tau_{\mathrm{III}}^{rel'}(\hat{d})$ in place of $\tau_{\mathrm{I}}^{rel'}(\hat{d})$.

(1-1-c) If at least one $rel_i(\hat{d}_{1,i}, \hat{d}_{2,i})$ has the relation type $\tau_{\mathrm{IV}}^{rel_i}(\hat{d}_{1,i}, \hat{d}_{2,i})$, this means that we have $m(J) \models \dots, \overline{rel}(\hat{d}_{1,i}, \hat{d}_{2,i}), \dots$. Thus, for $rel'(\hat{d})$ we have $m(J) \not\models rel'(\hat{d})$ but $m(J) \models \overline{rel}'(\hat{d})$, and

$$m(J) \cup \mathcal{T}_m \models \overline{rel}'(\hat{d}), \tau_{\mathrm{IV}}^{rel'}(\hat{d}). \tag{4.30}$$

By the same reasoning in (1-1-a), we get that the non-ground choice rule created by step (c) amounts to $m(l) \leftarrow m(B(r)), \overline{rel}'(\hat{d}), \tau_{\mathrm{IV}}^{rel'}(\hat{d})$ in $\widehat{\Pi}^{\widehat{I} \cup \mathcal{T}_m}$, and thus we reach a contradiction.

(1-2) This case is handled the same as in proof (1-2) of Theorem 4.15.

Now, we focus on the case $I \not\models m(B(r))$. As $I \models B(r)$, we know that $\hat{I} \models m(B^+(r))$ should hold. Then $\hat{I} \not\models m(B(r))$ means that for a non-empty set $L \subseteq B^-(r)$ of negative literals $\hat{I} \models l_i$ for each $l_i \in L$, while $I \not\models l_i$, i.e., $l_i \notin I$. So we get $\hat{I} \models m(B^{sh}_L(r))$. Assume we further have a set L_c of literals involved in a negative loop. We also get $\hat{I} \models m(B^{sh}_{L,L_c}(r))$ (4.28), since the set $L' = L \cap L_c$ of literals gets omitted from $B^-(r)$, and we get $\hat{I} \models not\ B^-(r) \setminus L'$.

Then there are the following cases for $m(J)$: (2-1) $m(J) \models m(B^{sh}_{L,L_c}(r))$, or (2-2) $m(J) \not\models m(B^{sh}_{L,L_c}(r))$.

(2-1) As $m(J) \models m(B^{sh}_{L,L_c}(r))$, similar to proof (1-1) above and (2-1) of Theorem 4.15, the abstraction $rel_1(\hat{d}_{1,1}, \hat{d}_{2,1}), \dots, rel_k(\hat{d}_{1,k}, \hat{d}_{2,k})$ on relations is considered, and the contradiction $J \models l$ is achieved.

(2-1-a) By (1-1-a) and (1-1-b), we get the case (4.29) and same for $\tau_{\mathrm{III}}^{rel'}(\hat{d})$. We know that as $m(J) \models m(l_i)$ and as $l_i \notin J$ (since $J \subseteq I$ and $l_i \notin I$, this means that $isCluster(\hat{j})$ holds true in $m(J)$ and $\hat{I}$ for some $j \in arg(l_i)$. Thus we have $m(J) \cup \mathcal{T}_m \models rel'(\hat{d}), isCluster(\hat{j})$, which means that $m(l) \in \hat{I}$, thus $l \in I$ and by definition of J, $l \in J$, which is a contradiction.

(2-1-b) By (1-1-c), we get the case (4.30) and by a similar reasoning as in (2-1-a) we also have $m(J) \models isCluster(\hat{j})$, hence $m(J) \cup \mathcal{T}_m \models \overline{rel}'(\hat{d}), \tau_{\mathrm{IV}}^{rel'}(\hat{d}), isCluster(\hat{j})$. Thus we similarly achieve a contradiction.

(2-2) We show that there is no possibility to have $m(J) \not\models m(B^{sh}_{L,L_c}(r))$, while $\hat{I} \models m(B^{sh}_{L,L_c}(r))$. As $J \models B(r)$, we know that $m(J) \models m(B^{+}(r))$ should hold. So $m(J) \not\models m(B^{sh}_{L,L_c}(r))$ means $m(J) \not\models m(l_i)$ for some $l_i \in L \setminus L_c$, while $\hat{I} \models m(l_i)$. We do the same recursive reasoning as in proof (2-2) of Theorem 4.15 over the literals not in L_c. Thus, the process eventually ends and achieves that $m(J) \models m(l_i)$ actually holds, and that $m(J) \not\models m(B^{sh}_{L,L_c}(r))$ is not possible.

□

Example 4.35 (ctd)**.** The original program Π has the below answer sets (with facts omitted).

$$\begin{aligned}
I_1 =& \{c(2), c(4), d(1), d(3), d(5), e(2))\} \cup S_b\\
I_2 =& \{c(2), d(1), d(3), d(4), d(5), e(2), b(1,4), b(3,4)\} \cup S_b\\
I_3 =& \{c(4), d(1), d(2), d(3), d(5), b(1,2), b(3,2)\} \cup S_b\\
I_4 =& \{d(1), d(2), d(3), d(4), d(5), b(1,2), b(1,4), b(3,2), b(3,4)\} \cup S_b
\end{aligned}$$

where $S_b = \{b(1,1), b(1,3), b(1,5), b(3,5), b(3,1), b(3,3)\}$. The constructed abstract program Π^m has the answer sets (with abstract facts omitted)

$$\begin{aligned}
\hat{I}_1 =& \{d(k), b(k,k)\} \qquad \hat{I}_2 = \{c(k)\} \qquad \hat{I}_3 = \{c(k), d(k), b(k,k)\}\\
\hat{I}_4 =& \{c(k), e(k)\} \qquad \hat{I}_5 = \{c(k), d(k), e(k), b(k,k)\}
\end{aligned}$$

where $m(I_1) = m(I_2) = \hat{I}_5$, $m(I_3) = \hat{I}_3$, $m(I_4) = \hat{I}_1$.

The abstraction yields in general an over-approximation of the answer sets of a program. This motivates the following notion.

Definition 4.17 (cf. Definition 4.2)**.** *An abstract answer set* $\widehat{I} \in AS(\Pi^m)$ *is* concrete, *if there exists an answer set* $I \in AS(\Pi)$ *such that* $\widehat{I} = m(I) \cup \mathcal{T}_m$*, else it is* spurious.

A spurious abstract answer set has no corresponding concrete answer set.

Example 4.36 (ctd)**.** The abstract answer sets $\hat{I}_2 = \{c(k)\}$ and $\hat{I}_4 = \{c(k), e(k)\}$ are spurious.

Refining Abstractions After checking an abstract answer set, one can either continue finding other abstract answer sets and check their correctness, or *refine* the abstraction to reach an abstraction where less spurious answer sets occur.

Definition 4.18. *Given a domain mapping $m: D \to D'$, a mapping $m': D \to D''$ is a* refinement *of m if for all $x \in D$, $m'^{-1}(m'(x)) \subseteq m^{-1}(m(x))$.*

That is, refinement is on dividing the abstract clusters to a finer grained domain.

Example 4.37 (ctd)**.** The mapping $m' = \{\{1\}/k_1, \{2,3,4\}/k_2, \{5\}/k_3\}$ is a refinement of mapping m. Furthermore, there does not exist an answer set $I' \in AS(\Pi^{m'})$ such that $m(m'^{-1}(I')) = \hat{I}_2$, thus the spurious answer set $\hat{I}_2$ of Π^m is eliminated.

Faithful abstraction An abstract program that does not have a spurious answer set is a faithful abstraction of the original program.

Example 4.38 (Example 4.1 ctd)**.** In the graph coloring problem for the graph in Figure 4.1a, the mapping $m = \{\{4,5,6\}/\hat{4}\}$ which maps nodes 1,2,3 to singleton clusters constructs an abstract program Π^m that has 42 answer sets which consists of the combination of 6 possible correct colorings of nodes 1-3 with 7 possible colorings $\{\{red\},\{blue\},\{green\},\{red,blue\},\{red,green\},\{green,blue\},\{red,green,blue\}\}$ of the node cluster $\hat{4}$, thus resulting in a faithful abstraction.

Ideally, we have faithfulness, but this is hard to achieve in general (see Section 4.3.6 on complexity).

4.3.4 Syntactic Extensions and Further Considerations

Treating Choice Rules

Choice rules are treated specially by ensuring that the abstraction is done on the body, and the choice over the head is kept.

Definition 4.19. *Given a rule $r: \{l\} \leftarrow B(r), rel(t_1,t_2)$ and a domain mapping m, the set r^m contains the rules of Definition 4.13 for steps (b)-(d), and for step (a), it contains*

$$\{m(l)\} \leftarrow m(B(r)), rel(\hat{t}_1,\hat{t}_2), \tau_{\mathrm{I}}^{rel}(\hat{t}_1,\hat{t}_2).$$

More sophisticated choice rules that involve cardinality constraints, e.g., $m \leq \{l\} \leq n \leftarrow B(r)$, can not immediately be treated similarly. Lifting the cardinality constraints the same to the abstract rule causes to force the occurrence of abstract atoms to ensure the lower bound.

Example 4.39 (ctd)**.** Consider instead of (4.11) the rule

$$2 \leq \{b(X,Y) : d(Y)\} \leq 4 \leftarrow a(X), dom(X).$$

which gets lifted to the same abstract rule. However, for the mapping $m = \{\{1,2,3,4,5\}/k\}$, if $a(k)$ and $d(k)$ holds true, this would cause to have $b(k,k)$ hold true and no other atoms with the same predicate. Thus, the lower bound can not be satisfied, causing the abstract program to become unsatisfiable.

The issue arises from the fact that if the atom in the choice head is involved with some non-singleton cluster, then it is possible that more than one original atom can be mapped to it, thus still satisfying the lower bound constraint in the original program. Such choice rules can be treated by modifying the lower bound of the choice rule in the abstract program and adding a constraint to ensure that the original lower bound is met if the atom is only involved with singleton clusters.

Definition 4.20. *Given a rule $r: \; m \leq \{l\} \leq n \leftarrow B(r), rel(t_1,t_2)$, in the abstraction procedure the choice head is changed to $\{m(l)\} \leq n$, and an additional constraint of the following form is added.*

$$\begin{aligned} \perp \leftarrow & \{m(l) : \mathit{isSingleton}(\hat{t}_1), \ldots, \mathit{isSingleton}(\hat{t}_n)\} < m, & (4.31)\\ & \{m(l) : \mathit{isCluster}(\hat{t}_1); \ldots; m(l) : \mathit{isCluster}(\hat{t}_n)\} < 1. & (4.32)\end{aligned}$$

where $arg(l) = \{t_1, \ldots, t_n\}$,

The idea with the additional constraint is to ensure that if the lower bound m is not satisfied through literals mapped to singleton clusters (4.31), then some literal with a non-singleton cluster (4.32) should also occur.

Example 4.40 (ctd)**.** Instead of lifting the choice rule as in Example 4.39, we add in the abstract program the below rules.

$$\begin{aligned} &\{b(X,Y) : d(Y)\} \leq 4 \leftarrow a(X), dom(X).\\ &\qquad \perp \leftarrow \{b(X,Y) : \mathit{isSingleton}(X), \mathit{isSingleton}(Y)\} < 2,\\ &\qquad\quad \{b(X,Y) : \mathit{isCluster}(X); b(X,Y) : \mathit{isCluster}(Y)\} < 1.\end{aligned}$$

This way there is no lower bound on the number of occurrence of $b(X,Y)$ that causes unsatisfiability at the abstract program. Furthermore, for mapping $m = \{\{1\}/k_1, \{2,3,4,5\}/k_2\}$, if an answer set contains $b(k_1,k_1)$, then the constraint ensures that the answer set also contains some $b(\hat{d}_1,\hat{d}_2)$, where $\hat{d}_1$ or $\hat{d}_2$ is a cluster, so that the original lower bound is met.

Other Forms of Relations

The programs can also consist of relations that are not binary, such as addition, multiplication. These relations can be treated as follows: (1) rewrite the relations by adding instead auxiliary atoms to represent the relations, (2) standardize apart the auxiliary atoms arguments similarly as the remaining atoms, and (3) add to the original program facts of the auxiliary atom to show for which domain elements the relation holds true. In the abstraction procedure, the introduced facts will be lifted to the abstract domain, and the abstraction is handled over relations for the arguments which were standardized apart.

Example 4.41. Consider the rule

$$b(X,Y) \leftarrow a(X), d(Y), X+1=Y, int(X), int(Y).$$

For the addition relation, an auxiliary atom $plusOne(X,Y)$ is introduced by adding the set $\{plusOne(1,2), plusOne(2,3), plusOne(3,4), plusOne(4,5)\}$ of facts to Π to show on which domain elements this relation holds.

The respective rule gets standardized apart into the following form.

$$b(X,Y) \leftarrow a(X), d(Y), plusOne(X_1,Y_1), X{=}X_1, Y{=}Y_1, int(X), int(Y), int(X_1), int(Y_1).$$

The requirement for being able to lift the relations to the abstract domain is to have the built-in relations also defined over the abstract domain. For example, if an ordered domain $\{1,2,3,4,5\}$ is mapped to some domain $\{a,b,c\}$ where the relation $<$ is undefined, then lifting the relation $<$ to the abstract domain will not be feasible. For such cases, the above approach of introducing auxiliary atoms must be taken.

Concreteness with Projection

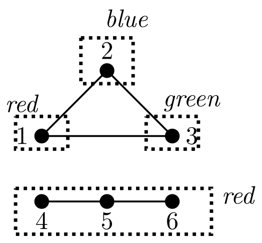
Usually the problem encodings contain auxiliary atoms that are not significant in the computed answer set. When constructing the abstract program, such auxiliary rules are also treated the same, by introducing choices whenever there is an uncertainty. However, this then causes to have many spurious guesses over the auxiliary atoms, and making sure that the abstract answer set is concrete w.r.t. all of these atoms becomes too ambitious, as encountering a concrete abstract answer set among many spurious ones is more difficult. For this reason, we consider a projected notion of determining concreteness of an abstract answer set by only focusing on a certain set of atoms.

Definition 4.21. *For a set A of atoms, an abstract answer set $\widehat{I} \in AS(\Pi^m)$ is* concrete w.r.t. *A, if $\widehat{I}|_{\hat{A}} = m(I|_A) \cup \mathcal{T}_m$ for an answer set $I \in AS(\Pi)$, where $\hat{A} = m(A)$.*

Example 4.42. Consider a modified instance for the graph coloring problem where the isolated nodes are connected with edges as shown in Figure 4.6. For the abstraction, the abstract coloring is spurious since the nodes in the cluster $\{4,5,6\}$ cannot all be colored to *red* in the original graph due to the edges. However, the abstract coloring is concrete w.r.t. the nodes $\{1,2,3\}$.

Such a notion of concreteness becomes useful when abstraction is applied to analyze problems as one can focus on the atoms that are believed to be of importance. For this, the user should have an idea of the significant atoms in the problem description that

Figure 4.6: Concreteness w.r.t. projection over nodes $\{1, 2, 3\}$



are used in determinining a valid solution. For example, when considering an planning problem, this notion can help in focusing on the actions and the directly affected objects, which are used in describing a solution, and obtaining abstract answer sets that have concrete truth assignments of these atoms, while the auxiliary atoms and their concrete truth assignments becomes irrelevant.

4.3.5 Properties of Domain Abstraction

We now consider some basic semantic properties of our formulation of program abstraction. (Non-)existing spurious answer sets allow us to infer properties of the original program.

Proposition 4.17. *For any program* Π*,*

(i) $AS(\Pi^{m_{id}}) = \{I \cup \mathcal{T}_{m_{id}} \mid I \in AS(\Pi)\}$ *for identity* $m_{id} = \{\{x\}/x \mid x \in D\}$*.*

(ii) $AS(\Pi^m) = \emptyset$ *implies that* $AS(\Pi) = \emptyset$*.*

(iii) $AS(\Pi) = \emptyset$ *iff some* Π^m *has only spurious answer sets.*

Proof. (i) Having the identity mapping *id* causes to only have singletom clusters in the abstract domain, thus resulting in only τ_{I} and τ_{II} type facts in $\mathcal{T}_{m_{id}}$. This causes for only the rules of step (a) in Definitions 4.13 and 4.19 to remain when the rules are grounded to the relation types. Hence, the same answer sets are obtained.

(ii) Corollary of Theorem 4.16.

(iii) Similar to the proof of Proposition 4.3(iv) (in omission abstraction). If $AS(\Pi) = \emptyset$, then no $\hat{I} \in AS(\Pi^m)$ for any m has a concrete answer set in Π; thus, all abstract answer sets of Π^m are spurious. Now assume the latter holds but $AS(\Pi) \neq \emptyset$. Then Π has some answer set I, and by Theorem 4.16 $m(I) \cup \mathcal{T}_m \in AS(\Pi^m)$, which would contradict that Π^m has only spurious answer sets.

$\square$

The abstract program is built by a syntactic transformation. The abstraction over the domain can also be done incrementally which in the end amounts to the overall abstraction.

Lemma 4.18. *For any program Π and mapping m for which there are two mappings m_1, m_2 such that $m_2(m_1(D)) = m(D)$, we have $grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2}) = grd_{\mathcal{T}_m}(\Pi^m)$, where $grd_{\mathcal{T}}$ denotes the grounding of the program to the relation type facts $\mathcal{T}$.*

For proving Lemma 4.18, we need to have the following result.

Lemma 4.19. *For a relation $rel(d_1, d_2)$ and a mapping m for which there are two mappings m_1, m_2 such that $m_2(m_1(D)) = m(D)$, we have $\mathcal{T}^{rel}_{m_2,m_1} = \mathcal{T}^{rel}_m$.*

Proof. The relation type computation $\mathcal{T}^{rel}_{m_1}$ is done for $rel(m_1(d_1), m_1(d_2))$, and then the relation type computation $\mathcal{T}^{rel}_{m_2,m_1}$ is done for $rel(m_2(m_1(d_1)), m_2(m_1(d_2))) = rel(m(d_1), m(d_2))$, resulting in the same relation type facts of $\mathcal{T}^{rel}_m$. □

Proof of Lemma 4.18. From the rules of Π^{m_1}, the rules for $(\Pi^{m_1})^{m_2}$ will be constructed according to Definitions 4.13 and 4.19. Consider a rule r with body $B(r), rel(t_1, t_2)$ in Π. The set $r^{m_1} \in \Pi^{m_1}$ contains rules with body $m_1(B(r)), rel(\hat{t}_1, \hat{t}_2), \tau^{rel}_{\mathrm{i}}(\hat{t}_1, \hat{t}_2)$ where $\hat{t}_k = m_1(t_k)$ if t_k is a constant; $\hat{t}_k = t_k$ otherwise.

For the set r^{m_1} of rules, a new set $(r^{m_1})^{m_2}$ will be constructed. Let $r' \in r^{m_1}$, its body will be abstracted to

$$m_2(B(r')), rel(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau^{rel}_{\mathrm{j}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2) \tag{4.33}$$

where $m_2(B(r')) = m_2(m_1(B(r))), m_2(\tau^{rel}_{\mathrm{i}}(\hat{t}_1, \hat{t}_2))$ and $\hat{\hat{t}}_j = m_2(m_1(t_k))$ if t_k is a constant; $\hat{\hat{t}}_k = t_k$ otherwise. Since $m_2(\tau^{rel}_{\mathrm{i}}(\hat{t}_1, \hat{t}_2)) = \tau^{rel}_{\mathrm{i}}(m_2(\hat{t}_1), m_2(\hat{t}_2)) = \tau^{rel}_{\mathrm{i}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2)$, (4.33) will take the form

$$m(B(r)), \tau^{rel}_{\mathrm{i}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2), rel(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau^{rel}_{\mathrm{j}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2).$$

where $\hat{\hat{t}}_k = m(t_k)$ if t_k is a constant; $\hat{\hat{t}}_k = t_k$ otherwise.

The rules in $(r^{m_1})^{m_2}$ where types of the relation differs, i.e., $i \neq j$ for $\tau^{rel}_{\mathrm{i}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2), \tau^{rel}_{\mathrm{j}}(\hat{\hat{t}}_1, \hat{\hat{t}}_2)$, are insignificant as the atoms can not both hold true in $\mathcal{T}_{m_2,m_1}$, i.e., they do not appear in $grd_{\mathcal{T}_{m_2,m_1}}((r^{m_1})^{m_2})$. As for the remaining rules in $(r^{m_1})^{m_2}$, they correspond to the rules in r^m. Thus, by Lemma 4.19 and $\{m_2(m_1(p(\vec{c}))). \mid p(\vec{c}). \in \Pi\} = \{m(p(\vec{c})). \mid p(\vec{c}). \in \Pi\}$, we obtain $grd_{\mathcal{T}_{m_2,m_1}}(\Pi^{m_1})^{m_2}) = grd_{\mathcal{T}_m}(\Pi^m)$. □

Example 4.43 (Example 4.15 ctd)**.** Applying first the mapping $m_1 = \{\{1,2\}/k_0, \{3,4\}/k_1, \{5\}/k_2\}$ and then the mapping $m_2 = \{\{k_0, k_1\}/a_0, \{k_2\}/a_1\}$ obtains the mapping $m = \{\{1, 2, 3, 4\}/a_0, \{5\}/a_1\}$. Figure 4.7 shows the constructed abstract programs. Notice that the program in Figure 4.7b is the same as the non-ground program in Example 4.34 updated for the mapping m, i.e., k is replaced with a_1, when it is grounded to $\mathcal{T}_{m_2,m_1}$.

An easy induction argument shows then the possibility of doing abstraction sequentially, by having abstract mappings defined over previously abstracted domains.

Figure 4.7: Abstract programs of Example 4.43

$$
\begin{aligned}
c(k_0) &\leftarrow k_0 < k_2, \mathit{not}\ d(k_0).\\
c(k_1) &\leftarrow k_1 < k_2, \mathit{not}\ d(k_1).\\
\{c(X)\} &\leftarrow X < k_2, \mathit{dom}(X), \mathit{isCluster}(X).\\
d(X) &\leftarrow \mathit{dom}(X), \mathit{not}\ c(X).\\
\{d(X)\} &\leftarrow \mathit{dom}(X), \mathit{isCluster}(X).\\
b(X,Y) &\leftarrow a(X), d(Y), \mathit{dom}(X), \mathit{dom}(Y).\\
\{e(k_0)\} &\leftarrow c(k_0), a(k_0), k_0 \le k_0.\\
\{e(k_1)\} &\leftarrow c(k_1), a(k_1), k_1 \le k_1.\\
e(k_0) &\leftarrow c(k_0), a(k_1), k_0 \le k_1.\\
e(k_0) &\leftarrow c(k_0), a(k_2), k_0 \le k_2.\\
e(k_1) &\leftarrow c(k_1), a(k_2), k_1 \le k_2.\\
e(k_2) &\leftarrow c(k_2), a(k_2), k_2 \le k_2.\\
\bot &\leftarrow b(k_2,k_2), e(k_2), k_2 = k_2.
\end{aligned}
$$

(a) $grd_{\mathcal{T}_{m_1}}(\Pi^{m_1})$

$$
\begin{aligned}
c(a_0) &\leftarrow a_0 < a_1, \mathit{not}\ d(a_0).\\
\{c(X)\} &\leftarrow X < a_1, \mathit{dom}(X), \mathit{isCluster}(X).\\
d(X) &\leftarrow \mathit{dom}(X), \mathit{not}\ c(X).\\
\{d(X)\} &\leftarrow \mathit{dom}(X), \mathit{isCluster}(X).\\
b(X,Y) &\leftarrow a(X), d(Y), \mathit{dom}(X), \mathit{dom}(Y).\\
\{e(a_0)\} &\leftarrow c(a_0), a(a_0), a_0 \le a_0.\\
e(a_0) &\leftarrow c(a_0), a(a_1), a_0 \le a_1.\\
e(a_1) &\leftarrow c(a_1), a(a_1), a_1 \le a_1.\\
\bot &\leftarrow b(a_1,a_1), e(a_1), a_1 = a_1.
\end{aligned}
$$

(b) $grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2})$

Proposition 4.20. *For any program* Π *and mapping* m *for which there are mappings* $m_1, \dots, m_n$ *such that* $m_n(\dots(m_1(D))) = m(D)$*, we have* $grd_{\mathcal{T}_{m_n,\dots,m_1}}(((\Pi^{m_1})^{\cdots})^{m_n}) = grd_{\mathcal{T}_m}(\Pi^m)$*.*

In Chapter 6, we demonstrate further uses of having a hierarchy of abstractions.

Note that properties of spurious answer sets such as their convexity and the non-reoccurrence after elimination, mentioned for omission abstraction (Proposition4.7-4.8) also apply in domain abstraction, as these are general properties of over-approximation.

Abstraction over Sorts

Applications of ASP usually contain *sorts* that form subdomains of the Herbrand universe. For example, in graph coloring there are sorts for nodes and colors. We define an abstraction over a sort as follows.

Definition 4.22. *An abstraction is limited to a sort* $D_i \subseteq D$*, if all elements* $x \in D \setminus D_i$ *form singleton clusters* $\{x\}/x$*.*

Example 4.44. In the graph coloring problem, we have sorts *node* and *color* in the domain $\{1, \dots, 6, red, green, blue\}$ for the instance in Figure 4.1a. An abstraction mapping m limited to the sort *node* means $m(x) = \{x\}$ for $x \in \{red, blue, green\}$.

In order to obtain much coarser abstractions, applying abstraction over multiple sorts is also possible, given that the individual sorts fulfill the following property.

Definition 4.23 (Sort independence)**.** *For a program Π and domain D, subdomains $D_1, \dots, D_n \subseteq D$ are* independent*, if $D_i \cap D_j = \emptyset$ for all $i \neq j$.*

For independent sorts, abstractions can be composed.

Proposition 4.21. *For domain mappings m_1 and m_2 over independent domains D_1 and D_2, $grd_{\mathcal{T}_{m_1,m_2}}((\Pi^{m_2})^{m_1}) = grd_{\mathcal{T}_{m_2,m_1}}((\Pi^{m_1})^{m_2})$.*

Proof. The mapping $m_i : D \mapsto \hat{D}$ is of form $\{\{x\}/x \mid x \in D \setminus D_i\} \cup m_{D_i}$, $i \in \{1, 2\}$, where m_{D_i} describes the mapping over D_i to the abstract domain $\hat{D}_i$. We know that $m_i(D \setminus D_i) = D \setminus D_i$, and since D_1 and D_2 are independent, we have $D_1 \subseteq m_2(D)$ and $D_2 \subseteq m_1(D)$. Knowing this, we can apply the mappings independently from each other as $m_2(m_1(D)) = m_1(m_2(D))$ to achieve an abstract domain $\hat{\hat{D}} = (D \setminus (D_1 \cup D_2)) \cup \hat{D}_1 \cup \hat{D}_2$. Another mapping m can then be defined to map D to $\hat{\hat{D}}$. By Lemma 4.18 we get the result. □

Cartesian Abstraction

Given domain mappings $m_1, \dots, m_n$ limited to subdomains $D_1, \dots, D_n$, respectively, a *cartesian abstraction* of the mappings corresponds to the abstract domain $m(D_1) \times \cdots \times m(D_n)$. Assuming that the subdomains $D_1, \dots, D_n$ are independent, Definition 4.13 can be altered to be applied over a rule of form

$$r : \ l \leftarrow B(r), rel^{D_1}(t_1, t_2), \dots, rel^{D_n}(t_1, t_2)$$

by considering all possible combinations of $\tau_j^{rel^{D_i}}(\hat{t}_1, \hat{t}_2), j \in \{1, \dots, n\}$. Alternatively, we can define cartesian abstraction by applying abstraction over each subdomain one step at a time, by extending Proposition 4.21 to multiple sorts.

Proposition 4.22. *For domain mappings $m_1, \dots, m_n$ over independent domains $D_1, \dots, D_n$, $\Pi^{m_1 \times \dots \times m_n} = ((\Pi^{m_{\pi(1)}})^{\cdots})^{m_{\pi(n)}}$ where π is any permutation of $\{1, \dots, n\}$.*

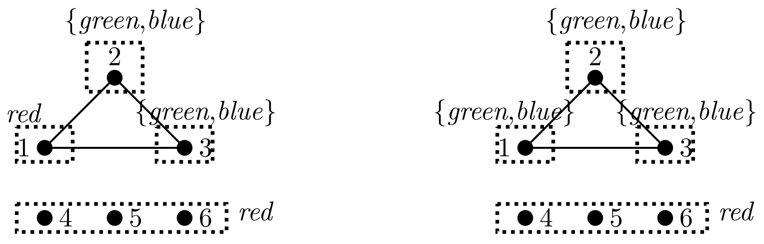
Example 4.45 (Example 4.1 ctd)**.** In the graph coloring problem for the graph in Figure 4.1a, consider the mappings $m_n = \{\{4, 5, 6\}/\hat{4}\}$ and $m_c = \{\{red\}/\hat{r}, \{green, blue\}/\hat{gb}\}$ over the sorts nodes and colors, respectively. The abstract program $(\Pi^{m_n})^{m_c}$ has the concrete answer set (shown in Figure 4.8a)

$$\{chosenColor(1, \hat{r}), chosenColor(2, \hat{gb}), chosenColor(3, \hat{gb}), chosenColor(\hat{4}, \hat{r})\}$$

that chooses the color cluster $\hat{gb}$ for nodes 2 and 3, which matches the intuition of coloring the neighbor nodes of node 1 to some color different than its own color.

Notably, $(\Pi^{m_n})^{m_c}$ also has the spurious answer set (shown in Figure 4.8b)

Figure 4.8: Abstraction over nodes and colors



(a) A concrete answer set (b) A spurious answer set

$$\{chosenColor(1, \hat{gb}), chosenColor(2, \hat{gb}), chosenColor(3, \hat{gb}), chosenColor(\hat{4}, \hat{r})\}$$

due to the guesses introduced for the uncertainty.

In Chapter 5 we demonstrate further uses of such an multi-step abstraction over the subdomains in analyzing problems.

4.3.6 Computational Complexity

In this section, we turn to the computational complexity of reasoning tasks that are associated with program abstraction. We build on the complexity results in [DEGV01, EFFW07].

Lemma 4.23. *Given an arbitrary non-ground program Π, a mapping m, and an abstract interpretation I, checking whether $I \in AS(\Pi^m)$ holds is feasible and in Δ_2^p.*

Proof. By definition, we need to check (1) that I is a model of $(\Pi^m)^I$ and (2) that I is minimal, no $J \subset I$ is a model of $(\Pi^m)^I$.

As for (1), we can refute the property by guessing a rule $r \in \Pi^m$ and a variable substitution θ and verifying that I does not satisfy $(r\theta)^I$, where $r\theta$ denotes the ground instance of r obtained by applying θ to its variables; note that in this case $(r\theta)^I \in (\Pi^m)^I$ holds.

Each rule $r \in \Pi^m$ has polynomial size in the input. Checking whether $r \in \Pi^m$ holds is feasible in polynomial time, as computing the set of atoms that occur in negative cycles is feasible in polynomial time as well. Furthermore, checking whether $r' = r\theta$ is in $(\Pi^m)^I$ is feasible in polynomial time. Overall, refuting (1) is in NP.

As for (2), I is minimal if each atom $a \in I$ has a proof, given by a sequence $r_1, r_2, \dots r_k$ of applications of rules from $r_i \in (\Pi^m)^I$ where each positive body literal of r_i occurs in some head of r_j, $j < i$. Note that w.l.o.g. $I = \{a_1, \dots, a_k\}$ and a_i has as proof $r_1, \dots, r_i$, $i = 1, \dots, k$. As the proof can be guessed and nondeterministically verified in polynomial time, it follows that (2) is in NP. Hence it follows that the problem is in Δ_2^p (more precisely, in the class DP). □

Next we consider the problem of identifying concrete abstract answer sets.

Theorem 4.24. *Given a program Π, a domain mapping m, and an abstract interpretation $\widehat{I}$, deciding whether $\widehat{I}$ is a concrete abstract answer set of Π^m is* NEXP*-complete in general and Σ_2^p-complete for bounded predicate arities. Furthermore, the complexity remains unchanged if $\widehat{I} \in AS(\Pi^m)$ is asserted.*

Proof. To show that $\widehat{I}$ is a concrete abstract answer set of Π^m, we can guess an interpretation J of Π and check that (a) $m(J) = \widehat{I}$, (b) $m(J) \in AS(\Pi^m)$, and (c) $J \in AS(\Pi)$. Testing (a) is clearly polynomial in the size of J, and by Lemma 4.23, (b) and (c) are feasible in Δ_2^p in the size of J and Π (and thus in exponential time in the size of $\widehat{I}$ and Π); consequently, deciding whether $\widehat{I}$ is a concrete abstract answer set of Π^m is in NEXP. For bounded predicate arities, the guess for J has polynomial size in the input, and we can check the conditions (b) and (c) by Lemma 4.23 with an NP oracle in polynomial time; this establishes Σ_2^p membership.

The matching lower bounds are shown by a reduction from deciding whether a given non-ground program Π has some answer set, which is NEXP-complete in the general case and Σ_2^p-complete for bounded predicate arities [DEGV01, EFFW07].

Without loss of generality, Π involves a single predicate p (which can be achieved by reification and padding arguments) and contains some fact $p(\vec{d})$. The mapping we define is $m = \{\{d_1, \dots, d_n\}/\hat{d}\}$ where $d_1, \dots, d_n$ form the Herbrand domain. Then $\widehat{I} = \{p(\hat{d}, \dots, \hat{d})\}$ is a concrete abstract answer set of Π^m iff Π has some answer set. Note that actually $\widehat{I} \in AS(\Pi^m)$ holds; thus the overall complexity does not change if this property is asserted. This proves the result. $\square$

That is, the worst case complexity is the one of answer set existence for non-ground programs; the two problems can be reduced to each other in polynomial time. However, it drops to Σ_2^p if the domain size $|D|$ is polynomial in the abstracted domain size $|\widehat{D}|$; e.g., if each abstract cluster is small (and multiple clusters exist).

The following result is on deciding whether the constructed abstract program has a spurious answer set.

Theorem 4.25. *Given a program Π and a domain mapping m, deciding whether some $\widehat{I} \in AS(\Pi^m)$ exists that is spurious is* $\mathsf{NEXP}^{\mathsf{NP}}$*-complete in general and Σ_3^p-complete for programs with bounded predicate arities (i.e., bounded by a constant).*

Proof. For the membership, one can guess an interpretation $\hat{I}$ of Π^m such that $\hat{I}$ is an answer set of Π^m, and then check whether $\hat{I}$ is spurious. By Theorem 4.24, the spuriousness check can be done with a coNEXP oracle in general and with a Σ_2^p oracle in the bounded predicate case. However, by applying standard padding techniques,[1] it

[1]The input x to the oracle is changed to (x, y), where y is an (exponentially) long string y, and the oracle query considers x from the input only. This artificially lowers the time bound within the query (measured in the size of (x, y)) can be answered.

follows that a coNP oracle is sufficient in the general case. This proves membership of the problem in $\mathsf{coNEXP}^{\mathsf{NP}}$ in the general case and in Σ_3^p in the bounded predicate case, respectively.

The $\mathsf{coNEXP}^{\mathsf{NP}}$-hardness in the general case is shown by a reduction from evaluating second-order logic formulas of a suitable form over finite relational successor structures, i.e., relational structures $S = (D, R^S)$ with a universe D and interpretations R_i^S for the relations R_i in $R = R_1, \dots, R_k$, which include the relations $\mathit{first}(x)$, $\mathit{next}(x, y)$ and $\mathit{last}(x)$ associated with a linear ordering $\leq$ of D.

Lemma 4.26. *Given a second-order (SO) sentence of the form $\Phi = \exists P \forall Q.\varphi$ where $P = P_1, \dots, P_{m_1}$ and $Q = Q_1, \dots, Q_{m_2}$ are predicate variables and $\varphi = \bigvee_j \varphi_j$ is FO such that each φ_j is of the form $\varphi_j = \exists x_1, \dots, x_n l_{j,1} \cdots \wedge \cdots \wedge l_{j,k}$ where each $l_{i,j}$ is a FO-literal, and a finite relational successor structure S, deciding whether $S \models \Phi$ is* $\mathsf{NEXP}^{\mathsf{NP}}$*-complete.*

This lemma can be obtained from the facts that (1) evaluating SO-sentences of the form $\Psi \exists P \forall Q.\varphi$, where ψ is a first-order formula, over finite relational successor structures is $\mathsf{NEXP}^{\mathsf{NP}}$-complete, cf. [GLV99], and (2) that Ψ can be transformed into some Φ of the form described in polynomial time; the latter is possible using second-order skolemization and auxiliary predicates for quantifier elimination, cf. [EGG96] and for denoting subformulas, such that $\varphi(\vec{x}) \equiv P_\varphi(\vec{x})$ and $\varphi(\vec{x}) = \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})$ is expressed by $P_\varphi(\vec{x}) \equiv P_{\varphi_1}(\vec{x}) \wedge P_{\varphi_2}(\vec{x})$ etc.

We first describe how to encode evaluating the sentence $\Phi' = \exists P \exists Q \neg \varphi$ into an ordinary program Π_0, and then extend the encoding to prove the result. We define the rules of Π_0 as follows:

$$P_{j,i}(X_1, \dots, X_n) \leftarrow \mathit{not}\ \overline{P_{j,i}}(X_1, \dots, X_n), D(X_1), \dots, D(X_n). \quad \text{for each } P_{j,i} \in P \tag{4.34}$$

$$\overline{P_{j,i}}(X_1, \dots, X_n) \leftarrow \mathit{not}\ P_{j,i}(X_1, \dots, X_n), D(X_1), \dots, D(X_n). \quad \text{for each } P_{j,i} \in P \tag{4.35}$$

$$Q_{j,i}(Y_1, \dots, Y_n) \leftarrow \mathit{not}\ \overline{Q_{j,i}}(Y_1, \dots, Y_n), D(Y_1), \dots, D(Y_n). \quad \text{for each } Q_{j,i} \in Q \tag{4.36}$$

$$\overline{Q_{j,i}}(Y_1, \dots, Y_n)\} \leftarrow \mathit{not}\ \ Q_{j,i}(Y_1, \dots, Y_n), D(Y_1), \dots, D(Y_n). \quad \text{for each } Q_{j,i} \in Q \tag{4.37}$$

$$\mathit{sat} \leftarrow l_{j,1}^{\neg/\mathit{not}} \wedge \cdots \wedge l_{j,k}^{\neg/\mathit{not}}. \tag{4.38}$$

$$\mathit{ok} \leftarrow \mathit{not}\ \mathit{ok}. \tag{4.39}$$

$$\mathit{ok} \leftarrow \mathit{not}\ \mathit{sat}. \tag{4.40}$$

where $l^{\neg/\mathit{not}}$ denotes the replacement of $\neg$ in l by not.[2]

Informally, the rules (4.34),(4.35) and (4.36),(4.37) guess extensions for the predicates in P and Q, respectively, while the rules (4.38) evaluate the formula φ. A guess for P

[2]To make the rules safe, domain predicates $D(X)$ can be added for unsafe variables X.

and Q yields an answer set of Π_0 augmented with S (provided as positive facts) iff φ evaluates over S to false; in this case, no rule (4.38) fires and thus *sat* can not be derived, which means in turn that *ok* can be derived by (4.40) and thus the constraint (4.39) is satisfied. On the other hand, deriving *ok* is necessary to have an answer set, which means that *sat* must not be derived from the guess for P and Q.

We extend the program Π_0 now for spuriousness checking. To this end, we introduce for the domain $D = \{x_1, \dots, x_n\}$ at hand a copy $D' = \{y_1, \dots, y_n\}$ and link x_i to y_i via a predicate $eq(x, y)$ that holds for $x, y \in D \cup D'$ iff $x = x_i \wedge y = y_i$ for some $i = 1, \dots, n$. The idea is to use D and D' in the predicates from P and Q, respectively, and to abstract D' into a single element, such that for every guess χ for P, some abstract answer set $\hat{I}_\chi$ of the abstract program Π^m will exist; and that, moreover, $\hat{I}_\chi$ will be concrete if for some guess for Q, we have an answer set of Π, where the latter program is equivalent to Π_0; thus $\hat{I}_\chi$ will be spurious iff no guess for Q will yield an answer set of Π_0, which means that the formula $\exists P \forall Q.\varphi$ evaluates to true.

We make the following adjustments.

1. First, we replace in (4.36) and (4.37) the predicate D with D'.
2. Next, for each rule r from (4.38) we add for each term t that occurs in the rule body a "typing" atom $D(t)$, we replace each term t that occurs in a Q-literal with a fresh variable X_t and add the atoms $D'(X_t)$ and $eq(t, X_t)$.
3. To each rule r obtained from the previous step we add $not\ succ(y_1, y_1)$ in the body (this literal evaluates to true with no abstraction).
4. We add facts $eq(x_i, y_i)$, for $i = 1, \dots, n$.
5. We add facts $Q_{j,i}(y_0, \dots, y_0)$, $\overline{Q_{j,i}}(y_0, \dots, y_0)$ for all $Q_{j,i} \in Q$, where y_0 is a fresh constant.

It is not hard to establish that the answer sets I of the resulting program Π (over S) correspond to the answer sets I_0 of Π_0 over S; each I is obtained from some I_0 by replacing in the $Q_{j,i}$- and $\overline{Q_{j,i}}$-atoms the constant x_l with the corresponding y_l, adding all facts $Q_{j,i}(y_0, \dots, y_0)$, $\overline{Q_{j,i}}(y_0, \dots, y_0)$ and stripping off the *eq*-atoms.

The mapping that we construct is $m = \{\{x_1\}/x_1, \dots, \{x_n\}/x_n\} \cup \{\{y_0, y_1, \dots, y_n\}/\hat{y}\}$. In the abstract program Π^m, the rules (4.34), (4.35) are carried over, while the modified rules (4.36), (4.37) are turned into rules to derive abstract atoms over $Q_{j,i}$ resp. $\overline{Q_{j,i}}$. However, since Π^m contains the abstracted facts $Q_{j,i}(\hat{y}, \dots, \hat{y})$, $\overline{Q_{j,i}}(\hat{y}, \dots, \hat{y})$, these rules are redundant.

The modified rules (4.38) are turned into guessing rules for *sat*, while the other rules (4.39) and (4.40) remain unchanged. The abstract answer sets of Π^m correspond to guesses χ for P to which *ok* and all $Q_{j,i}(\hat{y}, \dots, \hat{y})$, $\overline{Q_{j,i}}(\hat{y}, \dots, \hat{y})$ are added (*sat* is guessed false); denote this answer set by I_χ.

The answer set I_χ is concrete, if there is some guess μ for Q such that we obtain an answer set I of the program Π that is mapped to I_χ, i.e., $m(I_1) = I_\chi$; this I_1 corresponds to some answer set I_0 as described above. Thus I_χ is spurious, if no such guess μ for Q exists.

Putting it all together, it holds that Π has with respect to the mapping $m = \{\{x_1\}/x_1, \dots, \{x_n\}/x_n\} \cup \{\{y_0, y_1, \dots, y_n\}/\hat{y}\}$ some spurious answer iff the formula Φ in Lemma 4.26 evaluates over S to true. Since Π and m are constructable in polynomial time from Φ and S, this proves $\mathsf{coNEXP}^{\mathsf{NP}}$ hardness in the general case.

For the bounded predicate arities case, the evaluation of a formula Φ as in Lemma 4.26 is Σ_3^p-complete; furthermore, all steps in producing the program Π preserves bounded arities. Thus with the same argument, we obtain Σ_3^p-hardness for deciding whether some spurious answer set exists for bounded predicate arities. This proves the result. $\square$

As for testing faithfulness, we note the following result that is an immediate consequence.

Corollary 4.27. *Given a program* Π *and a domain mapping* m*, deciding whether* Π^m *is faithful, i.e., has no spurious abstract answer set, is* $\mathsf{coNEXP}^{\mathsf{NP}}$*-complete in general and* Π_3^p*-complete for bounded predicate arities (i.e., bounded by a constant).*

4.3.7 Existential Abstraction on Relations

Until now, we focused on constructing an abstract program by lifting the built-in relations of the original program and treating the uncertainties among the behavior of the relations due to the abstraction over the domain elements. Section 4.3.4 described how to handle the cases that do not adhere to the restricted case of having binary built-in relations defined over the abstract domain. In this section, we introduce a different way of constructing the abstract programs in order to avoid having such a restriction over the given program.

The idea is to apply abstraction over the relations in the spirit of existential abstraction. For this, we introduce an abstract relation $\widehat{rel}$ for a relation rel as follows:

$$(\forall \hat{d}_i \in \widehat{D})\widehat{rel}(\hat{d}_1, \dots, \hat{d}_k) \Leftrightarrow \exists x_i \in m^{-1}(\hat{d}_i).rel(x_1, \dots, x_k). \tag{4.41}$$

$$(\forall \hat{d}_i \in \widehat{D})neg\_\widehat{rel}(\hat{d}_1, \dots, \hat{d}_k) \Leftrightarrow \exists x_i \in m^{-1}(\hat{d}_i).\neg rel(x_1, \dots, x_k). \tag{4.42}$$

I.e., $\widehat{rel}(\hat{d}_1, \dots, \hat{d}_k)$ is true if for some corresponding original values the original relation holds; the negation of $\widehat{rel}(\hat{d}_1, \dots, \hat{d}_k)$ is true otherwise. Notably, both versions may hold simultaneously, depending on the abstract domain clusters.

Example 4.46 (Example 4.15 ctd)**.** For the mapping $\{1, \dots, 5\}/k$, the abstract relation $k \widehat{\leq} k$ holds true, since $X \leq Y$ for all X, Y mapped to k. The abstract relation $k \widehat{=} k$ and its negation both hold true, since $X_1 = X_2$ holds only for some X_1, X_2 values mapped to k.

Notice that having both rel and neg_rel hold means an uncertainty on the truth value of the relation in the abstract clusters. This brings us to determining the types of the relations over the abstract clusters, similar as before.

Abstract relation types. The following cases $\tau_{\mathrm{I}} - \tau_{\mathrm{III}}$ occur in a mapping for the abstract relation predicates $\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k)$ and $neg\_\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k)$:

$$\begin{array}{lrl}
\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1,\dots,\hat{d}_k): & \widehat{rel}(\hat{d}_1,\dots,\hat{d}_k) \wedge & not\ neg\_\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k) \\
\tau_{\mathrm{II}}^{\widehat{rel}}(\hat{d}_1,\dots,\hat{d}_k): & neg\_\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k) \wedge & not\ \widehat{rel}(\hat{d}_1,\dots,\hat{d}_k) \\
\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1,\dots,\hat{d}_k): & \widehat{rel}(\hat{d}_1,\dots,\hat{d}_k) \wedge & neg\_\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k)
\end{array} \tag{4.43}$$

Type I is the case where the abstraction does not cause uncertainty for the relation, thus the rules that contain $\widehat{rel}$ with type I can remain the same in the abstract program. Type II shows the cases where $\widehat{rel}$ does not hold in the abstract domain. Type III is the uncertainty case, which needs to be dealt with when creating the abstract rules. To ensure that an over-approximation is achieved, the head of the respective rule will be changed into a choice.

For an abstraction m, we compute the set $\mathcal{T}_m$ of all atoms $\tau_{\iota}^{\widehat{rel}}(\hat{d}_1,\dots,\hat{d}_k)$ where $\iota \in \{\mathrm{I},\mathrm{II},\mathrm{III}\}$ is the type of $\widehat{rel}(\hat{d}_1,\dots,\hat{d}_k)$ for m.

Abstraction procedure

For simplicity and ease of presentation, we consider programs with rules having (i) a single relation atom; and (ii) no cyclic dependencies between non-ground literals.

Definition 4.24 (rule abstraction). *Given a rule $r:\ l \leftarrow B(r), rel(t_1,\dots,t_k)$ and a domain mapping m, the set r^m contains the following rules.*

(a) $m(l) \leftarrow m(B(r)), \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1,\dots,\hat{t}_k).$

(b) $\{m(l)\} \leftarrow m(B(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\dots,\hat{t}_k).$

(c) For all $L \subseteq B^-(r)$*:*

$\bigcup_{j\in arg(l_i), l_i\in L}\left\{\{m(l)\} \leftarrow m(B_L^{sh}(r)), \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1,\dots,\hat{t}_k), isCluster(\hat{j}).\right\}$

$\bigcup_{j\in arg(l_i), l_i\in L}\left\{\{m(l)\} \leftarrow m(B_L^{sh}(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\dots,\hat{t}_k), isCluster(\hat{j}).\right\}$

where $B_L^{sh}(r) = B^+(r) \cup L, not\, B^-(r) \setminus L.$

The idea is to introduce guesses when there is an uncertainty over the relation holding in the abstract domain (b), or over the negated atoms due to the abstract clusters (c) (by considering all combinations of the negative literals), and otherwise just abstracting the rule (a).

The abstraction procedure introduced in Definition 4.24 obtains semantically the same abstract program as in Definition 4.13 for rules of form

$$l \leftarrow B(r), rel(t_1, t_2).$$

with binary relations $=, \neq, \leq, <$. We denote the latter by Π_0^m.

Proposition 4.28. *For any domain mapping m of a program Π, $AS(\Pi^m)$ and $AS(\Pi_0^m)$ coincide (modulo auxiliary atoms).*

Proof. For an assignment I, we need to show that $I \cup \mathcal{T}_m$ is a minimal model of $(\Pi^m)^I$ if and only if $I \cup \mathcal{T}_{m_0}$ is a minimal model of $(\Pi_0^m)^I$.

($\Rightarrow$) Towards a contradiction, assume $I \cup \mathcal{T}_m$ is a minimal model of $(\Pi^m)^I$ but $I \cup \mathcal{T}_{m_0}$ is not a minimal model of $(\Pi_0^m)^I$. Then either (i) $I \cup \mathcal{T}_{m_0}$ is not a model of $(\Pi_0^m)^I$, or (ii) $I \cup \mathcal{T}_{m_0}$ is not a minimal model of $(\Pi_0^m)^I$.

(i) There is a rule $\hat{r} \in (\Pi_0^m)^I$ such that $I \cup \mathcal{T}_{m_0} \models B(\hat{r})$ but $I \not\models H(\hat{r})$. By construction of Π_0^m, $\hat{r}$ is only obtained by step (a) of Definition 4.13, otherwise $\hat{r}$ would be a choice rule with head $H(\hat{r}) = \{m(l)\}$, and $\hat{r}$ would be satisfied. Consequently $\hat{r}$ is a rule from step (a) for r in Π. Thus, we have $I \cup \mathcal{T}_{m_0} \models m(B(r)), rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)$. Since the definitions of relation type I for lifted relations and abstract relations correspond to each other, we have $\mathcal{T}_{m_0} \models \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2) \iff \mathcal{T}_m \models \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$. This means we get $I \cup \mathcal{T}_m \models m(B(r)), \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ which is the abstract rule of r constructed by step (a) of Definition 4.24. Since $I \cup \mathcal{T}_m$ is a minimal model of $(\Pi^m)^I$, $I \models H(\hat{r})$. Hence, we reach a contradiction.

(ii) Let there be $J \subset I$ such that $J \cup \mathcal{T}_{m_0}$ is a model of $(\Pi_0^m)^I$. We claim that $J \cup \mathcal{T}_m$ is a model of $(\Pi^m)^I$, which would contradict $I \cup \mathcal{T}_m \in AS(\Pi^m)$. Assume $J \cup \mathcal{T}_m \not\models (\Pi^m)^I$. Then there is a rule $\hat{r} \in (\Pi^m)^I$ such that $J \cup \mathcal{T}_m \models B(\hat{r})$ but $J \not\models H(\hat{r})$, while $I \models H(\hat{r})$.We need to show that there is a corresponding rule in $(\Pi_0^m)^I$ for $\hat{r}$, which would then achieve the contradiction that is $J \models H(\hat{r})$. Below, we denote by $B(\hat{r}) \setminus \hat{\Gamma}_r$, the abstract body excluding the abstracted relation (and its relation type atom).

- If $\hat{r}$ contains $\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ (step (a) or (c) of Definition 4.24), then since we know $\mathcal{T}_{m_0} \models \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2) \iff \mathcal{T}_m \models \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{t}_1, \hat{t}_2)$, we achieve $J \cup \mathcal{T}_{m_0} \models rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)$ (also $J \cup \mathcal{T}_{m_0} \models rel(\hat{d}_1, \hat{d}_2)$). Since $\hat{r} \in (\Pi^m)^I$, we know that $I \models B(\hat{r})$ and also $I \models B(\hat{r}) \setminus \hat{\Gamma}_r$. Thus we get $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)$ (also $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, rel(\hat{d}_1, \hat{d}_2)$) in $(\Pi_0^m)^I$. Since $J \cup \mathcal{T}_{m_0}$ is a model of $(\Pi_0^m)^I$, we get $J \models H(\hat{r})$, which is a contradiction.

- If $\hat{r}$ contains $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ (step (b) or (c) of Definition 4.24), then $J \cup \mathcal{T}_m \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ means that there exist some $d_{11}, d_{12} \in m^{-1}(\hat{d}_1)$, $d_{21}, d_{22} \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \models rel(d_{11}, d_{21})$ and $J' \not\models rel(d_{12}, d_{22})$. There are the following cases for $rel(\hat{d}_1, \hat{d}_2)$: (1) $J \models rel(\hat{d}_1, \hat{d}_2)$, or (2) $J \not\models rel(\hat{d}_1, \hat{d}_2)$.
  (1) Since we know $J' \not\models rel(d_{12}, d_{22})$, this case obtains $\tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_{m_0} \models rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$. With similar reasoning as above on obtaining $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$ in $(\Pi_0^m)^I$ (also $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, rel(\hat{d}_1, \hat{d}_2)$ in $(\Pi_0^m)^I$), we achieve $J \models H(\hat{r})$, a contradiction.

(2) Since we know $J' \models rel(d_{11}, d_{21})$, this case obtains $\tau_{\mathrm{IV}}^{rel}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_{m_0} \models \overline{rel}(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{IV}}^{rel}(\hat{d}_1, \hat{d}_2)$. With similar reasoning as above we reach a contradiction.

($\Leftarrow$) Towards a contradiction, assume $I \cup \mathcal{T}_{m_0}$ is a minimal model of $(\Pi_0^m)^I$ but $I \cup \mathcal{T}_m$ is not a minimal model of $(\Pi^m)^I$. Then either (i) $I \cup \mathcal{T}_m$ is not a model of $(\Pi^m)^I$, or (ii) $I \cup \mathcal{T}_m$ is not a minimal model of $(\Pi^m)^I$.

(i) There is a rule $\hat{r} \in (\Pi^m)^I$ such that $I \cup \mathcal{T}_m \models B(\hat{r})$ but $I \not\models H(\hat{r})$. By construction of Π^m, $\hat{r}$ is only obtained by step (a) of Definition 4.24. With an analogous reasoning as above item (i), we achieve a contradiction.

(ii) Let there be $J \subset I$ such that $J \cup \mathcal{T}_m$ is a model of $(\Pi^m)^I$. We claim that $J \cup \mathcal{T}_{m_0}$ is a model of $(\Pi_0^m)^I$, which would contradict $I \cup \mathcal{T}_{m_0} \in AS(\Pi_0^m)$. Assume $J \cup \mathcal{T}_{m_0} \not\models (\Pi_0^m)^I$. Then there is a rule $\hat{r} \in (\Pi_0^m)^I$ such that $J \cup \mathcal{T}_{m_0} \models B(\hat{r})$ but $J \not\models H(\hat{r})$, while $I \models H(\hat{r})$. We need to show that there is a corresponding rule in $(\Pi^m)^I$ for $\hat{r}$, which would then achieve the contradiction that $J \models H(\hat{r})$.

- If $\hat{r}$ contains $rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)$ (step (a) of Definition 4.13), an analogous reasoning as above item (ii) obtains $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \tau_{\mathrm{I}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ in $(\Pi^m)^I$ which achieves $J \models H(\hat{r})$ a contradiction.

- If $\hat{r}$ contains $rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$ (step (b) of Definition 4.13), then $J \cup \mathcal{T}_{m_0} \models rel(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$ means that $J \models rel(\hat{d}_1, \hat{d}_2)$ and there exist some $d_1 \in m^{-1}(\hat{d}_1)$, $d_2 \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \not\models rel(d_1, d_2)$. This obtains abstract relation type $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, thus $J \cup \mathcal{T}_m \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$. Notice that also $J \models isCluster(\hat{d}_i)$ holds for some $i \in \{1, 2\}$. With similar reasoning as above on obtaining $H(\hat{r}) \leftarrow B(\hat{r}) \setminus \hat{\Gamma}_r, \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$ in $(\Pi^m)^I$, we achieve $J \models H(\hat{r})$, a contradiction.

- If $\hat{r}$ contains $\overline{rel}(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{IV}}^{rel}(\hat{d}_1, \hat{d}_2)$ (step (c) or (d-ii) of Definition 4.13), then $J \cup \mathcal{T}_{m_0} \models \overline{rel}(\hat{d}_1, \hat{d}_2), \tau_{\mathrm{IV}}^{rel}(\hat{d}_1, \hat{d}_2)$ means that $J \not\models rel(\hat{d}_1, \hat{d}_2)$ and there exist some $d_1 \in m^{-1}(\hat{d}_1)$, $d_2 \in m^{-1}(\hat{d}_2)$ and some $J' \in m^{-1}(J)$ such that $J' \models rel(d_1, d_2)$. This again obtains abstract relation type $\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, i.e., $J \cup \mathcal{T}_m \models \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{d}_1, \hat{d}_2)$, thus reaches a contradiction as above.

- If $\hat{r}$ contains only $rel(\hat{d}_1, \hat{d}_2)$ (step (d-i) of Definition 4.13), then this means either $J \cup \mathcal{T}_m \models \tau_{\mathrm{I}}^{rel}(\hat{d}_1, \hat{d}_2)$ or $J \cup \mathcal{T}_m \models \tau_{\mathrm{III}}^{rel}(\hat{d}_1, \hat{d}_2)$ holds. Also we know that $J \models isCluster(\hat{d}_i)$ holds for some $i \in \{1, 2\}$. So similar as above, we achieve a contradiction.

$\square$

Generalization to multiple relation atoms and handling cyclic dependencies by removing the restrictions (i)-(ii) can be done similarly as in cases (G-ii) and (G-iii) of Section 4.3.3.

Example 4.47 (Example 4.15 ctd)**.** For the program Π in (4.9)-(4.13) with the mapping $m = \{\{1,\dots,5\}/k\}$, the constructed program Π^m becomes as below.

$$
\begin{aligned}
&c(X) \leftarrow \mathit{not}\ d(X), \tau_{\mathrm{I}}^{<}(X,k), \widehat{int}(X).\\
&\{c(X)\} \leftarrow \mathit{not}\ d(X), \tau_{\overline{\mathrm{III}}}^{<}(X,k), \widehat{int}(X).\\
&\{c(X)\} \leftarrow \tau_{\mathrm{I}}^{<}(X,k), \mathit{isCluster}(X), \widehat{int}(X).\\
&\{c(X)\} \leftarrow \tau_{\overline{\mathrm{III}}}^{<}(X,k), \mathit{isCluster}(X), \widehat{int}(X).\\
&d(X) \leftarrow \mathit{not}\ c(X), \widehat{int}(X).\\
&\{d(X)\} \leftarrow \mathit{isCluster}(X), \widehat{int}(X).\\
&b(X,Y) \leftarrow a(X), d(Y), \widehat{int}(X), \widehat{int}(Y).\\
&e(X) \leftarrow c(X), a(Y), \tau_{\mathrm{I}}^{\leq}(X,Y), \widehat{int}(X), \widehat{int}(Y).\\
&\{e(X)\} \leftarrow c(X), a(Y), \tau_{\overline{\mathrm{III}}}^{\leq}(X,Y), \widehat{int}(X), \widehat{int}(Y).\\
&\bot \leftarrow b(X,Y), e(X_1), \tau_{\mathrm{I}}^{=}(X,X_1), \widehat{int}(X), \widehat{int}(X_1), \widehat{int}(Y).
\end{aligned}
$$

with $\mathcal{T}_m = \{\tau_{\overline{\mathrm{III}}}^{\leq}(k,k), \tau_{\overline{\mathrm{III}}}^{=}(k,k), \tau_{\overline{\mathrm{III}}}^{<}(k,k)\}$ and abstract facts $\{a(k), \widehat{int}(k)\}$. Note that the atom $d(X)$ is omitted instead of shifting the polarity.

The abstract program consists of the same answer sets as the one constructed by lifting the relations in Example 4.34.

When treating n-ary relations for $n > 2$, it is possible to modify Definition 4.24 to create finer abstractions.

Example 4.48. Consider the argument Z of the following rule involving addition:

$$r : e(Z) \leftarrow c(X), a(Y), Z = X + Y. \tag{4.44}$$

We denote $Z = X + Y$ with the relation $\mathit{plus}(X,Y,Z)$. Regarding the arguments, we have $\mathit{arg}(e(Z)) \cap \mathit{arg}(\mathit{plus}(X,Y,Z)) = \{Z\} \neq \emptyset$ while $\mathit{arg}(e(Z)) \cap \{X,Y\} = \emptyset$, where X, Y are the shared arguments of the body literals with the relation plus, i.e., $\mathit{arg}(B(r)) \cap \mathit{arg}(\mathit{plus}(X,Y,Z)) = \{X,Y\}$. Consider the mapping $m : \{1\} \mapsto a_1, \{2,3\} \mapsto a_{23}, \{4,5\} \mapsto a_{45}$ and $X{=}a_1, Y{=}a_1$. For the abstract relation $\hat{\mathit{plus}}$, both $\hat{\mathit{plus}}(a_1,a_1,a_{23})$ and $\mathit{neg\_}\hat{\mathit{plus}}(a_1,a_1,a_{23})$ hold true, due to $1+1=2$ and $1+1 \neq 3$. As Z is not used in the body literals, it does not cause uncertainties for applying the rule in the abstraction, which is caught by

$$e(Z) \leftarrow c(X), a(Y), \tau_{\mathrm{III}}^{\widehat{rel}(X,Y,Z)}, \mathit{isSingleton}(X), \mathit{isSingleton}(Y).$$

By adding in Definition 4.24 the rule

- $m(l) \leftarrow m(B(r)), \tau_{\mathrm{III}}^{\widehat{rel}}(\hat{t}_1,\dots,\hat{t}_k), \bigwedge_{\hat{t}_i \in \mathit{arg}_i(\mathit{rel}) \setminus \mathit{arg}(l)} \mathit{isSingleton}(\hat{t}_i).$
  if $\mathit{arg}(l) \cap \mathit{arg}(\mathit{rel}) \neq \emptyset$ and $\mathit{arg}(l) \cap (\mathit{arg}(B(r)) \cap \mathit{arg}(\mathit{rel})) = \emptyset$.

the guess in (b) can be avoided, when all arguments of *rel* that are not involved with the head l are singleton clusters.

The use of abstract relations opens a wide range of possible applications, as it simplifies the use of a given program without preprocessing it to match the restrictions over the forms of the relations for the previous abstraction method.

4.4 Refinement by Debugging

Over-approximation of an answer set program unavoidably introduces spurious answer sets, which makes it necessary to have an abstraction refinement method. In CEGAR, the decision in a refinement step depends on the correctness checking of the spurious abstract solution, through which the problematic part of the abstraction is detected. Inspired from this, we define an alternative way of checking the correctness of an abstract answer set which can then be used in determining how the refinement should be made.

Correctness checking using constraints For an abstract answer set $\hat{I}$ to be spurious means that the original program does not contain an answer set that matches $\hat{I}$. In other words, querying the original program for a match to an abstract answer set $\hat{I}$ would return unsatisfiable if $\hat{I}$ is spurious.

Definition 4.25 (Query of an answer set)**.** *Given an abstract answer set $\widehat{I}$ and a mapping m, a* query $Q^m_{\widehat{I}}$ *for an answer set that matches $\widehat{I}$ is described by the following constraints.*

$$\bot \leftarrow \{\alpha \mid m(\alpha) = \hat{\alpha}\} \leq 0. \qquad \hat{\alpha} \in \widehat{I} \tag{4.45}$$

$$\bot \leftarrow \alpha. \qquad \hat{\alpha} \notin \widehat{I}, m(\alpha) = \hat{\alpha} \tag{4.46}$$

Here (4.45) ensures that a witnessing answer set I of Π (i.e., $m(I) = \widehat{I}$) contains for every abstract atom in $\widehat{I}$ some atom that is mapped to it. The constraint (4.46) ensures that I has no atom that is mapped to an abstract atom not in $\widehat{I}$.

The query $Q^m_{\widehat{I}}$ is slightly syntactically modified for the two different abstraction approaches:

- For the omission mapping m that omits A, the query $Q^m_{\widehat{I}}$ gets the following set of constraints for A:

$$Q^{\overline{A}}_{\hat{I}} = \{\bot \leftarrow not\ \alpha \mid \alpha \in \hat{I}\} \cup \{\bot \leftarrow \alpha \mid \alpha \in \overline{A} \setminus \hat{I}\}. \tag{4.47}$$

  in order to enforce an answer set that concides on the non-omitted atoms with $\hat{I}$.

- If m is a domain mapping, then the set of atoms $\widehat{I}$ in (4.45) and (4.46) is changed to $\widehat{I} \setminus \mathcal{T}_m$ to ensure that the check is done only for non-τ_ι abstract atoms.

The following is then easy to establish.

Proposition 4.29. *Suppose m is an abstraction mapping for a program Π.*

(i) If m is an omission mapping that omits a set A of atoms, then an abstract answer set $\hat{I} \in AS(omit(\Pi, A))$ is spurious iff $\Pi \cup Q_{\hat{I}}^{\overline{A}}$ is unsatisfiable.

(ii) If m is a domain mapping, then an abstract answer set $\widehat{I} \in AS(\Pi^m)$ is spurious iff $\Pi \cup Q_{\widehat{I}}^{m}$ is unsatisfiable.

Proof. (i) As $\hat{I}$ is spurious, there exists no $I \in AS(\Pi)$ such that $I = \hat{I} \cup X$ for $X \subseteq A$, i.e., there is no match of an original answer set for $\hat{I}$ with the atoms from $\overline{A}$ contained in it and excluded from it. $Q_{\hat{I}}^{\overline{A}}$ enforces such a match, thus returns unsatisfiability.

Having no match for $\hat{I}$ means that no extension of it will make an answer set of Π, thus $\hat{I}$ is spurious.

(ii) As $\hat{I}$ is spurious, there exists no $I \in AS(\Pi)$ such that $m(I) = \hat{I} \setminus \mathcal{T}_m$, i.e., there is no match of an original answer set I for $\hat{I}$ where the atoms in I can be mapped to the abstract atoms contained in $\hat{I} \setminus \mathcal{T}_m$ and the atoms not in I can be mapped to the abstract atoms not contained in $\hat{I} \setminus \mathcal{T}_m$. $Q_{\hat{I}}$ enforces such a match, thus returns unsatisfiability.

Having no match for $\hat{I}$ means that no original answer set can be mapped to it, thus $\hat{I}$ is spurious.

□

Debugging the correctness checking In this section, we show how to employ an ASP debugging approach in order to debug the inconsistency of the original program Π caused by checking a spurious answer set $\hat{I}$, referred to as *inconsistency of* Π *w.r.t.* $\hat{I}$, in order to get hints for refining the abstraction. Different from a usual ASP program debugging approach, we need to shift the focus of the debugging from "debugging the original program" to "debugging the inconsistency caused by the spurious answer set". Unfortunately an immediate application of the available ASP debugging tools is not possible. For our purposes, we make use of the meta-level debugging language introduced by [BGP+07] which is based on a tagging technique that allows one to control the building of answer sets and to manipulate the evaluation of the program.

The meta-program constructed by `spock` [BGP+07] introduces *tags* to control the building of answer sets. Given a program Π over $\mathcal{A}$ and a set $\mathcal{N}$ of names for all rules in Π, it creates an enriched alphabet $\mathcal{A}^+$ obtained from $\mathcal{A}$ by adding atoms such as $ap(n_r), bl(n_r), ok(n_r), ko(n_r)$ where $n_r \in \mathcal{N}$ for each $r \in \Pi$. The atoms $ap(n_r), bl(n_r)$ express whether a rule r is applicable or blocked, respectively, while $ok(n_r), ko(n_r)$ are used for manipulating the application of r.

For omission-based abstraction, we alter the meta-program so that through debugging, some of the atoms can be determined as *badly omitted* and thus should be added back in the refinement. As for domain abstraction, debugging the non-ground program has its own difficulties. The approach in [BGP$^+$07] is on the propositional level, thus can not be immediately applied. However, non-ground program debugging approaches such as [OPT10, DGM$^+$15] are not easily adjustable due to the need for shifting the focus towards debugging the correctness checking. In addition, such a meta-programming approach gives the power of choosing the importance of the type of violations which becomes useful in getting hints for the refinement of the domain abstraction.

4.4.1 Bad Omission of Ground Atoms

For omission abstraction, we describe the constructed debugging program that is able to shift the focus to the omitted atoms, and how we use it in determining bad omission of atoms.

Debugging Meta-Program

The (altered) meta-program with an enriched vocabulary $\mathcal{A}^+$ (by omitting the atoms $ok(n_r)$ of [BGP$^+$07], as they are not needed) that is created is as follows.

Definition 4.26. *Given* Π*, the program* $\mathcal{T}_{meta}[\Pi]$ *consists of the following rules for* $r \in \Pi, \alpha_1 \in B^+(r), \alpha_2 \in B^-(r)$*:*

$$\begin{aligned} H(r) &\leftarrow ap(n_r), not\ ko(n_r). \\ ap(n_r) &\leftarrow B(r). \\ bl(n_r) &\leftarrow not\ \alpha_1. \\ bl(n_r) &\leftarrow not\ not\ \alpha_2. \end{aligned}$$

Here the last rule uses double (nested) negation $not\ not\ \alpha_2$ [LTT99], which in the reduct w.r.t. an interpretation I is replaced by $\top$ if $I \models \alpha_2$ and by $\bot$ otherwise. The role of $ko(r)$ is to avoid the application of the rule $H(r) \leftarrow ap(r), not\ ko(r)$ if necessary. We use it for the rules that are changed due to some atom omitted from the body.

The following properties follow from [BGP$^+$07].

Proposition 4.30 ([BGP$^+$07])**.** *For a program* Π *over* $\mathcal{A}$*, and an answer set* X *of* $\mathcal{T}_{meta}[\Pi]$*, the following holds for any* $r \in \Pi$ *and* $a \in \mathcal{A}$*:*

1. $ap(n_r) \in X$ *iff* $r \in \Pi^X$ *iff* $bl(n_r) \notin X$*;*
2. *if* $a \in X$*, then* $ap(n_r) \in X$ *for some* $r \in def(a, \Pi)$*;*
3. *if* $a \notin X$*, then* $bl(n_r) \in X$ *for all* $r \in def(a, \Pi)$*.*

The relation between the auxiliary atoms and the original atoms are described below.

Theorem 4.31 ([BGP$^+$07])**.** *For a program* Π *over* $\mathcal{A}$*, the answer sets* $AS(\Pi)$ *and* $AS(\mathcal{T}_{meta}[\Pi])$ *satisfy the following conditions:*

1. *If $X \in AS(\Pi)$, then $X \cup \{ap(n_r) \mid r \in \Pi^X\} \cup \{bl(n_r) \mid r \in \Pi \setminus \Pi^X\} \in AS(\mathcal{T}_{meta}[\Pi])$.*
2. *If $Y \in AS(\mathcal{T}_{meta}[\Pi])$, then $Y \cap \mathcal{A} \in AS(\Pi)$.*

Abnormality atoms are introduced to indicate the cause of inconsistency: $ab_p(r)$ signals that rule r is falsified under some interpretation, $ab_c(\alpha)$ points out that α is true but has no support, and $ab_l(\alpha)$ indicates that α may be involved in a faulty loop (unfounded or odd).

Definition 4.27. *Given a program Π over $\mathcal{A}$, and a set $A \subseteq \mathcal{A}$ of atoms, the following additional meta-programs are constructed:*

1. *$\mathcal{T}_P[\Pi]$: for all $r \in \Pi$ with $B(r) \cap A \neq \emptyset$, $H(r) \nsubseteq A$ and $H(r) \neq \bot$:*

$$\begin{aligned} &ko(n_r).\\ &\{H(r)\} \leftarrow ap(n_r).\\ &ab_p(n_r) \leftarrow ap(n_r), not\ H(r). \end{aligned}$$

2. *$\mathcal{T}_C[\Pi, \mathcal{A}]$: for all $\alpha \in \mathcal{A} \setminus A$ with the defining rules $def(\alpha, \Pi) = \{r_1, ..., r_k\}$:*

$$\begin{aligned} \{\alpha\} &\leftarrow bl(n_{r_1}), ..., bl(n_{r_k}).\\ ab_c(\alpha) &\leftarrow \alpha, bl(n_{r_1}), ..., bl(n_{r_k}). \end{aligned}$$

3. *$\mathcal{T}_A[\mathcal{A}]$: for all $\alpha \in \mathcal{A}$:*

$$\begin{aligned} \{ab_l(\alpha)\} &\leftarrow not\ ab_c(\alpha).\\ \alpha &\leftarrow ab_l(\alpha). \end{aligned}$$

The difference from the abnormality atoms in [BGP$^+$07] is that the auxiliary atoms $ab_p(n_r)$ are only created for the rules which will be changed in the abstraction (but not omitted) due to A, denoted by $\Pi^c_A = \{r \mid r \in \Pi, B(r) \cap A \neq \emptyset, H(r) \nsubseteq A, H(r) \neq \bot\}$, and the auxiliary atoms $ab_c(a)$ are created only for the non-omitted atoms. This helps the search of a concrete interpretation for the partial/abstract interpretation by avoiding "bad" (i.e., non-supported) guesses of the omitted atoms. Notice that for the rules r_i with $H(r_i) = \alpha$ and empty body, we also put $bl(n_{r_i})$ so that $ab_c(\alpha)$ does not get determined, since one can always guess over α in Π.

Having $ab_l(\alpha)$ indicates that α is determined through a loop, but it does not necessarily show that the loop is unfounded (as described through *loop formulas* in [BGP$^+$07]). By checking whether α only gets support by itself, the unfoundedness can be caught. In some cases, α could be involved in an odd loop that was disregarded in the abstraction due to omission, which requires an additional check.

The basic properties of the abnormality atoms follows from [BGP$^+$07].

Proposition 4.32 ([BGP$^+$07]). *Consider a program Π over $\mathcal{A}$, a set $A \subseteq \mathcal{A}$ of atoms, and an answer set X of $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$.*

For each rule $r \in \Pi^c_A$:

1. *$ab_p(n_r) \in X$ iff $ap(n_r) \in X, bl(n_r) \notin X$, and $H(r) \notin X$;*
2. *$ab_p(n_r) \notin X$ if $ab_c(H(r)) \in X$ or $ab_l(H(r)) \in X$.*

Moreover, for every $a \in \mathcal{A} \setminus A$, it holds that:

1. *$ab_c(a) \in X$ and $ab_l(a) \notin X$ iff $a \in X$ and $(X \cap A) \not\models (\bigvee_{r \in def(a,\Pi)} B(r))$;*
2. *$ab_c(a) \notin X$ if $a \in X$ and $(X \cap A) \models (\bigvee_{r \in def(a,\Pi)} B(r))$;*
3. *$ab_c(a) \notin X$ and $ab_l(a) \notin X$ if $a \notin X$;*
4. *$ab_c(a) \notin X$ if $ab_l(a) \in X$.*

The next result shows that the answer sets of the translated program that are free from abnormality atoms correspond to the answer sets of the correctness checking of an abstract answer set $\hat{I}$ over Π using the query $Q^{\overline{A}}_{\hat{I}}$. We denote by $AB_A(\Pi)$ the set of abnormality atoms according to the omitted atoms A, i.e., $AB_A(\Pi) = \{ab_p(n_r) \mid r \in \Pi, B(r) \cap A \neq \emptyset, H(r) \nsubseteq A, H(r) \neq \bot\} \cup \{ab_c\}$.

Theorem 4.33. *For a program Π over $\mathcal{A}$, a set $A \subseteq \mathcal{A}$ of atoms and answer set $\hat{I}$ of $omit(\Pi, A)$, the following holds.*

1. *If X is an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$, then*
$$X \cup \{ko(n_r) \mid r \in \Pi^c_A\} \cup \{ap(n_r) \mid r \in \Pi^X\} \cup \{bl(n_r) \mid r \in \Pi \setminus \Pi^X\}$$
is an answer set of $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}] \cup Q^{\overline{A}}_{\hat{I}}$.
2. *If Y is an answer set of $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}] \cup Q^{\overline{A}}_{\hat{I}}$ such that $(Y \cap AB_A(\Pi)) = \emptyset$, then $(Y \cap \mathcal{A})$ is an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$.*

Proof. 1. Assume towards a contradiction that $X' = X \cup \{ko(n_r) \mid r \in \Pi^c_A\} \cup \{ap(n_r) \mid r \in \Pi^X\} \cup \{bl(n_r) \mid r \in \Pi \setminus \Pi^X\}$ is not answer set of $\Pi' \cup Q^{\overline{A}}_{\hat{I}}$, where $\Pi' = \mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$. This means that either (i) X' is not a model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$, or (ii) X' is not a minimal model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$.

(i) There is some rule $r \in (\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$ such that $X' \models B(r)$, but $X' \not\models H(r)$. We know that X is an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$, and thus $X \in AS(\Pi)$. By Theorem 4.31, we know that $X \cup \{ap(n_r) \mid r \in \Pi^X\} \cup \{bl(n_r) \mid r \in \Pi \setminus \Pi^X\}$ is an answer set of $\mathcal{T}_{meta}[\Pi]$. As X' contains no ab atoms, r can not be in $\mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$. So r must be in $Q^{\overline{A}}_{\hat{I}}$.

The rule r can be in two forms: (a) $\bot \leftarrow not\ \alpha.$ for some $\alpha \in \hat{I}$, or (b) $\bot \leftarrow \alpha.$ for some $\alpha \in \overline{A} \setminus \hat{I}$.

(a) As $X' \models B(r)$, then $\alpha \notin X'$ which means $\alpha \notin X$. However having $r \in (\Pi \cup Q^{\overline{A}}_{\hat{I}})^X$ contradicts that X is an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$.

(b) Similarly as (a), we reach a contradiction.

(ii) Let $Y' \subset X'$ be a model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$, for some $Y' = Y \cup \{ko(n_r) \mid r \in \Pi^c_A\} \cup \{ap(n_r) \mid r \in \Pi^X\} \cup \{bl(n_r) \mid r \in \Pi \setminus \Pi^X\}$. As the auxiliary atoms are fixed, $Y \subset Y'$ must hold. We claim that Y is then a model of $(\Pi \cup Q^{\overline{A}}_{\hat{I}})^X$, which is a contradiction. Assume Y is not such a model. Then there is a rule $r \in (\Pi \cup Q^{\overline{A}}_{\hat{I}})^X$ such that $Y \models B(r)$ but $Y \not\models H(r)$. There are two cases: (a) $r \in \Pi$, or (b) $r \in Q^{\overline{A}}_{\hat{I}}$.

(a) By definition of Y', this means that $Y' \models B(r)$ and $Y' \not\models H(r)$. However, this contradicts that Y' is a smaller model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$ than X' since $H(r)' \in Y'$.

(b) In both versions of r in $Q^{\overline{A}}_{\hat{I}}$, we get that $r \in (\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$ which contradicts that Y' is a model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^{X'}$.

2. Assume towards a contradiction that $(Y \cap \mathcal{A})$ is not an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. This means that either (i) $(Y \cap \mathcal{A})$ is not a model of $(\Pi \cup Q^{\overline{A}}_{\hat{I}})^{(Y \cap \mathcal{A})}$, or (ii) $(Y \cap \mathcal{A})$ is not a minimal model of $(\Pi \cup Q^{\overline{A}}_{\hat{I}})^{(Y \cap \mathcal{A})}$.

(i) There is some rule $r \in (\Pi \cup Q^{\overline{A}}_{\hat{I}})^{(Y \cap \mathcal{A})}$ such that $(Y \cap \mathcal{A}) \models B(r)$ but $(Y \cap \mathcal{A}) \not\models H(r)$. As we have $(Y \cap \mathcal{A}^+) \in AS(\mathcal{T}_{meta}[\Pi])$, by Theorem 4.31, we get $(Y \cap \mathcal{A}) \in AS(\Pi)$, thus r can not be in Π. However, $r \in Q^{\overline{A}}_{\hat{I}}$ also can not hold, since then r will be in $(Q^{\overline{A}}_{\hat{I}})^Y$ and we know that $Y \models Q^{\overline{A}}_{\hat{I}}$. Thus $(Y \cap \mathcal{A})$ must be a model of $(\Pi \cup Q^{\overline{A}}_{\hat{I}})^{(Y \cap \mathcal{A})}$.

(ii) Assume there exists some $Z \subset (Y \cap \mathcal{A})$ such that $Z \models (\Pi \cup Q^{\overline{A}}_{\hat{I}})^{(Y \cap \mathcal{A})}$. We claim that then $Z' = Z \cup \{ko(n_r) \mid r \in \Pi^c_A\} \cup \{ap(n_r) \mid r \in \Pi'^Y\} \cup \{bl(n_r) \mid r \in \Pi' \setminus \Pi'^Y\}$ is a model of $(\Pi' \cup Q^{\overline{A}}_{\hat{I}})^Y$, which achieves a contradiction. Now let us assume that this is not the case. Then there is some rule $r \in (\Pi' \cup Q^{\overline{A}}_{\hat{I}})^Y$ such that $Z' \models B(r)$ and $Z' \not\models H(r)$. The rule r can not be in $(Q^{\overline{A}}_{\hat{I}})^Y$, since it contradicts that $Y \models (Q^{\overline{A}}_{\hat{I}})^Y$. The rest of the cases for r also results in a contradiction.

(a) If $r \in \mathcal{T}_{meta}[\Pi]^Y$, then r can only be of form $H(r) \leftarrow ap(n_r), not\ ko(n_r)$, where $H(r) \neq \bot$. So we have $ap(n_r) \in Z'$, $ko(n_r) \notin Z'$ and $H(r) \notin Z'$. For rule r, rules of form 1 in Definition 4.27 are created in $\mathcal{T}_P[\Pi]$. However, since having $H(r) \notin Y$ causes to have the rule $ab_p(n_r) \leftarrow ap(n_r), not\ H(r)$ in $\mathcal{T}_P[\Pi]^Y$, $H(r) \in Y \setminus Z'$ should hold, which however contradicts that $Z \subset (Y \cap \mathcal{A})$, as then $H(r)' \in Z'$ would hold.

(b) If $r \in \mathcal{T}_P[\Pi]^Y$, then r can only be of form $H(r) \leftarrow ap(n_r)$. As $Z' \not\models H(r)$ we have $H(r)' \in Z'$ which contradicts that $Z \subset (Y \cap \mathcal{A})$. A similar contradiction is reached if $r \in \mathcal{T}_C[\Pi, \mathcal{A}]^Y$, since that means $\alpha \in Z'$ while $\alpha \notin Y$.

(c) Having $r \in \mathcal{T}_A[\mathcal{A}]^Y$ means that $Z' \not\models ab_l(\alpha)'$ for some $\alpha \in \mathcal{A}$, i.e., $ab_l(\alpha) \in Z'$, which contradicts $Y \cap AB_A(\Pi) = \emptyset$.

□

Determining Bad-Omission Atoms

Whether or not the program Π is consistent, our focus is on debugging the cause of inconsistency introduced through checking for a spurious answer set $\hat{I}$, i.e., evaluating the program $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. We reason about the inconsistency by inspecting the reason for having $\hat{I} \in AS(omit(\Pi, A))$ due to some modified rules.

Definition 4.28. *Let $r : \alpha \leftarrow B$ be a rule in Π such that $B \cap A \neq \emptyset$ and $\alpha \notin A$. The abstract rule $\hat{r} : \{\alpha\} \leftarrow m_A(B)$ in $omit(\Pi, A)$ introduces w.r.t. an abstract interpretation $\hat{I} \in AS(omit(\Pi, A))$*

(i) a spurious choice, *if $\hat{I} \models m_A(B)$ and $\hat{I} \models \overline{\alpha}$, i.e., $\hat{I} \not\models \alpha$, but some model I of $\Pi \setminus \{r\}$ exists s.t. $I|_{\overline{A}} = \hat{I}$ and $I \models B$.*

(ii) a spurious support, *if $\hat{I} \models m_A(B)$ and $\hat{I} \models \alpha$, but some model I of Π exists s.t. $I|_{\overline{A}} = \hat{I}$ and for all $r' \in def(\alpha, \Pi), I \not\models B(r')$.*

Any occurrence of the above cases shows that $\hat{I}$ is spurious. In case (i), due to $\hat{I} \not\models \alpha$, the rule r is not satisfied by I while I is a model of the remaining rules. In case (ii), an I that matches $\hat{I} \models \alpha$ does not give a supporting rule for α.

Definition 4.29. *Let $r : \alpha \leftarrow B$ be a rule in Π such that $B \cap A \neq \emptyset$. The abstract rule $\hat{r} = omit(r, A)$ introduces a* spurious loop-behavior *w.r.t. $\hat{I}$, if some model I of Π exists s.t. $I|_{\overline{A}} = \hat{I}$ and $I \models r$, but α is involved in a loop that is unfounded or is odd, due to some $\alpha' \in A \cap B$.*

The need for reasoning about the two possible faulty loop behaviors is shown by the following examples.

Example 4.49. Consider the programs Π_1, Π_2 and their abstractions $\widehat{\Pi}_1 = \widehat{\Pi}_{1\overline{\{a\}}}$, $\widehat{\Pi}_2 = \widehat{\Pi}_{2\overline{\{a,b\}}}$.

Π_1	$\widehat{\Pi}_1$	Π_2	$\widehat{\Pi}_2$
$r1 : a \leftarrow b.$		$r1 : a \leftarrow b.$	
$r2 : b \leftarrow not\ c, a.$	$\{b\} \leftarrow not\ c.$	$r2 : b \leftarrow not\ a, c.$	
		$r3 : c.$	$c.$

The program Π_1 has the single answer set $\emptyset$, and omitting a creates a spurious answer set $\{b\}$ disregarding that b is unfounded. The program Π_2 is unsatisfiable due to the odd loop of a and b. When both atoms are omitted, this loop is disregarded, which causes a spurious answer set $\{c\}$.

Bad omission of atoms are then defined as follows.

Definition 4.30 (bad omission atoms)**.** *An atom $\alpha \in A$ is a* bad omission *w.r.t. a spurious answer set $\hat{I}$ of $omit(\Pi, A)$, if some rule $r \in \Pi$ with $\alpha \in B(r)$ exists s.t. $\hat{r} = omit(r, A)$ introduces either (i) a spurious choice, (ii) a spurious support or (iii) a spurious loop-behavior w.r.t. $\hat{I}$.*

Intuitively, for case (i) of Definition 4.28, as $\overline{\alpha}$ was decided due to choice in $H(\hat{r})$, we infer that the omitted atom which caused r to become a choice rule is a bad omission. Also for case (ii), as α is decided with $\hat{I} \models B(\hat{r})$, we infer that the omitted atom that caused $B(r)$ to be modified is a bad omission. As for case (iii), it shows that the modification made on r (either omission or change to choice rule) ignores an unfoundedness or an odd loop. Case (i) also catches issues that arise due to omitting a constraint in the abstraction.

We now describe how we determine when an omitted atom is a bad omission.

Definition 4.31 (bad omission determining program)**.** *The bad omission determining program $\mathcal{T}_{bo}$ is constructed using the abnormality atoms obtained from $\mathcal{T}_P[\Pi]$, $\mathcal{T}_C[\Pi, \mathcal{A}]$ and $\mathcal{T}_A[\mathcal{A}]$ as follows:*

1. *A bad omission is inferred if the original rule is not satisfied, but applicable (and satisfied) in the abstract program:*

$$badomit(X, type1) \leftarrow ab_p(R), absAp(R), changed(R), omittedAtomFrom(X, R).$$

2. *A bad omission is inferred if the original rule is blocked and the head is unsupported, while it is applicable (and satisfied) in the abstract program:*

$$\begin{aligned} badomit(X, type2) \leftarrow\ & bl(R), head(R, H), ab_c(H), absAp(R), changed(R), \\ & omittedAtomFrom(X, R). \end{aligned}$$

3. *A bad omission is inferred in case there is unfoundedness or an involvement of an odd loop, via an omitted atom:*

$$\begin{aligned} faulty(X) &\leftarrow ab_l(X), inOddLoop(X, X_1), omittedAtom(X_1). \\ faulty(X) &\leftarrow ab_l(X), inPosLoop(X, X_1), omittedAtom(X_1). \\ badomit(X_1, type3) &\leftarrow faulty(X), head(R, X), modified(R), absAp(R), \\ & \quad omittedAtomFrom(X_1, R). \end{aligned}$$

where $absAp(r)$ is an auxiliary atom to keep track of which original rule becomes applicable with the remaining non-omitted atoms for the abstract interpretation, $changed(r)$ shows that r is changed to a choice rule in the abstraction, and $modified(r)$ shows that r is either changed or omitted in the abstraction, and $omittedAtomFrom(x, r)$ is an auxiliary atom that states which atoms are omitted from a rule.

For defining *type3*, we check for loops using the encoding in [Syr06] and determine *inOddLoop* and (newly defined) *inPosLoop* atoms of Π (see Figure 4.9).

Figure 4.9: Loop checking

$$
\begin{array}{ll}
\mathit{posEdge}(H,A) & \leftarrow \mathit{head}(R,H), \mathit{posBody}(R,A).\\
\mathit{negEdge}(H,B) & \leftarrow \mathit{head}(R,H), \mathit{negBody}(R,B).\\
\mathit{even}(X,Y) & \leftarrow \mathit{posEdge}(X,Y).\\
\mathit{odd}(X,Y) & \leftarrow \mathit{negEdge}(X,Y).\\
\mathit{even}(X,Z) & \leftarrow \mathit{posEdge}(X,Y), \mathit{even}(Y,Z), \mathit{atom}(Z).\\
\mathit{odd}(X,Z) & \leftarrow \mathit{posEdge}(X,Y), \mathit{odd}(Y,Z), \mathit{atom}(Z).\\
\mathit{odd}(X,Z) & \leftarrow \mathit{negEdge}(X,Y), \mathit{even}(Y,Z), \mathit{atom}(Z).\\
\mathit{even}(X,Z) & \leftarrow \mathit{negEdge}(X,Y), \mathit{odd}(Y,Z), \mathit{atom}(Z).\\
\mathit{inOddLoop}(X,Y) & \leftarrow \mathit{odd}(X,Y), \mathit{even}(Y,X).\\
\\
\mathit{posDep}(X,Y) & \leftarrow \mathit{posEdge}(X,Y).\\
\mathit{posDep}(X,Z) & \leftarrow \mathit{posEdge}(X,Y), \mathit{posDep}(Y,Z), \mathit{atom}(Z).\\
\mathit{inPosLoop}(X,Y) & \leftarrow \mathit{posDep}(X,Y), \mathit{posDep}(Y,X).
\end{array}
$$

The cases for *type2* and *type3* introduce as bad omissions the omitted atoms of all the rules that add to $ab_c(H)$ being true, or of all rules that have X in the head for $ab_l(X)$, respectively. Modifying *badomit* determination to have a choice over such rules to be refined (and their omitted atoms to be *badomit*) and minimizing the number of *badomit* atoms reduces the number of added back atoms in a refinement step, at the cost of increasing the search space.

In order to avoid the guesses of ab_l for omitted atoms even if there is no faulty loop behavior related with them (i.e., this is not the cause of inconsistency of $\hat{I}$), we add the constraint $\leftarrow ab_l(X), not\ someFaulty.$ with the auxiliary definition $someFaulty \leftarrow faulty(X)$.

With all this in place, the program for debugging a spurious answer set is composed as follows.

Definition 4.32 (spurious answer set debugging program)**.** *For an abstract answer set $\hat{I}$, we denote by $\mathcal{T}[\Pi,\hat{I}]$ the program $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi,\mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}] \cup \mathcal{T}_{bo} \cup Q^{\overline{A}}_{\hat{I}}$.*

We denote by $\mathcal{A}^*_A$ the vocabulary of $\mathcal{T}[\Pi,\hat{I}]$ which consists of $\mathcal{A}^+ \cup AB_A(\Pi) \cup HB_{\mathcal{T}_{bo}}$.

From the answer sets of $\mathcal{T}[\Pi,\hat{I}]$, we can see bad omissions and their types.

Example 4.50. For the following program Π, $\hat{I} = \{b\}$ is a spurious answer set of the abstraction for $A = \{a,d\}$:

Figure 4.10: Meta-programs for Example 4.50

$$
\begin{array}{ll}
c & \leftarrow ap(r1), not\ ko(r1).\\
ap(r1) & \leftarrow not\ d.\\
bl(r1) & \leftarrow not\ not\ d.\\
\\
d & \leftarrow ap(r2), not\ ko(r2).\\
ap(r2) & \leftarrow not\ c.\\
bl(r2) & \leftarrow not\ not\ c.\\
\\
a & \leftarrow ap(r3), not\ ko(r3).\\
ap(r3) & \leftarrow not\ d, c.\\
bl(r3) & \leftarrow not\ c.\\
bl(r3) & \leftarrow not\ not\ d.\\
\\
b & \leftarrow ap(r4), not\ ko(r4).\\
ap(r4) & \leftarrow a.\\
bl(r4) & \leftarrow not\ a.\\
\bot & \leftarrow falsum.
\end{array}
$$

(a) $\mathcal{T}_{meta}[\Pi]$

$$
\begin{array}{ll}
ko(r1). & \\
\{c\} & \leftarrow ap(r1).\\
ab_p(r1) & \leftarrow ap(r1), not\ c.\\
ko(r4). & \\
\{b\} & \leftarrow ap(r4).\\
ab_p(r4) & \leftarrow ap(r4), not\ b.\\
\\
\{b\} & \leftarrow bl(r4).\\
ab_c(b) & \leftarrow bl(r4), b.\\
\{c\} & \leftarrow bl(r1).\\
ab_c(c) & \leftarrow bl(r1), c.\\
\\
\{ab_l(a)\} & \leftarrow not\ ab_c(a).\\
a & \leftarrow ab_l(a).\\
\{ab_l(b)\} & \leftarrow not\ ab_c(b).\\
b & \leftarrow ab_l(b).\\
\{ab_l(c)\} & \leftarrow not\ ab_c(c).\\
c & \leftarrow ab_l(c).\\
\{ab_l(d)\} & \leftarrow not\ ab_c(d).\\
d & \leftarrow ab_l(d).
\end{array}
$$

(b) $\mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$

Π	$\widehat{\Pi}_{\overline{a,d}}$
$r1 : c \leftarrow not\ d.$	$\{c\}.$
$r2 : d \leftarrow not\ c.$	
$r3 : a \leftarrow not\ d, c.$	
$r4 : b \leftarrow a.$	$\{b\}.$

Figure 4.10 shows the constructed meta programs of Π. $\mathcal{T}[\Pi, \hat{I}]$ gives the answer set that contains $\{ap(r2), bl(r1), bl(r4), bl(r3), ab_c(b), badomit(a, type2)\}$. The answer set shows that since $c \notin \hat{I}$, the rule $r1$ gets blocked and the rule $r2$ becomes applicable (which means d is derived). However as the rule $r3$ is blocked, a can not be derived, and thus the occurrence of b is unsupported in Π (w.r.t $\{b, d\}$), which was avoided in $\widehat{\Pi}_{\overline{a,d}}$ due to (badly) omitting a from the body of $r4$.

The next example shows the need for reasoning about the disregarded positive loops and odd loops, due to omission.

Figure 4.11: Meta-programs for Example 4.51

$$\begin{array}{ll}
a & \leftarrow \mathit{ap}(r1), \mathit{not}\ \mathit{ko}(r1).\\
\mathit{ap}(r1) & \leftarrow b.\\
\mathit{bl}(r1) & \leftarrow \mathit{not}\ b.\\
 & \\
b & \leftarrow \mathit{ap}(r2), \mathit{not}\ \mathit{ko}(r2).\\
\mathit{ap}(r2) & \leftarrow \mathit{not}\ c, a.\\
\mathit{bl}(r2) & \leftarrow \mathit{not}\ a.\\
\mathit{bl}(r2) & \leftarrow \mathit{not}\ \mathit{not}\ c.\\
 & \\
\mathit{ko}(r2). & \\
\{b\} & \leftarrow \mathit{ap}(r2).\\
\mathit{ab}_p(r2) & \leftarrow \mathit{ap}(r2), \mathit{not}\ b.\\
 & \\
\{b\} & \leftarrow \mathit{bl}(r2).\\
\mathit{ab}_c(b) & \leftarrow \mathit{bl}(r2), b.\\
\{c\}. & \\
\mathit{ab}_c(c) & \leftarrow c.\\
 & \\
\{\mathit{ab}_l(a)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(a).\\
a & \leftarrow \mathit{ab}_l(a).\\
\{\mathit{ab}_l(b)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(b).\\
b & \leftarrow \mathit{ab}_l(b).\\
\{\mathit{ab}_l(c)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(c).\\
c & \leftarrow \mathit{ab}_l(c).
\end{array}$$

(a) $\mathcal{T}_{meta}[\Pi_1] \cup \mathcal{T}_P[\Pi_1] \cup \mathcal{T}_C[\Pi_1, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$

$$\begin{array}{ll}
a & \leftarrow \mathit{ap}(r1), \mathit{not}\ \mathit{ko}(r1).\\
\mathit{ap}(r1) & \leftarrow b.\\
\mathit{bl}(r1) & \leftarrow \mathit{not}\ b.\\
 & \\
b & \leftarrow \mathit{ap}(r2), \mathit{not}\ \mathit{ko}(r2).\\
\mathit{ap}(r2) & \leftarrow \mathit{not}\ a, c.\\
\mathit{bl}(r2) & \leftarrow \mathit{not}\ c.\\
\mathit{bl}(r2) & \leftarrow \mathit{not}\ \mathit{not}\ a.\\
 & \\
c & \leftarrow \mathit{ap}(r3), \mathit{not}\ \mathit{ko}(r3).\\
\mathit{ap}(r3). & \\
\bot & \leftarrow \mathit{falsum}.\\
 & \\
\{c\} & \leftarrow \mathit{bl}(r3).\\
\mathit{ab}_c(c) & \leftarrow \mathit{bl}(r3), c.\\
 & \\
\{\mathit{ab}_l(a)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(a).\\
a & \leftarrow \mathit{ab}_l(a).\\
\{\mathit{ab}_l(b)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(b).\\
b & \leftarrow \mathit{ab}_l(b).\\
\{\mathit{ab}_l(c)\} & \leftarrow \mathit{not}\ \mathit{ab}_c(c).\\
c & \leftarrow \mathit{ab}_l(c).
\end{array}$$

(b) $\mathcal{T}_{meta}[\Pi_2] \cup \mathcal{T}_P[\Pi_2] \cup \mathcal{T}_C[\Pi_2, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$

Example 4.51 (Example 4.49 ctd)**.** Recall that the program Π_1 has an unfounded loop between a and b, and the abstraction $\widehat{\Pi}_1 = \widehat{\Pi}_{1\overline{\{a\}}}$ has the spurious answer set $\{b\}$. Figure 4.11a shows the constructed meta-programs $\mathcal{T}_{meta}[\Pi_1] \cup \mathcal{T}_P[\Pi_1] \cup \mathcal{T}_C[\Pi_1, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$. The overall program $\mathcal{T}[\Pi_1, \{b\}]$ yields $\mathit{inPosLoop}(b, a)$, $\mathit{ap}(r1)$, $\mathit{ap}(r2)$, $\mathit{ab}_l(b)$, $\mathit{badomit}(a, \mathit{type3})$. Omitting from the program Π_2 the loop atoms a, b causes the spurious answer set $\{c\}$. Figure 4.11b shows the constructed meta-programs for Π_2. Accordingly, $\mathcal{T}[\Pi_2, \{c\}]$ yields $\mathit{ap}(r3)$, $\mathit{inOddLoop}(b, a)$, $\mathit{inOddLoop}(a, b)$, $\mathit{ab}_l(b)$, $\mathit{ap}(r1)$, $\mathit{bl}(r2)$, $\mathit{badomit}(a, \mathit{type3})$, $\mathit{badomit}(b, \mathit{type3})$, as desired.

The program $\mathcal{T}[\Pi, \hat{I}]$ always returns an answer set for $\hat{I}$, due to relaxing Π by tolerating abnormalities that arise from checking the concreteness for $\hat{I}$.

Proposition 4.34. *For each abstract answer set $\hat{I}$ of omit(Π, A), the program $\mathcal{T}[\Pi, \hat{I}]$ has an answer set I such that $I \cap \overline{A} = \hat{I}$.*

Proof. Let X be an interpretation over $\mathcal{A}^*_A$ with $X \cap \overline{A} = \hat{I}$. We will show that with the help of the auxiliary rules/atoms, some interpretation X' which is a minimal model of $(\mathcal{T}[\Pi, \hat{I}])^{X'}$ can be reached starting from X. We have the cases (i) $X \not\models (\mathcal{T}[\Pi, \hat{I}])^X$, and (ii) $X \models (\mathcal{T}[\Pi, \hat{I}])^X$.

(i) Let r be a ground unsatisfied rule in $(\mathcal{T}[\Pi, \hat{I}])^X$. This means that $X \models B(r)$ and $X \not\models H(r)$. We show that X can be changed to some interpretation X' that avoids the condition for X not satisfying r. First, observe that since $X \cap \overline{A} = \hat{I}$ we have $X \models (Q^{\overline{A}}_{\hat{I}})^X$.

(a) Assume r is in $\mathcal{T}'^X = (\mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}] \cup \mathcal{T}_{bo})^X$. The rule r can not be an instantiation of the choice rules in $\mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}] \cup \mathcal{T}_A[\mathcal{A}]$, as it would be instantiated for X, and hence be satisfied. Thus r can either (a-1) have $H(r) \in AB_A(\Pi)$ and be in $(\mathcal{T}_P[\Pi] \cup \mathcal{T}_C[\Pi, \mathcal{A}])^X$, (a-2) have $H(r) = ko(n_{r'})$ for some $r' \in \Pi$ and be in $\mathcal{T}_P[\Pi]^X$, (a-3) be in $(\mathcal{T}_{bo})^X$, or (a-4) be of form $\alpha \leftarrow ab_l(\alpha)$ for some $\alpha \in \mathcal{A}$ in $\mathcal{T}_A[\mathcal{A}])^X$. For cases (a-1),(a-2),(a-3) we can construct $X' = X \cup \{H(r)\}$ so that $X' \models H(r)$ and the reduct $\mathcal{T}'^{X'}$ will not have further rules.

As for case (a-4), if $\alpha \in \overline{A}$, this means α is determined to be false by $\hat{I}$, so we construct $X' = (X \setminus \{ab_l(\alpha)\}) \cup \{ab_l(\alpha)'\}$ so that r does not occur in $\mathcal{T}'^{X'}$. If $\alpha \notin \overline{A}$, then we construct $X' = X \cup \{\alpha\}$.

(b) Assume r is in $(\mathcal{T}_{meta}[\Pi])^X$.

(b-1) If the rule is of form $H(r') \leftarrow ap(n_{r'}), not\ ko(n_{r'})$. where $B(r') \cap A \neq \emptyset$, $H(r') \not\subseteq A$ and $H(r') \neq \bot$ for some $r' \in \Pi$, this means $ko(n_{r'}) \notin X$. However, rules for r' are added in $\mathcal{T}_P[\Pi]$ which uses the rule $ko(n_{r'})$. to deactivate the meta-rule in $\mathcal{T}_{meta}[\Pi]$, which is then also unsatisfied in the reduct $(\mathcal{T}_P[\Pi])^X$. So we construct $X' = X \setminus \cup\{ko(n_{r'})\}$. Thus, the rule r does not appear in $(\mathcal{T}_{meta}[\Pi])^{X'}$.

(b-2) Let the rule be of form $H(r') \leftarrow ap(n_{r'}), not\ ko(n_{r'})$. for some $r' \in \Pi$ different than the one in (b-1). We have $ap(n_{r'}) \in X$. Assume $X \models B(r')$. If $H(r') = \bot$, then we must have $B(r') \cap A \neq \emptyset$, since otherwise r' would occur in $omit(\Pi, A)$ and contradicts that $\hat{I}$ is an answer set. Then if $B^-(r') \cap A \neq \emptyset$, we construct $X' = X \cup \{\alpha\}$ for some $\alpha \in B^-(r') \cap A$; otherwise, we $X' = X \setminus \{\alpha\}$ for some $\alpha \in B^+(r') \cap A$. If $X \not\models B(r')$, we construct $X' = X \setminus ap(n_{r'})$.

(b-3) If r is of form $ap(n'_r) \leftarrow B(r')$ for some $r' \in \Pi$, then we construct $X' = X \cup \{ap(n'_r)\}$. If r is of the remaining forms with $bl(n'_r)$, we construct $X' = X \cup \{bl(n'_r)\}$

(ii) If X is a minimal model, then X is an answer set of $\mathcal{T}[\Pi, \hat{I}]$, which achieves the result. We assume this is not the case, and that there exists $Y \subset X$ such that $Y \models (\mathcal{T}[\Pi, \hat{I}])^X$. So, we have $Y \cap \overline{A} = \hat{I}$. Thus, there exists $\alpha \in X \setminus Y$ such that

$\alpha \in \mathcal{A}^*_A \setminus \overline{A}$. Assume $\alpha \in A$. Then $ap(n_r) \notin Y$ should hold for all $r \in def(\alpha, \Pi)$ (to satisfy the corresponding meta-rules in $\mathcal{T}_{meta}[\Pi]^X$). Also $(\mathcal{T}_A[\mathcal{A}])^Y$ does not contain the rule $\alpha \leftarrow ab_l(\alpha)$ (since otherwise it would not be satisfied). So we have $ab_l(\alpha) \notin Y$, but then we get $ab_l(\alpha)' \in Y \setminus X$ which is a contradiction.

If the case $\alpha \in \mathcal{A}^*_A \setminus \mathcal{A}$ occurs, then we pick Y as the interpretation. If $\alpha \in AB_A(\Pi) \cup HB_{\mathcal{T}_{bo}}$, then the reduct $(\mathcal{T}[\Pi, \hat{I}])^Y$ will not have further rules. If $\alpha \in \mathcal{A}^+ \setminus \mathcal{A}$, then we apply the above reasoning for Y. When we recursively continue with this reasoning, eventually, this case will not be applicable, and thus we would construct a minimal model.

□

The following result shows that $\mathcal{T}[\Pi, \hat{I}]$ flags in its answer sets always bad omission of atoms, which can be utilized for refinement.

Proposition 4.35. *If the abstract answer set $\hat{I}$ is spurious, then for every answer set $S \in AS(\mathcal{T}[\Pi, \hat{I}])$, $badomit(\alpha, i) \in S$ for some $\alpha \in A$ and $i \in \{type1, type2, type3\}$.*

Proof. Note that by Proposition 4.29 we know that the program $\Pi \cup Q^{\overline{A}}_{\hat{I}}$ is unsatisfiable. Thus $S \cap \mathcal{A}$ is not an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. By Theorem 4.33, we know that having $S \cap AB_A(\Pi) = \emptyset$ contradicts with the spuriousness of $\hat{I}$. Thus, we have $S \cap AB_A(\Pi) \neq \emptyset$.

(a) If $ab_p(n_r) \in S$ for some rule $r \in \Pi$, then the rule $ap(n_r) \leftarrow ap(n_r), not\ H(r)$ is in $(\mathcal{T}[\Pi, \hat{I}])^S$, i.e., $ap(n_r) \in S$ and $H(r) \notin S$. This unsatisfied rule is then a reason for $S \cap \mathcal{A}$ not being an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. Since $B(r) \cap A \neq \emptyset$, for the changed rule $\hat{r} \in omit(\Pi, A)$, we have $B(\hat{r}) = B(r) \setminus A$ and thus $S \models B(\hat{r})$, i.e., the auxiliary atom $absAp(n_r)$ is true. Then by definition, $badomit(\alpha, type1) \in S$ for $\alpha \in B(r) \cap A$.

(b) If $ab_c(\alpha) \in S$ for some atom $\alpha \in \overline{A}$, then the rule $ab_c(\alpha) \leftarrow \alpha, bl(n_{r_1}), \dots, bl(n_{r_k})$, for $def(\alpha, \Pi) = \{r_1, \dots, r_k\}$, is in $(\mathcal{T}[\Pi, \hat{I}])^S$, i.e., $\alpha \in S$ and $bl(n_{r_1}), \dots, bl(n_{r_k}) \in S$. This unsupported atom α is then a reason for $S \cap \mathcal{A}$ not being an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$. We know that α is also in $\hat{I}$, due to $S \models (Q^{\overline{A}}_{\hat{I}})^S$. This means that the abstraction $\hat{r}_i$ of some rule r_i is in $omit(\Pi, A)^{\hat{I}}$, i.e., the auxiliary atom $absAp(n_{r_i})$ is true, while $bl(n_{r_i}) \in S$. Thus $B(r) \cap A \neq \emptyset$ must hold. Then by definition, $badomit(\alpha', type2) \in S$ for $\alpha' \in B(r) \cap A$.

(c) If $ab_l(\alpha) \in S$ for some atom $\alpha \in \mathcal{A}$, then $\alpha \in S$ and $ab_c(\alpha) \notin S$. Assume that $S \cap \mathcal{A}$ is not an answer set of $\Pi \cup Q^{\overline{A}}_{\hat{I}}$ due to an odd or unfounded loop L containing α.

We distinguish the cases for α. Let $\alpha \in \overline{A}$. As $ab_c(\alpha) \notin S$, for some rule r_i in $def(\alpha, \Pi)$ we have $bl(n_{r_i}) \notin S$, i.e., $ap(n_{r_i}) \in S$ and thus $S \models B(r_i)$. We know that α is also in $\hat{I}$, and since $\hat{I}$ is an answer set of $omit(\Pi, A)$, we conclude that there exists some $\alpha' \in B(r) \cap A$ such that $\alpha' \in L$. This way, for the abstract rule $\hat{r}_i$ we have $\hat{I} \models B(\hat{r}_i)$, i.e., the auxiliary atom $absAp(n_{r_i})$ is true. By definition, we get $badomit(\alpha', type3) \in S$.

Now, let $\alpha \in A$. Then the rule r_i in $\mathit{def}(\alpha, \Pi)$ with $B(n_{r_i}) \cap L \neq \emptyset$ is omitted. Say $\alpha' \in B(n_{r_i}) \cap L$. If $\alpha' \in \overline{A}$, by the above reasoning we get $\mathit{badomit}(\alpha'', \mathit{type3}) \in S$ for some $\alpha'' \in B(r') \cap A$, for $r' \in \mathit{def}(\alpha', \Pi)$. If $\alpha' \in A$, we recursively do the same reasoning. Since L is a loop, eventually we reach a rule $r_i'^m$ with $\alpha \in B(r_i'^m)$, and thus get $\mathit{badomit}(\alpha, \mathit{type3}) \in S$.

If there is no such loop L with $\alpha \in L$, then cases (a) or (b) applies for $S \cap \mathcal{A}$ not being an answer set of $\Pi \cup Q_{\hat{I}}^{\overline{A}}$.

□

The badly omitted atoms $A_o \subseteq A$ w.r.t. a spurious $\hat{I} \in AS(\mathit{omit}(\Pi, A))$ are added back to refine m_A. If $\hat{I}$ still occurs in the refined program $\mathit{omit}(\Pi, A \setminus A_o)$, i.e., some $\hat{I}' {\in} AS(\mathit{omit}(\Pi, A \setminus A_o))$ with $\hat{I}'|_{\overline{A}} {=} \hat{I}$ exists, then $\mathcal{T}[\Pi, \hat{I}']$ finds another possible bad omission. In the worst case, all omitted atoms A are put back to eliminate $\hat{I}$.

Let $A_0 = A$ and $A_{i+1} = A_i \setminus BA_i$, where BA_i are the badly omitted atoms for $\mathit{omit}(\Pi, A_i)$ w.r.t. an abstract answer set $\hat{I}_i$ of $\mathit{omit}(\Pi, A_i)$.

Corollary 4.36. *For a spurious answer set $\hat{I}$, after at most $k = |A|$ steps, $\mathit{omit}(\Pi, A_k)$ will have no answer set that matches $\hat{I}$.*

Adding back a badly omitted atom may cause a previously omitted rule to appear as a changed rule in the refined program. Due to this choice rule, the spurious answer set might not get eliminated. To give a (better) upper bound for the number of required iterations in order to eliminate a spurious answer set, a trace of the dependencies among the omitted rules is needed.

The *rule dependency graph* of Π, denoted $G_\Pi^{rule} = (V, E)$, shows the positive/negative dependencies similarly as in G_Π, but at a rule-level, where the vertices V are the rules $r \in \Pi$ and an edge from r to r' exists in E if $H(r') \in B^{\pm}(r)$ holds, which is called negative if $H(r') \in B^-(r)$ and positive otherwise. For a set A of atoms, let n_A denote the maximum length of a (non-cyclic) path in G_Π^{rule} from some rule r with $B(r) \cap A \neq \emptyset$ backwards through rules r' with $H(r') \in A$. The number n_A shows the maximum level of direct or indirect dependency between omitted atoms and their respective rules.

Proposition 4.37. *Given a program Π, a set A of atoms, and a spurious $\hat{I} \in AS(\mathit{omit}(\Pi, A))$, $\mathit{omit}(\Pi, A_i)$ will have no abstract answer set matching $\hat{I}$ after at most $i = n_A$ iterations.*

Proof. Let r_0 be a rule with $\alpha \in B(r_0) \cap A$ that is changed to a choice rule due to m_A. Let $r_0, r_1, \dots, r_{n_A}$ be a dependency path in G_Π^{rule} where $H(r_i) \cap A \neq \emptyset$ and $B(r_i) \cap A \neq \emptyset$, $0 \leq i < n_A$. Let $\hat{I} {\in} AS(\mathit{omit}(\Pi, A))$, assume r_0 has spurious behavior w.r.t. $\hat{I}$, and w.l.o.g. assume $\hat{I} \models B(r_i) \setminus A$ for all $i {\leq} n_A$.

Due to inconsistency via r_0, $\mathit{badomit}(\alpha, i) \in AS(\mathcal{T}[\Pi, \hat{I}])$ for some $i \in \{\mathit{type1}, \mathit{type2}, \mathit{type3}\}$. For $A' {=} A \setminus \{\alpha\}$, $\mathit{omit}(r_0, A')$ is unchanged, while $\mathit{omit}(r_1, A')$ becomes a choice rule (with

n_A-1 dependencies left). Thus, some $I' \in AS(omit(\Pi, A'))$ with $I'|_{\overline{A}} = \hat{I}$ can still exist. Since r_1 introduces spuriousness w.r.t. I', there is $badomit(\alpha') \in AS(\mathcal{T}[\Pi, I'])$ for $\alpha' \in B(r_1) \cap A'$.

By iterating this process n_A times, all omitted rules on which r_0 depends are traced and eventually no abstract answer set matching $\hat{I}$ occurs. □

We remark that in case more than one dependency path $r_0, \ldots, r_{n_A}$ with several rules causing inconsistencies exists, the returned set of *badomit*s from $\mathcal{T}[\Pi, \hat{I}]$ allows one to refine the rules in parallel.

Recall that Proposition 4.8 ensures that adding back further omitted atoms will not reintroduce a spurious answer set. Further heuristics on the determination of bad omission atoms can be applied in order to ensure that a spurious answer set is eliminated in one step.

4.4.2 Non-Ground Spuriousness

Debugging non-ground programs is not as straightforward as in the propositional case. Moreover, there is the additional need to debug the checking for an original answer set that can be mapped to the given abstract answer set. After discussing the inapplicability of the available non-ground ASP program debuggers, we describe our approach to help with debugging the checking and determining a refinement of a mapping.

On using available debuggers Debugging non-ground ASP programs through a meta-programming [GPST08] approach has been studied by [OPT10], with the drawback of considering all possible explanations for a given answer set I not being an answer set of the program Π. For the given input I, in order to prove that I is not an answer set of Π, the debugging considers many possible guesses of variable assignments that matches I with a faulty behavior. In our case, the input I is an abstract answer set stating that there should be some original answer set I' of Π such that each atom in I' can be mapped to some abstract atom $\hat{\alpha}$ in I. This adds an additional guess of some original atom that could be mapped to $\hat{\alpha}$. However, since the debugging aims at showing that I is not an answer set of Π, when this additional guessing comes into play, the debugging makes guesses of original atoms to create some faulty behavior for I even if these atoms do not even occur in an original answer set. Thus, an immediate application of the meta-programming approach is not possible.

In order to use the available non-ground debugging tools off-the-shelf, one possibility is to first guess all possible combinations of the original atoms to match the abstract I, and then separately debug each of them. If I is in fact spurious, this will be caught as all possible guesses would return some inconsistency. If I is concrete, then at some point some guess will correspond to an original answer set, with no inconsistency. However, this approach is too cumbersome, as there can be many possible concrete guesses for an abstract I and checking each of them one by one until a concrete one is found (if exists) is highly inefficient.

Our approach to debugging As the existing non-ground debugging tools are not immediately applicable, we approach the debugging of the unsatisfiability of $\Pi \cup Q^m_{\widehat{I}}$ for a spurious abstract answer set $\widehat{I}$ by following the debugging approach based on [BGP$^+$07] from the previous section.

- As a first step, we consider a simplified debugging approach inspired from the *ko* atoms of [BGP$^+$07], which is based on detecting the rules that have to be deactivated in order to keep the satisfiability while checking the concreteness of an abstract answer set $\widehat{I}$, in case it is spurious. We discuss how the debugging information can be used for refinement, which later in Section 5.2.2 we show on some benchmark problems.
- As the naive debugging can not address all debugging cases, we also show an extension of the refinement method by lifting the `spock` [BGP$^+$07] debugging approach to the non-ground case, confining to tight programs (i.e., we omit unfounded loop checking).

When demonstrating the different debugging approaches, we use a non-ground version of $Q^m_{\widehat{I}}$.

Definition 4.33 (Non-ground query)**.** *Given an abstract answer set $\hat{I}$ and a mapping m expressed as a set of facts of form $m(x,a)$ (where $m(x) = a$), a* (non-ground) query *for an answer set that matches $\hat{I}$ is described by the following constraints*

$$\bot \leftarrow in(\hat{\alpha}), \{\alpha : m(X_1, \hat{X}_1), \dots, m(X_k, \hat{X}_k)\} \leq 0. \tag{4.48}$$

$$\bot \leftarrow \alpha, not\ in(\hat{\alpha}), m(X_1, \hat{X}_1), \dots, m(X_k, \hat{X}_k). \tag{4.49}$$

where $\alpha = p(X_1, ..., X_k)$ and $\hat{\alpha} = p(\hat{X}_1, ..., \hat{X}_k)$, and $m(X_i, \hat{X}_i)$ expresses the abstract mapping, plus the facts

$$in(\hat{\alpha}).,\quad \hat{\alpha} \in \widehat{I} \setminus \mathcal{T}_m. \tag{4.50}$$

Example 4.52 (Example 4.15 ctd)**.** For the program Π and the mapping $m = \{\{1,2,3,4,5\}/k\}$ expressed with facts $\{m(1,k), m(2,k), m(3,k), m(4,k), m(5,k)$, the abstract program Π^m has an answer set $\widehat{I} = \{a(k), c(k)\}$. The query $Q^m_{\widehat{I}}$ is as follows:

$$\begin{array}{ll}
\bot \leftarrow not\ in(d(A_1)), d(X_1), m(X_1, A_1). & \bot \leftarrow in(d(A_1)), \{d(X_1) : m(X_1, A_1)\} \leq 0. \\
\bot \leftarrow not\ in(c(A_1)), c(X_1), m(X_1, A_1). & \bot \leftarrow in(c(A_1)), \{c(X_1) : m(X_1, A_1)\} \leq 0. \\
\bot \leftarrow not\ in(a(A_1)), a(X_1), m(X_1, A_1). & \bot \leftarrow in(a(A_1)), \{a(X_1) : m(X_1, A_1)\} \leq 0. \\
\bot \leftarrow not\ in(e(A_1)), e(X_1), m(X_1, A_1). & \bot \leftarrow in(e(A_1)), \{e(X_1) : m(X_1, A_1)\} \leq 0. \\
\bot \leftarrow not\ in(b(A_1, A_2)), b(X_1, X_2), m(X_1, A_1), m(X_2, A_2). &
\end{array}$$

$\bot \leftarrow in(b(A_1, A_2)), \{b(X_1, X_2) : m(X_1, A_1), m(X_2, A_2)\} \leq 0.$
$in(a(k)). \qquad in(c(k)).$

Naive Non-ground Debugging

We approach the debugging of $\Pi \cup Q^m_{\widehat{I}}$ through catching the rules that need to be deactivated in order to keep satisfiability of the correctness checking, in case $\hat{I}$ is spurious. For this, we add abnormality atoms, $ab_{n_r}(t_1, \dots, t_n)$, in the rules of Π that contain arguments from the domain.

Definition 4.34. *Let Π be a non-ground program and a set $\mathcal{N}$ of names for all non-ground rules in Π. For each $r \in \Pi$, the program Π_{ab} consists of the following rule:*

$$r' = \begin{cases} H(r) \leftarrow B(r), not\ ab_{n_r}(c_1, \dots, c_n) & B(r) \neq 0 \land arg(H(r)) = \{c_1, \dots, c_n\} \neq \emptyset \\ H(r) \leftarrow B(r), not\ ab_{n_r}(c_1, \dots, c_n) & H(r) = \bot \land arg(B(r)) = \{c_1, \dots, c_n\} \neq \emptyset \\ r & otherwise \end{cases}$$

where $n_r \in \mathcal{N}$. Additionally, for each $ab_{n_r}(c_1, \dots, c_n)$, Π_{ab} contains:

$$\{ab_{n_r}(c_1, \dots, c_n)\}.$$

Having additional rules to assign costs for the existence of the *ab* atoms in the answer set, e.g., as

$$\bot :\sim ab_{n_r}(c_1, \dots, c_n).[1, c_1, \dots, c_n],$$

helps in obtaining the debugging answer set with the minimal number of *ab* atoms.

Example 4.53 (ctd)**.** The program Π in Example 4.15 is converted into Π_{ab}:

$c(X) \leftarrow not\ d(X), X < 5, int(X), not\ ab_{r1}(X).$
$d(X) \leftarrow not\ c(X), int(X), not\ ab_{r2}(X).$
$b(X, Y) \leftarrow a(X), d(Y), int(X), int(Y), not\ ab_{r3}(X, Y).$
$e(X) \leftarrow c(X), a(Y), X \leq Y, int(X), int(Y), not\ ab_{r4}(X, Y).$
$\bot \leftarrow b(X, Y), e(X), int(X), int(Y), not\ ab_{r5}(X, Y).$
$\{ab_{r1}(X)\}. \quad \{ab_{r2}(X)\}. \quad \{ab_{r3}(X, Y)\}. \quad \{ab_{r4}(X, Y)\}. \quad \{ab_{r5}(X, Y)\}.$

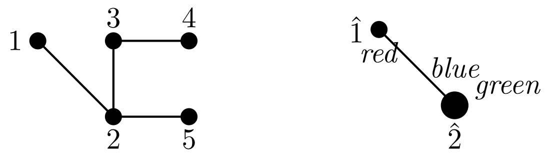
and an additional set of constraints are added:

$$\begin{array}{ll} \bot :\sim ab_{r1}(X).[1, X] & \bot :\sim ab_{r3}(X, Y).[1, X, Y] \\ \bot :\sim ab_{r2}(X).[1, X] & \bot :\sim ab_{r4}(X, Y).[1, X, Y] \\ & \bot :\sim ab_{r5}(X, Y).[1, X, Y] \end{array}$$

Checking the spuriousness of $\widehat{I}$ by $\Pi_{ab} \cup Q^m_{\widehat{I}}$ gives an optimum answer set

$$\{ab_{r4}(1, 1), ab_{r4}(1, 3), ab_{r4}(2, 3), ab_{r4}(3, 3), ab_{r2}(5)\},$$

Figure 4.12: A graph coloring instance



(a) Original graph (b) Abstract graph coloring

showing that the rules $r4$ and $r2$ that derive the atoms e and d, respectively, are actually involved in the computation of the original answer sets and had to be deactivated for certain domain elements so that a match to the spurious $\widehat{I}$ could be computed.

The obtained debugging atoms during a correctness check gives hints on which domain elements should not be involved in a cluster.

Definition 4.35. *Given a set AB of (non-ground) abnormality atoms used in the program* Π_{ab}*, the refinement-hint gathering (non-ground) program* $\Pi_{hint}(AB)$ *contains for each* $ab_{n_r}(c_1,\dots,c_n) \in AB$ *and* $c_i \in arg(ab_{n_r}(c_1,\dots,c_n))$*, the rule*

$$refine(c_1,\dots,c_n) \leftarrow ab_{n_r}(c_1,\dots,c_n), m(c_i, ac_i), isCluster(ac_i).$$

We discuss later how to make use of these *refine* atoms to determine a refinement. First, we show that for the constructed Π_{ab}, we can not guarantee that the correctness checking $\Pi_{ab} \cup Q^m_{\widehat{I}}$ always returns some ab atom if $\widehat{I}$ is spurious, and thus, the need for a more sophisticated debugging approach.

Shortcomings Although this debugging can be used for common benchmark problems (which we demonstrate in Section 5.2.2), the approach to focus on finding rules to deactivate during the correctness checking is unable to address all debugging cases.

Example 4.54. Consider the graph coloring example with an additional set of rules requiring that there must be a node that is colored to red and has two neighbors which are colored to blue and green, so that all three colors are used in the coloring.

The below encoding defines an auxiliary atom *usedRedGreenBlue* and ensures with a constraint that the atom must hold.

$$usedRedGreenBlue \leftarrow color(X, red), hasEdgeTo(X, green), hasEdgeTo(X, blue).$$
$$\bot \leftarrow not\ usedRedGreenBlue.$$

For the graph instance shown in Figure 4.12a, the mapping $\{\{1\}/\hat{1}, \{2,3,4,5\}/\hat{2}\}$ gives the abstract graph shown in Figure 4.12b. For the coloring shown in Figure 4.12b, the node $\hat{1}$ has possible edges to a node which is colored to blue and also to a node which is colored to green, thus an abstract answer set contains *usedRedGreenBlue*. However, when the abstract coloring is mapped back to the original graph, since node 1 only has one edge, the condition *usedRedGreenBlue* can not hold true.

However, due to the constraint (4.48) which aims to ensure that all occurring atoms in $\hat{I}$ should have a corresponding original atom, the query $\Pi_{ab} \cup Q^m_{\hat{I}}$ returns unsatisfiable, since the rule that defines *usedRedGreenBlue* can not be applicable.

The case shown in Example 4.54 is caused by the need to activate some original atoms, even if all the rules that can derive them are not applicable, in order to match the spurious abstract answer set. Next, we introduce a more sophisticated debugging approach to be able to address the shortage of the naive debugging.

Non-Ground Debugging using Tagging

We extend the refinement method by lifting the "tagging" approach of spock [BGP+07] to the non-ground case, confining to tight programs (i.e., we omit unfounded loop checking). Given Π, we construct the meta program $\mathcal{T}_{meta}[\Pi]$ similar to spock [BGP+07], but with an extension of having arguments in the ap_{n_r}, bl_{n_r} atoms to have information for which constants the rules are applicable and blocked.

Definition 4.36. *Given a non-ground program Π, the program $\mathcal{T}_{meta}[\Pi]$ consists of the following rules for $r \in \Pi$ with $arg(H(r)) = \{c_1, \dots, c_n\}$ and $arg(B(r)) = \{d_1, \dots, d_m\}$:*

$$\begin{array}{ll}
\textit{If } B(r) = \emptyset: & r \\[2ex]
\textit{If } H(r) \neq \bot \wedge n > 0: & \begin{array}{l} H(r) \leftarrow ap_{n_r}(c_1, \dots, c_n), not\ ko_{n_r}. \\ ap_{n_r}(c_1, \dots, c_n) \leftarrow B(r). \\ bl_{n_r}(c_1, \dots, c_n) \leftarrow not\ ap_{n_r}(c_1, \dots, c_n). \end{array} \\[6ex]
\textit{If } H(r) = \bot \vee n = 0: & \begin{array}{l} H(r) \leftarrow ap_{n_r}(d_1, \dots, d_m), not\ ko_{n_r}. \\ ap_{n_r}(d_1, \dots, d_m) \leftarrow B(r). \\ ap_{n_r} \leftarrow ap_{n_r}(d_1, \dots, d_m). \\ bl_{n_r} \leftarrow not\ ap_{n_r}. \end{array}
\end{array}$$

In case the head of rule r is $\bot$ or does not contain arguments in the atom, we use the arguments from the body to know whether r is applicable.

We have abnormality atoms to indicate the actions to avoid the inconsistency:

- $ab\_deact_{n_r}$ signals that r was applicable under some interpretation, but had to be deactivated;
- similarly for $ab\_deactCons_{n_r}$ which only talks about the constraints; and
- $ab\_act(\alpha)$ says that atom α has to made true although it had no support.

Definition 4.37. *Given a non-ground program* Π *over* $\mathcal{A}$*, the following additional meta-programs are constructed:*

1. $\mathcal{T}_{deact}[\Pi]$*: for all* $r \in \Pi$ *with* $B(r) \neq \emptyset$ *and* $H(r) \neq \bot$*:*

$$\begin{aligned} ko_{n_r}&. \\ \{H(r)\} &\leftarrow ap_{n_r}(c_1, \dots, c_n). \\ ab\_deact_{n_r}(c_1, \dots, c_n) &\leftarrow ap_{n_r}(c_1, \dots, c_n), not\ H(r). \end{aligned}$$

2. $\mathcal{T}_{deactCons}[\Pi]$*: for all* $r \in \Pi$ *with* $H(r) = \bot$*:*

$$\begin{aligned} &\{ko_{n_r}\}. \\ ab\_deactCons_{n_r}(c_1, \dots, c_n) &\leftarrow ap_{n_r}(c_1, \dots, c_n), ko_{n_r}. \end{aligned}$$

3. $\mathcal{T}_{act}[\Pi, \mathcal{A}]$*: for all* $\alpha \in \mathcal{A}$*, with* $def(\alpha, \Pi) = \{r_1, ..., r_k\}$*:*

$$\begin{aligned} \{\alpha\} &\leftarrow bl_{n_{r_1}}(c_1, \dots, c_n), ..., bl_{n_{r_k}}(c_1, \dots, c_n). \\ ab\_act(\alpha) &\leftarrow \alpha, bl_{n_{r_1}}(c_1, \dots, c_n), ..., bl_{n_{r_k}}(c_1, \dots, c_n). \end{aligned}$$

Notice the similarity of the *ab_deact* atom to the *ab* atom from the naive debugging (Definition 4.34). The difference of these atoms is in their arguments. The *ab* atoms of Definition 4.34 contain arguments related with the body of the rule, while the arguments of *ab_deact* only contain the ones from the head of the rule. This is a representation choice, to avoid dealing with many variables involved in the body while only few of them are used in the head of the rule. For the definition of *ab_deactCons* however, the variables of the body must be used. Having a different representation for the deactivation of the constraints will allow to steer the debugging towards the constraints by assigning different costs for their occurrence when computing the answer sets with the smallest number of *ab* atoms.

Definition 4.38. *For a program* Π *over atoms* $\mathcal{A}$*, we denote by* Π_{debug} *the program* $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}]$.

We denote by $\mathcal{A}^*$ the vocabulary of Π_{debug} which consists of $\mathcal{A}^+ \cup AB(\Pi)$.

We use Π_{debug} for checking the correctness of an abstract answer set and then deciding on the refinement. Adding weak constraints over the abnormality atoms yields an answer set with fewest *ab* atoms.

Example 4.55 (Example 4.52 ctd)**.** The programs $\mathcal{T}_{meta}[\Pi]$ and $\mathcal{T}_{deact}[\Pi]$ with additional weak constraints over the abnormality atoms are shown in Figures 4.13a and 4.13b, respectively. The minimal answer set of $\mathcal{T}_{meta}[\Pi] \cup \mathcal{T}_{deact}[\Pi] \cup Q^m_{\widehat{I}}$ is then

$$\{ab\_deact_{r2}(5), ab\_deact_{r4}(1), ab\_deact_{r4}(3), ab\_deact_{r4}(2)\}.$$

Observe that this is similar to the debugging output achieved with the naive approach in Example 4.53.

Figure 4.13: Debugging programs for Example 4.52

$$
\begin{array}{ll}
c(X) & \leftarrow ap_{r1}(X), \mathit{not}\ ko_{r1}.\\
ap_{r1}(X) & \leftarrow X < 5, \mathit{not}\ d(X).\\
bl_{r1}(X) & \leftarrow \mathit{not}\ ap_{r1}(X).\\
 & \\
d(X) & \leftarrow ap_{r2}(X), \mathit{not}\ ko_{r2}.\\
ap_{r2}(X) & \leftarrow \mathit{not}\ c(X).\\
bl_{r2}(X) & \leftarrow \mathit{not}\ ap_{r2}(X).\\
 & \\
b(X,Y) & \leftarrow ap_{r3}(X,Y), \mathit{not}\ ko_{r3}.\\
ap_{r3}(X,Y) & \leftarrow a(X), d(Y).\\
bl_{r3}(X,Y) & \leftarrow \mathit{not}\ ap_{r3}(X,Y).\\
 & \\
e(X) & \leftarrow ap_{r4}(X), \mathit{not}\ ko_{r4}.\\
ap_{r4}(X) & \leftarrow c(X), a(Y), X \leq Y.\\
bl_{r4}(X) & \leftarrow \mathit{not}\ ap_{r4}(X).\\
 & \\
\mathit{falsum} & \leftarrow ap_{r5}(X,Y), \mathit{not}\ ko_{r5}.\\
ap_{r5}(X,Y) & \leftarrow b(X,Y), e(X).\\
ap_{r5} & \leftarrow ap_{r5}(X,Y).\\
bl_{r5} & \leftarrow \mathit{not}\ ap_{r5}.\\
\perp & \leftarrow \mathit{falsum}.
\end{array}
$$

(a) $\mathcal{T}_{meta}[\Pi]$

$$
\begin{array}{ll}
ko_{r1}. & \\
\{c(X)\} & \leftarrow ap_{r1}(X).\\
ab\_deact_{r1}(X) & \leftarrow ap_{r1}(X), \mathit{not}\ c(X).\\
\perp & :\sim ab\_deact_{r1}(X).\\
 & \\
ko_{r2}. & \\
\{d(X)\} & \leftarrow ap_{r2}(X).\\
ab\_deact_{r2}(X) & \leftarrow ap_{r2}(X), \mathit{not}\ d(X).\\
\perp & :\sim ab\_deact_{r2}(X).\\
 & \\
ko_{r3}. & \\
\{b(X,Y)\} & \leftarrow ap_{r3}(X,Y).\\
ab\_deact_{r3}(X,Y) & \leftarrow ap_{r3}(X,Y), \mathit{not}\ b(X,Y).\\
\perp & :\sim ab\_deact_{r3}(X,Y).\\
 & \\
ko_{r4}. & \\
\{e(X)\} & \leftarrow ap_{r4}(X).\\
ab\_deact_{r4}(X) & \leftarrow ap_{r4}(X), \mathit{not}\ e(X).\\
\perp & :\sim ab\_deact_{r4}(X).
\end{array}
$$

(b) $\mathcal{T}_{deact}[\Pi]$ with weak constraints

This debugging approach is also able to handle the mentioned shortcomings of the naive approach in Example 4.12, as $\mathcal{T}_{act}[\Pi, \mathcal{A}]$ is used to activate original atoms if it is necessary for achieving satisfiability for $\Pi_{debug} \cup Q^m_{\widehat{I}}$.

We show that $\Pi_{debug} \cup Q^m_{\widehat{I}}$ always returns an answer set for $\hat{I}$, with a similar reasoning as in Proposition 4.34 for debugging bad omission.

Proposition 4.38. *Given a tight program Π and a mapping m, for each answer set $\hat{I} \in AS(\Pi^m)$, $\Pi_{debug} \cup Q^m_{\widehat{I}}$ has an answer set.*

Proof. Let X be an interpretation over $\mathcal{A}^*$. We will show that with the help of the auxiliary rules/atoms, some interpretation X' which is a minimal model of $(\Pi_{debug} \cup Q^m_{\widehat{I}})^{X'}$ can be reached starting from X. We have the cases (i) $X \not\models (\Pi_{debug} \cup Q^m_{\widehat{I}})^X$, and (ii) $X \models (\Pi_{debug} \cup Q^m_{\widehat{I}})^X$.

(i) We show that X can be changed to some interpretation X' that avoids the conditions for X not satisfying a rule $(\Pi_{debug} \cup Q^m_{\hat{I}})^X$, by doing a case analysis on where the rule may occur.

(a) Let r be an unsatisfied rule in $(\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}])^X$. This means that $X \models B(r)$ and $X \not\models H(r)$. The rule r can not be an instantiation of the choice rules in $\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}]$, as it would be instantiated for X, and hence be satisfied. Thus r can either (a-1) have $H(r) \in AB(\Pi)$ or (a-2) have $H(r) = ko_{n_{r'}}$ for some $r' \in \Pi$ and be in $\mathcal{T}_{deact}[\Pi]^X$. For both of these cases, we construct $X' = X \cup \{H(r)\}$, and the reduct $(\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}])^{X'}$ will not have further rules, since the added $H(r)$ does not occur in the body of any auxiliary rule. Note that adding $ko_{n_{r'}}$ also avoids having the corresponding meta-rule of r' in the reduct $(\mathcal{T}_{meta}[\Pi])^{X'}$. So no further rules will be added in $(\Pi_{debug} \cup Q^m_{\hat{I}})^{X'}$.

(b) Let r be an unsatisfied rule in $(\mathcal{T}_{meta}[\Pi])^X$.

(b-1) If $B(r) = \emptyset$, we construct $X' = X \cup H(r)$. Thus, we get $X' \models H(r)$.

(b-2) If the rule is of form $H(r') \leftarrow ap_{n_{r'}}(c_1, \dots, c_n), \mathit{not}\ ko_{n_{r'}}.$ where $H(r') \neq \bot$ for some $r' \in \Pi$, this means $ko_{n_{r'}} \notin X$. However, rules for r' are added in $\mathcal{T}_{deact}[\Pi]$ which uses the rule $ko_{n_{r'}}.$ to deactivate the meta-rule in $\mathcal{T}_{meta}[\Pi]$, which is then also unsatisfied in the reduct $(\mathcal{T}_{deact}[\Pi])^X$. So we construct $X' = X \setminus \cup\{ko_{n_{r'}}\}$. Thus, the rule r does not appear in $(\mathcal{T}_{meta}[\Pi])^{X'}$.

(b-3) Let the rule be of form $H(r') \leftarrow ap_{n_{r'}}(d_1, \dots, d_m), \mathit{not}\ ko_{n_{r'}}.$, where $H(r') = \bot$ for some $r' \in \Pi$. The rules in $\mathcal{T}_{deactCons}[\Pi]$ are added for r'. $X \models B(r)$ means that $ko_{n_{r'}} \notin X'$. So the choice rule $\{ko_{n_{r'}}\}.$ in $\mathcal{T}_{deactCons}[\Pi]$ gets instantiated to $ko'_{n_{r'}}.$ in $(\mathcal{T}_{deactCons}[\Pi])^X$, i.e., $ko'_{n_{r'}} \in X$. Now consider the interpretation $X' = (X \setminus \{ko'_{n_{r'}}\}) \cup \{ko_{n_{r'}}\}$. Since $ko_{n_{r'}} \in X'$, the rule r does not appear in $\mathcal{T}_{meta}[\Pi]^{X'}$.

(b-4) If r is of form $ap_{n_{r'}}(c_1, \dots, c_n) \leftarrow B(r')$ (or $ap_{n_{r'}}(d_1, \dots, d_m) \leftarrow B(r')$) for some $r' \in \Pi$, then we construct $X' = X \cup \{ap_{n_{r'}}(c_1, \dots, c_n)\}$ (or $X' = X \cup \{ap_{n_{r'}}(d_1, \dots, d_m)\}$). If r is of the remaining forms involving ab and bl, for some $r' \in \Pi$, we first check if $X \models B(r')$, then either add the respective ab atoms or the respective bl atom to X'.

(c) Assume $X \not\models (Q^m_{\hat{I}})^X$. As $Q^m_{\hat{I}}$ has two forms of rules, X not satifying a rule in $(Q^m_{\hat{I}})^X$ means that either (a) some $\alpha \in X \cap \mathcal{A}$ exists while $m(\alpha) \notin \hat{I}$, or (b) for some $\hat{\alpha} \in \hat{I}$, no $\alpha \in X \cap \mathcal{A}$ exists such that $m(\alpha) = \hat{\alpha}$.

(c-1) The literal $m(\alpha)$ not occurring in $\hat{I}$ means that all rules in $\mathit{def}(m(\alpha), \Pi^m)$ are not applicable for $\hat{I}$. This means that for each $r \in \mathit{def}(\alpha, \Pi)$ we have $B(r) \neq \emptyset$, and for some $r \in \mathit{def}(\alpha, \Pi)$ the choice rule $\{\alpha\} \leftarrow ap_{n_r}(c_1, \dots, c_n)$ in $\mathcal{T}_{deact}[\Pi]$ gets instantiated to $\alpha \leftarrow ap_{n_r}(c_1, \dots, c_n)$ in Π^X_{debug}. Now consider the interpretation $X' = (X \setminus \{\alpha\}) \cup \{\alpha'\}$, for which the choice rule gets instantiated to $\alpha' \leftarrow ap_{n_r}(c_1, \dots, c_n)$ in $\Pi^{X'}_{debug}$. Thus an interpretation X' is constructed where case (c-1) is avoided.

(c-2) Similarly as above, depending on whether the original rules in $def(\alpha, \Pi)$ for all $\alpha \in m^{-1}(\hat{\alpha})$ are applicable or blocked for X, using the auxiliary choice rules in $\mathcal{T}_{deact}[\Pi]$ and $\mathcal{T}_{act}[\Pi, \mathcal{A}]$ some X' can be constructed that avoids the case of having no $\alpha \in X'$ with $m(\alpha) = \hat{\alpha}$.

For the constructed X' case (a-1) may occur, and can be avoided as shown by adding *ab* atoms to the interpretation.

By iterating the reasoning on the constructed X', since none of the cases will undo a construction step that is previously made, some interpretation X'' that satisfies $(\Pi_{debug} \cup Q^m_{\hat{I}})^X$ will eventually be achieved.

(ii) If X is a minimal model, then X is an answer set of $\Pi_{debug} \cup Q^m_{\hat{I}}$, which achieves the result. We assume this is not the case, and that there exists $Y \subset X$ such that $Y \models (\Pi_{debug} \cup Q^m_{\hat{I}})^X$. As $Y \models (Q^m_{\hat{I}})^X$, we have $m(Y \cap \mathcal{A}) = \hat{I}$. Let $\alpha \in X \setminus Y$, then we have two cases: (a) $\alpha \in \mathcal{A}$, (b) $\alpha \in \mathcal{A}^* \setminus \mathcal{A}$, i.e., α is an auxiliary debugging atom.

(a) Let $\alpha = p(c_1, \dots, c_n)$. As $\alpha \in X \setminus Y$ and $\alpha \in \mathcal{A}$, this means that there is some $\hat{\alpha} = p(\hat{c}_1, \dots, \hat{c}_n)$ in $\hat{I}$ and some $\alpha_1 = p(c'_1, \dots, c'_n)$ in Y (and $\alpha_1 \in X$) such that $m(\alpha_1) = m(\alpha) = \hat{\alpha}$. However, since $\hat{\alpha} \in \hat{I}$ and $\hat{I}$ is an answer set of Π^m, there exists a rule instantiation $\hat{r} \in (\Pi^m)^{\hat{I}}$ such that $H(\hat{r}) = \hat{\alpha}$. Then, since $\alpha, \alpha_1 \in X$ the abstract rule $\hat{r}$ has corresponding rules $r, r_1 \in \mathcal{T}_{meta}[\Pi]^X$ where $r : p(c_1, \dots, c_n) \leftarrow ap_{n_{r'}}(c_1, \dots, c_n), not\ ko_{n_{r'}}$. and $r_1 : p(c'_1, \dots, c'_n) \leftarrow ap_{n_{r''}}(c'_1, \dots, c'_n), not\ ko_{n_{r''}}$ for some $r', r'' \in \Pi$.

Now, as we assume $\alpha \notin Y$, then $ap_{n_{r'}}(c_1, \dots, c_n) \notin Y$ should hold (to satisfy the rule $r \in \mathcal{T}_{meta}[\Pi]^X$). Then in order to avoid not satisfying the rule $ap_{n_{r'}}(c_1, \dots, c_n) \leftarrow B(r')|_{c_1,\dots,c_n}$ in $\mathcal{T}_{meta}[\Pi]^X$, where $B(r')|_{c_1,\dots,c_n}$ denotes that $B(r')$ is instantiated over $c_1, \dots, c_n$, either (a-1) for some $\beta \in B^+(r')|_{c_1,\dots,c_n}$ we must have $\beta \notin Y$ or (a-2) for some $\beta \in B^-(r')|_{c_1,\dots,c_n}$ we must have $\beta \in Y$. As case (a-2) contradicts $Y \subset X$, let us consider case (a-1). For $\beta \notin Y$, we do the same reasoning as above. By doing this recursive reasoning, since Π is tight, we eventually reach a rule in which case (a-1) is not applicable, and thus by case (a-2) we reach a contradiction.

(b) Let $\alpha \in \mathcal{A}^* \setminus \mathcal{A}$. If α is in $AB(\Pi)$, then we pick Y as the interpretation, and the reduct $(\Pi_{debug} \cup Q^m_{\hat{I}})^Y$ will not have further rules, since the *ab* atoms of $AB(\Pi)$ do not occur in body of any auxiliary rule. If α is in $\mathcal{A}^+ \setminus \mathcal{A}$, then we pick Y as the interpretation and apply the above reasoning for Y. When we recursively continue with this reasoning, eventually, the case (ii-b) will not be applicable, and thus we would construct a minimal model.

□

Note that in the proof of Proposition 4.38, since the auxiliary programs $\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}]$ can always have a model (as they do not contain constraints), one can start with an interpretation X which models $(\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}])^X$.

However, since we construct another interpretation Y from X, a check on whether Y models $(\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}])^Y$ then becomes necessary.

The following result shows that we can use $\Pi_{debug} \cup Q^m_{\widehat{I}}$ to obtain hints for the spuriousness reason of $\hat{I}$.

Proposition 4.39. *Given a tight program* Π *and a mapping* m*, if an answer set* $\hat{I} \in AS(\Pi^m)$ *is spurious, then for every answer set* $S \in AS(\Pi_{debug} \cup Q^m_{\widehat{I}})$ *either (i)* $ab\_deact_{n_r}(c_1, \dots, c_n) \in S$ *or* $ab\_deactCons_{n_r}(c_1, \dots, c_n) \in S$ *for some* $r \in \Pi$*, or (ii)* $ab\_act(\alpha(c_1, \dots, c_n)) \in S$*.*

Proof. If $\hat{I}$ is spurious, by Proposition 4.29 the program $\Pi \cup Q^m_{\widehat{I}}$ is unsatisfiable. We focus on debugging the cause of inconsistency introduced by $Q^m_{\widehat{I}}$. Since Π is tight, this inconsistency can either be due to (i) an unsatisfied rule or (ii) an unsupported atom.

(i) Let $r \in \Pi$ be an unsatisfied rule w.r.t. S. This means that the constraints in $Q^m_{\widehat{I}}$ is causing $H(r)$ to be false while $B(r)$ is satisfied. By the program $\mathcal{T}_{meta}[\Pi]$, depending on r, either $ap_{n_r}(c_1, \dots, c_n)$ or $ap_{n_r}(d_1, \dots, d_m)$ is true. By $\mathcal{T}_{deact}[\Pi]$, we get $ab\_deact(r, c_1, \dots, c_n) \in S$. If $H(r) = \bot$, then by $\mathcal{T}_{deactCons}[\Pi]$, we have $ko_{n_r} \in S$ (else $\bot \in S$ by $\mathcal{T}_{meta}[\Pi]$), and we get $ab\_deactCons(r, c_1, \dots, c_n) \in S$.

(ii) Let $\alpha = \alpha(c_1, \dots, c_n) \in S$ be an unsupported atom in Π w.r.t S for the domain elements $c_1, \dots, c_n$. Then, for each rule instance r deriving α, we have $bl_{n_r}(c_1, \dots, c_n) \in S$ and by $\mathcal{T}_{act}[\Pi, \mathcal{A}]$, we have $ab\_act(\alpha) \in S$.

$\square$

Indeed for non-tight programs, debugging the corretness checking could result in unsatisfiability.

Example 4.56. Consider the below program which is unsatisfiable and also contains a positive loop.

$$\begin{aligned}
&a(X) \leftarrow not\ a(X), int(X).\\
&a(X) \leftarrow a(X), int(X).\\
&int(1).\ int(2).\ int(3).
\end{aligned}$$

For the mapping $m = \{\{1, 2, 3\}/k\}$, the constructed abstract program Π^m becomes

$$\begin{aligned}
a(X) &\leftarrow not\ a(X), \widehat{int}(X).\\
\{a(X)\} &\leftarrow isCluster(X), \widehat{int}(X).\\
a(X) &\leftarrow a(X), \widehat{int}(X).\\
\widehat{int}&(k).
\end{aligned}$$

causing to have the abstract answer set $\hat{I} = \{a(k)\}$. Checking the correctness using $\Pi_{debug} \cup Q^m_{\widehat{I}}$ results in unsatisfiability, because it requires to have some $a(c)$ for $c \in m^{-1}(k)$ to hold true through a loop, which is not covered in the definition of Π_{debug}.

Handling the unfounded loop checking can be done by introducing an additional abnormality atom, say ab_{loop} as in [BGP+07] and lifting it to the non-ground settings as follows:

$$\begin{aligned}\{ab_{loop}(\alpha)\} &\leftarrow not\ ab\_act(\alpha).\\ \alpha &\leftarrow ab_{loop}(\alpha).\end{aligned}$$

However, this solution causes to have further guessing rules involved in the non-ground debugging. Also the existence of $ab_{loop}(\alpha)$ sometimes does not even indicate that a violation of a loop formula exists and just makes the search more difficult due to considering many possibilities of the guesses. Therefore, we choose to focus only on tight programs and concentrate on the determination of a refinement.

Definition 4.35 on obtaining refinement hints then gets updated to use the newly introduced *ab* atoms.

Definition 4.39. *The refinement-hint gathering program $\Pi_{hint}(AB)$ contains the following rules:*

- *For $c_i \in arg(ab\_deact_{n_r}(c_1,\dots,c_n))$:*

$$\{refine(c_1,\dots,c_n) \leftarrow ab\_deact_{n_r}(c_1,\dots,c_n), m(c_i,a_i), isCluster(a_i).\}$$

- *For $c_i \in arg(ab\_deactCons_{n_r}(c_1,\dots,c_n))$:*

$$\{refine(c_1,\dots,c_n) \leftarrow ab\_deactCons_{n_r}(c_1,\dots,c_n), m(c_i,a_i), isCluster(a_i).\}$$

- *For $c_i \in arg(\alpha(c_1,\dots,c_n))$:*

$$\{refine(c_1,\dots,c_n) \leftarrow ab\_act(\alpha(c_1,\dots,c_n)), m(c_i,a_i), isCluster(a_i).\}$$

By using Π_{hint}, we get as hints the domain elements that are mapped to cluster abstract elements and that cause to obtain *ab* atoms in the debugging.

Deciding on a Refinement

The introduced debugging approaches find a set of abnormality atoms in case the abstract answer set is spurious. We consider two forms of using the obtained debugging output for deciding on a refinement.

(v1) The number of *ab* atoms occurring in the answer set with the smallest number of *ab* atoms is assigned as a cost to the corresponding mapping.
(v2) The inferred *refine* atoms are used to decide on a refinement of the abstraction.

In approach (v1), the assigned costs are then used for a local search among the possible refinements of an abstraction, where the mapping with the minimum cost is picked as the refinement. Approach (v2) is more closer to the CEGAR-like approach [CGJ+03], where a refinement is determined from the spuriousness check. We now describe the approaches in more detail, and then in Section 5.2.2 we report on their comparisons.

Algorithm 4.1: *decideRefinement* with Search

```
Input: $\Pi$, $m$
Output: $m'$
if $m$ has non-singleton clusters then
    $refinecosts = []$; $allrefs = []$;
    /* compute all 1-distance refinements of $m$ */
    $refs = computeRefinements(m, 1)$
    forall $m' \in refs$ do
        $c = getCostOfMapping(\Pi, m')$;
        if $c = 0$ then /* found a concrete abstract answer set */
            return $m'$;
        else
            $refinecosts.append(c)$; $allrefs.append(m')$
    $minrefs = getRefsMinCost(refinecost, allrefs)$
    $m' = pickRandomRef(minrefs)$
return $m'$
____________________________________________________________
def $getCostOfMapping(\Pi, m)$
    $\Pi' = constructAbsProg(\Pi, m)$;
    $\Pi_{debug} = constructDebugProg(\Pi)$;
    Pick some $I \in AS(\Pi')$
    Find optimum answer set $I'$ of $\Pi_{debug} \cup Q_m(I)$
    return $|I'|_{ab}|$
```

(v1) Local Refinement Search The idea is to search among possible refinements of a mapping for deciding on a refinement. For this, we consider a notion of distance between an abstraction and its refinement, which is the difference in the number of abstract clusters.

Definition 4.40. *A* distance *of a mapping m_1 to a refinement m_2 is $|D_2| - |D_1|$ for the corresponding abstract domains $D_i, i \in \{1, 2\}$.*

Example 4.57. Each mapping $m' \in \bigcup_{d \in \{\{1\},\{1,2\},\{1,2,3\},\{1,2,3,4\}\}} \{d/k_1, \{1, \dots, 5\} \setminus d/k_2\}$ is a 1-distance refinement of $m = \{\{1, 2, 3, 4, 5\}/k\}$.

Algorithm 4.1 shows the procedure of deciding on a refinement for a given mapping m by doing a distance-based search among all possible refinements of the mapping and picking the one with the least cost. All 1-distance refinements of m are computed, and then the cost of each of the refinements is determined, by calling the function $getCostOfMapping$. This function first constructs the abstract program Π' according to the mapping, and then picks an abstract answer set I. It then finds the answer set with smallest number of *ab* atoms of the program $\Pi_{debug} \cup Q_m(I)$ and returns the number of occurring *ab* atoms. If some refinement has cost 0, the mapping is returned. Otherwise, all the costs of the refinements are collected. In Line 10 the refinements with minimum cost are gathered, and then a random pick is made over them. If the given mapping contains only singleton clusters, this means the original domain has been reached.

Example 4.58 (ctd)**.** Running Algorithm 4.1 to decide on a refinement for m of Example 4.53 (using the implementation explained in the upcoming Section 5.1) results in the following steps. The algorithm starts computing the cost of each refinement (Example 4.57) one by one. The cost of refinement $m' = \{\{1\}/k_1, \{2,3,4,5\}/k_2\}$ is 3 due to the picked answer set of $\Pi^{m'}$ being

$$\{a(k_1), a(k_2), d(k_1), c(k_2), b(k_1,k_1), b(k_2,k_1)\}.$$

For the refinement $m'' = \{\{1,2\}/k_1, \{3,4,5\}/k_2\}$ the picked answer set of $\Pi^{m''}$ is

$$\{a(k_1), a(k_2), d(k_1), d(k_2), b(k_1,k_1), b(k_2,k_1), b(k_1,k_2), b(k_2,k_2)\},$$

which is concrete, thus m'' has cost 0. Hence, the refinement m'' is chosen without continuing further.

(v2) Abstraction Refinement Using Hints The abstract answer set correctness checking returns *ab*-atoms that contain the domain elements involved in the debugging of the unsatisfiability. These domain elements can be used as hints on which part of the mapping to refine. The idea of the *refine* atoms is to get the hints about which domain elements should not be involved in a cluster. For a mapping m, given a hint atom $\mathit{refine}(c_1,\dots,c_n)$, we consider two actions to describe a refinement m':

(1) For $\mathit{refine}(c_1,\dots,c_n)$ with $c_i, 1 \leq i \leq n$, such that $m^{-1}(m(c_i)) > 1$, the refinement m' satisfies $m'^{-1}(m'(c_i)) = 1$.

(2) For $\mathit{refine}(c_1,\dots,c_n)$ with $c_i \neq c_j, 1 \leq i \leq j \leq n$, and $m(c_i) = m(c_j)$, the refinement m' satisfies $m'(c_i) \neq m'(c_j)$.

Example 4.59 (ctd)**.** For Example 4.53, the hint atoms are $\{\mathit{refine}(1,1), \mathit{refine}(1,3), \mathit{refine}(2,3), \mathit{refine}(3,3), \mathit{refine}(5)\}$. Applying refinement action (1) means to map the elements $1,2,3,5$, which are mapped to the cluster k in m, to singletons in m'. Thus, we obtain the refinement mapping $m' = \{\{1\}/k_1, \{2\}/k_2, \{3\}/k_3, \{4\}/k_4, \{5\}/k_5\}$. It is clear that with this trivial abstraction mapping the spurious answer set $\hat{I}$ is removed.

If we apply refinement action (2), we need to separate the clusters of $1,3$ and $2,3$. The refinement $m' = \{\{1,2\}/k_1, \{3,4,5\}/k_2$ satisfies this. The answer sets of $\Pi^{m'}$ do not contain a match to $\hat{I}$, i.e., $\nexists I' \in AS(\Pi^{m'})$ such that $I' \in m'(m^{-1}(\hat{I}))$.

The obtained hint atoms from Example 4.55 are $\{\mathit{refine}(1), \mathit{refine}(2), \mathit{refine}(3), \mathit{refine}(5)\}$ on which refinement action (2) would not result in a refinement, as there are no pairs to separate.

Note that it is not guaranteed to always obtain some *refine* atom during the correctness checking whenever $\hat{I}$ is spurious, since sometimes the *ab* atoms can contain only domain elements that are mapped to singleton clusters. In this case, another abstract answer set $\hat{I}' \in \widehat{\Pi}^m$ can be picked for the correctness checking.

Algorithm 4.2: *Omission-Abs&Ref*

```
Input: Program $\Pi$, set $A_{init}$ of atoms to omit
Output: $\Pi' = omit(\Pi, A')$, $A'$, and either an abstract answer set $I$ of $\Pi'$, or $\Pi'$ is
        unsatisfiable
1  $A' = A_{init}$;
2  $\Pi' = constructAbsProg(\Pi, A')$;
3  while $AS(\Pi') \neq \emptyset$ do
4      Get $I \in AS(\Pi')$;
5      $\Pi_{debug} = constructDebugProg(\Pi, A', I)$;
6      $S = getASWithMinBadOmit(\Pi_{debug})$;
7      if $S|_{badomit} = \emptyset$ then /* $I$ concrete */
8          return $\Pi', A', I$
9      else /* refine the abstraction */
10         $A' = A' \setminus S|_{badomit}$;
11         $\Pi' = constructAbsProg(\Pi, A')$;
   /* reached an unsatisfiable $\Pi'$ */
12 return $\Pi', A', \emptyset$
```

4.5 Overall Methodology

As seen in the previous sections, the methods to construct the abstract program and decide on a refinement by checking the correctness of an abstract answer set differs for the two abstraction methods. We now describe in more detail how the methods work for the particular abstractions.

4.5.1 Omission Abstraction and Refinement

For omission abstraction, the mapping m is represented with the set A of atoms to be omitted by m and the input program Π is ground. Algorithm 4.2 shows the abstraction refinement method for omission abstraction. Given a ground program Π and a set A_{init} of atoms to be omitted, first the abstract program $\Pi' = omit(\Pi, A_{init})$ is constructed (Line 2). If the abstract program is unsatisfiable, the program and the set of omitted atoms are returned (Line 12). Otherwise, an answer set $I \in AS(omit(\Pi, A_{init}))$ is computed. In the implementation, the first answer set is picked. In order to check whether I is concrete, the meta-program $\Pi_{debug} = \mathcal{T}[\Pi, \hat{I}]$ as described in Section 4.4.1 is constructed (Line 5). Then a search over the answer sets of $\mathcal{T}[\Pi, \hat{I}]$ for a minimum number of *badomit* atoms is carried out (Line 6). If an answer set with no *badomit* atoms exists, then this shows that I is concrete, and the abstract program and the set of omitted atoms are returned (Line 8). Otherwise, the set of omitted atoms is refined by removing the atoms that are determined as badly omitted, and a new abstract program is constructed with the refined abstraction A'. This loop continues until either the abstract program Π' constructed at Line 11 is unsatisfiable or its first answer set is concrete.

Algorithm 4.3: *Domain-Abs&Ref*

```
Input: Program $\Pi$, domain mapping $m_{init}$
Output: $\Pi^m$, a mapping $m$ that refines $m_{init}$, and either an abstract answer set $\hat{I}$
        of $\Pi^m$, or $\Pi^m$ is unsatisfiable.
1  $m = m_{init}$;
2  $\Pi^m = constructAbsProg(\Pi)$;
3  $\mathcal{T}_m = computeRelTypes(\Pi, m)$;
4  $\Pi_{debug} = constructDebugProg(\Pi)$;
5  while $AS(\Pi^m, \mathcal{T}_m) \neq \emptyset$ do
6      Get $I \in AS(\Pi^m, \mathcal{T}_m)$;
7      $C = getASWithMinAbAtoms(\Pi_{debug}, m, I)$;
8      if $C|_{ab} = \emptyset$ then /* $I$ concrete */
9          return $\Pi^m, m, I$
10     $m = decideRefinement(m, C)$;
11     $\Pi^m = constructAbsProg(\Pi)$;
12     $\mathcal{T}_m = computeRelTypes(\Pi, m)$;
13 return $\Pi^m, m, \emptyset$
```

4.5.2 Domain Abstraction and Refinement

The abstraction and refinement method for domain abstraction is shown in Algorithm 4.3. For domain abstraction, additionally, the relation type computation of the relations according to the abstraction m is conducted (Line 3). The construction of the debugging program does not depend on the abstract answer set as in omission abstraction, thus, is done before an abstract answer set is computed (Line 4). The concreteness checking (Line 7) is conducted with *getASWithMinAbAtoms* which uses the non-ground query $Q^m_{\hat{I}}$ (Definition 4.33) constructed for m and I, and searches for an answer set of $\Pi_{debug} \cup Q^m_{\hat{I}}$ with minimal number of *ab* atoms. The refinement step (Line 10) is done by deciding on a refinement for m given the *ab* atoms occurring in the concreteness checking if I is spurious.

If the rules in Π do not contain constants from the domain to be abstracted, then it is enough to construct the abstract program Π^m once and only recompute the relation types $\mathcal{T}_m$ according to the new mapping m, since the constructed Π^m is non-ground. Thus, for such cases Line 11 can be omitted. We remark that if a rule contains a constant c, it can be replaced with a variable X by also adding a new auxiliary atom $p_c(X)$ to the rule body and using the additional fact $p_c(c)$.

In order to apply abstraction over a particular subdomain/sort by keeping the rest of the domain elements untouched, Algorithm 4.3 can be extended to consider as input the domain predicate name s for the sort. This way the relation type computation, the abstract program and the debugging program construction can only focus on the domain elements related with s.

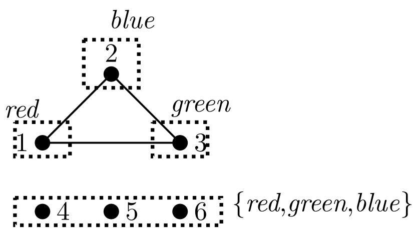
Algorithm 4.4: *Domain-Abs&Ref-Diverse*

```
Input: Program $\Pi$, domain mapping $m_{init}$
Output: $\Pi^m$, a mapping $m$ that refines $m_{init}$, and either an abstract answer set $\hat{I}$
        of $\Pi^m$, or $\Pi^m$ is unsatisfiable.
$m = m_{init}$;
$\Pi^m = constructAbsProg(\Pi)$;
$\mathcal{T}_m = computeRelTypes(\Pi, m)$;
$\Pi_{debug} = constructDebugProg(\Pi)$;
while $AS(\Pi^m, \mathcal{T}_m) \neq \emptyset$ do
    $A = getDiverseAnsSets(\Pi^m, \mathcal{T}_m)$;
    $C_{list} = []$;
    for $I \in A$ do
        $C = getASWithMinAbAtoms(\Pi_{debug}, m, I)$;
        if $C|_{ab} = \emptyset$ then /* $I$ concrete */
            return $\Pi^m, m, I$
        else
            $C_{list}.append(C)$;
    $m = decideRefinement(m, C_{list})$;
    $\Pi^m = constructAbsProg(\Pi)$;
    $\mathcal{T}_m = computeRelTypes(\Pi, m)$;
return $\Pi^m, m, \emptyset$
```

Projecting correctness checking to relevant atoms Projecting the correctness checking of abstract answer sets to the *relevant* atoms that describe the solution is achieved by constructing $Q^m_{\hat{I}}$ only for these atoms. Thus an abstract answer set will then be decided to be concrete as long as it describes a concrete solution with respect to these relevant atoms.

Diverse abstract answer sets The refinement step can also be extended to checking multiple abstract answer sets before deciding on a refinement. That is, a refinement decision is not made by checking one answer set, but by gathering the results of checking multiple answer sets. Algorithm 4.4 shows an updated version of Algorithm 4.3 by collecting the results of checking a set of abstract answer sets. An additional function *getDiverseAnsSets* is used to pick a set of abstract answer sets. Then *decideRefinement* uses the collected results for deciding on a refinement. For refinement approach v1, the cost of a mapping is determined by checking multiple abstract answer sets and then picking the one with the least cost. Refinement approach v2 decides on the domain to refine by choosing the most occurring $refine$ atom.

Figure 4.14: Joint abstraction of nodes and colors



4.6 Multi-Dimensional Domain Abstraction

As we have seen from Proposition 4.22, it is possible to construct an abstract program over multiple sorts in the manner of cartesian abstraction. This is achieved by doing abstraction over the sorts one at a time, which means one can not adhere to structure during the abstraction. However, in certain cases an abstraction mapping that takes into account certain interdependency among the sorts may be needed.

Example 4.60 (Example 4.45 ctd)**.** An interesting abstraction would be to assign a color cluster $\hat{rgb}$ only for the nodes $\{4,5,6\}$ while for nodes $\{1,2,3\}$ the original colors are considered and also having a node cluster $\{4,5,6\}/\hat{4}$ as shown in Figure 4.14. Such an abstraction can not be achieved with a cartesian style abstraction, since the color cluster $\hat{rgb}$ is only meant to be considered for the node cluster $\hat{4}$. Thus, the desired abstraction can only be defined with a multi-dimensional mapping $m : D_n \times D_c \rightarrow \widehat{D}_n \times \widehat{D}_c$ as follows:

$$m(i,j) = \begin{cases} (i,j) & i \in \{1,2,3\}, j \in \{red, green, blue\} \\ (\hat{4}, \hat{rgb}) & i \in \{4,5,6\}, j \in \{red, green, blue\} \end{cases} .$$

Grid-Cell Domains To further motivate the need for multi-dimensionality, let us consider a representation of grid-cell domains which is commonly used in problems. Usually the grid-cells are represented by using two sorts *row* and *column*. For example, the following rules show the part of a Sudoku encoding that guesses an assignment of symbols to the cells, and ensures that each cell has a number.

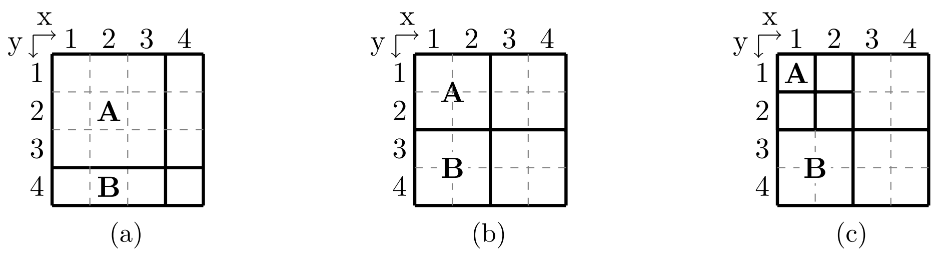
$$\begin{aligned} \{sol(X,Y,N)\} &\leftarrow not\ occupied(X,Y), num(N), row(X), column(Y). \\ hasNum(X,Y) &\leftarrow sol(X,Y,N), row(X), column(Y). \\ \bot &\leftarrow not\ hasNum(X,Y), row(X), column(Y). \end{aligned} \tag{4.51}$$

Further constraints ensure that cells in the same column (4.52) or same row (4.53) do not contain the same symbol.

$$\bot \leftarrow sol(X,Y_1,M), sol(X,Y_2,M), Y_1 < Y_2. \tag{4.52}$$

$$\bot \leftarrow sol(X_1,Y,M), sol(X_2,Y,M), X_1 < X_2. \tag{4.53}$$

Figure 4.15: Abstractions over grid-cells



An additional more involved constraint (see Appendix A) ensures that the cells in the same sub-region also satisfies this condition.

An abstraction over the grid-cells would be to cluster the rows and columns together in order to define an abstract grid-cell. Although abstraction over the sorts one at a time is able to achieve certain abstract cell structures, to obtain more sophisticated abstractions these sorts must be *jointly* abstracted.

Example 4.61. Consider the abstractions in Figure 4.15. Achieving those in Figures 4.15a-4.15b is possible by a mapping over the rows and columns independently such as $m_{row} = m_{col} = \{\{123\}/a_{123}, \{4\}/a_4\}$ and $m_{row} = m_{col} = \{\{12\}/a_{12}, \{34\}/a_{34}\}$. For a given program Π, one can construct the abstract program $(\Pi^{m_{row}})^{m_{col}}$. However to achieve Figure 4.15c, rows and columns must be *jointly* abstracted. While the cells $(a_i, b_j), 1 \leq i, j \leq 2$ are singletons mapped from (i, j), the other abstract regions are only given by

$$m_{row,col}(x, y) = \begin{cases} (a_{12}, b_{34}) & x \in \{1,2\}, y \in \{3,4\} \\ (a_{34}, b_{12}) & x \in \{3,4\}, y \in \{1,2\} \\ (a_{34}, b_{34}) & x \in \{3,4\}, y \in \{3,4\} \end{cases} \tag{4.54}$$

Observe that the abstract row a_{12} describes a cluster that abstracts over the individual abstract rows a_1, a_2. The original rows $\{1, 2\}$ are mapped to $\{a_{12}\}$ only in combination with columns $\{3, 4\}$, otherwise they are mapped to $\{a_1, a_2\}$.

Need for existential abstraction Consider the rule (4.52) standardized apart over rows and columns, to have the relations $X_1 = X_2$ and $Y_1 < Y_2$. For the mapping $m_{row,col}$ (Fig. 4.15c), if these relations are lifted following Section 4.3.2, although the relation over the y-axis is still defined (as A is located above of B), i.e., $A_Y \leq B_Y$, $A_X = B_X$ is unclear as the abstract clusters for X-values are different due to different levels of abstraction.

An existential abstraction over the relations as described in Section 4.3.7 gives the ability to introduce domain mappings over multiple sorts such as

$$m : D_1 \times \ldots \times D_n \to \hat{D}_1 \times \ldots \times \hat{D}_n,$$

and to handle relations over different levels of abstraction.

Computing Joint Abstract Relation Types

Abstract relations can be easily employed with abstraction mappings over several sorts in the domain as $m : D_1 \times \cdots \times D_n \to \hat{D}_1 \times \cdots \times \hat{D}_n$. If a rule has relations over the sorts, a joint abstract relation combining them must be computed. We show an example of grid-cell abstraction for illustration and then extend to the multi-dimensional case.

Example 4.62 (Abstracting grid-cells)**.** Consider the relations $rel_1(X_1, X_2)$: $X_1 = X_2$ and $rel_2(Y_1,\ Y_2)$: $Y_1 < Y_2$ for $X_1, X_2 \in row, Y_1, Y_2 \in column$, from standardizing apart the variables in (4.52). The rules to compute the types $\tau_{\mathrm{I}}^{\widehat{rel}}, \tau_{\mathrm{III}}^{\widehat{rel}}$, where $\widehat{rel}$ combines rel_1 and rel_2, are as follows:

1. Define the abstract relations. This step corresponds to the existential abstraction (4.41).

$$\widehat{rel_1}((\hat{X}_1, \hat{Y}_1), (\hat{X}_2, \hat{Y}_2)) \leftarrow rel_1(X_1, X_2), m((X_1, Y_1), (\hat{X}_1, \hat{Y}_1)), m((X_2, Y_2), (\hat{X}_2, \hat{Y}_2)).$$
$$\widehat{rel_2}((\hat{X}_1, \hat{Y}_1), (\hat{X}_2, \hat{Y}_2)) \leftarrow rel_2(Y_1, Y_2), m((X_1, Y_1), (\hat{X}_1, \hat{Y}_1)), m((X_2, Y_2), (\hat{X}_2, \hat{Y}_2)).$$

   Similarly, the negations $\neg\widehat{rel_1}, \neg\widehat{rel_2}$ are computed as (4.42).

$$\neg\widehat{rel_1}((\hat{X}_1, \hat{Y}_1), (\hat{X}_2, \hat{Y}_2)) \leftarrow \neg rel_1(X_1, X_2), m((X_1, Y_1), (\hat{X}_1, \hat{Y}_1)), m((X_2, Y_2), (\hat{X}_2, \hat{Y}_2)).$$
$$\neg\widehat{rel_2}((\hat{X}_1, \hat{Y}_1), (\hat{X}_2, \hat{Y}_2)) \leftarrow \neg rel_2(Y_1, Y_2), m((X_1, Y_1), (\hat{X}_1, \hat{Y}_1)), m((X_2, Y_2), (\hat{X}_2, \hat{Y}_2)).$$

2. Compute the types of each abstract relation $\widehat{rel_i}, i \in \{1,2\}$ with the objects $\hat{C}_i = (\hat{X}_i, \hat{Y}_i), i \in \{1, 2\}$ as (4.43).

$$\tau_{\mathrm{I}}^{\widehat{rel_i}}(\hat{C}_1, \hat{C}_2) \leftarrow \widehat{rel_i}(\hat{C}_1, \hat{C}_2), not\ \neg\widehat{rel_i}(\hat{C}_1, \hat{C}_2).$$
$$\tau_{\mathrm{II}}^{\widehat{rel_i}}(\hat{C}_1, \hat{C}_2) \leftarrow not\ \widehat{rel_i}(\hat{C}_1, \hat{C}_2), \neg\widehat{rel_i}(\hat{C}_1, \hat{c}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel_i}}(\hat{C}_1, \hat{C}_2) \leftarrow \widehat{rel_i}(\hat{C}_1, \hat{C}_2), \neg\widehat{rel_i}(\hat{C}_1, \hat{C}_2).$$

3. Compute the types of the joint abstract relation $\widehat{rel}$ over $\widehat{rel_i}, i \in \{1, 2\}$:

$$\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{C}_1, \hat{C}_2) \leftarrow \tau_{\mathrm{I}}^{\widehat{rel1}}(\hat{C}_1, \hat{C}_2), \tau_{\mathrm{I}}^{\widehat{rel2}}(\hat{C}_1, \hat{C}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{C}_1, \hat{C}_2) \leftarrow not\ \tau_{\mathrm{II}}^{\widehat{rel1}}(\hat{C}_1, \hat{C}_2), \tau_{\mathrm{III}}^{\widehat{rel2}}(\hat{C}_1, \hat{C}_2).$$
$$\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{C}_1, \hat{C}_2) \leftarrow \tau_{\mathrm{III}}^{\widehat{rel1}}(\hat{C}_1, \hat{C}_2), not\ \tau_{\mathrm{II}}^{\widehat{rel2}}(\hat{C}_1, \hat{C}_2).$$

The mapping (4.54) shown in Figure 4.15c gives the types $\tau_{\mathrm{I}}^{\widehat{rel}}((a_1, b_1), (a_1, b_2)), \tau_{\mathrm{I}}^{\widehat{rel}}((a_2, b_1), (a_2, b_2))$ and $\tau_{\mathrm{III}}^{\widehat{rel}}$ for the remaining abstract pairs.

Note that for the joint abstract relation $\widehat{rel}$, type $\tau_{\mathrm{II}}^{\widehat{rel}}$ computation is not needed, as the abstract rule construction only deals with types I and III.

Multi-dimensional relation types Computing relation types is easily lifted to relations $rel_1, \dots, rel_n$ over variables from $\hat{D}_1, \dots, \hat{D}_n$, respectively. Assuming for simplicity a uniform arity k, the abstract k-tuple relations are computed by

$$\widehat{rel}_i((\hat{d}_1^1, \dots, \hat{d}_1^n), \dots, (\hat{d}_k^1, \dots, \hat{d}_k^n)) \leftarrow rel_i(d_1^i, \dots, d_k^i), \textstyle\bigwedge_{j=1}^k m(((d_j^1, \dots, d_j^n)), ((\hat{d}_j^1, \dots, \hat{d}_j^n))).$$

for $i = 1, \dots, n$. We compute the types of these auxiliary abstract relations, for objects $\hat{c}_j = (\hat{d}_j^1, \dots, \hat{d}_j^n), 1 \leq j \leq k$, as follows:

$$\begin{aligned}
\tau_{\mathrm{I}}^{\widehat{rel}_i}(\hat{c}_1, \dots, \hat{c}_k) &\leftarrow \widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t), not\ \neg\widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t).\\
\tau_{\mathrm{II}}^{\widehat{rel}_i}(\hat{c}_1, \dots, \hat{c}_t) &\leftarrow not\ \widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t), \neg\widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t).\\
\tau_{\mathrm{III}}^{\widehat{rel}_i}(\hat{c}_1, \dots, \hat{c}_t) &\leftarrow \widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t), \neg\widehat{rel}_i(\hat{c}_1, \dots, \hat{c}_t).
\end{aligned}$$

The types of the joint abstract relation $\widehat{rel}$ over the objects $\hat{c}_j = (\hat{d}_j^1, \dots, \hat{d}_j^n), 1 \leq j \leq k$ (i.e. $\tau_{\mathrm{I}}^{\widehat{rel}}$ and $\tau_{\mathrm{III}}^{\widehat{rel}}$), are then computed as below.

$$\begin{aligned}
&\tau_{\mathrm{I}}^{\widehat{rel}}(\hat{c}_1, \dots, \hat{c}_k) \leftarrow \tau_{\mathrm{I}}^{\widehat{rel}_1}(\hat{c}_1, \dots, \hat{c}_k), \dots, \tau_{\mathrm{I}}^{\widehat{rel}_n}(\hat{c}_1, \dots, \hat{c}_k).\\
&\tau_{\mathrm{III}}^{\widehat{rel}}(\hat{c}_1, \dots, \hat{c}_k) \leftarrow \tau_{\mathrm{III}}^{\widehat{rel}_i}(\hat{c}_1, \dots, \hat{c}_k), \textstyle\bigwedge_{j=1:j\neq i}^n not\ \tau_{\mathrm{II}}^{\widehat{rel}_j}(\hat{c}_1, \dots, \hat{c}_k), 1 \leq i \leq n.
\end{aligned}$$

To emphasize on the abstracted relations, we sometimes denote $\widehat{rel}$ in $\tau^{\widehat{rel}}$ with the combination of the relations that the abstract relation is built on; e.g., for Example 4.62 we have $\tau_{\mathrm{I}}^{=,<}, \tau_{\mathrm{III}}^{=,<}$.

The multi-dimensional abstraction constructs an abstract structure, i.e., *object*, over the abstracted sorts where not all combinations of the abstract sorts yields a valid object. For example, in Example 4.60 the color cluster $\widehat{rgb}$ can only be considered with the node cluster $\hat{4}$. This also needs to be taken into account when constructing the abstract program.

Example 4.63. The abstract program for Sudoku (4.51)-(4.53), where the occurrences of $row(X)$, $column(Y)$ are replaced by $cell(X,Y)$, is as follows.

$$\begin{aligned}
hasNum(X,Y) &\leftarrow sol(X,Y,N), cell(X,Y).\\
\{sol(X,Y,N)\} &\leftarrow not\ occupied(X,Y), num(N), cell(X,Y).\\
\{sol(X,Y,N)\} &\leftarrow occupied(X,Y), num(N), isCluster(X), cell(X,Y).\\
\{sol(X,Y,N)\} &\leftarrow occupied(X,Y), num(N), isCluster(Y), cell(X,Y).\\
\bot &\leftarrow not\ hasNum(X,Y), cell(X,Y).\\
\bot \leftarrow sol(X_1,Y_1,M), sol(X_2,Y_2,M), \tau_{\mathrm{I}}^{=,<}&(X_1,Y_1,X_2,Y_2), cell(X,Y_1), cell(X_2,Y_2).\\
\bot \leftarrow sol(X_1,Y_1,M), sol(X_2,Y_2,M), \tau_{\mathrm{I}}^{<,=}&(X_1,Y_1,X_2,Y_2), cell(X_1,Y), cell(X_2,Y_2).
\end{aligned}$$

Different abstraction levels on variables Remember that the bodies of the original rules have to be standardized apart over the sorts on which the abstraction is made, in order to ensure that over-approximation is achieved. However, when the variables can get involved with different levels of abstraction, further adjustments on the rule has to be made.

For example, consider an additional rule for the Sudoku encoding as below

$$middle(X, Y_3, M_1, M_2) \leftarrow sol(X, Y_1, M_1), sol(X, Y_2, M_2), Y_2 < Y_3, Y_3 < Y_1.$$

which distinguishes the cells between (X, Y_1) and (X, Y_2). When the variables in the body are standardized apart over *row* and *column*, we get

$$middle(X, Y_3, M_1, M_2) \leftarrow sol(X, Y_1, M_1), sol(X_1, Y_2, M_2), X = X_1, Y_2 < Y_3, Y_3 < Y_1.$$

where the head atom contains the variable X while also the variable X_1 occurs in the body due to the standardization. For regions B and A in Figure 4.15c, type $\tau_{\mathrm{III}}^{=}(a_{12}, a_1)$ holds for the X values, and for the region that is below A and above B, type $\tau_{\mathrm{I}}^{<,<}(b_1, b_2, b_{34})$ holds for the Y values. If the atoms $sol(a_{12}, b_{34}, m_1)$ and $sol(a_1, b_1, m_2)$ hold true for some numbers m_1, m_2, then the atom $middle(a_1, b_2, m_1, m_2)$ is expected to hold true. However, since the head of the rule only contains the variable X and (a_{12}, b_2) is not a valid cell, this rule is unable to derive the expected atom; an over-approximation can not be obtained.

There are different ways to overcome this issue. One way is to apply standardization apart over the whole rule (with the head), and to treat the uncertainty over the relations. An alternative way is to add additional symmetric rules for the other variables that occur in the body due to the standardization. For the above example, the additional rule

$$middle(X_1, Y_3, M_1, M_2) \leftarrow sol(X, Y_1, M_1), sol(X_1, Y_2, M_2), X = X_1, Y_2 < Y_3, Y_3 < Y_1.$$

gets added to also have the variable X_1 in the head. This way no possible valuation of the (non-ground) head atom is lost in the abstraction process.

In Chapter 6, we describe in detail the implementation for this type of abstraction and investigate its use in understanding problems over grid-cells.

4.7 Discussion

In this chapter, we have introduced two notions of abstraction to Answer Set Programming and we have defined methods to apply them on a given answer set program in order to achieve an over-approximation of the original program. Such an abstraction notion opens a wide-range of applications, as it can be used in getting rid of irrelevant details of a program by preserving its original behavior. Depending on the abstraction, the over-approximation may result in obtaining spurious abstract answer sets. To tackle this, we have described a CEGAR-style abstraction refinement methodology that starts with a coarse abstraction and refines the abstraction upon encountering spurious abstract answer sets. This way, one can start with some initial abstraction in mind, and then let the method find a refinement of it which returns a concrete solution.

Two Forms of Relation Types In domain abstraction, we introduced two approaches in applying abstraction to the rules. The first approach is more intuitive in the sense that the built-in relations in the rules are meant to be kept when constructing the abstract rules. This requires to lift the relations to the abstract domain, and the result of this lifting has to be taken into account to achieve an over-approximation. The steps needed to achieve this as described in Section 4.3.1 are intuitive. With this approach, the abstract program still contains the built-in relations in the rules, which keeps the similarity for the original rules.

The existential abstraction approach takes a more straightforward view on abstracting the relations. Given that there are fewer relation types to consider, we conducted experiments to observe whether there is any gain in computation. The experiments showed no improvement in the computation effort, since type III for abstract relations is a combination of type III and IV of lifted relations. With this approach, the constructed abstract program loses the format of the relations in the rules by only keeping the auxiliary relation type atom. However, such an existential abstraction gives the ability to handle different levels of abstractions among the abstract elements as shown in Section 4.6.

4.7.1 Other Abstraction Possibilities

Composing the Abstractions

The two abstraction notions can easily be combined to obtain an abstraction that omits certain details and also abstracts over some part of the domain. This can be achieved with the current definitions, by first applying the desired domain abstraction to the program and then grounding the constructed non-ground abstract program to omit some of the atoms from it. After constructing the abstract ground program and computing an answer set, the correctness checking would have to be done on the original non-ground program. However, since some of the atoms are omitted, the abstract answer set will not contain these atoms. Thus, the correctness checking needs to take into account the omitted atoms, and check for the concreteness only over the remaining atoms.

Deciding on a refinement can still be achieved by non-ground debugging, where there will be two main causes for abnormalities. One will be the bad clustering of domain elements, while the other one will be the bad omission of atoms. Thus, obtaining the refinement hints has to be altered to take into account these both cases. Depending on the outcome of debugging the correctness checking, a decision can be made either to refine the domain, or to add back some atoms, or to do both.

Predicate Abstraction

Applying predicate abstraction to ASP would be to introduce new literals that desribe an abstraction of some of the original literals, and rewriting the given answer set program to only mention the newly introduced literals. The naive way of replacing the literals with the abstract ones would not always achieve an over-approximation, due to the same reasoning as for domain abstraction.

Example 4.64 (Example 4.15 ctd)**.** Consider a predicate abstraction that maps the atoms $a(X)$ and $d(X)$ to $ad(X)$, i.e., if $a(d)$ or $d(d)$ holds true for some $d \in D$, then $ad(d)$ holds true. When we replace the a- and d-atoms in Example 4.15, we get the following program:

$$\begin{aligned}
c(X) &\leftarrow \mathit{not}\ ad(X), X < 5, \mathit{int}(X).\\
ad(X) &\leftarrow \mathit{not}\ c(X), \mathit{int}(X).\\
b(X,Y) &\leftarrow ad(X), ad(Y), \mathit{int}(X), \mathit{int}(Y).\\
e(X) &\leftarrow c(X), ad(Y), X \leq Y, \mathit{int}(X), \mathit{int}(Y).\\
\bot &\leftarrow b(X,Y), e(X), \mathit{int}(X), \mathit{int}(Y).\\
ad(1).\ \ ad(3).&\ \ \mathit{int}(0), \dots, \mathit{int}(5).
\end{aligned}$$

However, in all of the answer sets of this program $b(5,1)$ and $b(5,3)$ holds true, and none of the original answer sets has a matching abstract answer set.

Further considerations are necessary according to the types of the abstract literals due to the abstraction, similar to how it is done for domain abstraction.

A simple way to achieve predicate abstraction is to apply reification of predicates of the original program. E.g., $p(X,Y)$ gets written into $x(p,X,Y)$. Then one considers the constants for the predicate names as a sort. By standardizing apart the variables, the clustering of the predicate names can be handled over the relations.

Example 4.65 (ctd)**.** A rewriting of the atoms $a(X)$ and $d(X)$ to the form $x(a,X)$ and $x(d,X)$ achieves the program

$$\begin{aligned}
c(X) &\leftarrow \mathit{not}\ x(d,X), X < 5, \mathit{int}(X).\\
x(d,X) &\leftarrow \mathit{not}\ c(X), \mathit{int}(X).\\
b(X,Y) &\leftarrow x(P_1,X), x(P_2,Y), P_1 = a, P_2 = d, \mathit{pred}(P_1), \mathit{pred}(P_2), \mathit{int}(X), \mathit{int}(Y).\\
e(X) &\leftarrow c(X), x(P,Y), P = a, \mathit{pred}(P), X \leq Y, \mathit{int}(X), \mathit{int}(Y).\\
\bot &\leftarrow b(X,Y), e(X), \mathit{int}(X), \mathit{int}(Y).\\
x(a,1).&\ \ x(a,3).\ \ \mathit{int}(0), \dots, \mathit{int}(5).\\
\mathit{pred}(a).&\ \ \mathit{pred}(d).
\end{aligned}$$

Then an abstraction m over the sort pred such as $\{a,d\}/ad$ can be applied.

This works for predicate abstraction where the corresponding literals have arguments from the same sort in the same index. In case a literal contains fewer arguments, then dummy values can be used to fill in the remaining indices.

Another possibility to achieve predicate abstraction is by following the motivation behind existential abstraction of the relations and to introduce a new set of predicates by also introducing their relation types according to the abstraction. Then the abstract rules will be formed for all combinations of the types of the abstract predicates, i.e., no choice in the head for having only type I relations of all abstract predicates, and adding choice to the head for the remaining cases.

Example 4.66 (ctd)**.** Similar to Example 4.64 the abstract predicate name *ad* is introduced with the relation type τ_{III}^{ad} and the additional fact $\mathit{isCluster}^{ad}$. Note that the arguments of the literal are not of importance, as the abstraction is over the predicate name, not on a domain element. The constructed abstract program then becomes as follows.

$$\begin{aligned}
c(X) &\leftarrow not\ ad(X), X < 5, int(X).\\
\{c(X)\} &\leftarrow ad(X), \mathit{isCluster}^{ad}, X < 5, int(X).\\
ad(X) &\leftarrow not\ c(X), int(X).\\
\{b(X,Y)\} &\leftarrow ad(X), ad(Y), \tau_{\mathrm{III}}^{ad}, int(X), int(Y).\\
\{e(X)\} &\leftarrow c(X), ad(Y), \tau_{\mathrm{III}}^{ad}, X \leq Y, int(X), int(Y).\\
\bot &\leftarrow b(X,Y), e(X), int(X), int(Y).\\
ad(1).&\ \ ad(3).\ \ int(0), \dots, int(5).
\end{aligned}$$

We remark that this approach is similar to using the rewriting of the original program with reification of predicates and applying existential abstraction on the relations.

4.7.2 Related Work in ASP

The most related work to abstraction in ASP are the simplification methods that strive for preserving the semantics. These methods have been extensively studied over the years, here we give an overview of the notions. Notice that, different from these simplification methods, abstraction may lead to an over-approximation of the models (answer sets) of a program, which changes the semantics, in a modified language.

Over-approximation by abstraction reduces the vocabulary which makes it different than the *relaxation* methods [LZ04, GLM04]. These methods translate a ground program into its completion [Cla78] and search for an answer set over the relaxed model. As they focus only on ground programs, they can be compared with the abstraction that omits atoms from the program, which does not need to take into account the computation of the loop formulas when searching for a concrete abstract answer set. However, finding the reason for spuriousness of an abstract answer set is trickier than finding the reason for a model of the completion not being an answer set of the original program, since the abstract answer set contains fewer atoms and a search over the original program is needed to detect the reason why no matching answer set can be found.

Equivalence-based rewriting

The equivalence of logic programs is considered in the sense of the answer set semantics: a program Π_1 is equivalent to a program Π_2 if $AS(\Pi_1) = AS(\Pi_2)$. *Strong equivalence*

[LPV01] is a much stricter condition over the two programs: Π_1 and Π_2 are strongly equivalent if, for any set R of rules, the programs $\Pi_1 \cup R$ and $\Pi_2 \cup R$ are equivalent. This is the notion that makes it possible to simplify a part of a logic program without looking at the rest of it: if a subprogram Q of Π is strongly equivalent to a simpler program Q', then the Q is replaced by Q'. The works [ONA02, Tur03, EFTW04, Pea04] show ways of transforming programs by ensuring that the property holds. A more liberal notion is *uniform equivalence* [Mah86, Sag87] where R is restricted to a set of facts. Then, a subprogram Q in Π can be replaced by a uniformly equivalent program Q' and the main structure will not be affected [EF03].

In terms of abstraction, there is the abstraction mapping that needs to be taken into account, since the constructed program may contain a modified language and the mapping makes it possible to relate it back to the original language. Thus, in order to define equivalence between the original program Π and its abstraction $\widehat{\Pi}^m$ according to a mapping m, we need to compare $m(AS(\Pi))$ with $AS(\Pi^m)$. The equivalence of Π and $\widehat{\Pi}^m$ then becomes similar to the notion of faithfulness. However, as we have shown, even if the abstract program $\widehat{\Pi}^m$ is faithful, refining m may lead to an abstract program that contains spurious answer sets. Thus, simply lifting the current notions of equivalence to abstraction may not achieve useful results.

Refinement-safe faithfulness however is a property that would allow one to make use of $\widehat{\Pi}^m$ instead of Π, since it preserves the answer sets. This property can immediately be achieved when a constructed abstract program is unsatisfiable (which then shows that original program was unsatisfiable). However, for original programs that are consistent, reaching an abstraction that is refinement-safe faithful is not easy; adding an atom back or dividing the domain cluster may immediately cause to reach a guessing that introduces spurious solutions.

Other program transformations

Other transformation methods, especially to help with the grounding and solving of the programs, have also been investigated. A preprocessing technique was considered in [GKNS08] which transforms a program into a simpler one, along with an assignment and a relation expressing equivalences among the assignable constituents of the program. Another form of preprocessing in [MW12] (later extended to full ASP syntax [BMW16]) was applied to each rule of a program by computing the tree decomposition of a rule, and then splitting the rule up into multiple, smaller rules according to this decomposition.

The unfolding method for disjunctive programs in [JNS$^+$06] is similar in spirit to our approach of introducing choice to the head for uncertainties. For a given disjunctive program P, the authors create a *generating program* that preserves completeness. With this program they generate model candidates (but they may also get “extra” candidate models, which do not match the stable models of P). Then they *test* for stability of these candidates. For testing a candidate model M, another normal program $\mathit{Test}(P, M)$ is built such that $\mathit{Test}(P, M)$ has no stable models if and only if M is a stable model of the original disjunctive program P. Thus, testing the stability is reduced to testing the nonexistence of stable models for $\mathit{Test}(P, M)$. However, this approach does not consider omission of atoms from the disjunctive rules when creating the new program; the authors further extend the vocabulary with auxiliary atoms. They build the model candidate gradually by starting from an empty partial interpretation and extending it step by step. For this, they use the observation that if for the extension M of the partial interpretation by assigning false to the undefined atoms, $\mathit{Test}(P, M)$ has a stable model, then P has no stable model $M' \supset M$. When compared with the notions introduced in the omission-based abstraction, this technique would give a more restricted notion of spuriousness of an abstract answer set, since the omitted atoms would be assigned to false.

Forgetting

Forgetting is an important operation in knowledge representation and reasoning, which has been studied for many formalisms and is a helpful tool for a range of applications, cf. [Del17, EKI18]. The aim of forgetting is to reduce the signature of a knowledge base, by removing symbols from the formulas in it (while possibly adding new formulas) such that the information in the knowledge base, given by its semantics that may be defined in terms of models or a consequence relation, is invariant with respect to the remaining symbols; that is, the models resp. consequences for them should not change after forgetting.

Due to nonmonotonicity and minimality of models, forgetting in ASP turned out to be a nontrivial issue. It has been extensively studied in the form of introducing specific operators that follow different principles and obey different properties; we refer to [GKLW17, Lei17] for a survey and discussion. The main aim of forgetting in ASP as such is to remove/hide atoms from a given program, while preserving its semantics for the remaining atoms. As atoms in answer sets must be derivable, this requires to maintain dependency links between atoms. For example, forgetting the atom b from the program $\Pi = \{a \leftarrow b.; b \leftarrow c.\}$ is expected to result in a program Π' in which the link between a and c is preserved; this intuitively requires to have the rule $a \leftarrow c$ in Π'. The various properties that have been introduced as postulates or desired properties for an ASP forgetting operator mainly serve to ensure this outcome; forgetting in ASP is thus subject to more restrictive conditions than abstraction.

Relation with omission abstraction Atom omission as we consider it is different from forgetting in ASP as it aims at a deliberate over-approximation of the original program that may not be faithful; furthermore, our omission does not resort to language extensions such as nested logic programs that might be necessary in order to exclude non-faithful abstraction; notably, in the ASP literature under-approximation of the answer sets was advocated if no language extensions should be made [EW08].

Only more recently, over-approximation has been considered as a property of forgetting in ASP in [DW15], which was later named *Weakened Consequence (WC)* in [GKL16]:

(WC) Let Π be a disjunctive logic program, let A be a set of atoms, and let X be an answer set for Π. Then $X \setminus A$ is an answer set for $\mathit{forget}(\Pi, A)$.

That is, $AS(\Pi)|_{\overline{A}} \subseteq AS(\mathit{forget}(\Pi, A))$ should hold. This property amounts to the notion of over-approximation that we achieve in Theorem 4.2. However, according to [GKL16], this property is in terms of proper forgetting only meaningful if it is combined with further axioms. Our results may thus serve as a base for obtaining such combinations; in turn, imposing further properties may allow us to prune spurious answer sets from the abstraction.

Unsatisfiable Cores in ASP

Before we conclude, we remark the relation of unsatisfiable cores with our notion of abstraction and its use in finding the cause for unsatisfiability.

A well-known notion for unsatisfiability are minimal unsatisfiable subsets (MUS), also known as *unsatisfiable cores* [LS08, LS04]. The latter are based on computing, given a set of constraints respectively formulas, a minimal subset of the constraints that explains why the overall set is unsatisfiable. Unsatisfiable cores are helpful in speeding up automated reasoning, but have beyond many applications and a key role e.g. in model-based diagnosis [Rei87] and in consistent query answering [ABC99].

In ASP, unsatisfiable cores have been used in the context of computing optimal answer sets [AD16, AKMS12], where for a given (satisfiable) program, weak constraints are turned into hard constraints; an unsatisfiable core of the modified program that consists of rewritten constraints allows one to derive an underestimate for the cost of an optimal answer set, since at least one of the constraints in the core can not be satisfied. However, if the original program is unsatisfiable, such cores are pointless. In the recent work [ADJ$^+$18], unsatisfiable core computation has been used for implementing cautious reasoning. The idea is that modern ASP solvers allow one to search, given a set of assumption literals, for an answer set. In case of failure, a subset of these literals is returned that is sufficient to cause the failure, which constitutes an unsatisfiable core. Cautious consequence of an atom amounts then to showing that the negated atom is an unsatisfiable core.

Relation to spurious answer sets in omission abstraction Intuitively, unsatisfiable cores are similar in nature to spurious abstract answer sets, since the latter likewise to not permit to complete a partial answer set to the whole alphabet. More formally, their relationship is as follows.

Technically, an *unsatisfiable (u-) core* for a program Π is an assignment I over a subset $C \subseteq \mathcal{A}$ of the atoms such that Π has no answer set J that is compatible with I, i.e., such that $J|_C = I$ holds. We then have the following property.

Proposition 4.40. *Suppose that for a program* Π *and a set* A *of atoms* $\hat{I} \in AS(\mathit{omit}(\Pi, A))$ *holds. If* $\hat{I}$ *is spurious, then* $\hat{I}$ *is a u-core of* Π *(w.r.t.* $\mathcal{A} \setminus A$*). Furthermore, if* A *is* *maximal, i.e., no* $A' \supset A$ *exists such that* $omit(\Pi, A')$ *has some spurious answer set* $\widehat{I}'$ *such that* $\hat{I}|_{\overline{A'}} = \widehat{I}'$*, then* $\hat{I}$ *is a minimal core.*

Proof. The abstract answer set $\hat{I}$ describes an assigment over $\mathcal{A} \setminus A$; $\hat{I}$ being spurious means that there is no answer set J in Π such that $J|_{\mathcal{A}\setminus A} = \hat{I}$, thus $\hat{I}$ is a u-core. Now assume A is maximal, and that there exists an assignment $\hat{I}' \subset \hat{I}$ which is a u-core for Π w.r.t. $\mathcal{A} \setminus A'$ for some $A' \supset A$. Let $\alpha \in \hat{I} \setminus \hat{I}'$, then $\alpha \in A' \setminus A$. Thus, by over-approximation, we know that $\hat{I}|_{\overline{A'}} = \hat{I}' \in AS(omit(\Pi, A'))$ must hold. Since $\hat{I}'$ is an answer set of $AS(omit(\Pi, A'))$, the fact that it is a u-core means that $\hat{I}'$ is spurious. By this, we reach a contradiction to the assumption that A is maximal. □

That is, spurious answer sets are u-cores; however, the converse fails in that cores C are not necessarily spurious answer sets of the corresponding omission $A = \mathcal{A} \setminus \mathcal{A}(C)$, where $\mathcal{A}(C)$ are the atoms that occur in C. E.g., for the program with the single rule

$$r : a \leftarrow b, not\, a.$$

the set $C = \{b\}$ is a core, while C is not an answer set of $omit(\{r\}, \{a\})) = \emptyset$. Intuitively, the reason is that C lacks foundedness for the abstraction, as it assigns b true while there is no way to derive b from the rules of the program, and thus b must be false in every answer set. As C is a minimal u-core, the example shows that also minimal u-cores may not be spurious answer sets.

Thus, spurious answer sets are a more fine-grained notion of relative inconsistency than (minimal) u-cores, which accounts for a notion of weak satisfiability in terms of the abstracted program. In case of an unsatisfiable program Π, each blocker set C for Π naturally gives rise to u-cores in terms of arbitrary assignments I to the atoms in $\mathcal{A} \setminus C$; in this sense, blocker sets are conceptually a stronger notion of inconsistency explanation than u-cores, in which minimal blocker sets and minimal u-cores remain unrelated in general.

CHAPTER 5

Applications in Problem Analysis

In this chapter, we investigate possible applications of abstraction in understanding the key elements of problems, by abstracting away as many irrelevant details as possible that may be traced before finding a solution of a problem or realizing that it is unsolvable.

Outline In Section 5.1, we provide details of our implementations for the abstraction and refinement methods introduced in Chapter 4 for different types of abstractions. Section 5.2 shows the evaluation results by focusing on the achievement of abstract solutions to the problems. Section 5.2.1 shows the results of using omission abstraction in finding satisfiability blockers of programs, and Section 5.2.2 reports about the achieved non-trivial domain abstractions and the results of having variations in the methodology. In Section 5.3, we discuss the use of domain abstraction in understanding planning problems expressed in ASP. We conclude with a discussion in Section 5.4.

5.1 Implementation

We have implemented the following three prototypical systems to instantiate and experiment with the abstraction and refinement methods described in Chapter 4:

1. ASPARO - Omission Abstraction and Refinement for ASP Programs
2. DASPAR - Domain Abstraction for ASP Programs
3. mDASPAR - Multi-dimensional Domain Abstraction for ASP Programs

The systems are implemented using Python and Clingo, and are available at `http://www.kr.tuwien.ac.at/research/systems/abstraction/`.

Figure 5.1 shows the system structure of the tools according to procedures shown in Section 4.5, with the respective components (for ASPARO and (m)DASPAR, they are slightly different). The arcs model both control and data flow within the tool. The

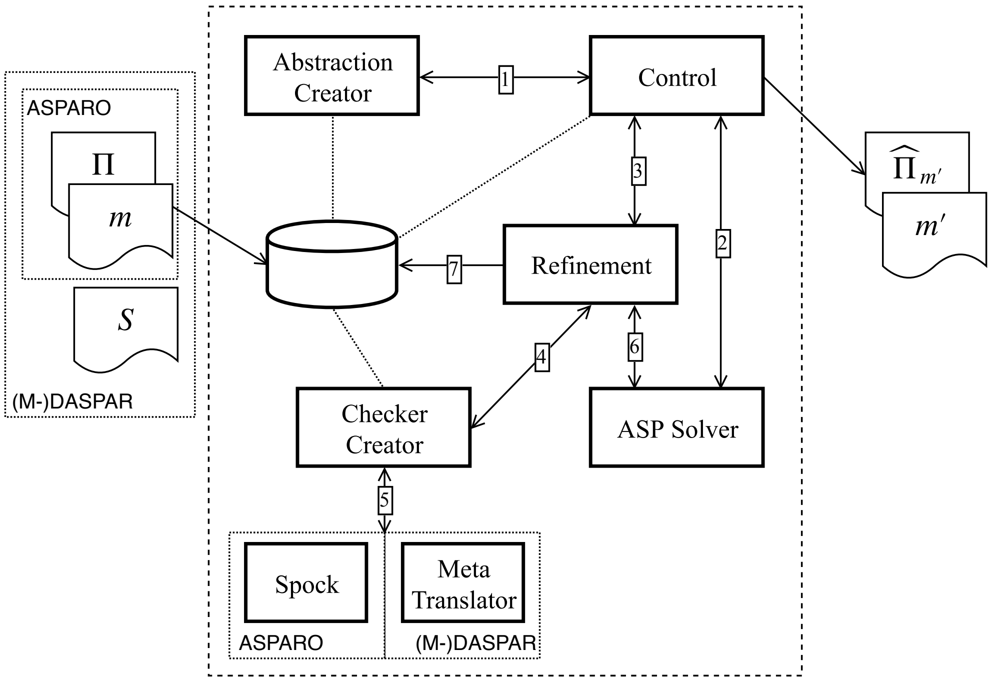


Figure 5.1: System structure of the implementation

general workflow of the tools is as follows. First, the input program Π and the mapping m are read. For (m)DASPAR an additional input S of which sorts to do the abstraction is given. Then the *control component* calls the *abstraction creator* component which uses Π and m to create the abstract program $\widehat{\Pi}_m$ $\boxed{1}$. The controller then calls the *ASP Solver* to get an answer set of $\widehat{\Pi}_m$ $\boxed{2}$. If the solver finds no answer set, the controller outputs the abstract program and m. Otherwise, it calls the *refinement* component with the abstract answer set $\hat{I}$ to check spuriousness and to decide whether or not to refine the abstraction $\boxed{3}$. The refinement component calls the *checker creator* $\boxed{4}$ to create the debugging program, which for ASPARO uses Spock $\boxed{5}$ and for (m)DASPAR uses a MetaTranslator to obtain the reification of the program. Then the ASP solver is called to check whether $\hat{I}$ is concrete $\boxed{6}$. If not, i.e., when $\hat{I}$ is spurious, it refines the abstraction by updating m $\boxed{7}$. Otherwise, the controller returns the outputs.

For further implementation issues considered in mDASPAR and its use for problem solving, we refer to Chapter 6.

5.1.1 ASPARO

ASPARO is called with two parameters, the ground program Π and a set A of atoms to be omitted from Π, describing m. The format of Π should adhere to the input requirements of SPOCK[1].

Given the input, ASPARO first creates the abstract program as described in Section 4.2 and gets an abstract answer set. In order to check the correctness of the answer set, it calls SPOCK to get the debugging program and alters it as described in Section 4.4.1. In case the answer set is spurious, the debugging program determines some atoms to be badly omitted, then updates the abstraction by adding them back to the vocabulary in the next iteration. The loop continues until either unsatisfiability is observed, or a concrete abstract answer set is encountered.

Example 5.1. Consider the program $\Pi = \{c \leftarrow \mathit{not}\ d., d \leftarrow \mathit{not}\ c., a \leftarrow \mathit{not}\ b, c., b \leftarrow d, e., e \leftarrow \mathit{not}\ a.\}$ and an initial omission abstraction of $\{b, e\}$. The input program (say `ex.lp`) of ASPARO would be

```
r1: c :- not d.
r2: d :- not c.
r3: a :- not b,c.
r4: b :- d,e.
r5: e :- not a.
```

ASPARO is called with

```
python asparo.py ex.lp b,e.
```

An example run of the system is as follows. First the abstract program $\Pi|_{\overline{\{b,e\}}} = \{c \leftarrow \mathit{not}\ d., d \leftarrow \mathit{not}\ c., \{a\} \leftarrow c.\}$ is created, and the (spurious) abstract answer set $\{c\}$ is computed. When checking its correctness, the atom b is determined as badly omitted. Then the abstraction is refined by adding b back to the vocabulary. In the new abstract program $\Pi|_{\overline{\{e\}}} = \{c \leftarrow \mathit{not}\ d., d \leftarrow \mathit{not}\ c., a \leftarrow \mathit{not}\ b, c.\{b\} \leftarrow d.\}$, the concrete abstract answer set $\{c, a\}$ is encountered, which ends the run.

5.1.2 DASPAR

DASPAR consists of three separate Python scripts. There is a script that creates the abstract program following the method described in Sections 4.3.2-4.3.3, which is called with three parameters: a non-ground program Π, the abstraction mapping m described as facts, and the name of the sort to be abstracted. Another script is used to create the debugging program following Section 4.4.2. The main script applies the overall abstraction refinement methodology.

In order to guarantee obtaining an over-approximation, the format of the input program Π should adhere to certain restrictions. Each variable in a rule should be guarded by a domain predicate. If there are several sorts in the program and the abstraction should

[1] `http://www.kr.tuwien.ac.at/research/systems/debug/index.html`. The most important restriction is to have a labelling of the rules, to allow the debugging mechanism to explicitly refer to rules.

be done on some subset S of sorts, then the variables in the rules referring to the sorts in S have to be standardized apart. For example, a rule of form

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_2), X \leq X_1.$$

needs to be converted into

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_3), X \leq X_1, X_2 = X_3.$$

where each variable is supported by the domain predicate *dom*. In order to support the case of having more than one relation, a syntactic change on the rule has to be made. These relations need to be combined into an auxiliary relation atom which represents the combination of the relations. The above rule needs to be converted into

$$a(X) \leftarrow b(X, X_1), c(X_2), d(X_3), \mathit{leqEqu4}(X, X_1, X_2, X_3).$$

where $\mathit{leqEqu4}(X, X_1, X_2, X_3)$ is an auxiliary atom which holds true whenever the respective relation holds true for its arguments. The supported auxiliary relation combinations can be found in the description, and more can easily be added.

The abstraction mapping m needs to be described in a separate file. This file should contain facts of the form `dom(sort,original element)` describing the original domain and `mapTo(sort,original element,abstract element)` describing the mapping. For example, the mapping $m = \{\{1, 2, 3\}/a_0, \{4, 5\}/a_1\}$ is described as

```
dom(dom,1..5).  mapTo(dom,1..3,a_0).[2]  mapTo(dom,4..5,a_1).
```

When DASPAR applies an abstraction mapping over a sort, it does not consider the full mapping over the whole domain (i.e. Herband universe) that maps the remainder elements different from the sort elements to singletons. Therefore, DASPAR currently only supports programs with rules that contain no relation atoms with arguments as constants or variables from different sorts.

Initially, in v0.1, DASPAR was built following the naive debugging method. In v0.2, the sophisticated debugging method is considered. For this, DASPAR uses the MetaTranslator of the debugging tool Ouroboros [OPT10] to obtain the reification of the program, and then constructs the debugging program.

DASPAR is invoked as follows.

```
python daspar.py prog mapping pred ref_type <focus_atoms>
```

Here `prog` should contain the original program in the input format and the `mapping` should contain the abstraction mapping information. Currently, DASPAR supports abstraction on one sort, thus `pred` should be the name of the sort to the abstracted. The parameter `ref_type` is the option of specifying whether the refinement should respect an order (`1`) or not (`0`). If `1` is given, the refinement step considers only splitting the domain, while when `0` is given the refinement step is not restricted. The parameter <`focus_atoms`> is an optional input in which one can give the predicate names of the atoms that are of importance. DASPAR uses this information when checking the

[2]One can use `1..3` (a range) to compactly state facts.

correctness of the abstract answer sets by projecting the checking only to these focus atoms.

There are different settings of DASPAR for picking abstract answer sets and for deciding on a refinement. For picking an abstract answer set to check for correctness, the default setting is to pick the first computed answer set. It is also possible to change this setting to consider a diverse set of abstract answer sets. To foster diversity a Clingo API is used to guide the answer set seach by randomly picking one atom in an answer set and adding it as *nogood*, which excludes re-occurrence of the atom. In case `focus_atoms` is provided, the diversity is done only on these atoms. With the diverse setting, DASPAR computes at most 50 diverse abstract answer sets, and randomly picks 5 of them for checking. For deciding on a refinement, the two forms mentioned in Section 4.4.2 are implemented. Either a search over the cost of the refinements is done and the refinement with the minimum cost is picked, or hints are obtained from the debugging and refinement is made according to the hints. Later, we evaluate the effects of having these different settings in the methodology on the achieved resulting abstractions.

For practical purposes, sorts can use overlapping elements of the domain, provided that all occurrences of the sort are guarded by domain predicates. For example, the blocksworld problem contains sorts *block* and *time* where both sorts can use integers. Note that this restriction is to help the machine know about the relations of the arguments, which is something that the user implicitly knows when encoding the problem. With this guidance, it becomes clear over which arguments in the rule the abstraction should focus on.

5.2 Evaluation

The assessment of the introduced abstraction and refinement procedure focuses on the achievement of abstract solutions to the problems. We evaluated the omission-based abstraction approach for unsatisfiable problems to observe its use in finding the unsatisfiability reason of a problem by keeping the atoms relevant for the unsatisfiability. The evaluation of the domain abstraction approach focuses on the achieved (non-trivial) abstractions on which a concrete solution to the problem is encountered.

5.2.1 Finding Satisfiability Blockers of Programs

The computation of a $\subseteq$-minimal blocker set of an unsatisfiable program (introduced in Section 4.2.5), given an initial set of omission atoms A, is shown in Algorithm 5.1. It takes into account that minimal blocker sets amount to minimal put-back sets for unsatisfiability. The procedure checks whether omitting an atom $\alpha \in \mathcal{A} \setminus A$ from Π preserves unsatisfiability. If yes, the atom is added to A and the search continues from the newly constructed abstract program $omit(\Pi, \{\alpha\})$. Once all the atoms are examined, the atoms that are chosen not to be omitted, $\mathcal{A} \setminus A$, form a $\subseteq$-minimal blocker set, provided that $AS(\Pi, A)$ is unsatisfiable.

Algorithm 5.1: *ComputeMinBlocker*

```
   Input: $\Pi$, $\mathcal{A}$, $A$ s.t. $AS(\Pi, A) = \emptyset$
   Output: a $\subseteq$-minimal blocker set $C_{\min} \subseteq \mathcal{A} \setminus A$
1  forall $\alpha \in \mathcal{A} \setminus A$ do
2      $\Pi' = constructAbsProg(\Pi, \{\alpha\})$;
3      if $AS(\Pi') = \emptyset$ then
4          $A = A \cup \{\alpha\}$;
5          $\Pi = \Pi'$;
6  return $C_{\min} = \mathcal{A} \setminus A$
```

Experiments

In our experiments, we wanted to observe the use of abstraction in catching the part of the program which causes unsatisfiability. We aimed at studying how the abstraction and refinement method behaves in different benchmarks in terms of the computed final abstractions and the needed refinement steps, when starting with an initial omission of a random set of atoms.

For the refinement step, we used the debugging approach described in Section 4.4.1 and searched for an answer set with minimum number of *badomit* atoms, which are the debugging atoms that tell which omitted atoms should be added back in the refinement. As we expected this search to be difficult, we wanted to investigate whether different minimizations over the *badomit* atom number makes a difference in the final abstractions reached.

Additionally, we were interested in computing the $\subseteq$-minimal blocker sets of the programs and observing the difference in size of the $\subseteq$-minimal blocker sets depending on the problems. For finding $\subseteq$-minimal blocker sets, we moreover compared a *top-down* method to a *bottom-up* method, to see their effects on the quality of the resulting $\subseteq$-minimal blocker sets:

- The top-down method proceeds by calling the function *ComputeMinBlocker* with the original program Π, the set $\mathcal{A}$ of atoms and $A = \{\}$, so that the search for a $\subseteq$-minimal blocker set starts from the top.
- The bottom-up method initially chooses a certain percentage of the atoms to omit, A_{init}, and calls the function *Omission-Abs&Ref* with Π and A_{init} to refine the abstraction and find an unsatisfiable abstract program, $omit(\Pi, A_{final})$. Then, a search for $\subseteq$-minimal blocker sets is done with the remaining atoms, by calling the function *ComputeMinBlocker* with $omit(\Pi, A_{final})$, $\mathcal{A}$ and A_{final}.

We wanted to observe whether there are cases where the bottom-up method helps in reaching $\subseteq$-minimal blocker sets of better quality in terms of smaller size than those obtained with the top-down method.

Benchmarks We considered five benchmark problems with a focus on the unsatisfiable instances. Two of the problems are based on graphs, two are scheduling and planning problems, respectively, and the fifth one is a subset selection problem.

Graph Coloring (GC). We obtained the generator for the graph coloring problem[3] that was submitted to the ASP Competition 2013 [ACC+13], and we generated 35 graph instances with node size varying from 20 to 50 with edge probability 0.2 to 0.6, which are not 2 or 3-colorable. The respective colorability tests are added as superscripts to GC, i.e, GC^2, GC^3.

Abstract Argumentation (AA). Abstract argumentation frameworks are based on graphs to represent and reason about arguments [Dun95]. The abstract argumentation research community has a broad collection of benchmarks with different types of graph classes, which are also being used in competitions [GLMW16]. We obtained the Watts-Strogatz (WS) instances [WS98] that were generated by [CGV16] and are unsatisfiable for existence of so called stable extensions.[4] We focused on the unsatisfiable (in total 45) instances with 100 arguments (i.e., nodes) where each argument is connected (i.e., has an edge) to its $n \in \{6, 12, 18\}$ nearest neighbors with probability 1 and to the remaining arguments with a probability $\beta \in \{0.10, 0.30, 0.50, 0.70, 0.90\}$.

Disjunctive Scheduling (DS). As a non-graph problem, we considered the task scheduling problem from the ASP Competition 2011 [CIR+11] and generated 40 unsatisfiable instances with $t \in \{10, 20\}$ tasks within $s \in \{20, 30\}$ time steps, where $d \in \{10, 20\}$ tasks are randomly chosen to not to have overlapping schedules.

Strategic Companies (SC). We considered the strategic companies problem with the encoding and simple instances provided in [ELM+98]. In order to achieve unsatisfiability, we added a constraint to the encoding that forbids having all of the companies that produce one particular product to be strategic. *SC* is a canonic example of a disjunctive program that has presumably higher computational cost than normal logic programs, and no polynomial time encoding into such programs is feasible. We have thus split rules with disjunctive heads, e.g., $a \vee b \leftarrow c$, into choice rules $\{a\} \leftarrow c; \{b\} \leftarrow c$ at the cost of introducing spurious guesses and answer sets. The resulting split program can be seen as an over-approximation of the original program, and thus causes for unsatisfiability of the split program can be seen as approximative causes for unsatisfiability of the original program.

15-puzzle (PZ). Inspired from the Unsolvability International Planning Competition,[5] we obtained the ASP encoding for the Sliding Tiles problem from the ASP Competition 2009 [DVB+09], which is named as 15-puzzle. We altered the encoding in order to avoid having cardinality constraints in the rules, and to make it possible to solve also non-square

[3]`www.mat.unical.it/aspcomp2013/GraphColouring`

[4]`www.dbai.tuwien.ac.at/research/project/argumentation/systempage/Data/stable.dl`

[5]`https://unsolve-ipc.eng.unimelb.edu.au/`

Figure 5.2: Experimental results for the base case (i.e., with upper limit on *badomit* # per step). The three entries in a cell, e.g., 0.49 / 0.74 / 1.00 in cell (GC^2, $\frac{|A_{init}|}{|\mathcal{A}|}$), are for 50% / 75% / 100% initial omission.

Π	$\frac{\lvert A_{init}\rvert}{\lvert\mathcal{A}\rvert}$	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathcal{A}\rvert}$	t (sec)
GC^2	0.49	0.49	0.02	0.81	0.10	0.80
	0.74	0.63	0.51	1.13	0.10	0.51
	1.00	0.18	3.03	3.60	0.10	1.63
	top-down				0.10	2.30
GC^3	0.49	0.40	0.82	1.83	0.17	1.68
	0.72	0.31	2.46	5.87	0.16	2.04
	1.00	0.11	4.18	6.54	0.17	3.47
	top-down				0.16	4.32
AA	0.50	0.19	3.70	7.20	0.38	8.90
	0.75	0.20	4.19	8.41	0.37	8.67
	1.00	0.01	2.00	4.07	0.38	11.74
	top-down				0.38	11.75
DS	0.50	0.39	1.62	3.36	0.10	1.89
	0.72	0.40	3.49	6.77	0.09	2.09
	1.00	0.45	4.90	9.57	0.07	1.99
	top-down				0.09	4.15
SC	0.49	0.48	0.03	0.59	0.10	0.34
	0.74	0.42	0.65	1.14	0.10	0.41
	1.00	0.43	1.00	2.65	0.11	0.40
	top-down				0.12	0.82
PZ	0.36	0.32	3.76	65.10	0.29	150.10
	0.54	0.45	8.47	154.10	0.27	103.70
	0.76	0.54	22.85	448.60	0.26	80.00
	top-down				0.30	281.40

instances. We used the 20 unsolvable instances from the planning competition, which consist of 10 instances of 3x3 and 10 instances of 4x3 tiles.

The collection of all encodings and benchmark instances can be found at `http://www.kr.tuwien.ac.at/research/systems/abstraction/`

Results The tests were run on an Intel Core i5-3450 CPU @ 3.10GHz machine using Clingo 5.3, under a 600 secs time limit and 7 GB memory limit. The initial omission, A_{init}, is done by choosing randomly 50%, 75% or 100% of the nodes in the graph problems GC, AA, of the tasks in DS, of the companies in SC, and of the tiles in PZ, as well as by omitting all the atoms related with the chosen objects. We show the overall average of

10 runs for each instance in Figure 5.2.

The first three rows under each category show the bottom-up approach for 50%, 75%, and 100% initial omission, respectively. The columns $|A_{init}|/|\mathcal{A}|$ and $|A_{final}|/|\mathcal{A}|$ show the ratio of the initial omission set A_{init} and the final omission set A_{final} that achieves unsatisfiability after refining A_{init} (with shown number of refinement steps and time). The second part of the columns is on the computation of a $\subseteq$-minimal blocker set $C_{\min}$. For the bottom-up approach, the search starts from A_{final} while for the top-down approach, it starts from $\mathcal{A}$. In each refinement step, the number of determined *badomit* atoms is minimized to be at most $|A|/2$; Figure 5.3 shows results for different upper limits and its full minimization.

Figure 5.2 shows that, as expected, there is a minimal part of the program which contains the reason for unsatisfiability of the program by projecting away the atoms that are not needed (sometimes more than 90% of all atoms). Observe that when 100% of the objects in the problems are omitted, refining the abstraction until an unsatisfiable abstract program is obtained takes the most time. This shows that a naive way of starting with an initial abstraction by omitting every relevant detail is not efficient in reaching an unsatisfiable abstract program. We can observe that larger A_{final} results in having less time spent in computing $\subseteq$-minimal blocker sets, as a smaller number of atoms must be checked. For example, for PZ, starting with 32% of omitted atoms takes longer to compute a $\subseteq$-minimal blocker set compared to starting with 54% of omitted atoms. Additionally, with a bottom-up method it is possible to reach a $\subseteq$-minimal blocker set which is smaller in size than the ones obtained with the top-down method.

The graph coloring benchmarks ($GC^{2,3}$) show that more atoms are kept in the abstraction to catch the non-3-colorability than the non-2-colorability, which matches our intuition. For example, in GC^2 omitting 50% of the nodes (49% of the atoms in A_{init}) already reaches an unsatisfiable program, since no atoms were added back in A_{final}. However, for GC^3 an average of only 9% of the omitted atoms were added back until unsatisfiability is caught.

For the $GC^{2,3}$, SC and PZ benchmarks, we can observe that omitting 50% of the objects ends up easily in reaching some unsatisfiable abstract program, with refinements of the abstractions being relatively small. For example, for GC^2 the size of A_{final} is the same as for A_{init}, and for PZ an average of only 4% of the atoms is added back in A_{final}. However, this behavior is not observed when initially omitting 75% of the objects.

We can also observe that some problems (AA and PZ) have larger $\subseteq$-minimal blocker sets than others. This shows that these problems have a more complex structure than others, in the sense that more atoms are related with each other and have to be considered together for obtaining unsatisfiability.

Badomit minimization In a refinement step, minimizing the number of *badomit* atoms gives the smallest set of atoms to put back. However, the minimization makes the search more difficult, hence it may hit a timeout; e.g., no optimal solution for 45

Figure 5.3: Experimental results with different upper limits on *badomit* #. The three entries in a cell, e.g., 0.21 / 0.24 / 0.23 in cell (AA, $\frac{|A_{final}|}{|\mathcal{A}|}$) of *badomit* # $\leq |\mathcal{A}|/5$, are for 50% / 75% / 100% initial omission.

	badomit # $\leq \lvert\mathcal{A}\rvert/5$					*badomit* # $\leq \lvert\mathcal{A}\rvert/10$				
Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathcal{A}\rvert}$	t (sec)	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathcal{A}\rvert}$	t (sec)
	0.21	4.84	9.49	0.37	8.93	0.23	6.90	13.59	0.36	8.69
AA	0.24	5.93	11.92	0.36	8.38	0.29	8.61	17.84	0.35	7.86
	0.23	5.87	11.93	0.36	8.88	0.33	10.27	22.30	0.34	7.36

	min_badomit #				
Π	$\frac{\lvert A_{final}\rvert}{\lvert\mathcal{A}\rvert}$	Ref #	t (sec)	$\frac{\lvert C_{\min}\rvert}{\lvert\mathcal{A}\rvert}$	t (sec)
	0.24	7.89	15.20	0.36	8.06
AA	0.30	10.65	34.10 (2)	0.34	7.06
	0.44	17.48	62.46 (1)	0.34	5.86

nodes in *GC* was found in 10 mins. Figure 5.3 shows the results of giving different upper bounds on the number of *badomit* atoms and also applying the full minimization in the refinement for the *AA* instances. The numbers in the parentheses show the number of instances that reached a timeout. As more minimization is imposed, we can observe an increase in the size of the final omissions A_{final} and also a decrease in the size of the ⊆-minimal blocker set. For example, for 75% initial omission, we can see that the size of the computed final omission increases from 0.20 (Figure 5.2) to 0.24, 0.29 and finally to 0.30. Also the size of the ⊆-minimal blocker set decreases from 0.37 (Figure 5.2) to 0.36, 0.35 and finally to 0.34. As expected, adding the smallest set of *badomit* atoms back makes it possible to reach a larger omission A_{final} that keeps unsatisfiability (e.g., *min_badomit#* third row (100% A_{init}): A_{final} is 44% instead of 0.01% as in Figure 5.2). On the other hand, such minimization over the number of *badomit* atoms causes to have more refinement steps (Ref #) to reach some unsatisfiable abstract program, which also adds to the overall time.

The ⊆-minimal blocker search algorithm relies on the order of the picked atoms. We considered the heuristics of ordering the atoms according to the number of rules in which each atom shows up in the body, and starting the minimality search by omitting the least occurring atoms. However, this did not provide better results than just picking an atom arbitrarily.

Sliding Tiles (15-puzzle) Studying the resulting abstract programs with ⊆-minimal blockers showed that finding out whether the problem instance is unsolvable within the

Figure 5.4: Unsolvable sliding tiles problem instance

Initial State

0	3	2
8	5	4
1	6	7

Goal State

0	1	2
3	4	5
6	7	8

Initial state

0	3	*
*	5	4
*	*	*

Goal State

0	*	*
3	4	5
*	*	*

given time frame does not require to consider every detail of the problem. Omitting the details about some of the tiles still reaches a program which is unsolvable, and shows the reason for unsolvability through the remaining tiles. Figure 5.4 shows an instance from the benchmark, which is unsolvable in 10 steps. Applying omission abstraction achieves an abstract program that only contains atoms relevant with the tiles 0,3,4,5 and is still unsatisfiable; this matches the intuition behind the notion of pattern databases introduced in [CS98].

Summary The results show that the notion of abstraction is useful in computing the part of the problem which causes unsatisfiability, as all of the benchmarks contain a blocker set that is smaller than the original vocabulary. We observed that different program structures cause the $\subseteq$-minimal blocker sets to be different in size with respect to the original vocabulary size. Computation of these $\subseteq$-minimal blocker sets can sometimes result in smaller sizes with the bottom up approach. However, starting with an 100% initial omission to use the bottom-up approach appears to be unreasonable due to the time difference compared to the top-down approach, even though sometimes it computes $\subseteq$-minimal blocker atoms sets of smaller size. The abstraction & refinement approach can also be useful for finding some (not necessarily minimal) blocker, as most of the time, starting with an initial omission of 50% or 75% results in computing some unsatisfiable abstraction in few refinement steps.

5.2.2 Obtaining Abstract Solutions

The main aim of this evaluation was to see whether the introduced domain abstraction and refinement method automatically finds non-trivial domain abstractions that yield concrete answer sets. We also wanted to observe the effect of having variations over the method w.r.t. the picked abstract answer sets (mentioned in Section 4.5.2) or the refinement decisions (shown in Section 4.4.2).

Experiments

We used DASPAR v0.2 for the experiments, which uses the sophisticated debugging program for concreteness checking.[6] The variations we considered are as follows.

[6]The results from version v0.1 were reported in [SSE19].

- When computing answer sets of the abstract program to be submitted for concreteness checking, we either **(s)** pick a single abstract answer set or **(div)** pick a (diverse) set of answer sets w.r.t. the focus atoms.
- Deciding on a refinement is either done by (**v1**) assigning costs to possible refinements and picking the one with the smallest cost or by **(v2)** using the hints obtained from the debugging atoms while checking. For (v2), it is ensured in the refinement that the domain elements occurring in the picked debugging atom do not occur in the same cluster.

We conducted experimens on two benchmark problems from ASP competitions, viz. graph coloring and disjunctive scheduling. For graph coloring, we randomly generated 20 graphs on 10 nodes with edge probability $0.1, 0.2, \ldots, 0.5$ each; out of the 100 graphs, 74 were 3-colorable. We used two different graph coloring encodings shown in Figure 5.5, to see their effect in the resulting abstractions. In the first encoding GC_{enc1} (Figure 5.5a), guessing a color assignment to each node is done as (5.1)-(5.3) with the common approach of using default negation and the auxiliary atom $\mathit{hasEdgeTo}(X, C)$ shows which colors C the node X has as its neighbors. The second encoding GC_{enc2} (Figure 5.5b) uses a choice rule (5.8) to make a guess of an assignment and then ensures that a node is not assigned more than one color with (5.9). The rules (5.10)-(5.11) are an alternative way of writing the rule $\bot \leftarrow \mathit{chosenColor}(X_1, C), \mathit{chosenColor}(X_2, C), \mathit{edge}(X_1, X_2), X_1 < X_2$. so that when the variables are standardized apart for the sort *node*, fewer relation atoms occur in one rule. Also notice that GC_{enc2} imposes an order relation among the nodes, to reduce duplications of the constraints.

For disjunctive scheduling, for each $t \in \{10, 20, 30\}$, we generated 20 instances with 5 tasks over time $\{1, \ldots, t\}$. We used the encoding [7] from ASP Competition 2011 and precomputed deterministic part (i.e., not involved in unstratified negation resp. guesses) of the program, so that they are lifted to the abstract program without introducing (unnecessary) nondeterminism (see Appendix A). The initial abstraction mapping provided to DASPAR is the single-cluster abstraction, i.e., clustering all nodes into one for graph coloring and all time points into one for disjunctive scheduling.

Results

We report the average results over 10 runs for each variation. For simplicity of the presentation, we discuss the results for each benchmark separately by concentrating on the different observations made throughout the experimental evaluation.

Graph coloring. The evaluation results of the obtained abstractions are presented in Table 5.1. The first two rows show the average number of refinement steps and the average domain size (i.e., the number of clusters) of the resulting abstractions. The best abstraction (i.e., with smallest domain size) found for each instance in the runs is further

[7]`https://www.mat.unical.it/aspcomp2011/files/DisjunctiveScheduling/disjunctive_scheduling.enc.asp`

Figure 5.5: Two encodings of the Graph Coloring problem

$$chosenColor(X,r) \leftarrow not\ chosenColor(X,g), not\ chosenColor(X,y), node(X). \tag{5.1}$$
$$chosenColor(X,g) \leftarrow not\ chosenColor(X,r), not\ chosenColor(X,y), node(X). \tag{5.2}$$
$$chosenColor(X,y) \leftarrow not\ chosenColor(X,g), not\ chosenColor(X,r), node(X). \tag{5.3}$$
$$hasEdgeTo(X,C) \leftarrow edge(X,Y), chosenColor(Y,C). \tag{5.4}$$
$$\perp \leftarrow hasEdgeTo(X,C), chosenColor(X,C). \tag{5.5}$$
$$colored(X) \leftarrow chosenColor(X,C). \tag{5.6}$$
$$\perp \leftarrow node(X), not\ colored(X). \tag{5.7}$$

(a) GC$_{enc1}$

$$\{chosenColor(X,C)\} \leftarrow node(X), color(C). \tag{5.8}$$
$$\perp \leftarrow chosenColor(X,C_1), chosenColor(X,C_2), C_1 \neq C_2. \tag{5.9}$$
$$adj(X,Y) \leftarrow edge(X,Y), X < Y. \tag{5.10}$$
$$\perp \leftarrow adj(X,Y), chosenColor(X,C), chosenColor(Y,C). \tag{5.11}$$
$$colored(X) \leftarrow chosenColor(X,C). \tag{5.12}$$
$$\perp \leftarrow node(X), not\ colored(X). \tag{5.13}$$

(b) GC$_{enc2}$

checked for faithfulness, to observe whether the corresponding abstract program only contains concrete answer sets. The domain size of the abstractions which are faithful is shown in the third row. The percentage distribution of the abstractions which are trivial (thus faithful), non-trivial and faithful, and non-faithful is shown in the last three rows.

The left column shows the results of full concreteness checking with different variations. We can observe that picking a single abstract answer set to decide on a refinement results in finer abstractions (i.e., with larger domain size) when compared to making a decision according to a set of abstract answer sets. The percentage of the trivial abstractions obtained also decreases when (div) is considered, due to making better decisions and the increase in the chance of encountering a concrete abstract answer set. The latter feature also causes to obtain more non-faithful abstractions, since a concrete answer set is encountered within the diverse answer sets (by chance) without refining the abstraction to a finer one with less spurious answer sets. As for using hints (v2) to decide on refinements, we can observe that it does not help in obtaining coarser abstractions than using the minimal cost (v1) method. Also more trivial abstractions are obtained with (v2), since splitting the domain in each refinement step to ensure certain abstract elements are no longer clustered together quickly ends up with the original domain.

Table 5.1: Experimental results for graph coloring

		full				projected			
		(s)		(div)		(s)		(div)	
		(v1)	(v2)	(v1)	(v2)	(v1)	(v2)	(v1)	(v2)
GC_{enc1}	number of steps	7.38	7.83	7.04	7.69	5.24	6.48	4.83	6.14
	abs. domain size	8.38	8.84	8.04	8.69	6.24	7.48	5.83	7.14
	faithful abs. dom. size	6.84	8.04	6.12	7.51	6.02	5.71	5.65	5.82
	trivial abstractions (*id*)	13%	23%	4%	12%	2%	1%	2%	2%
	faithful & non-*id* abs.	30%	32%	29%	27%	56%	61%	50%	47%
	non-faithful abs.	57%	45%	67%	61%	42%	38%	48%	51%
GC_{enc2}	number of steps	7.01	6.40	6.56	6.37	3.53	3.76	3.40	3.52
	abs. domain size	8.01	8.64	7.56	8.29	4.53	6.73	4.40	6.36
	faithful abs. dom. size	8.88	8.62	7.97	8.66	4.86	5.44	4.75	5.72
	trivial abstractions (*id*)	19%	13%	5%	13%	3%	2%	3%	2%
	faithful & non-*id* abs.	22%	24%	25%	22%	54%	59%	54%	48%
	non-faithful abs.	59%	63%	70%	65%	43%	39%	43%	50%

The right column shows the results of considering a projected notion of concreteness that limits the checking to a set of relevant atoms. We picked the nodes 1,2,3 and their assigned colors to be the focus elements, to observe how it affects the obtained abstractions. As expected, a concrete abstract answer set is encountered in much coarser abstractions, since the color assignment of the remaining nodes are not of relevance. In case of projection, the trivial abstraction is reached much less than in the full case; moreover, more non-trivial faithful abstractions are reached, which is beneficial, as the computed abstractions can be used to obtain (concrete) solutions over the focused nodes.

The main difference of the various encodings is on the size of the achieved abstract domains. GC_{enc2} requires less number of refinement steps to achieve an abstraction with a concrete solution, as the need to preserve the order relation among the nodes creates less number of refinement possibilities. The average resulting abstractions are coarser than GC_{enc1}, however the domain sizes of the faithful abstractions are larger. This can be due to the choice rule in GC_{enc2} causing to obtain spurious answer sets which have to be treated by further refining the abstraction.

Disjunctive scheduling. We compared the effects of the variations w.r.t. the resulting abstractions and the calls to the ASP solver to obtain an abstract answer set or to check concreteness with debugging. Table 5.2 shows the collected results. For the refinement search, we considered besides (v1) and (v2), an alteration of (v2), called (v2'), where each abstract element appearing in the obtained debugging atom is mapped to a singleton cluster in the refinement.

As expected, the minimal cost method (v1) causes to have much more calls to the ASP solver, due to the need to compute the cost for each possible refinement. Although in some cases, it achieves coarser abstractions, having such large number of calls is a

Table 5.2: Experimental results for scheduling

time		(s) (v1)	(s) (v2)	(s) (v2')	(div) (v1)	(div) (v2)	(div) (v2')
$t = 10$	number of steps	7.22	4.81	3.56	6.04	4.81	3.54
	abs. domain size	8.22	8.48	8.46	7.04	8.38	8.35
	call abs. prog.	41.35	5.81	4.56	40.72	5.81	4.54
	call debug prog.	40.90	5.36	4.11	56.30	6.87	5.44
$t = 20$	number of steps	14.71	7.65	5.47	12.00	7.53	5.33
	abs. domain size	15.71	14.16	14.12	13.00	14.16	13.81
	call abs. prog.	168.48	8.65	6.47	157.41	8.53	6.33
	call debug prog.	168.28	8.45	6.27	244.45	12.08	8.74
$t = 30$	number of steps	22.82	9.57	7.76	20.57	9.56	7.68
	abs. domain size	23.82	19.02	19.12	21.57	19.07	18.68
	call abs. prog.	391.88	10.57	8.76	366.09	10.56	8.68
	call debug prog.	391.43	10.12	8.31	580.23	14.59	12.24

disadvantage. For example, for $t = 20$ - (div), (v1) achieves the average abstract domain of 13.00 clusters while calling the ASP solver over 400 times, while (v2') achieves an average of 13.81 clusters with only around 15 calls.

For the instances with $t = 20$, refinement through hints (v2) achieves coarser abstractions than (v1) when single abstract answer sets are picked. This is due to the hints guiding the refinement much better than the assigned cost according to a single abstract answer set. When the costs are assigned according to a diverse set of abstract answer sets, then much coarser abstractions are achieved. However, for the instances with $t = 30$, we can observe that the cost approach results in much finer abstractions. This shows that making a local search over the 1-step refinements does not always provide the best outcome. A refinement using hints can obtain much finer abstractions in one step which shows to guide the method much better.

We can also observe that (v2') mostly achieves much coarser abstractions than (v2); immediately singling out the domain elements connected with the spuriousness helps. It also causes to have smallest number of refinement steps compared to other approaches, as it reaches a concrete solution much faster with the refinement decisions.

The results show that with larger domains, the effect of the abstraction can be seen much better, e.g., the average abstract domain size reached for $t = 30$ is 62% (=18.68/30) of the original domain, while for $t = 10$, it shrinks to 70.4%.

Summary. The results show that with domain abstraction it is possible to achieve concrete solutions while abstracting over some of the details of the program. Reaching faithful abstractions is desired, however does not occur often, unless a projected notion of concreteness check is considered, that only distinguishes the details relevant to describing the solution to the problem. Deciding on a refinement by obtaining hints from a set of abstract spurious answer sets, instead of just one, results in better decisions, and thus coarser abstractions. Investigating more sophisticated refinement methods that achieve better abstractions would be an interesting research direction.

5.3 Abstraction in ASP Planning

In this section, we discuss further on the use of domain abstraction in understanding planning problems expressed in ASP, by abstracting over the unnecessary details.

Planning problems in ASP are represented by having a `time` sort to describe the sequence of states and the changes according to the taken actions. These problems usually contain two types of objects, represented with different sort types:

- certain objects on which the actions have direct effect on, e.g., the blocks in the blocksworld which can be moved, and
- the remaining objects which are not affected by the actions, but are involved in the decision making, e.g., the table in the blocksworld on which a block can be moved to.

We discuss how domain abstraction can be used for abstracting over these objects to be able to talk about abstract states and abstract plans.

Describing actions in ASP

First, we emphasize the different ways of expressing planning problems in ASP, through the blocksworld problem. For example, the effects of moving a block on top of another block can be expressed with the following rules

$$\mathit{onB}(B, B_1, T+1) \leftarrow \mathit{moveToBlock}(B, B_1, T). \tag{5.14}$$
$$\neg \mathit{onB}(B, B_2, T) \leftarrow \mathit{onB}(B, B_1, T), B_1 \neq B_2. \tag{5.15}$$

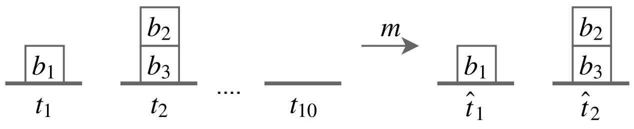
where 5.14 shows the direct effect and 5.15 shows the indirect effect. Alternatively, the following rules can also be used to express all the effects as direct effects.

$$\mathit{onB}(B, B_1, T+1) \leftarrow \mathit{moveToBlock}(B, B_1, T). \tag{5.16}$$
$$\neg \mathit{onB}(B, B_2, T) \leftarrow \mathit{moveToBlock}(B, B_1, T), B_1 \neq B_2. \tag{5.17}$$

The preconditions of an action can either be described through constraints, or as a condition for an action to become applicable. For example, the condition that a block cannot sit on a block which is smaller can be expressed as a constraint

$$\bot \leftarrow \mathit{onB}(B, B_1, T), B_1 \leq B.$$

Figure 5.6: Initial state of a blocksworld with multiple tables (concrete $\overset{m}{\to}$ abstract).



which forbids that a block can be located on smaller block. Alternatively, the respective action can be forbidden to apply in case the condition is not satisfied using the following rules.

$$\begin{aligned}\bot &\leftarrow \mathit{moveToBlock}(B, B_1, T), \mathit{not}\ \mathit{precondmtb}(B, B_1, T).\\ \mathit{precondmtb}(B, B_1, T) &\leftarrow B < B_1, \mathit{block}(B), \mathit{block}(B_1).\end{aligned}$$

Note that the alternative version is much closer to the PDDL-style encoding. Remember that the law of inertia is then descibed by the rule

$$\mathit{onB}(B, B_1, T{+}1) \leftarrow \mathit{onB}(B, B_1, T), \mathit{not}\ \neg \mathit{onB}(B, B_1, T).$$

5.3.1 Abstracting over Irrelevant Details

We first show the abstraction possibility over the details of objects that are indirectly affected by the actions. For demonstration, we consider two planning problems, that are well-known problems extended with objects out of main focus for the plan computation.

- *Multi-table blocksworld*: The Blocksworld problem extended with multiple tables, where the blocks can be moved on to any of the tables. A plan needs to be found that piles up the blocks on a given specific table.
- *Package delivery with checkpoints*: The Package Delivery problem of carrying packages from an initial location to a goal location, while also passing through some checkpoint reachable from the initial location.

Multi-table blocksworld. Figure 5.6 illustrates an example instance, where the blocks need to piled up on table t_1. From the given initial state, reaching the goal state does not rely on which table the blocks are moved to in between. However, when computing a plan in the original program, the planner has to consider all possible movements.

Figure 5.7 shows a (natural) encoding of the problem which contains the actions $\mathit{moveToT}(B, \mathit{Ta}, T)$ and $\mathit{moveToB}(B, B', T)$ that represent moving block B onto table Ta and onto block B', resp., at time T. Consider the initial state shown in Figure 5.6:

$$\mathit{onT}(b_1, t_1), \mathit{onB}(b_2, b_3), \mathit{onT}(b_3, t_2), \mathit{chosenTable}(t_1).$$

Figure 5.7: Encoding for Multi-table Blocksworld

% action choice

$\{moveToB(B, B_1, T) : bl(B), bl(B1); moveToT(B, L, T) : bl(B), tbl(L)\} \leq 1 \leftarrow T < t_{max}.$

% no gaps between moves

$done(T) \leftarrow moveToB(B, B1, T).$

$done(T) \leftarrow moveToT(B, L, T).$

$\bot \leftarrow done(T{+}1), not\ done(T).$

% preconditions

$\bot \leftarrow moveToB(B, B_2, T), onB(B_1, B, T).$

$\bot \leftarrow moveToB(B_1, B_2, T), onB(B_1, B_2, T).$

$\bot \leftarrow moveToT(B, L, T), onB(B_1, B, T).$

$\bot \leftarrow moveToT(B, L, T), onT(B, L, T).$

% effects

$onB(B, B_1, T{+}1) \leftarrow moveToB(B, B_1, T), T{<}t_{max}.$

$onT(B, L, T{+}1) \leftarrow moveToT(B, L, T), T{<}t_{max}.$

$\neg onB(B, B_2, T) \leftarrow onB(B, B_1, T), B_1{\neq}B_2.$

$\neg onT(B, L, T) \leftarrow onB(B, B_1, T).$

$\neg onB(B, B_1, T) \leftarrow onT(B, L, T).$

$\neg onT(B, L_1, T) \leftarrow onT(B, L_2, T), L_1{\neq}L_2.$

% inertia

$onB(B, B_1, T{+}1) \leftarrow onB(B, B_1, T),$
$\quad not\ \neg onB(B, B_1, T{+}1), T{<}t_{max}.$

$onT(B, L, T{+}1) \leftarrow onT(B, L, T),$
$\quad not\ \neg onT(B, L, T{+}1), T{<}t_{max}.$

% state constraints

$\bot \leftarrow onB(B_1, B, T),$
$\quad onB(B_2, B, T), B_1{\neq}B_2.$

$\bot \leftarrow onB(B, B_1, T),$
$\quad onB(B, B_2, T), B_1{\neq}B_2.$

$\bot \leftarrow onB(B, B_1, T), onT(B, L, T).$

$\bot \leftarrow onB(B, B_1, T), B_1 \leq B.$

% goal constraints

$notblockgoal(T) \leftarrow onT(B, L, T),$
$\quad onT(B_1, L_1, T), B{\neq}B_1.$

$\bot \leftarrow notblockgoal(T), maxTime(T).$

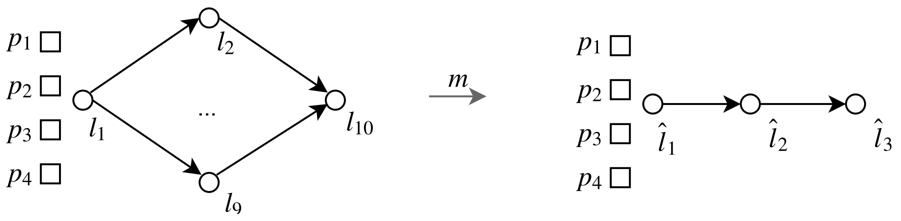
$\bot \leftarrow not\ notblockgoal(T), onT(B, L, T),$
$\quad not\ chosenTable(L).$

After ensuring that all of the variables are guarded by domain predicates and that the variables related with the *table* sort are standardized apart, we run DASPAR with the initial mapping $\{\{t_1, \ldots, t_n\}/\hat{t}\}$. The abstraction reached is the one shown in Figure 5.6, which distinguishes the chosen table $\hat{t}_1$ and clusters the remaining tables into $\hat{t}_2$. This makes it possible to compute a concrete abstract answer set

$$\{moveToT(b_2, \hat{t}_2, 0), moveToT(b_3, \hat{t}_1, 1), moveToB(b_2, b_3, 2), moveToB(b_1, b_2, 3)\}.$$

that describes a plan without going into detail on which table the blocks are moved. The abstraction shows that, for solving the problem, it is essential to distinguish the picked table from all others and that the number of tables is irrelevant. Furthermore, this abstraction is a faithful abstraction when the answer sets are projected to the actions *moveToB*, *moveToT*.

Figure 5.8: Initial state of a package delivery with checkpoints (concrete $\xrightarrow{m}$ abstract).



Package delivery with checkpoints Figure 5.8 illustrates an example instance, where the packages in location l_1 need to be carried to location l_{10}. As these locations are not connected, the truck has to pass through a middle point; through which point the truck passes does not make a difference in reaching the goal state.

For this problem, we used the Nomystery encoding from ASPCOMP2015 and altered it to have no fuel computation. Furthermore, for a $drive(T, L_1, L_2, S)$ action to be possible we added an additional condition that the locations L_1 and L_2 should be connected by an edge, $edge(L_1, L_2)$. Consider the initial state shown in Figure 5.8: $atT(t, l_1, 0)$. $atP(p_1, l_1, 0)$. $atP(p_2, l_1, 0)$. $atP(p_3, l_1, 0)$. $atP(p_4, l_1, 0)$. $goal(p_1, l_{10})$. $goal(p_2, l_{10})$. $goal(p_3, l_{10})$. $goal(p_4, l_{10})$. with the demonstrated *edge* facts. Running DASPAR with the mapping $\{\{l_1, \ldots, l_{10}\}/\hat{l}\}$ over the sort *location* results in the abstraction mapping $\{\{l_1\}/\hat{l}_1, \{l_2, \ldots, l_9\}/\hat{l}_2, l_{10}/\hat{l}_3\}$ (shown in Figure 5.8). With this abstraction, the below concrete abstract answer set is computed:

$$\begin{aligned}
&\{load(d, t, \hat{l}_1, 1), load(c, t, \hat{l}_1, 2), load(a, t, \hat{l}_1, 3), load(b, t, \hat{l}_1, 4),\\
&\ drive(t, \hat{l}_2, \hat{l}_1, 5), drive(t, \hat{l}_1, \hat{l}_3, 6),\\
&\ unload(p_3, t, \hat{l}_3, 7), unload(p_1, t, \hat{l}_3, 8), unload(p_4, t, \hat{l}_3, 9), unload(p_2, t, \hat{l}_3, 10)\}
\end{aligned}$$

which describes a plan that loads all the packages, moves to the middle cluster location, moves to the goal location, and unloads the packages. Furthermore, this abstraction is a faithful abstraction when the abstract answer sets are projected to the actions $load, unload, drive$.

The abstraction we demonstrated was over details that are not of importance for the plan computation. The faithful abstraction gives an understanding of the problem by realizing its focus points. If however there are further constraints over these details which need to be distinguished to compute a plan, then faithfulness might not be achieved.

5.3.2 Computing Abstract Plans

Abstracting over the objects that are directly affected by the actions would make it possible to talk about abstract plans. However, in ASP-style encodings, abstracting only over the object sort causes the abstract program to compute plans with the original *time* sort over the abstracted object sort. For example, say in the Package Delivery

Figure 5.9: Encoding for Package Delivery

% action choice
$\{unload(P,T,L,S) : package(P), truck(T), loc(L);$
$load(P,T,L,S) : package(P), truck(T), loc(L);$
$drive(T,L_1,L_2,S) : edge(L_1,L_2), loc(L_1), loc(L_2), truck(T)\} \leq 1 \leftarrow step(S), S > 0.$

% no gaps between moves
$done(S) \leftarrow unload(P,T,L,S).$
$done(S) \leftarrow load(P,T,L,S).$
$done(S) \leftarrow drive(T,L_1,L_2,S).$
$\bot \leftarrow done(S{+}1), not\ done(S).$
% effects
$atP(P,L,S) \leftarrow unload(P,T,L,S).$
$\neg in(P,T,S) \leftarrow unload(P,T,L,S).$
$\neg atP(P,L,S) \leftarrow load(P,T,L,S).$
$in(P,T,S) \leftarrow load(P,T,L,S).$
$\neg atT(T,L_1,S) \leftarrow drive(T,L_1,L_2,S).$
$atT(T,L_2,S) \leftarrow drive(T,L_1,L_2,S).$

% precondition check
$\bot \leftarrow unload(P,T,L,S), not\ precondu(P,T,L,S).$
$precondu(P,T,L,S) \leftarrow atT(T,L,S{-}1),$
$in(P,T,S{-}1).$
$\bot \leftarrow load(P,T,L,S), not\ precondl(P,T,L,S).$
$precondl(P,T,L,S) \leftarrow atT(T,L,S{-}1),$
$atP(P,L,S{-}1).$
$\bot \leftarrow drive(T,L_1,L_2,S), not\ precondd(T,L_1,L2,S).$
$precondd(T,L_1,L_2,S) \leftarrow atT(T,L_1,S{-}1).$
% inertia
$atT(T,L,S) \leftarrow atT(T,L,S{-}1), not\ \neg atT(T,L,S).$
$atP(P,L,S) \leftarrow atP(P,L,S{-}1), not\ \neg atP(P,L,S).$
$in(P,T,S) \leftarrow in(P,T,S{-}1), not\ \neg in(P,T,S).$
% goal check
$\bot \leftarrow goal(P,L), not\ atP(P,L,S), maxstep(S).$

problem (without checkpoints and only two locations l_1, l_2) we cluster the packages into one abstract package, $\hat{p}_0$. Then, the abstract program will contain the abstract actions $load(\hat{p}_0,t,l,s), unload(\hat{p}_0,t,l,s)$ which then causes to find a plan

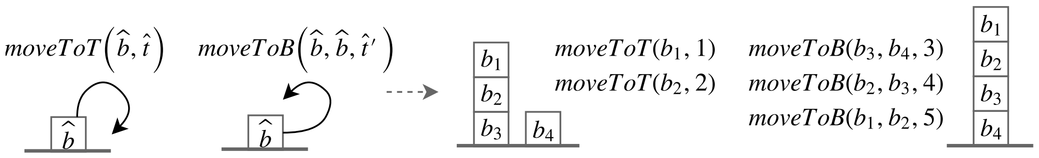
$$load(\hat{p}_0,t,l_1,1), drive(t,l_1,l_2,2), unload(\hat{p}_0,t,l_2,3).$$

However, this plan is immediately spurious since it is not possible to find an original action to match $load(\hat{p}_0,t,l_1,1)$ that loads all the packages in one step. Thus, doing abstraction only on one sort results in the occurrence of many spurious answer sets. In order to avoid this, an additional abstraction over the *time* sort becomes necessary.

By abstracting over different sorts in the program, it becomes possible to talk about abstract instances of actions that abstract from the concrete order of applications. Given that the sorts are independent, i.e., blocks, time or packages, time, multiple calls of DASPAR to abstract over each sort one-by-one achieves the desired abstract program.

For the Package Delivery problem, consider two abstraction mappings $m_{package} = \{\{p_1,p_2,p_3,\ p_4\}/\hat{p}\}$ and $m_{time} = \{\{1,2,3,4\}/\hat{t}_1, \{5\}/\hat{t}_2, \{6,7,8,9\}/\hat{t}_3\}$. The constructed

Figure 5.10: Abstract and concrete plan of Example 5.2



abstract program contains the abstract plan

$$load(\hat{p},t,l_0,\hat{t}_1), drive(t,l_1,l_2,\hat{t}_2), unload(\hat{p},t,l_2,\hat{t}_3)$$

which abstracts over the order of the package loading/unloading by talking about the abstract actions over the time clusters.

Unfortunately, achieving an abstraction over multiple sorts, especially if one sort is over the *time* domain, is not trivial. The abstraction over time with the time clusters steers the plan computation and the determination of the action orders. For example, a time mapping as $\{\{1\}/\hat{t}_1, \{2,3,\}/\hat{t}_2, \{4,5,6,7,8,9\}/\hat{t}_3\}$ will be unable to obtain an abstract plan as shown before.

A policy as discussed in Chapter 3 can also be used to help with the decision making for the next sequence of actions, which we demonstrate in the next example.

Example 5.2 (Blocksworld with a Policy)**.** Consider the blocksworld problem with a single table in Figure 5.10. The encoding in Figure 5.7 is modified by removing the table argument from *onT* and is standardized apart according to the block sort and the time sort. Suppose further rules realize a policy that first puts all blocks on the table and piles them up in a second phase.

$existsOnBlock(T) \leftarrow onB(B,B_1,T).$
$allOnTable(T) \leftarrow not\ existsOnBlock(T), time(T).$
$atPhase2(T_1) \leftarrow allOnTable(T), T < T_1.$
$1\{moveToT(B,T) : onB(B,B_1,T)\} \leftarrow T < t_{max}, not\ atPhase2(T), not\ allOnTable(T).$
$1\{moveToB(B,B_1,T) : onT(B,T), block(B_1)\} \leftarrow T < t_{max}, allOnTable(T).$
$1\{moveToB(B,B_1,T) : onT(B,T), onB(B_1,B_2,T)\} \leftarrow T < t_{max}, atPhase2(T).$

Given the initial state $\{onT(b_4,1), onT(b_3,1), onB(b_2,b_3,1), onB(b_1,b_2,1)\}$ and time domain $\{1,\dots,6\}$, we abstract using the block mapping $\{\{b_1,\dots,b_4\}/\hat{b}\}$ and the time mapping $\{\{1,2\}/\hat{t}, \{3,\dots,6\}/\hat{t}'\}$. The abstract program has 8 answer sets, including

$$\{moveToT(\hat{b},\hat{t}), onT(\hat{b},\hat{t}), onB(\hat{b},\hat{b},\hat{t}), onT(\hat{b},\hat{t}'), onB(\hat{b},\hat{b},\hat{t}'), moveToB(\hat{b},\hat{b},\hat{t}')\},$$

which contains abstract actions: $moveToT(\hat{b},\hat{t})$ and $moveToB(\hat{b},\hat{t}')$ (Figure5.10).

5.4 Discussion

In this chapter, we described the implemented systems for the abstraction and refinement method. We conducted experiments with different problems to observe the use of our method in achieving an abstraction that permits to find a concrete solution without the need to refine back to the original domain. For omission abstraction, we investigated the computation of blocker sets for unsatisfiable problems, and for domain abstraction, we investigated different approaches of refinement decision making and their effects in the resulting abstractions.

The experiments show that by using domain abstraction, it is possible to achieve abstract solutions to the problem which are concrete, without having to refine the abstraction back to the original domain. The rules that occur in the abstract program w.r.t. the mapping and the abstracted facts of the program show the possibility to achieve a concrete solution. For unsolvable problems, detecting the unsolvability in the abstraction immediately results in faithfulness. The abstract program then contains the reason for unsolvability of the original program. An ASP debugging technique can then be used in finding the particular reason for not computing an answer set at this abstract level. For satisfiable problems, if a concrete abstract answer set is computed in the abstract program, then a justification technique in ASP can be used to understand why that answer set is computed at the abstract level. If a faithful abstraction is achieved, such a justification technique can help in distinguishing the unnecessary details of the domain that are irrelevant while finding concrete solutions to the problem.

Our focus here is on evaluating the usefulness of the abstraction approach for ASP, and not as a scalability technique. Further implementation improvements and optimization techniques in future work will make it possible to argue about efficiency. On the other hand, as the state-of-the-art ASP solvers are quite efficient in solving problems, it can not be expected to out perform these solvers in all problems with the abstraction refinement methodology. The methodology may require many checking and refinement steps until a concrete abstract solution is found. Nevertheless, there are problems that the current ASP solvers also fail to solve efficiently, such as optimization problems or problems that create huge search spaces. For such problems, abstraction could be useful, but achieving a good abstraction that could help with solving is not trivial. In the method presented here, some initial mapping is provided to the system, which then automatically finds an abstraction that provides a concrete solution. One interesting aspect would be to find a good initial abstraction to start with, which could help in reaching a concrete solution is few steps. For this, one needs to investigate properties of the programs needed to obtain such abstractions.

We perform the abstract answer set checking externally, using a debugging program. An improvement of this process could be to embed this check into the ASP solver while also obtaining hints during the check. However, doing this is not trivial, especially for the non-ground case, since the checking involves a guess of an answer set over the original program to could match with the abstract answer set w.r.t. the abstraction mapping.

CHAPTER 6

Abstracting Problems over Grid-Cells

In this chapter, we focus on problems involving grid-cell structures to observe the use of a two-dimensional abstraction over the grids in focusing on the essential parts of the problem and achieving abstract solutions.

Outline We begin by describing the problem types that we focus on in Section 6.1. In Section 6.2, we define the 2-dimensional abstraction on grid-cells based on quad-trees. Section 6.3 describes mDASPAR and considered implementation issues. We evaluate our approach in Section 6.4 on unsatisfiable problems and conduct a user study for comparing the obtained abstract explanations. In Section 6.5 we discuss the application of abstraction to the problem of policy refutation. We conclude with a discussion in Section 6.6.

6.1 Problems in Focus

As we have shown in Chapter 5, abstraction can become handy when finding the cause for unsatisfiability by abstracting over the irrelevant details and also for finding abstract solutions to the problems that help in having an understanding of the problem. Grid-cell environments are a particular type of environment which describes a structure. For problems over grid-cells, it is often the case that certain parts of the environment are crucial to finding a solution to the problem. For example, suppose one wants to check whether all cells are reachable from a given starting point in a grid with obstacles. In case there are unreachable cells, this is due to the obstacles surrounding them. It would be enough for a person to look at the area with the obstacles to realize that there are unreachable cells. As another example, consider the problem of checking whether the agent always manages to find a missing person with a given policy. If the policy does not work, then a counterexample trajectory over some part of the environment will show this. By looking at the area which shows the trajectory, one can conclude that the policy does not work. Depending on the problem, the focus points may have different nature. For the reachability example, the focused area in the environment remains can remain local, while for the person search example the path of a trajectory needs to be distinguished.

Inspired from the capability of humans in focusing on certain areas in a grid-cell environment, by abstracting away the rest of the details, when solving a problem or realizing that there is no solution, we investigate the application of domain abstraction to such problems. We aim to observe the abstractions the machine can automatically obtain given the capability of doing abstraction over the grid-cells. For this, we consider problems of two types:

(i) Problems that are unsolvable for a given instance, due to some facts causing to violate some constraints.
(ii) Problems with constraints/restrictions that steer the solution finding.

For problems of type (i), we investigate whether a domain abstraction over the grid-cell can focus on the parts of the problem instance that are essential to reach unsatisfiability. For problems of type (ii), we investigate whether an abstraction can put the focus on where the constraints/restrictions cause to achieve a solution.

For demonstration we have the following running example.

Example 6.1 (Reachability)**.** Below encoding computes the obstacle-free cells (i.e., *points*) that are reachable from the starting point.

$$point(X,Y) \leftarrow not\ obsAt(X,Y), row(X), column(Y). \tag{6.1}$$
$$reachable(X,Y) \leftarrow start(X,Y). \tag{6.2}$$
$$reachable(X_1,Y) \leftarrow reachable(X,Y), point(X_1,Y), neighbor(X,Y,X_1,Y). \tag{6.3}$$
$$neighbor(X,Y,X_1,Y) \leftarrow |X-X_1|=1, column(Y). \tag{6.4}$$
$$neighbor(X,Y,X,Y_1) \leftarrow |Y-Y_1|=1, row(X). \tag{6.5}$$

Consider two versions of the reachability problem.

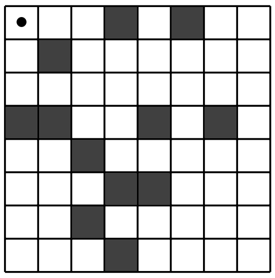
(i) Having an additional constraint to check if all points are reachable.

$$\perp \leftarrow point(X,Y), not\ reachable(X,Y), row(X), column(Y). \tag{6.6}$$

(ii) Redefining reachability (6.3-6.5) by prioritizing the east neighbor over the rest, and in case the east neighbor has an obstacle, choosing the south neighbor.

$$neighborE(X,Y,X_1,Y) \leftarrow X_1-X=1, column(Y). \tag{6.7}$$
$$neighborS(X,Y,X,Y_1) \leftarrow Y_1-Y=1, row(X). \tag{6.8}$$
$$\begin{aligned} reachableE(X,Y,X_1,Y) \leftarrow\ & reachable(X,Y), point(X_1,Y), \\ & neighborE(X,Y,X_1,Y). \end{aligned} \tag{6.9}$$

Figure 6.1: Original domain



$$hasNeighborE(X,Y) \leftarrow reachableE(X,Y,X_1,Y_1). \tag{6.10}$$
$$reachable(X_1,Y_1) \leftarrow reachableE(X,Y,X_1,Y_1). \tag{6.11}$$
$$reachable(X,Y_1) \leftarrow reachable(X,Y), point(X,Y_1), neighborS(X,Y,X,Y_1), not\ hasNeighborE(X,Y). \tag{6.12}$$

Figure 6.1 shows an instance of a grid-cell domain with obstacles. For this instance, Problem-(i) is unsatisfiable, due to the unreachable cells in the lower left area. As for Problem-(ii), the reachable cells are determined in the order $\rightarrow^2\downarrow\rightarrow^5\downarrow^6$.

6.2 Quad-tree Abstraction

Multi-dimensional abstraction allows us to express abstractions where one domain (e.g., an X coordinate) is abstracted depending on its context, i.e., depending on a second domain it occurs with (e.g., a Y coordinate). For a systematic refinement of abstractions on grid-cell environments, we consider a generic quad-tree representation (Figure 6.2), which is a concept used, e.g., in path planning [KD86].

Initially, an environment may be abstracted to four regions of $n/2 \times n/2$ grid-cells each. This amounts to a tree with four leaf nodes that correspond to the main regions, with level $log_2(n)$. Each region then contains 4 leaves of smaller regions. The leaves of the quad-tree are then the original cells of the grid-cell. A refinement of a region amounts to dividing the region into four subregions, i.e., expanding the representing leaf with its four leaves. Given the original X and Y coordinates $\{a_1,\dots,a_n\}$ and $\{b_1,\dots,b_n\}$ respectively, we represent the coordinates of an abstract region with level $log_2(k+1)$, for $0 \leq k < n$, defined over the cells within the coordinates $a_i,\dots,a_{i+k}$ and $b_j,\dots,b_{j+k}$, by the shorthand notation $(a_{i\dots i+k}, b_{j\dots j+k})$.

Example 6.2. Figure 6.3 shows different abstractions over the grid-cells. The facts that represent the original instance also get lifted to the abstract domain. Figure 6.3a is the initial abstraction of dividing the grid-cell into 4 regions. Figure 6.3b shows an abstraction that distinguishes the area which shows the obtacles that cause to have unreachable cells in the lower-left corner (Problem (i)). The abstraction shown in Figure 6.3c singles out

Figure 6.2: Quad-tree representation for regions

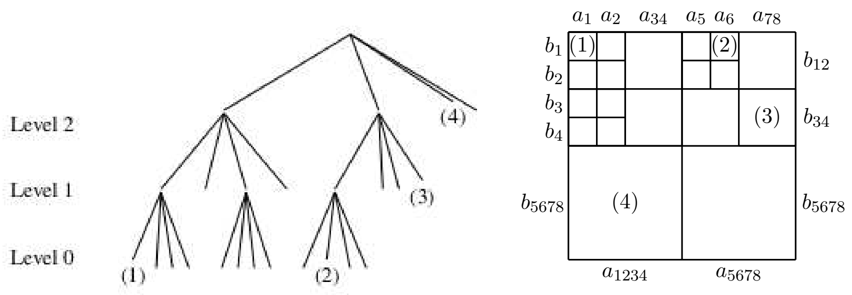
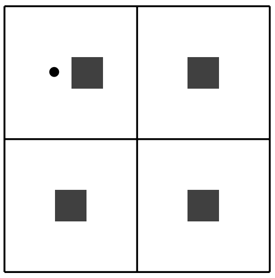


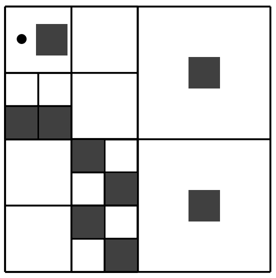
Figure 6.3: Abstractions over the grid-cell domain with obstacles in Figure 6.1

(a) Initial abstraction



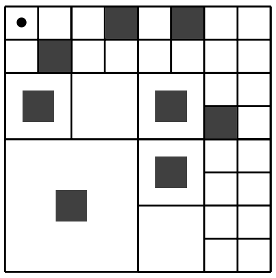
start(a_{1234}, b_{1234}).
obsAt(a_{1234}, b_{1234}).
obsAt(a_{1234}, b_{5678}).
obsAt(a_{5678}, b_{1234}).
obsAt(a_{5678}, b_{5678}).

(b) Distinguishing the obstacles that cause unreachability



start(a_{12}, b_{12}).
obsAt(a_1, b_4).
obsAt(a_2, b_4).
obsAt(a_3, b_5).
obsAt(a_4, b_6).
obsAt(a_3, b_7).
obsAt(a_4, b_8).
obsAt(a_{12}, b_{12}).
obsAt(a_{5678}, a_{1234}).
obsAt(a_{5678}, a_{5678}).

(c) Distinguishing the cells reachable w.r.t. the restrictions



start(a_1, b_1).
obsAt(a_4, b_1).
obsAt(a_6, b_1).
obsAt(a_2, b_2).
obsAt(a_7, b_4).
obsAt(a_{12}, a_{34}).
obsAt(a_{56}, a_{34}).
obsAt(a_{56}, a_{56}).
obsAt(a_{1234}, a_{5678}).

the area that contains the cells that are reachable according to the restrictions (Problem (ii)).

Starting with an initial abstraction of level $log_2(n)$, using quad-tree split operations as abstraction refinement operations, we can automatically search for suitable quad-tree-structured abstractions in grids (see Section 6.3). Importantly, multi-dimensional abstraction refinement is *structure aware*: refining one of the squares of a quad-tree (e.g., area (3) in Figure 6.2) maintains the structure of the abstraction of all other squares.

6.3 mDASPAR

In this section, we describe implementation details of mDASPAR, which is an extension of DASPAR (described in Chapter 5) that applies a multi-dimensional domain abstraction. Currently, mDASPAR handles 2-dimensional abstractions with a quad-tree style refinement process, and it can be applied to problems described over grid-cells. The input format described for DASPAR also applies to mDASPAR. We discuss several challenges of multi-dimensional abstractions that are tackled in the system.

Abstract objects A multi-dimensional abstraction creates abstract objects and not all combinations of the abstracted sorts, e.g., *row* and *column*, correspond to a valid object. To avoid non-valid combinations, the constructed abstract program should comply to only using the abstract objects in the rules. For this, mDASPAR applies a post-processing over the constructed abstract program by going over each rule and replacing the occurrence of the abstracted sorts with a new object name.

We remark that in order to have the objects "grouped" automatically and, most importantly, correctly, the system needs some guidance. For a given encoding, humans are capable of detecting the cells implicitly, whereas the machine can not do this. The user has to provide some guidelines for the machine to recognize the objects, by adjusting the encoding so that the grids are explicitly shown. For this, we impose some syntactic restrictions on the input program, on which the post-processing technique relies.

Given two sorts s_1, s_2 over which a 2-dimensional abstraction is desired, in order to achieve a correct object naming in the rules, i.e., $row(X_1), column(Y_1)$ changed to $cell(X_1, Y_1)$, the input program should adhere to following restrictions:

(1) The rules should have atoms that contain pairs X, Y of variables where $X \in s_1, Y \in s_2$, and

(2) the domain predicates for sorts s_1, s_2 should be written in the order of the pairs.

If these restrictions are satisfied, then mDASPAR is able to correctly convert the sort names to the abstract object name *cell*.

Example 6.3 (ctd)**.** The rule (6.3) will be standardized apart into

$$\begin{aligned} reachable(X_1, Y_1) \leftarrow\ & reachable(X, Y), point(X_1, Y_1), neighbor(X_2, Y_2, X_3, Y_3), \\ & X = X_2, X_1 = X_3, Y = Y_2, Y_1 = Y_3. \end{aligned}$$

Then the multiple relations related with a sort needs to be converted into an auxiliary relation atom, similarly as described in the input format for DASPAR (Section 5.1.2):

$$\begin{aligned} reachable(X_1, Y_1) \leftarrow\ & reachable(X, Y), point(X_1, Y_1), neighbor(X_2, Y_2, X_3, Y_3), \\ & equEqu4(X, X_2, X_1, X_3), equEqu4(Y, Y_2, Y_1, Y_3). \end{aligned}$$

Figure 6.4: Input program with the rules 6.1-6.6

```
point(X,Y) :-  not obsAt(X,Y), row(X), column(Y).
reachable(X,Y) :-  start(X,Y), row(X), column(Y).
reachable(X1,Y1) :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                    equEqu4(X,X2,X1,X3), equEqu4(Y,Y2,Y1,Y3),
                    row(X), column(Y), row(X1), column(Y1),
                    row(X2), column(Y2), row(X3), column(Y3).
:- point(X,Y), not reachable(X1,Y1), X=X1, Y=Y1,
   row(X), column(Y), row(X1), column(Y1).
neighbor(X,Y,X1,Y1) :-  dist1(X,X1), Y=Y1,
                        row(X), column(Y), row(X1), column(Y1).
neighbor(X,Y,X1,Y1) :-  X=X1, dist1(Y,Y1),
                        row(X), column(Y), row(X1), column(Y1).
```

The domain predicates for the rule above also need to be written in a format where the pairs X, Y, X_1, Y_1, X_2, Y_2, and X_3, Y_3 appear together. Figure 6.4 shows the resulting rule in the input program.

Relation type computation When constructing an abstract rule, mDASPAR gathers the relations in the original rule related with the abstracted sorts and creates an abstract relation atom with arguments from the original relations, following the description in Section 4.6. Auxiliary programs are used to compute the relation type facts.

mDASPAR is invoked as follows

```
python mdaspar.py prog mapping pred size ref_type <focus_atoms>
```

Different from DASPAR, there is also the parameter `size` which is the number $n = 2^k$, for $k \geq 2$, representing a grid of size $n \times n$.

The next example shows the input format of mDASPAR and the created abstract program.

Example 6.4 (ctd)**.** The rules (6.4)-(6.5) contain a relation that is not supported by the current approach. As shown in Section 4.3.4, a rewriting of the relation needs to be done, by introducing an auxiliary atom, say $dist1(X, X_1)$, and adding facts for the domain elements for which the relation $X + 1 = X_1$ holds true.

Figure 6.4 shows the input program of mDASPAR with the rules (6.1)-(6.6) where the variables are standardized apart. The constructed abstract non-ground program for abstracting over the sorts $row, column$ becomes as shown in Figure 6.5, where the occurrence of the sorts are renamed with a new object *cell*. Also the rules of the original program get numbered and in the abstract program the relation atoms are named w.r.t. the rule number. For example, for the rule (6.6), the standardization creates the relations $X = X_1$ and $Y = Y_1$, and in the abstraction the joint relation type atom becomes $\tau_{\text{I}}^{=,=}(X, Y, X_1, Y_1)$ for type I. Type III relation atom, i.e., $\tau_{\text{III}}^{=,=}(X, Y, X_1, Y_1)$, is not important as the abstracted constraint containing this atom in its body gets omitted in

Figure 6.5: Non-ground abstract program constructed by mDASPAR

```
point(X,Y) :- cell(X,Y), not obsAt(X,Y).
{point(X,Y)} :- cell(X,Y), obsAt(X,Y), isCluster(X).
{point(X,Y)} :- cell(X,Y), obsAt(X,Y), isCluster(Y).
reachable(X,Y) :- start(X,Y), cell(X,Y).
reachable(X1,Y1) :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                    cell(X2,Y2), cell(X3,Y3), cell(X,Y), cell(X1,Y1),
                    relr3(X,Y,X2,Y2,X1,Y1,X3,Y3,i).
{reachable(X1,Y1)} :- reachable(X,Y), point(X1,Y1), neighbor(X2,Y2,X3,Y3),
                      cell(X2,Y2), cell(X3,Y3), cell(X,Y), cell(X1,Y1),
                      relr3(X,Y,X2,Y2,X1,Y1,X3,Y3,iii).
:- point(X,Y),cell(X,Y),cell(X1,Y1),not reachable(X1,Y1),relr4(X,Y,X1,Y1,i).
neighbor(X,Y,X1,Y1) :- dist1(X,X1), cell(X,Y), cell(X1,Y1), relr5(Y,Y1,i).
{neighbor(X,Y,X1,Y1)} :- dist1(X,X1), cell(X,Y), cell(X1,Y1), relr5(Y,Y1,iii).
neighbor(X,Y,X1,Y1) :- dist1(Y,Y1), cell(X,Y), cell(X1,Y1), relr6(X,X1,i).
{neighbor(X,Y,X1,Y1)} :- dist1(Y,Y1), cell(X,Y), cell(X1,Y1), relr6(X,X1,iii).
```

the abstraction. For the input program (Figure 6.4) the rule (6.6) gets the rule name `r4` and thus the abstracted rule contains the relation atom `relr4(X,Y,X1,Y1,i)`.

Observe that rule (6.1), which has a deterministic computation of *point* atoms, gets abstracted to choice rules and causes to have further guesses. To avoid such unnecessary guessing, (6.1) can be removed from the input program and instead the *point* atoms can be precomputed to be added as facts.

Furthermore, by standardizing apart the variable of the negative literal in the constraint (6.6), its aim of making sure that all points are reachable gets relaxed to hold only when the abstraction on the domain elements is fine enough to satisfy the relation atom `relr4(X,Y,X1,Y1,i)`. Having the rule (6.6) without standardization would ensure that the constraint holds in coarser abstractions. For application on grid-cell problems, we applied standardization on the variables of the negative literals as well to obtain more fine-grained abstractions that distinguish the original cells to reach a concrete solution. This makes it easier to visualize the resulting abstractions and understand the found solutions.

6.3.1 Debugging and Refinement

The method for debugging and refinement is the same as with DASPAR, with further considerations and slight alterations.

Two-phase debugging The multi-dimensionality of the domain mapping causes to have many possibilities of a cause for spuriousness of an abstract answer set. Debugging the non-ground spuriousness by searching for the answer set with minimum number of *ab* atoms can become more difficult. To handle this, we implemented a two-phase debugging approach. In the first phase the debugging program Π_{debug} is created by modifying the debugging atoms $ab\_deact, ab\_deactCons$ of Definition 4.37 to only have the rule name as arguments. We denote this program by Π_{debug_0}. This then results in an easier computation of an answer set with minimal ab atoms. Once such an answer set I is computed, in the second phase, a new program Π_{debug} is created according to the original definition, but the ab atoms are only created for the rule names or atoms occurring in the ab atoms of I. This way, the search for an optimal answer set focuses on the trouble making rules/atoms.

Steer debugging towards constraints In problems we focus on, the constraints in the program cause to have unsatisfiability or to obtain a particular solution for a given instance. In order to help with reaching abstractions where the relevant constraints are distinguished, we give in the search for an optimal answer set during debugging less cost to obtaining answer sets with $ab\_deactCons$ atoms.

Getting hints Since the refinement of a region means to split it into four subregions, we only need to get the hint of which region to refine. This is different from the hints obtained for DASPAR, as there a decision of how the refinement should be relies on the domain elements occurring in the debugging atoms. We alter the *refine* atoms to get the information of which abstract domain occurs as a reason for spuriousness.

For demonstration, we show an example run of mDASPAR.

Example 6.5 (ctd)**.** We run mDASPAR with the input program (Figure 6.4) and the instance shown in Figure 6.1, with the initial mapping m of clustering the grid-cell into four regions (Figure 6.3a).

step 1 After constructing the non-ground abstract program (Figure 6.5) and computing the relation types, mDASPAR computes an abstract answer set

$$\{reachable(a_{1234}, b_{1234}), reachable(a_{5678}, b_{1234}), reachable(a_{1234}, b_{5678})\}.$$

step 2 The correctness checking is first done with Π_{debug_0} where the ab atoms only contain rule names (Figure 6.6) to obtain the optimal answer set

$$\{ab\_deactCons_{r3}, ab\_deact_{r2}.\}$$

step 3 Π_{debug} is constructed only for $r2$ and $r3$ where now the variables in the rule are also taken into account, by defining $ab\_deact_{r2}(X_1, Y_1)$ as

$$\begin{aligned}
ab\_deact_{r2}(X_1, Y_1) &\leftarrow ap_{r2}(X_1, Y_1), not\ reachable(X_1, Y_1).\\
\bot &:\sim ab\_deact_{r2}(X_1, Y_1).[1, X_1, Y_1]\\
\bot &:\sim ab\_deact_{r2}(X_1, Y_1), mapTo(X_1, Y_1, A_1, B_1),\\
&\quad isSingleton(A_1), isSingleton(B_1).[20, X_1, Y_1]\\
refine(A_1, B_1) &\leftarrow ab\_deact_{r2}(X_1, Y_1), mapTo(X_1, Y_1, A_1, B_1), isCluster(A_1).
\end{aligned}$$

Figure 6.6: Constructed debugging program $\mathcal{T}_{deact}[\Pi] \cup \mathcal{T}_{deactCons}[\Pi] \cup \mathcal{T}_{act}[\Pi, \mathcal{A}]$

$$\begin{array}{ll}
ko_{r1}. & \\
\{reachable(X,Y)\} & \leftarrow ap_{r1}(X,Y). \\
ab\_deact_{r1} & \leftarrow ap_{r1}(X,Y), not\ reachable(X,Y). \\
\bot & :\sim ab\_deact_{r1}.[1] \\
ko_{r2}. & \\
\{reachable(X_1,Y_1)\} & \leftarrow ap_{r2}(X_1,Y_1). \\
ab\_deact_{r2} & \leftarrow ap_{r2}(X_1,Y_1), not\ reachable(X_1,Y_1). \\
\bot & :\sim ab\_deact_{r2}.[1] \\
ko_{r3}. & \\
ab\_deactCons_{r3} & \leftarrow ko_{r3}, ap_{r3}(X,Y,X_1,Y_1). \\
\bot & :\sim ab\_deactCons_{r3}.[1] \\
ko_{r5}. & \\
\{neighbor(X,Y,X_1,Y_1)\} & \leftarrow ap_{r5}(X,Y,X_1,Y_1). \\
ab\_deact_{r5} & \leftarrow ap_{r5}(X,Y,X_1,Y_1), not\ neighbor(X,Y,X_1,Y_1). \\
\bot & :\sim ab\_deact_{r5}.[1] \\
ko_{r6}. & \\
\{neighbor(X,Y,X_1,Y_1)\} & \leftarrow ap_{r6}(X,Y,X_1,Y_1). \\
ab\_deact_{r6} & \leftarrow ap_{r6}(X,Y,X_1,Y_1), not\ neighbor(X,Y,X_1,Y_1). \\
\bot & :\sim ab\_deact_{r6}.[1] \\
& \\
\{neighbor(X,Y,X_1,Y_1)\} & \leftarrow bl_{r5}(X,Y,X_1,Y_1), bl_{r6}(X,Y,X_1,Y_1). \\
ab\_act(neighbor(X,Y,X_1,Y_1)) & \leftarrow bl_{r5}(X,Y,X_1,Y_1), bl_{r6}(X,Y,X_1,Y_1), \\
& \quad neighbor(X,Y,X_1,Y_1). \\
\bot & :\sim ab\_act(neighbor(X,Y,X_1,Y_1)).[1,X,Y,X_1,Y_1] \\
& \\
\{reachable(X,Y)\} & \leftarrow bl_{r1}(X,Y), bl_{r2}(X,Y). \\
ab\_act(reachable(X,Y)) & \leftarrow bl_{r1}(X,Y), bl_{r2}(X,Y), reachable(X,Y). \\
\bot & :\sim ab\_act(reachable(X,Y)).[1,X,Y]
\end{array}$$

$$refine(A_1,B_1) \leftarrow ab\_deact_{r2}(X_1,Y_1), mapTo(X_1,Y_1,A_1,B_1), isCluster(B_1).$$

and similarly $ab\_deactCons_{r3}(X,Y,X_1,Y_1)$. The correctness checking finds an optimal answer set with the atoms

$$\{refine(a_{1234}, b_{5678}), refine(a_{5678}, b_{5678})\}.$$

step 4 The region (a_{1234}, b_{5678}) is randomly picked to refine to $\{(a_{12}, b_{56}), (a_{12}, b_{78}), (a_{34}, b_{56}), (a_{34}, b_{78})\}$ by updating the corresponding mapping m.

step 5 Relation types according to the new mapping are computed and the loop goes back to step 1 to compute a new abstract answer set.

The loop continues until unsatisfiability is achieved. The abstraction shown in Figure 6.3b is one such abstraction where unsatisfiability is observed.

6.3.2 Incremental Concreteness Checking

In some cases, even the two-phase checking may not help with easily finding the optimal answer set during the debugging step as the original domain is large or many atoms with arguments cause to consider many possible concretizations of the abstract elements. To help with the difficulty of correctness checking on the original domain, we considered two approaches:

(1) For problems that contain a clear order on the atoms for the solution, checking is done incrementally over the order.
(2) If no such clear order exists, the checking is done via incrementally concretizing the abstract domain, following an iterative deepening style.

Approach (2) is applied to avoid making the concreteness check directly at the original domain. If the abstract answer set is spurious, this could be detected in the partially concretized domain, without the need to do the full check. The aim of Approach (1) is to avoid checking the whole ordered sequence of atoms, and catching the spuriousness in some of its prefix. To do this in the context of ASP, we require the following notion over the program.

Ordered modularity An incremental approach proposed by [GKK$^+$08], that builds on the concept of modules [OJ06], is on gradually increasing the bound to the solution size, represented by a parameter k, to help with both grounding and solving. In their case, they are searching for an answer set with minimum size over k, thus they increment the parameter until an answer set is computed. We use a similar idea to detect the spuriousness of an abstract answer set by gradually increasing the parameter. However, in our case, the increment is done until the spuriousness is realized with debugging, i.e., an answer set with an abnormality atom is obtained. We take a simpler view by limiting the generated grounding of the program to the parameter.

Let Π be a program with the Herbrand base $HB_\Pi = \mathcal{L}_B \cup \mathcal{L}_k$, for parameter k ranging over the natural numbers, where $\mathcal{L}_B$ represents the *static* literals with arguments independent of parameter k, and $\mathcal{L}_k$ represents the *dynamic* literals which have an argument k. For a set X of literals, we denote by $grd(\Pi)|_X$ the set $\{r \mid H(r) \cup B(r) \subseteq X\}$ of rules in $grd(\Pi)$ that contain only literals from X. Let $X_i \subseteq HB_\Pi$ denote the set of literals until the parameter value i, i.e., $X_i = \mathcal{L}_B \cup \bigcup_{j=0}^{i} \mathcal{L}_{k/j}$, where $\mathcal{L}_{k/j}$ denotes the set of literals with the respective argument of value j. The rules of $grd(\Pi)$ until parameter value i then are given by $grd(\Pi)|_{X_i}$, simply denoted $grd(\Pi)|_i$.

Let $I_{\leq i}$ denote the projection of an interpretation I to the literals related with the parameter value i, i.e., $I_{\leq i} = I \cap (\mathcal{L}_B \cup \bigcup_{0\leq j \leq i} \mathcal{L}_i)$. We say that Π is *ordered modular* if for each $I \in AS(\Pi)$, $I_{\leq i} \in AS(grd(\Pi)|_i)$ for all $0 \leq i \leq k$. This property means that determination of the occurrence of a literal l in an answer set $I_{\leq i}$ relies only on the decisions made until point i.

Proposition 6.1. *Let* Π *be an ordered modular program,* m *a domain mapping for* Π*, and let* $\hat{I} \in AS(\Pi^m)$*. If* $\hat{I}_{\leq i} \subseteq \hat{I}$ *is spurious for some* $i \leq n$*, then* $\hat{I}$ *is spurious.*

Proof. Assume $\hat{I}$ is concrete. This means that there exists some $I \in \Pi$ such that $m(I) = \hat{I}$. As Π is ordered modular, $I_{\leq i} \in AS(grd(\Pi)|_i)$. Thus, $m(I_{\leq i}) = \hat{I}_{\leq i}$ is concrete. □

We describe in detail the implementation of these approaches.

Incrementing *time*

Approach (1) is implemented in mDASPAR to handle planning problems with atoms having *time* arguments. By Proposition 6.1, we know that for a sequence of actions described in the abstract answer set, if the first few actions do not have a corresponding original plan, one can conclude that the plan described in the abstract answer set is spurious.

A common description of the planning problem in ASP uses two sorts *time*,*timea*, where *time* has the domain $\{0, \ldots, n\}$ and *timea* has the domain $\{0, \ldots, n-1\}$. The sort *timea* is used for action atoms, while *time* is used to define the rest of the fluents. For a given program Π with such a description of a planning problem that contains facts for *time* sort with domain $\{0, \ldots, n\}$, mDASPAR is implemented as follows. We denote by $\mathcal{T}_{meta}[\Pi]|_i$ the meta-program $\mathcal{T}_{meta}[\Pi]$ which contains *time* facts (resp. *timea*) until domain element i (resp. $i-1$), and by $\Pi_{debug}|_i$ we denote the similar restricted version of Π_{debug}. For a computed abstract answer set $\hat{I}$, which shows a plan $\langle s_0, a_0, s_1, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$ where the states s_i are described by a set of fluents that have a *time* argument and the actions a_i are described using action atoms that have a *timea* argument, we denote by $\hat{I}|_i$ the part of the plan until time point i.

Starting with $i = 1$, we continue the below iteration while $i \leq n$.

step (1) Create $\mathcal{T}_{meta}[\Pi]|_{i-1}$ and $\Pi_{debug}|_i$.

step (2) Check correctness of $\hat{I}|_i$ with $\Pi_{debug}|_i \cup \mathcal{T}_{meta}[\Pi]|_{i-1} \cup Q^m_{\hat{I}|_i}$.

step (3) If spurious, exit loop; otherwise, increase i by 1.

This way, we check the correctness of $\hat{I}$ for the action made at time i, by doing the debugging only for time point i as the guessing for time points $t < i$ is restricted by using $\mathcal{T}_{meta}[\Pi]|_{i-1}$. The time is increased incrementally while the partial solution gives a concrete partial plan. Once spuriousness is observed, the checking is stopped.

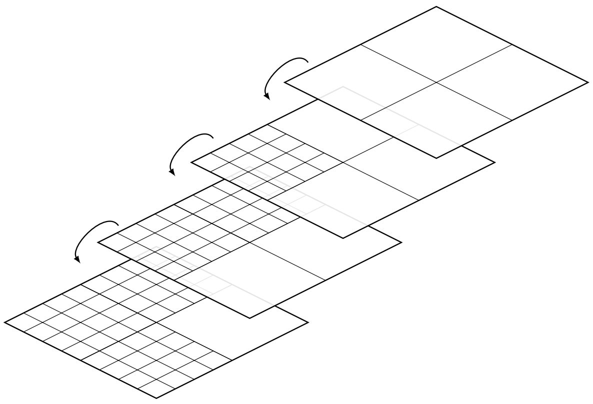


Figure 6.7: Step-wise partial concretization of a grid-cell abstraction

Partial concretization

For Approach (2), we make use of the possibility to have a hierarchy of abstractions mentioned in Proposition 4.20. The idea is to partially concretize the abstract domain, by fully concretizing certain regions and keeping the remaining regions at the abstract level. Figure 6.7 shows the hierarchy of some partial concretizations of the initial mapping. For a given mapping m, we consider a set of possible partial concretizations. We then check the correctness of an abstract answer set I over the program with partially concretized domain. Since the partially concretized domain still describes an abstraction compared to the original domain, this check can not be immediately done over the original program. For that, we have to do the correctness checking with the debugging over the abstract program w.r.t. the partial concretization.

The approach is implemented in mDASPAR as follows. For a given mapping m, starting with $j = 1$, the iteration focuses on concretizing j regions at a time, and checks the correctness in each such j-region combination. The iteration continues until spuriousness is detected or $m = m_{id}$:

```
step (1) Compute $j$-region concretizations of $m$, say $m_1, \dots, m_n$.
step (2) For every $m_i \in \{m_1, \dots, m_n\}$;
         1. Create $\Pi^{m_i}$ with $\mathcal{T}_{m_i}$ and the set $\{m_i(p(c)).|p(c). \in \Pi\}$ of facts, and $\Pi^{m_i}_{debug}$.
         2. Create the mapping $m'$ such that $m'(m_i(D)) = m(D)$.
         3. Check correctness of $I$ with $\Pi^{m_i}_{debug} \cup Q^{m'}_I$.
         4. If spurious, exit loop with debug answer $C$.
step (3) If $C \neq \emptyset$, refine $m$ according to $C$ and go back to step (1); otherwise, increase $j$
       by 1, and go back to step (1)
```

We do the correctness checking on the abstract level m_i by making use of Π^{m_i}. If I is concrete w.r.t. the partially concretized abstraction, this does not immediately guarantee that I is concrete; thus, the concretization is increased to redo the check. If however, spuriousness is detected, the mapping is refined and the partial concretization continues from the updated mapping.

6.4 Evaluation: Unsolvable Problem Instances

We investigate getting explanations of unsatisfiable grid-cell problems by achieving an abstraction over the instance to focus on the troubling area. We consider the following benchmark problems:

- *Reachability (R)*: This problem needs the neighboring cell information and can be encoded without introducing guesses. We check whether every cell is reachable in the given instance. The unsatisfiability can occur due to the layout of the obstacles.
- *Knight's Tour (KT)*: This problem is on finding a tour on which a knight visits each square of a board once and returns to the starting point. It is commonly used in ASP Competitions, with possible addition of forbidden cells. The cause of unsolvability is due to having forbidden cells that prevent the knight from moving. In ASP competitions, this problem is encoded by guessing a set of $move(X_1, Y_1, X_2, Y_2)$ atoms and ensuring that each cell has only one incoming and one outgoing movement.[1] There is no *time* sort (as in planning) which would describe an order.
- *Visitall*: We extended the planning problem of visiting every cell (without revisiting a cell) with obstacles. This problem needs the neighboring cell information and it can be encoded in two forms;

  (V) as a planning problem, in order to find a sequence of actions that visits every cell, or

  ($\mathrm{V_{KT}}$) as a combinatorial problem similar to the Knight's Tour encoding.

  In order to have smaller time domains, we encoded V by defining $go(X, Y, T)$ actions that can move horizontally/vertically to a cell X, Y (without passing through an obstacle) and the passed cells become visited. We set the time limit to be 30 time steps.
- *Sudoku (S)*: This problem has also been used in ASP competitions.[2] Its encoding consists of a guess of numbers in the cells combined with simple constraints such as one symbol per column, one symbol per subregion etc. The unsolvability occurs due to violation of these constraints.

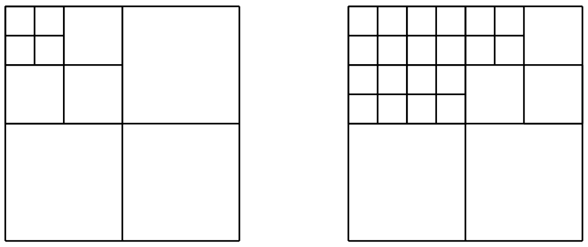
We generated 10 unsatisfiable instances complying to the following properties so that the unsolvability can be explained by focusing on a trouble making area:

[1] https://www.mat.unical.it/aspcomp2013/KnightTour

[2] https://dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/Sudoku.shtml

Figure 6.8: Measure for quality of a quadtree abstraction

(a) $c_m = \frac{4*2+3*1}{64*2+16*1} = 0.076$ (b) $c_m = \frac{20*2+3*1}{64*2+16*1} = 0.299$



- In Reachability instances, a group of neighboring cells are unreachable due to the obstacles surrounding them.
- For Knight's Tour instances, one or two cells are picked to have only one valid movement to an obstacle-free cell. This way, these cells and the obstacles that do not allow the valid movements become a reason for unsolvability.
- The Visitall instances consist of either two dead-end cells or areas with only one cell passage, so that one is forced to pass some cells more than once, which is not allowed.
- For Sudoku, we generated a layout of numbers that force to violate the constraints when solving the problem.

Measuring abstraction quality We consider a *quality measure* of the quad-tree abstraction by normalizing the number of abstract regions of a certain size and their level in the quadtree. The cost of a mapping m over an $n \times n$ grid is

$$c(m) = \sum_{i=0}^{\ell} r_{2^i}(m)(\ell - i) \ / \ \sum_{i=0}^{\ell} n^2 2^{-i^2}(\ell - i),$$

where $\ell = \log_2(n) - 1$, $r_{2^i}(m)$ is the number of abstract regions of size $2^i \times 2^i$ in m, and $n^2 2^{-i^2}$ is the number of abstract regions of size $2^i \times 2^i$ in the $n \times n$-sized cell. The factor $\ell - i$ is a weight that gives higher cost to abstractions with more low-level regions. The abstraction mapping with the smaller cost, i.e., the smaller level of detail, is considered to be of *better quality*.

Figure 6.8 shows measures of two abstraction mappings. The abstraction in Figure 6.8a is coarser than the one in Figure 6.8b, and this is reflected in the computed measures. Assigning more weight to having coarser regions would stress the importance of having a coarse abstraction even more. The computation of the measure is purely structural and domain-independent. Other measures can be defined that are dependent on the domain which considers further aspects, e.g., an abstraction that singles out smallest number of cells with obstacles is preferred.

Table 6.1: Evaluation results of applying different debugging approaches

	debugging type	average steps	average cost	minimum steps	minimum cost	best steps	best cost
R	default	5.4	0.227	5.4	0.227	5.0	0.208
	2-phase	5.5	0.233	5.3	0.222		
S	default	6.5	0.696	5.1	0.550	3.2	0.371
	2-phase	4.3	0.476	3.4	0.391		
KT	2-phase	14.3	0.643	10.4	0.460	5.6	0.245
	grid-inc	10.1	0.442	6.3	0.277		
V	2-phase[3]	16.2	0.708	13.9	0.608	8.7	0.360
	time-inc	16.3	0.712	13.5	0.569		
V_{KT}	2-phase	15.7	0.693	13.0	0.572	7.6	0.317
	grid-inc	13.0	0.569	10.3	0.449		

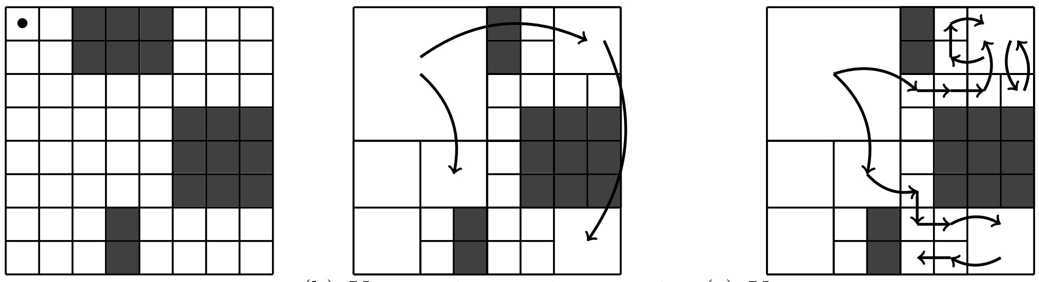
Evaluation Results

We compare different debugging approaches described in Section 6.3.1 in order to observe their effects in the resulting abstractions and the taken refinement steps. Due to their encodings and constraints, the Knight's Tour and Visitall problems are the challenging ones. To observe whether an incremental checking (Section 6.3.2) could help in deciding on a refinement and achieve better abstractions, we applied partial concretization for KT and V_{KT}, and incremental *time* checking for V. To evaluate the quality of the resulting abstraction on how far it is from the *best possible* abstraction that shows the unsolvability, we also checked the existence of a coarser abstraction that still preserves unsatisfiability.

Table 6.1 shows the main evaluation results. We compare different debugging approaches in terms of the average refinement steps and average costs of the resulting abstractions over 10 runs, and also on the best outcome obtained (with minimum refinement steps and minimum mapping cost) among the 10 runs. The right-most two columns are for checking the existence of a coarser abstraction from the best outcome obtained in the runs. The time to find an optimal solution when debugging the concreteness checking was limited by 50 seconds. If none is found within the time limit, the refinement is decided on the basis of suboptimal analyses.

For Reachability and Sudoku, we observe that abstractions close to the best possible ones can be obtained. Abstractions that are slightly better were obtained with 2-phase debugging, due to putting the focus on the right part of the abstraction after the first step. For Knight's Tour and Visitall, we observe that incremental checking can obtain better abstractions. For 2-phase debugging, the program mostly had to decide on suboptimal concreteness checking outputs, due to timeouts. Additionally, for problem Visitall with the V encoding, 2-phase debugging caused memory errors (when over 500 MB) on some

Figure 6.9: Spurious plans in abstractions that distinguish the single-cell passages



(a) inst. #10

(b) V - spurious action transitions

(c) V_{KT} - separate sequences of actions that visit the cells that are reachable with single-cell passages

runs for some instances, thus not all 10 runs could be completed.

We can also see a difference of the resulting abstractions for the different encodings of Visitall. The planning encoding causes to achieve unsatisfiability with finer abstractions. Guesses of spurious sequences of actions in the abstraction cause the debugging to decide on refinements that avoid these sequences. The focus moves towards the unsolvability when the abstract action sequence is not executable due to an obstacle. In some instances where the reason for unsolvability is not easily caught by having two dead-ends, shifting the focus on to the existence of some obstacles does not achieve unsatisfiability. The abstract encoding manages to find a plan passing through different sized regions by avoiding the constraints due to uncertainty. For these instances, the abstraction needs to be fine enough to get rid of most of the uncertainty.

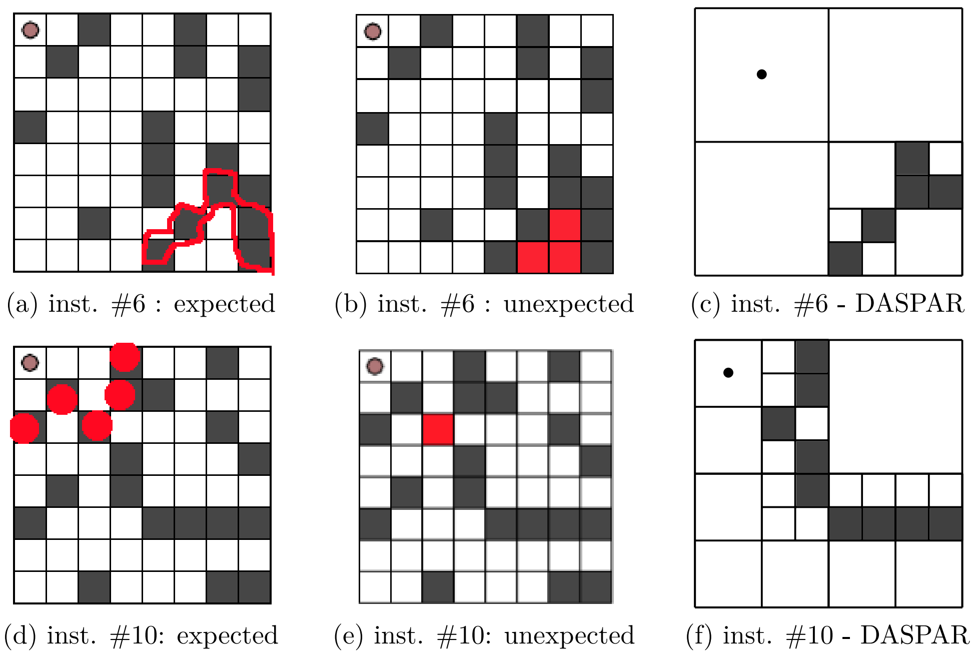
For example, Figure 6.9a shows such an instance. An abstraction that distinguishes the one-passage-entries and the obstacles that surround the cells cannot achieve unsatisfiability for V. Figure 6.9b shows some spurious action transitions that are determined in a plan found with the V encoding among regions by avoiding the constraints due to uncertainty. Unsatisfiability cannot be achieved for V_{KT} as well. This is due to guessing a set of *move* atoms, which achieves that every cell is visited, but actually does not have a corresponding original order of movements. Figure 6.9c shows the spurious order of movements that gets split in order to visit each cell that is only reachable through a one-passage-entry. If the abstraction is refined to distinguish the cells in the respective corners, then unsatisfiability is realized.

6.4.1 User Study on Unsatisfiability Explanations

We were interested in checking whether the obtained abstractions match the intuition behind a human explanation. For Reachability and Visitall, finding the reason for unsolvability of an instance is possible by looking at the obstacle layout. Thus, we

[3]A total of 16 runs could not be completed due to memory errors. The results are computed among the runs that have been completed.

Figure 6.10: Explanatory abstractions for unsatisfiable Reachability instances


(a) inst. #6 : expected (b) inst. #6 : unexpected (c) inst. #6 - DASPAR

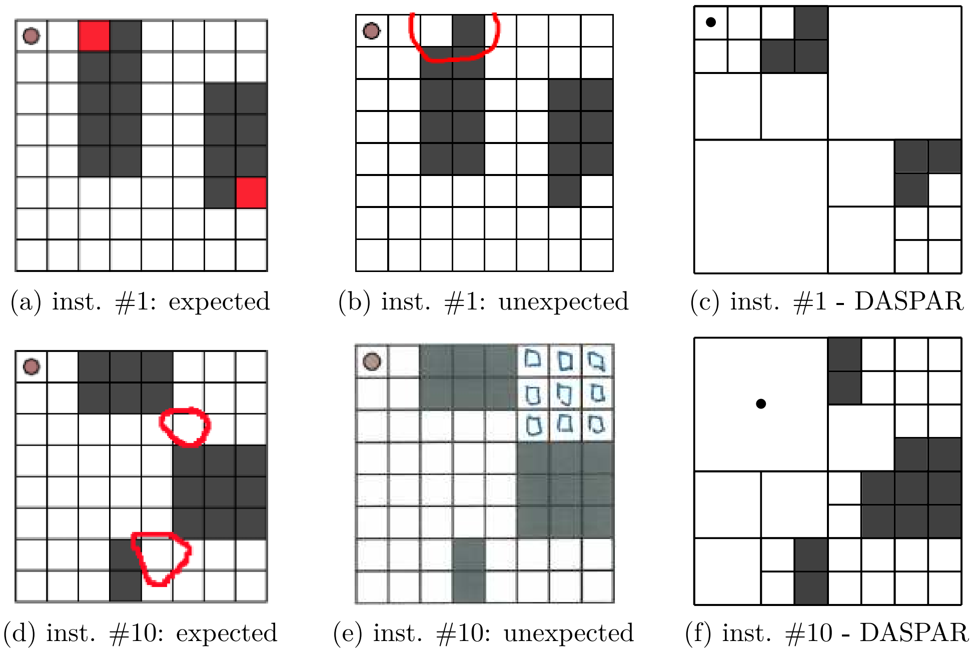
(d) inst. #10: expected (e) inst. #10: unexpected (f) inst. #10 - DASPAR

conducted a user study for these problems in order to obtain the regions that humans focus on to realize the unsolvability of the problem instance.

As participants, we had ten PhD students of Computer Science at TU Wien. We asked them to mark the area which shows the reason (if more than one exists mark with different colors) for having unreachable cells in the Reachability instances and the reason for not finding a solution that visits all the cells in the Visitall instances. Explanations for ten instances of each problem were collected. We discuss the results for both problems by showing two of the responses (expected and unexpected) and the best abstraction obtained from mDASPAR when starting with the initial mapping. All user explanations can be found in `http://www.kr.tuwien.ac.at/research/systems/abstraction/mdaspar_material.zip`.

Reachability. The expected explanations (e.g., Figures 6.10a and 6.10d) focus on the obstacles that surround the unreachable cells, as they prevent them from being reachable. When their respective abstraction mappings are given to mDASPAR, the constructed abstract program is also unsatisfiable. The explanation in Fig. 6.10b puts the focus on the unreachable cells themselves, and Fig. 6.10e distinguishes a particular obstacle as a reason. When the respective abstraction mappings are given to mDASPAR, it needs to refine further to distinguish more obstacles and achieve unsatisfiability. The mark in Figure 6.10e is a possible solution to the unreachability of the cells, since removing the marked obstacle makes all the cells reachable.

Figure 6.11: Explanatory abstractions for unsatisfiable Visitall instances


(a) inst. #1: expected (b) inst. #1: unexpected (c) inst. #1 - DASPAR
(d) inst. #10: expected (e) inst. #10: unexpected (f) inst. #10 - DASPAR

In ASP, checking whether all cells are reachable is straightforward, without introducing guesses. This is also observed to be helpful for mDASPAR, as most of the resulting abstractions were similar to the gathered answers. Since in the initial abstraction, the abstract program only knows that the agent is located in the upper-left abstract region, in instance #10, mDASPAR follows a different path in refining the abstraction, and reaches the abstraction shown in Figure 6.10f. Although not the same as the one given by the users, this abstraction also shows a reason for having unreachable cells. Humans use the implicit knowledge that the agent is located in the upper-left corner in order to determine the reason for unreachability of the cells, and thus focus on a different area than mDASPAR. Such an abstraction can also be achieved with the method, by influencing the refinement decisions towards singling out the initial location of the agent.

The abstractions achieved by mDASPAR are more general in the sense that the precise initial location of the agent is not necessary to distinguish the unreachable cells. The agent can be initially located in any of the cells mapped to the respective abstract region. The precise layout of the obstacles in the abstracted regions also does not play a role in determining the unreachability of the distingished cells.

Visitall. Most of the users pick two dead-end cells in the instances (if such occur) as an explanation for unsatisfiability. However, the explanations are given by marking these dead-end cells, instead of the obstacles surrounding them (see Fig. 6.11a), which are the actual cause for them to be dead-end cells. Even with abstraction mappings that also distinguish the surrounding obstacles of these dead-end cells, the corresponding abstract program still remains satisfiable. mDASPAR needs to refine further to distinguish the neighboring cells (as in Fig. 6.11c), to realize that it can only pass through one grid-cell when reaching the dead-end cells, and thus achieve unsatisfiability.

Unexpectedly, some users marked only one of the dead-end cell as an explanation (Figure 6.11b), which is actually focusing on a possible solution.

Some instances do not contain two dead-end cells, but single-cell passages to some regions. Fig. 6.11d shows an entry that distinguishes these passages, while again focusing only on the cells themselves. For these instances, the results of mDASPAR are quite different. We discussed this already over Figure 6.9 in the evaluation results. Fig. 6.11f shows the best abstraction achieved for V_{KT}. It distinguishes all the cells in the one-passage-entry regions to realize that a desired action sequence can not be found.

The generality of the achieved abstractions can also be observed here. The precise layout of the agent and the obstacles in the abstracted areas do not change the unsatisfiability result due to not being able to visit all the cells in the distinguished parts.

Observations The abstraction method can demonstrate the capability of human-like focus on certain parts of the grid to show the unsolvability reason. However, humans are also implicitly making use of their background knowledge and do not need to explicitly state the relations among the objects. Empowering the machine with such capabilities remains a challenge. The study also showed the difference in understanding the meaning of "explanation". For some, showing the solution to get rid of unsolvability is also seen as an explanation. This difference in understanding shows that one needs to clearly specify what they want (e.g., "mark only the obstacles that cause to have unreachable cells"), which would then deviate from studying the meaning of explanation.

6.5 Abstraction for Policy Refutation

We now focus on using the abstraction over grid-cells for the problem of checking policies on whether they manage to guide the agent towards the goal. We consider two versions of this problem and discuss the use of abstraction.

As a running example, we consider an agent trying to find its way in a maze towards a goal point (similar in spirit to the example of finding a missing person). For representing and generating the mazes, we use an altered version of the Maze Generation encoding from ASP Competition 2011.[4] The first example policy when talking about mazes that may come to one's mind is the "right-hand rule" policy. This policy is known to work in many maze instances, except the ones where the goal point is in the middle area and the agent is forced to loop due to the obstacle layout.

[4]`https://www.mat.unical.it/aspcomp2011/files/MazeGeneration/maze_generation.enc.asp`

Table 6.2: Policy checking in maze instances

	sat/unsat	ave. step	ave. cost	ave. best step	ave. best cost
naive	16/4	7.2	0.391	6.5	0.362
right-hand	6/14	12.5	0.630	11.8	0.599

Does the policy work on a given instance?

For fixed problem instances, this check is done by a search of a counterexample trajectory which follows the policy but does not reach the goal. If no such trajectory can be found (i.e., unsatisfiability is achieved), then this shows that the policy works on the instance. Abstraction can be used to focus on the part of the instance which is enough to show that the policy does not achieve the goal or that following the policy always achieves the goal. Notice that the case when the policy works then becomes similar to having unsatisfiable problems which was the focus in the previous section.

The granularity of the abstraction needed to show the result of the checking depends on the complexity of the policy. As demonstrated in Figure 6.13, refuting the well-known “right-hand rule” policy needs the abstraction to at least refine the outer area (if not more). To observe how the policy type affects the resulting abstraction, we did some experiments. To help with the refinement decisions, the initial abstraction distinguishes the starting point of the agent and abstracts over the rest.

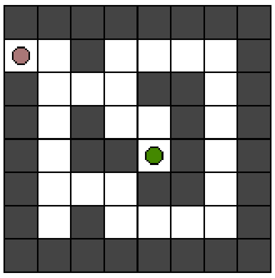
We consider the following two policies:

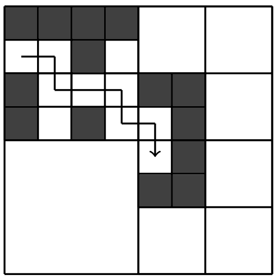
(A) Right-hand rule: Follow the wall on the right-hand side.
(B) Naive policy: Choose the direction to move to with the priority right $>$ down $>$ left $>$ up.

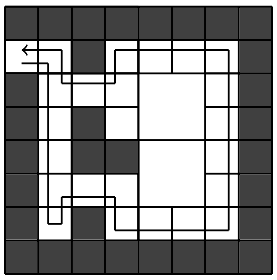
We generated 20 instances where on some of them both, some or none of the policies work. For the debugging method we picked time increment, since we wanted the debugging to focus on each step of the abstract trajectory starting from the beginning, and on whether or not they match the policy’s decisions in the corresponding original trajectory. Furthermore, the refinement decision is made only from the check of the first abstract answer set obtained. This is due to the fact that, if diverse answer sets are considered, a concrete answer set among spurious ones can be encountered. This would finalize the search and achieve an abstraction that definitely is not faithful. Thus, to increase the chances of achieving faithful abstractions, only one abstract answer set is picked.

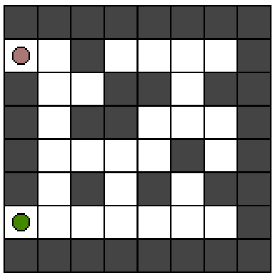
Table 6.2 shows the results of using mDASPAR to achieve an abstraction with a concrete solution. Obtaining SAT means that the program found a concrete solution, i.e., a concrete counterexample trajectory, which shows that the policy does not work, while having UNSAT means that the policy works. As expected, the naive policy failed to work for most of the instances. Since the right-hand rule forces to traverse the environment more, mDASPAR required to have finer abstractions to figure out the concrete solution. In both cases, the obtained abstractions were not too distant from the best possible ones,

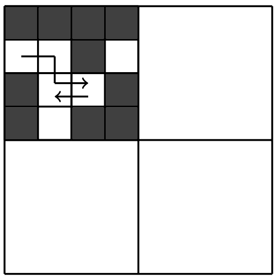
Figure 6.12: Abstractions on policy checking in maze instances (with the supporting/refuting paths)

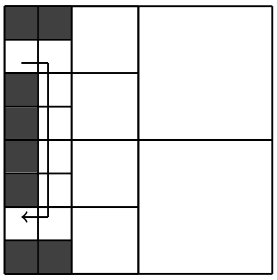

(a) naive: works, right-hand: doesn't work

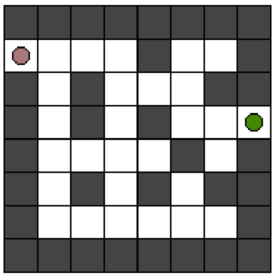

(b) Supporting the naive policy

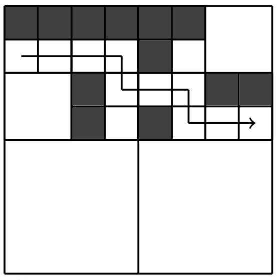

(c) Refuting the right-hand policy

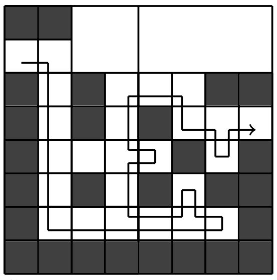

(d) naive: doesn't work, right-hand: works


(e) Refuting the naive policy


(f) Supporting the right-hand policy


(g) naive: works, right-hand: works


(h) Supporting the naive policy


(i) Supporting the right-hand policy

although still sometimes the focus was shifted to the irrelevant parts of the grid. All of the obtained resulting abstractions were faithful, which means that they were able to show the actual behavior of the policy. Figure 6.12 shows the resulting abstractions for three of the instances.

Does the policy always work?

This is a more involved check, since a set of possible instances has to be considered and a search of a counterexample trajectory among each instance needs to be done. If the policy works, then all possible policy trajectories in all instances have to be checked to conclude this result. For this case, considering an abstraction that focuses on a certain part of the grid may not be useful, since depending on the structure of the instances different parts of the grid may need to be singled out. However, if the policy does not work, it is enough to find an instance in which a counterexample policy trajectory can be shown. Thus, an abstraction that focuses on a certain part of the grid where some instance can show a counterexample would be useful.

In ASP, such a check can be done by making two sets of guesses: (1) choose a valid instance, by guessing the layout of the environment and the position of the goal, and (2) determine a counterexample trajectory, by guessing the movements of the agent following the policy which do not achieve the goal in the instance. If the policy is deterministic (i.e., chooses exactly one action at a state), then the second guessing part becomes straightforward. However, for nondeterministic policies, a choice of possible actions to take exists, which adds to the complexity of the search.

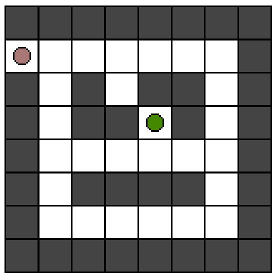
The experiments showed that having these guesses combined with the guesses introduced in the syntactic transformation causes to encounter many spurious abstract answer sets. These answer sets sometimes force the refinement decisions to be made towards useless parts of the grid. For example, when the general policy checking is done for the right-hand policy, mDASPAR needs to refine back to the original domain to catch some instance with a counterexample trajectory, since the policy forces to traverse the environment, and in the abstract encoding, the guesses of the instance and the movements cause to create many spurious trajectories. As for the naive policy, mDASPAR is capable of encountering a counterexample trajectory in few refinement steps. It is enough to realize that the naive policy does not work by creating a partial instance where the agent enters a dead-end and has to leave by moving left, then it starts looping by moving right and left.

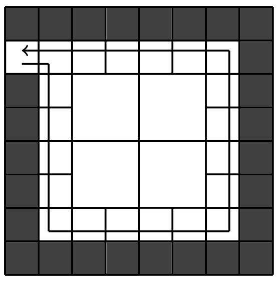
6.6 Discussion

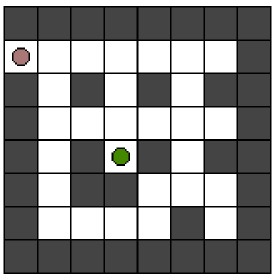
In this chapter, we have investigated the use of multi-dimensional domain abstraction on grid-cells. The quad-tree style abstraction respects the structure of the grid, and achieves a systematic abstraction refinement process. The resulting abstractions match the intuition behind human abstractions, by singling out the relevant part of the grid and abstracting away the rest.

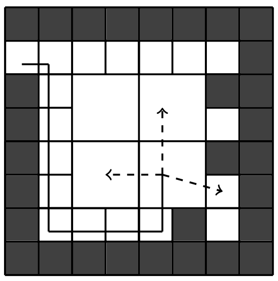
Achieving better has been abstractions shown to be possible with more sophisticated decision making for the refinement step. However, the multi-dimensionality adds to the difficulty of computing the types of the relations and the concreteness checking of abstract answer sets in the original domain. Thus, the current approach does not scale to large sized grids. For now, to aid in the relation type computation effort, once a type computation is conducted for a certain mapping, it is cashed and reused whenever the respective mapping appears. This way, the recomputations are avoided. This effort could be ameliorated by investigating analytical methods for computing and representing

Figure 6.13: Can we refute the right-hand rule policy in all maze instances with one abstraction?


(a) A counterexample instance


(b) Distinguishing the path of a counterexample trajectory


(c) A counterexample instance


(d) Spurious counterexample trajectories occur due to abstract regions

relation types. For concreteness checking, we discuss in Section 7.3 as future work a possible alternative approach to address the scalability issue.

The multi-dimensionality of the abstraction can be useful in other problems that bear a structure which needs to be respected. The approach can easily be applied to different problems, by modifying the refinement step description to respect the new structure.

Lastly, we want to emphasize that, in case the policy does not work, it is uncommon to have one abstraction mapping that can be applied with any possible instance and that is able to catch some counterexample trajectory to refute the given policy, especially one that is faithful when applied for any instance. Figure 6.13a shows an instance in which the right-hand policy is unable to reach the (green) goal point from the (red) entry point in the upper left corner. An abstraction such as Figure 6.13b is enough to realize that a loop occurs and a goal can not be reached (i.e., it is a faithful abstraction for this instance). However, this abstraction does not always distinguish the cells that force to obtain a counterexample trajectory in each possible refuting instances. For example, the instance in Figure 6.13c also forces the agent to loop, but with the same abstraction (Figure 6.13d) since there is uncertainty among the abstract regions it is still possible to create spurious counterexample trajectories in the abstract program. Thus, faithfulness can not be achieved. For this problem, the identity abstraction would be the one that can be used to (faithfully) refute the policy in all possible instances.

It would be interesting to investigate the properties needed in a policy and in the domain that makes it possible to have a universal (non-identity) abstraction to be used to (faithfully) refute the policy in all instances.

CHAPTER 7
Conclusion

7.1 Summary

In this thesis, we tackled the challenge of understanding the core elements of the behavior of an agent program by employing abstraction. We approached the problem from two directions. First, we investigated the representation of an abstraction for reactive agent behavior that follows a given policy. For this, we focused on a representation that gets rid of details irrelevant to the policy behavior and preserves the properties of the behavior, which makes it possible to do reasoning at the abstract level with a guarantee that the same holds at the original level. Second, we considered abstraction as an over-approximation, and employed the notion of abstraction in the context of Answer Set Programming, a knowledge representation and reasoning paradigm widely used in declarative problem solving, as well as modeling agent behavior, in order to be able to abstract over the irrelevant details of answer set programs.

We addressed the shortage of representations that are capable of modeling reactive policies which distinguish relevant details of the states and the transitions. Our ASP-based perspective on the representation integrates target development and online planning capabilities. These components allow one to describe a reactive behavior that decides the course of actions by determining targets as stepping stones to achieve during the interaction with the environment. Flexibility in these components does not restrict one to only using ASP or action languages, but allows for the use of other formalizations as well.

Depending on the agent's designed behavior and its determination of a course of actions at a state, some information in the state may not be necessary, relevant or even observable. For this, we considered a notion of indistinguishability which is on clustering the states that contain the same profile according to the policy. This equalization on the states shifts the focus towards the behavior that the policy enforces on the agent, by representing the transitions according to the plans determined by the policy as macro actions/big jumps between the equalized states. Since the original policy transitions are preserved, this makes it possible to reason about the behavior of the policy over all trajectories in the equalized transition system, especially to check if following the policy always manages to achieve the goal condition. In order to ensure that no new properties over the system behavior are introduced, a further condition over the equalization is introduced. This properness condition ensures that having a trajectory through the policy actions in the abstract level guarantees the existence of a concrete trajectory in the original level. Knowing that any such trajectory found in the equalized transition system exists in the original transition system gives the possibility to do further reasoning over the policy behavior. Especially, if a counterexample trajectory that shows that the policy cannot achieve the goal condition is found, then this is enough to conclude that the policy indeed does not work in the original system.

Our motivation has been to use ASP and ASP-based action languages in representing such an agent behavior that follows a desribed policy and to use abstraction to help in focusing on the key elements of the behavior. However, such a notion of abstraction has not been explored in the context of ASP before. This motivated to shift the focus of the thesis towards introducing this notion to ASP and investigating its possible uses.

We started with introducing abstraction of a program by constructing an abstract program with smaller vocabulary, and ensuring that the original program is over-approximated, i.e., every original answer set can be mapped to some abstract answer set. We described methods to construct abstract programs, while preserving the structure of the rules, by considering two approaches; one is omitting atoms from a ground program, while the other is about abstracting over the domain elements of a non-ground program. As having an over-approximation causes the possibility to have spurious answer sets which cannot be mapped to original answer sets, we proposed a method for refining the abstractions to get rid of the spuriousness. This method makes use of ASP-debugging techniques to obtain hints for refinements when checking the concreteness of the obtained abstract answer set.

We then integrated the abstraction and refinement notions and introduced a CEGAR-style methodology [CGJ+03] that starts with an initial abstraction and refines it repeatedly using hints that are obtained from checking the abstract answer sets, until a concrete solution (or unsatisfiability) is encountered. We have implemented the approach in the tools ASPARO, for omission-based abstraction, DASPAR, for domain abstraction, and mDASPAR that extends domain abstraction to handle multi-dimensionality. Given an answer set program and an initial abstraction, these tools are able to automatically produce an abstraction mapping that creates an abstract program where either a concrete answer set is encountered, or unsatisfiability is achieved.

The conducted experimental evaluations show the potential of the approaches in understanding the core elements of a program. For unsatisfiable ASP programs, we observe that automatic abstraction refinement is able to catch the unsatisfiability without refining back to the original program. With omission abstraction, this results in obtaining an abstract program where the focus of attention is narrowed down to the rules associated with obtaining unsatisfiability. As for domain abstraction, a multi-dimensional view of an abstraction makes it possible to zoom in to the area of the grid-cell which shows the reason for unsolvability. For satisfiable ASP programs, an abstraction over the domain allows for problem solving over abstract notions, which reflect relevant details only. The obtained abstract solutions can then be used in understanding the unnecessary details of the domain which does not make a difference in finding the solution.

7.2 Related Work

In this section, we describe the most related works to our thesis. Since the motivation of the thesis is to gain the capability of reasoning about the behavior of the agent, this makes it related to the area of agent verification, where the notion of abstraction has also been used. The aim of using abstraction to understand the core elements of problem solving makes the thesis also close to the works on using abstraction for planning and generalized planning. The main difference of the thesis lies in the focus on a highly expressive nonmonotonic logic formalism.

Agent Verification One view on representing agents has been by seeing them as *rational*, acting according to their beliefs, desires and intentions [Bra87, CL90]. The BDI model proposed by Rao and Georgeff [RG$^+$95, RG91] is a logical framework for such an agent theory, with a formalism that is based on branching time model. There are many different agent programming languages and platforms based on this model. Verifying properties of multi-agent systems represented in these languages has also received attention [BFVW06, DFWB12]. These approaches consider very complex architectures that even contain a plan library where plans are matched with the intentions or the agent's state and manipulate the intentions. Verification for such complex BDI architecture gets very challenging. In order to deal with large state spaces, Bordini et al. [BFVW04] represented each agent program by a graph describing the dependencies and proposed an algorithm to eliminate parts of the program that do not affect the property to be verified.

There have been works conducted on the verification of GOLOG programs [LRL$^+$97], a family of high-level action programming languages defined on top of action theories expressed in the situation calculus. The method of verifying properties of non-terminal processes are sound, but incomplete as the verification problem is undecidable [DGTR97, CL08]. By resorting to action formalisms based on description logic, decidability can be achieved [BZ13]. Another way of reaching decidability is to consider a fragment where actions affect only a finite number of objects [DGLP12, DGLPS16] or the dependencies between fluents in the successor state axioms are restricted [ZC16]. Mo et al. [MLL16] introduced a method that uses predicate abstraction for automatic verification of partial correctness of GOLOG programs.

Verifying temporal properties of dynamic systems in the context of data management was studied by Calvanese et al. [CDGMP13] for description logic knowledge bases under a “bounded-state” assumption. They considered a *generic* notion over the systems, which states that the properties of individuals are only those that can be inferred from the knowledge base, for which they introduced a notion of equivalence of ABoxes.

Abstraction was studied for situation calculus action theories by Banihashemi et al. [BDGL17], who imposed a bisimulation restriction on the abstraction in order to ensure that reasoning about agent’s actions at the abstract level can be mapped to a concrete reasoning. Later, in [BDGL18] they showed how this restricted notion of abstraction can be used in reasoning about a strategy for an agent to achieve a goal at the high level and then mapping it back into a low-level strategy. Their focus however was not on how such an abstraction can be found.

For verifying the behavior of multi-agent systems, the use of abstraction has been investigated by Lomuscio et al. for abstracting over each agent to construct an abstract system while preserving the properties expressed in a temporal-epistemic logic [CDLR09] or alternating-time temporal logic [LM14]. In [CDLR09] the focus is not on how such an abstraction can be built. In [LM14] an abstraction of the states is made by putting together the states that have the same possible actions to execute and action abstraction keeps the actions of certain agents, by omitting the rest. This is similar to the equalization we considered in the dynamic setting in Section 3.3, where we have abstracted over the environment actions. They considered a three-valued logic and the abstraction also preserves the behavior of not satisfying a property. The spuriousness may occur for the case of achieving an “uncertain” result for checking a specification in the abstract level, which then forces to refine the abstraction by splitting the states by investigating the subformulas of the specification. They later extended this work to infinite state models [LM16] and abstracted them to finite models using predicate abstraction by analysing the specification to be checked. Later they presented an interpolant-based refinement method [BLM16].

Verifying the correctness of ASP programs by representing them in first-order logic is an ongoing research topic of Lühne et al. with recent results on verifying strong equivalence [LLS19].

Abstraction in Planning Starting from the early years of AI planning, applications of abstraction to help with the search and planning for complex domains have received a lot of attention. One main research focus has been on hierarchical planning, which is on considering different abstraction levels over the problem space. A plan is searched at the abstract level and then the solution is refined successively to more detailed levels in the abstraction hierarchy, until a concrete plan is computed at the original level. Sacerdoti [Sac74] showed an abstraction notion that keeps the “critical” preconditions of actions and ignores the rest. Knoblock [Kno94] proposed an ordered monotonicity property to ensure that solving the subproblems by refining certain parts of the plan does not change the remaining parts of the abstract plan. A similar property was considered by Bacchus and Yang [BY94], which states that if the original problem is solvable, then any abstract solution must have a refinement. Anderson and Farley [AF88] construct operator hierarchies by having classes of operators that share common effects and forming new abstract operators with the shared preconditions.

Another research focus has been on using abstractions to compute heuristics, which are estimates of the distances to the solution that guide the search for plan. Pattern databases [CS98] are constructed from the results of projecting the state space to a set of variables of the planning task, called a pattern, which is to be solved optimally. The omission abstraction we introduced in Section 4.2 matches the intuition behind this projection notion. Edelkamp [Ede01] was the first to apply this technique in planning, and showed that a pre-compiled look up table consisting of the costs of abstract solutions can help with the heuristic search in finding optimal solutions. The merge & shrink abstraction method of Helmert et al. [HHH$^+$07] starts with a suite of single projections, and then computes an abstraction by merging them and shrinking. A CEGAR-inspired method was proposed by Seipp and Helmert [SH13] based on cartesian abstractions which form a general class of abstractions. The reason for spuriousness of the abstract plan is detected when trying to construct a concrete plan, and the abstraction is refined by splitting the states. Obtaining such a cartesian abstraction is also possible with domain abstraction introduced in Section 4.3, while we further empower the abstraction with a multi-dimensional handle in Section 4.6 which has the capability of representing a hierarchy of abstraction levels. As we showed in Chapter 6, such an abstraction is especially useful for grids.

Although not thoroughly investigated, the notion of domain abstraction has also been considered in heuristic-search planning. Hernádvölgyi and Holte [HH99] presented a domain abstraction notion over the states which are represented as fixed length vectors of labels. They also noted the possibility of encountering spurious states with some abstractions. Hoffman et al. [HSD06] considered variable domain abstraction by modifying the add and delete lists of the operators accordingly. They also argued that obtaining efficient results from applying abstraction in planning mostly relies on the amount of irrelevance that the problem contains. This is an observation we have also made with our experiments, and further investigations on the structure that problems must have in order to obtain good results, especially in the context of ASP, is an interesting research direction.

The notion of irrelevant information and its effects were analyzed for planning by Nebel et al. [NDK97], in which different heuristics were introduced to omit such information. Fox and Long [FL99] described a method for detecting symmetries in a problem which are then treated as indistinguishable to help the planner.

In the context of ASP and action languages, Dix et al. [DKN03] proposed a way of formulating and solving hierarchical planning under the ASP semantics, with a focus on *ordered task decomposition*, which is planning each step in the order it will later be executed. For a particular application of mobile robot planning, Zhang et al. [ZYKS15] performed hierarchical planning using the action language $\mathcal{BC}$.

Generalized planning Finding a plan that can achieve the goal in a class of problem instances can give an understanding of the relevant details of these problems. This plan can then be used in any particular instance of the problem without the need to do further search. Note that, as discussed in Section 5.3, the plans that are computed with our domain abstraction method can also be seen as a generalized plan, since these plans work for any original problem instance that can be mapped to the abstract instance.

Srivastava et al [SIZ11] proposed an abstraction method for constructing generalized plans with loops, by focusing on classical planning, but the problem of selecting a good abstraction was beyond the scope of their work. Bonet and Geffner [BG15] considered a setting where uncertainty is represented by a set of states, by clustering the states that provide the same observations. This view is similar to the indistinguishability notion we proposed in Section 3.1. They study the conditions for a policy (i.e., plan) to be general enough to work on other instances. Later they extended this notion with having trajectory constraints [BDGGR17].

Illanes and McIlraith [IM16] studied abstraction for numeric planning problems by compiling them into classical planning. Recently, they used abstraction for problems that contain quantifiable objects [IM19], e.g., some number of packages to deliver to points A and B, to find generalized plans by abstracting away from the quantification that works for multiple instances of the problem. For this, they build a quantified planning problem by identifying sets of indistinguishable objects using reformulation techniques [RDBF16] to reduce symmetry, and then use an algorithm to compute a general policy. While the quantifiability conditions of [IM19] restrict its possible applications, our method has the orthogonal potential drawback of producing spurious answers.

7.3 Future Work

Introducing the abstraction notion to representing agent behavior with an action language perspective, and in particular, to the context of Answer Set Programming offers various directions of future work.

One interesting research would be to relate such an abstraction notion in ASP with action domain descriptions, and to investigate the properties of obtaining abstractions over the states and actions of the transition system. Different properties may be achieved according to the structure of the problems, e.g., indirectly effected objects that do not affect the agent's decision making may be abstracted while preserving faithfulness. Such an over-approximation is to be built on the constructed equalized transition system, which can then be used in extracting the key elements of the agent's behavior according to the policy.

Regarding over-approximation in ASP, the current abstraction method can be made more sophisticated in order to avoid introducing too many spurious answer sets. This, however, will require to conduct a more extensive program analysis, as well as to have non-modular program abstraction procedures which do not operate on a rule by rule basis; to what extent the program structure can be obtained, and understanding the trade-off between program similarity and answer set similarity are interesting research questions. We plan to explore employing rule decomposition techniques [BMW16] as a preprocessing step in order to achieve programs with a certain structure, to observe their effects for the abstraction step.

The current method can benefit from significant performance improvement, as the current prototypes have been built on top of legacy code and tools of ASP-debugging approaches [BGP+07, OPT10] from previous works. Having the concreteness checking and refinement decision making done externally over the constructed meta programs also adds to the computation effort. Incorporating these steps into an ASP solver may lead to improvement in the performance of the methodology. The ASP debugging tools such as Ouroboros [OPT10] or DWASP [DGM+15] can also be employed for this purpose. Since the computation effort rises for the concreteness checking, e.g., for the grid-cell setting, doing an incremental check may be helpful. There are ASP solvers that are making use of the modularity of the program in searching for an answer set [GKK+08, GKKS19] or apply lazy grounding [TWF19], which could be used for our purposes as well. An optimized implementation may lead to view abstraction under a performance aspect, which then becomes part of a general ASP solving toolbox.

Different methods can be also explored to help with the decision making in the refinement step. The concreteness checking in the introduced method can be costly due to the need to ground the original program while checking. This can be ameliorated by approaching the checking problem from another perspective. Justification methods [PSE09, CFF14] can be used to first get an explanation of how the abstract answer set is computed. This explanation can then be checked on the original program by tracing the decision steps and searching for an original trace. In case an original trace fails at a certain step, this would show the reason for spuriousness of the abstract justification. This reason can then be used in the refinement of the abstraction.

An important aspect of the abstraction&refinement method is the inital abstraction mapping. Starting with too coarse abstractions may mislead the method into refining irrelevant parts of the abstraction. To overcome this, an understanding of a good initial abstraction needs to be investigated. Employing symmetry breaking techniques [DTW11, DBBD16] in order to get hints on a good initial abstraction is a promising subject of future research. Furthermore, as the use of abstraction depends on the problem structure at hand, characterizations of different problem types and the effects of abstraction are necessary.

Employing abstraction to obtain human-understandable explanations on problem solving using ASP programs requires further investigation. With the current method, an abstract program can be obtained, which can achieve concrete solutions to the problem. This alone can not be enough for a human to understand the behavior. The application on grid-cell problems showed that by visualizing the obtained abstractions for the corresponding ASP programs, similarities can be observed with the focus areas in human explanations to the problems. However, a general approach for obtaining explanations that can be applicable to variety of problems is an open issue.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[ABC99] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 68–79, 1999.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Elsevier, 1988.

[ACC$^+$13] Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, et al. The fourth answer set programming competition: Preliminary report. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, pages 42–53. Springer, 2013.

[AD16] Mario Alviano and Carmine Dodaro. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming*, 16(5-6):533–551, 2016.

[ADJ$^+$18] Mario Alviano, Carmine Dodaro, Matti Järvisalo, Marco Maratea, and Alessandro Previti. Cautious reasoning in ASP via minimal models and unsatisfiable cores. *Theory and Practice of Logic Programming*, 18(3-4):319–336, 2018.

[AF88] John S. Anderson and Arthur M. Farley. Plan abstraction based on operator generalization. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)*, pages 100–104, 1988.

[AKMS12] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming,* *ICLP 2012*, volume 17, pages 211–221. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[APG09] Alexandre Albore, Héctor Palacios, and Héctor Geffner. A translation-based approach to contingent planning. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[Bal96] José L. Balcázar. The complexity of searching implicit graphs. *Artificial Intelligence*, 86(1):171–188, 1996.

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[BCRT06] Piergiorgio Bertoli, Alessandro Cimatti, Marco Riveri, and Paolo Traverso. Strong planning under partial observability. *Artificial Intelligence*, 170(4):337–384, 2006.

[BD97] Stefan Brass and Jürgen Dix. Characterizations of the disjunctive stable semantics by partial evaluation. *The Journal of Logic Programming*, 32(3):207–228, 1997.

[BDGGR17] Blai Bonet, Giuseppe De Giacomo, Hector Geffner, and Sasha Rubin. Generalized planning: Non-deterministic abstractions and trajectory constraints. In *Proceedings of the 26th International Joint conference on Artificial intelligence (IJCAI 2017)*, pages 873–879, 2017.

[BDGL17] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Abstraction in situation calculus action theories. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI 2017)*, pages 1048–1055, 2017.

[BDGL18] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Abstraction of agents executing online and their abilities in the situation calculus. In *Proceedings of the 27th International Joint conference on Artificial intelligence (IJCAI 2018)*, pages 1699–1706, 2018.

[BEBN08] Chitta Baral, Thomas Eiter, Marcus Bjäreland, and Mutsumi Nakamura. Maintenance goals of agents in a dynamic environment: Formulation and policy construction. *Artificial Intelligence*, 172(12):1429–1469, 2008.

[BET11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BF97] Avrim L Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[BFVW04] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. State-space reduction techniques in agent verification. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004) - Volume 2*, pages 896–903. IEEE Computer Society, 2004.

[BFVW06] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous agents and multi-agent systems*, 12(2):239–256, 2006.

[BG00a] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence*, pages 257–279. Springer, 2000.

[BG00b] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems*, pages 52–61. AAAI Press, 2000.

[BG15] Blai Bonet and Hector Geffner. Policies that generalize: Solving many planning problems with the same policy. In *Proceedings of the 24th International Joint conference on Artificial intelligence (IJCAI 2015)*. AAAI Press, 2015.

[BGP$^+$07] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging asp programs by means of asp. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, pages 31–43. Springer, 2007.

[BJ95] Christer Backstrom and Peter Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proceedings of the 14th International Joint conference on Artificial intelligence (IJCAI 1995) - Volume 2*, pages 1599–1604. Morgan Kaufmann Publishers Inc., 1995.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BKS06] Daniel Bryce, Subbarao Kambhampati, and David E Smith. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, 26:35–99, 2006.

[BKT93] Samuel Buss, Jan Krajìček, and Gaisi Takeuti. On provably total functions in bounded arithmetic theories. In Peter Clote and Jan Krajìček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 116–61. Oxford University Press, 1993.

[BKT00] Chitta Baral, Vladik Kreinovich, and Raúl Trejo. Computational complexity of planning and approximate planning in the presence of incompleteness. *Artificial Intelligence*, 122(1-2):241–267, 2000.

[BLM16] Francesco Belardinelli, Alessio Lomuscio, and Jakub Michaliszyn. Agent-based refinement for predicate abstraction of multi-agent systems. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, pages 286–294. IOS Press, 2016.

[BMT11] Gerhard Brewka, Victor W. Marek, and Mirosław Truszczyński, editors. *Nonmonotonic reasoning: essays celebrating its 30th anniversary*. College Publ., 2011.

[BMW16] Manuel Bichler, Michael Morak, and Stefan Woltran. The power of non-ground rules in answer set programming. *Theory and Practice of Logic Programming*, 16(5-6):552–569, 2016.

[Bon10] Blai Bonet. Conformant plans and beyond: Principles and complexity. *Artificial Intelligence*, 174(3-4):245–269, 2010.

[Bra87] Michael Bratman. *Intention, plans, and practical reason*, volume 10. Harvard University Press Cambridge, MA, 1987.

[BY94] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[BZ13] Franz Baader and Benjamin Zarrieß. Verification of Golog programs over description logic actions. *Frontiers of Combining Systems*, pages 181–196, 2013.

[CDGMP13] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. Verification and synthesis in description logic based dynamic systems. In *Web Reasoning and Rule Systems*, pages 50–64. Springer, 2013.

[CDLR09] Mika Cohen, Mads Dam, Alessio Lomuscio, and Francesco Russo. Abstraction in model checking multi-agent systems. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 945–952. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[CFF14] Pedro Cabalar, Jorge Fandinno, and Michael Fink. Causal graph justifications of logic programs. *Theory and Practice of Logic Programming (TC)*, 14(4-5):603–618, 2014.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 1512–1542, 1994.

[CGV16] Federico Cerutti, Massimiliano Giacomin, and Mauro Vallati. Generating structured argumentation frameworks: AFBenchGen2. In Pietro Baroni, Thomas F. Gordon, Tatjana Scheffler, and Manfred Stede, editors, *Proceedings of the 6th International Conference on Computational Models of Argument (COMMA 2016)*, volume 287 of *Frontiers in Artificial Intelligence and Applications*, pages 467–468. IOS Press, 2016.

[CHVB18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking.* Springer, 2018.

[CIR+11] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, pages 388–403, 2011.

[CL90] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[CL08] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 589–599, 2008.

[Cla78] Keith L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[CR96] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

[CR00] Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

[Cra52] Kenneth James Williams Craik. *The nature of explanation*, volume 445. CUP Archive, 1952.

[CRB04] Alessandro Cimatti, Marco Roveri, and Piergiorgio Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206, 2004.

[CS98] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[DBBD16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. On local domain symmetry for model expansion. *Theory and Practice of Logic Programming*, 16(5-6):636–652, 2016.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001.

[Del17] James P. Delgrande. A knowledge level account of forgetting. *Journal of Artificial Intelligence Research*, 60:1165–1213, 2017.

[DFWB12] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.

[DGLP12] Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi. Bounded situation calculus action theories and decidable verification. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*. AAAI Press, 2012.

[DGLPS16] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Sebastian Sardina. Verifying ConGolog programs on bounded situation calculus theories. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, pages 950–956, 2016.

[DGM$^+$15] Carmine Dodaro, Philip Gasteiger, Benjamin Musitsch, Francesco Ricca, and Kostyantyn Shchekotykhin. Interactive debugging of non-ground asp programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, pages 279–293. Springer, 2015.

[DGTR97] Giuseppe De Giacomo, Eugenia Ternovskaia, and Ray Reiter. Non-terminating processes in the situation calculus. In *Working Notes of*

"Robots, Softbots, Immobots: Theories of Action, Planning and Control", *AAAI'97 Workshop*, 1997.

[DKN03] Jürgen Dix, Ugur Kuter, and Dana Nau. Planning in answer set programming using ordered task decomposition. In *Annual Conference on Artificial Intelligence*, pages 490–504. Springer, 2003.

[DTW11] Christian Drescher, Oana Tifrea, and Toby Walsh. Symmetry-breaking answer set solving. *AI Commun.*, 24(2):177–194, 2011.

[Dun95] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, 77(2):321–357, 1995.

[DVB$^+$09] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, pages 637–654, 2009.

[DW15] James P. Delgrande and Kewen Wang. A syntax-independent approach to forgetting in disjunctive logic programs. In *PProceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1482–1488, 2015.

[Ede01] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning (ECP 2001)*, pages 13–24, 2001.

[EF03] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In *International Conference on Logic Programming*, pages 224–238. Springer, 2003.

[EFFW07] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123, 2007.

[EFL$^+$03] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\text{k}}$ system. *Artificial Intelligence*, 144(1):157–211, 2003.

[EFL$^+$04] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.

[EFTW04] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir

Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, pages 87–99. Springer, 2004.

[EGG96] Thomas Eiter, Georg Gottlob, and Yuri Gurevich. Normal forms for second-order logic over finite structures, and classification of NP optimization problems. *Ann. Pure Appl. Logic*, 78(1-3):111–125, 1996.

[EGL16] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[EKI18] Thomas Eiter and Gabriele Kern-Isberner. A brief survey on forgetting from a knowledge representation and reasoning perspective. *KI – Künstliche Intelligenz*, 33(1):9–33, 2018.

[ELM$^+$98] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system dlv: Progress report, comparisons and benchmarks. *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR 1998)*, pages 406–417, 1998.

[ESS19] Thomas Eiter, Zeynep G. Saribatur, and Peter Schüller. Abstraction for zooming-in to unsolvability reasons of grid-cell problems. In *Proceedings of the Workshop on Explainable Artificial Intelligence (XAI)*, 2019. To appear.

[EW08] Thomas Eiter and Kewen Wang. Semantic forgetting in answer set programming. *Artificial Intelligence*, 172(14):1644–1672, 2008.

[FL99] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the 16th International Joint conference on Artificial intelligence (IJCAI 1999)*, volume 99, pages 956–961, 1999.

[FLP04] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2004.

[FS19] Jorge Fandinno and Claudia Schulz. Answering the "why" in answer set programming - A survey of explanation approaches. *Theory and Practice of Logic Programming*, 19(2):114–203, 2019.

[GK14] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach.* Cambridge University Press, 2014.

[GKK$^+$08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental ASP solver.

In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, pages 190–205, 2008.

[GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[GKL16] Ricardo Gonçalves, Matthias Knorr, and Joao Leite. The ultimate guide to forgetting in answer set programming. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR 2016)*, pages 135–144. AAAI Press, 2016.

[GKLW17] Ricardo Gonçalves, Matthias Knorr, João Leite, and Stefan Woltran. When you must forget: Beyond strong persistence when forgetting in answer set programming. *Theory and Practice of Logic Programming*, 17(5-6):837–854, 2017.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, pages 260–265. Springer, 2007.

[GKNS08] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Advanced preprocessing for answer set solving. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, volume 178, pages 15–19. IOS Press, 2008.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, volume 88, pages 1070–1080, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3):365–385, 1991.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2):301–321, 1993.

[GL98a] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.

[GL98b] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)*, pages 623–630, 1998.

[GLM04] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Sat-based answer set programming. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 61–66, 2004.

[GLMW16] Sarah Alice Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. Introducing the second international competition on computational models of argumentation. In *Proceedings of the International Workshop on Systems and Algorithms for Formal Argumentation (SAFA)*, pages 4–9, 2016.

[GLV99] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a source of complexity in logical formalisms. *Ann. Pure Appl. Logic*, 97(1-3):231–260, 1999.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[GPST08] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, volume 8, pages 448–453, 2008.

[GS88] Matthew L. Ginsberg and David E. Smith. Reasoning about action i: A possible worlds approach. *Artificial Intelligence*, 35(2):165–195, 1988.

[GW92] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[HB05] Jörg Hoffmann and Ronen Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS*, volume 2005, 2005.

[HB06] Jörg Hoffmann and Ronen I Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7):507–541, 2006.

[HH99] István T. Hernádvölgyi and Robert C. Holte. Psvn: A vector representation for production systems, 1999.

[HHH$^+$07] Malte Helmert, Patrik Haslum, Jörg Hoffmann, et al. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 176–183, 2007.

[HJ99] Patrik Haslum and Peter Jonsson. Some results on the complexity of planning with incomplete information. In *European Conference on Planning*, pages 308–318. Springer, 1999.

[Hob90] Jerry R. Hobbs. Granularity. In *Readings in qualitative reasoning about physical systems*, pages 542–545. Elsevier, 1990.

[HSD06] Jörg Hoffmann, Ashish Sabharwal, and Carmel Domshlak. Friends or Foes? an AI planning perspective on abstraction and search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 294–303, 2006.

[IM16] Leon Illanes and Sheila A. McIlraith. Numeric planning via search space abstraction. In *Proceedings of the Workshop on Knowledge-based Techniques for Problem Solving and Reasoning*, 2016.

[IM19] León Illanes and Sheila A. McIlraith. Generalized planning via abstraction: Arbitrary numbers of objects. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 2019.

[JL83] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness.* Number 6. Harvard University Press, 1983.

[JM16] Mikolás Janota and Joao Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artificial Intelligence*, 233:73–83, 2016.

[JNS$^+$06] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, 2006.

[KD86] Subbarao Kambhampati and Larry Davis. Multiresolution path planning for mobile robots. *IEEE Journal on Robotics and Automation*, 2(3):135–145, 1986.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[Kno94] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[Kra07] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[Lee05] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of the 19th International Joint conference on Artificial intelligence (IJCAI 2005)*, volume 5, pages 503–508, 2005.

[Lei17] João Leite. A bird's-eye view of forgetting in answer-set programming. In Marcello Balduccini and Tomi Janhunen, editors, *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *Lecture Notes in Computer Science*, pages 10–22. Springer, 2017.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, Saddek Bensalem, and David Probst. Property preserving abstractions for the verification of concurrent systems. *Formal methods in system design*, 6(1):11–44, 1995.

[Lif99a] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer, 1999.

[Lif99b] Vladimir Lifschitz. Answer set planning. In *Proceedings of the 1999 International Conference on Logic Programming (ICLP 1999)*, pages 23–37, 1999.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[Lif08a] Vladimir Lifschitz. Twelve definitions of a stable model. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, pages 37–51. Springer Berlin Heidelberg, 2008.

[Lif08b] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597, 2008.

[Llo87] John W. Lloyd. *Foundations of logic programming, Second Edition*. Springer, 1987.

[LLS19] Vladimir Lifschitz, Patrick Lühne, and Torsten Schaub. Verifying strong equivalence of programs in the input language of gringo. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, pages 270–283. Springer, 2019.

[LM14] Alessio Lomuscio and Jakub Michaliszyn. An abstraction technique for the verification of multi-agent systems against atl specifications. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR 2014)*. AAAI Press, 2014.

[LM16] Alessio Lomuscio and Jakub Michaliszyn. Verification of multi-agent systems via predicate abstraction against ATLK specifications. In *Proceedings of AAMAS*, pages 662–670, 2016.

[LPF$^+$06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[LPV01] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, October 2001.

[LRL$^+$97] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[LRS97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and computation*, 135(2):69–112, 1997.

[LS04] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[LT99] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. *Logic Programming and Nonmonotonic Reasoning*, pages 92–106, 1999.

[LTT99] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

[LZ04] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[Mah86] M. J. Maher. Equivalences of logic programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, pages 410–424. Springer Berlin Heidelberg, 1986.

[McC59] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91. MIT Press, 1959.

[McC81a] John McCarthy. Circumscription—a form of non-monotonic reasoning. In *Readings in Artificial Intelligence*, pages 466–472. Elsevier, 1981.

[McC81b] John McCarthy. Epistemological problems of artificial intelligence. In *Readings in artificial intelligence*, pages 459–465. Elsevier, 1981.

[McC86] John McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28(1):89–116, 1986.

[McC99] Norman McCain. The causal calculator, 1999.

[McD82] Drew McDermott. Nonmonotonic logic II: Nonmonotonic modal theories. *Journal of the ACM*, 29(1):33–57, 1982.

[MD80] Drew McDermott and Jon Doyle. Non-monotonic logic I. *Artificial Intelligence*, 13(1-2):41–72, 1980.

[MH69] John McCarthy and Patrick Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University USA, 1969.

[MLL16] Peiming Mo, Naiqi Li, and Yongmei Liu. Automatic verification of golog programs via predicate abstraction. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, pages 760–768. IOS Press, 2016.

[Moo85] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[MT98] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR 1998)*, pages 212–223. Morgan Kaufmann, 1998.

[MW12] Michael Morak and Stefan Woltran. Preprocessing of complex non-ground rules in answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming*, page 247, 2012.

[NDK97] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *European Conference on Planning*, pages 338–350. Springer, 1997.

[NL95] P. Pandurang Nayak and Alon Y. Levy. A semantic theory of abstractions. In *Proceedings of the 14th International Joint conference on Artificial intelligence (IJCAI 1995)*, pages 196–203, 1995.

[NS72] Allen Newell and Herbert A. Simon. *Human problem solving*, volume 104. Prentice-Hall Englewood Cliffs, NJ, 1972.

[OJ06] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, volume 6, pages 412–416. IOS Press, 2006.

[ONA02] Mauricio Osorio, Juan A. Navarro, and José Arrazola. Equivalence in answer set programming. In Alberto Pettorossi, editor, *Logic Based Program Synthesis and Transformation*, pages 57–75. Springer Berlin Heidelberg, 2002.

[OPT10] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10(4-6):513–529, 2010.

[Pap03] Christos H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[Pea04] David Pearce. Simplifying logic programs under answer set semantics. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming*, pages 210–224, 2004.

[PG09] Hector Palacios and Hector Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35:623–675, 2009.

[Prz90] Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13(4):445–463, 1990.

[PSE09] Enrico Pontelli, Tran Cao Son, and Omar Elkhatib. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56, 2009.

[RDBF16] Pat Riddle, Jordan Douglas, Mike Barley, and Santiago Franco. Improving performance by reformulating PDDL into a bagged representation. In *Proceedings of the Workshop on Heuristics and Search for Domain-independent Planning (HSDIP 2016)*, pages 28–36, 2016.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 1991)*, pages 473–484, 1991.

[RG$^+$95] Anand S Rao, Michael P Georgeff, et al. Bdi agents: from theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[Rin99] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[Sac74] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Sag87] Y. Sagiv. Optimizing datalog programs. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 349–362, New York, NY, USA, 1987. ACM.

[SB01] Tran Cao Son and Chitta Baral. Formalizing sensing actions – a transition function based approach. *Artificial Intelligence*, 125(1):19–91, 2001.

[SBE17] Zeynep G. Saribatur, Chitta Baral, and Thomas Eiter. Reactive maintenance policies over equalized states in dynamic environments. In *Proceedings of EPIA*, pages 709–723, 2017.

[SE16a] Zeynep G. Saribatur and Thomas Eiter. Reactive policies with planning for action languages. In Loizos Michael and Antonis Kakas, editors, *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA 2016)*, volume 10021 of *Lecture Notes in Computer Science*, pages 463–480. Springer, 2016.

[SE16b] Zeynep G. Saribatur and Thomas Eiter. Reactive policies with planning for action languages. In Gabriele Kern-Isberner and Renata Wassermann, editors, *Proceedings of the 16th International Workshop on Non-Monotonic Reasoning (NMR 2016)*, pages 143–152. TU Dortmund, 2016. Tech.Rep. in CS, 852.

[SE18a] Zeynep G. Saribatur and Thomas Eiter. Omission-based abstraction for answer set programs. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018)*, pages 42–51. AAAI Press, 2018.

[SE18b] Zeynep G. Saribatur and Thomas Eiter. Omission-based abstraction for answer set programs. In *LOGCOMP Research Report 18-06*, 2018.

[SE18c] Zeynep G. Saribatur and Thomas Eiter. Towards abstraction in asp with an application on reasoning about agent policies. In *Proceedings of the 12th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*, 2018.

[SH13] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 2013.

[SIZ11] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2):615–647, 2011.

[SN01] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, pages 434–438, 2001.

[SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[SSE19] Zeynep G. Saribatur, Peter Schüller, and Thomas Eiter. Abstraction for non-ground answer set programs. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*, Lecture Notes in Computer Science, pages 576–592. Springer, 2019.

[ST13] Claudia Schulz and Francesca Toni. ABA-based answer set justification. *Theory and Practice of Logic Programming (TC)*, pages 4–5, 2013.

[STB04] Tran Cao Son, Phan Huy Tu, and Chitta Baral. Planning with sensing actions and incomplete information using logic programming. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, pages 261–274. Springer, 2004.

[STGM05] Tran Cao Son, Phan Huy Tu, Michael Gelfond, and A Ricardo Morales. Conformant planning for domains with constraints-a new approach. In *AAAI*, volume 5, pages 1211–1216, 2005.

[SW98] David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)*, pages 889–896, 1998.

[Syr06] Tommi Syrjänen. Debugging inconsistent answer set programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR 2006)*, volume 6, pages 77–83, 2006.

[SZ13] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in artificial intelligence and complex systems*, volume 456. Springer, 2013.

[TSGM11] Phan Huy Tu, Tran Cao Son, Michael Gelfond, and A Ricardo Morales. Approximation of action theories and its application to conformant planning. *Artificial Intelligence*, 175(1):79–119, 2011.

[Tur02] Hudson Turner. Polynomial-length planning spans the polynomial hierarchy. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA 2002)*, volume 2424 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 2002.

[Tur03] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.

[TWF19] Richard Taupe, Antonius Weinzierl, and Gerhard Friedrich. Degrees of laziness in grounding. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, pages 298–311. Springer, 2019.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991.

[VHLP08] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*, volume 1. Elsevier, 2008.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of “small-world” networks. *Nature*, 393:440–442, 1998.

[ZC16] Benjamin Zarrieß and Jens Claßen. Decidable verification of Golog programs over non-local effect actions. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, pages 1109–1115, 2016.

[ZYKS15] Shiqi Zhang, Fangkai Yang, Piyush Khandelwal, and Peter Stone. Mobile robot planning using action language BC with an abstraction hierarchy. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, pages 502–516. Springer, 2015.

APPENDIX A

Encodings

In this appendix, we provide details about the slight modifications made from the existing (or common) encodings, in order to use them in our experiments. The full encodings can be found in `http://www.kr.tuwien.ac.at/research/systems/abstraction/`.

Disjunctive scheduling The problem is of the following form: given tasks I with fixed duration D ($task(I, D)$), earliest start time S ($est(I, S)$), latest end time E ($let(I, E)$), and disjunctive constraints ($disj(I, I')$) for tasks that cannot overlap, assign to each task a start time such that all constraints are satisfied. The encoding from ASPCOMP2011[1] contains many arithmetic expressions in order to compute the end times of the tasks when starting at certain time. The results of these computations are then compared with each other in order to ensure that there is no overlapping for the tasks that should remain disjoint:

$$\begin{aligned}\bot \leftarrow\ & disj(I_1, I_2), task(I_1, D_1), task(I_2, D_2), time(I_1, T_1), time(I_2, T_2),\\ & E_1 = T_1 + D_1, E_2 = T_2 + D_2, T_1 \leq T_2, T_2 < E_1. \qquad (A.1)\\ \bot \leftarrow\ & disj(I_1, I_2), task(I_1, D_1), task(I_2, D_2), time(I_1, T_1), time(I_2, T_2),\\ & E_1 = T_1 + D_1, E_2 = T_2 + D_2, T_2 \leq T_1, T_1 < E_2. \qquad (A.2)\end{aligned}$$

As DASPAR cannot handle such arithmetic expressions, we rewrote the above rules by introducing auxiliary atoms. The atom $aux\_endtime(I, T, E)$ means that the task I finishes at time E when it starts at time T and $aux\_starttime(I, T)$ shows that time T is a possible starting point of task I, i.e., if I starts at T then it will finish before the latest end time E given with $let(I, E)$.

$$aux\_endtime(I, T, E) \leftarrow task(I, D), times(T), E = T + D, times(E). \qquad (A.3)$$

[1]`https://www.mat.unical.it/aspcomp2011/files/DisjunctiveScheduling/disjunctive_scheduling.enc.asp`

$$aux\_starttime(I,T) \leftarrow aux\_endtime(I,T,E_1), let(I,E), E_1 \leq E. \qquad (A.4)$$

As the atoms $aux\_endtime(I,T,E)$ and $aux\_starttime(I,T)$ can be deterministically computed, for the evaluations in Section 5.2.2 we precompute these atoms for each instance and get rid of the rules (A.3)-(A.4), so that they are not treated in the abstraction process.

An time assignment of a task I is guessed over the possible starting points T that are later than the earliest start time S given with $est(I,S)$.

$$\{time(I,T)\} \leftarrow aux\_starttime(I,T), est(I,S), S \leq T.$$

The rules (A.1)-(A.2) are then represented with the following rules, by standardizing apart over the sort *times*.

$$\begin{aligned}
aux\_first(I_1,I_2,T,E_1) &\leftarrow disj(I_1,I_2), time(I_1,T_1), aux\_endtime(I_1,T_2,E_1), T_1 = T_2.\\
aux\_second(I_1,I_2,T,E_2) &\leftarrow disj(I_1,I_2), time(I_2,T_1), aux\_endtime(I_2,T_2,E_2), T_1 = T_2.\\
aux\_aux\_first(I_1,I_2,T_1) &\leftarrow aux\_first(I_1,I_2,T_1,E_1).\\
aux\_aux\_second(I_1,I_2,T_2) &\leftarrow aux\_second(I_1,I_2,T_2,E_2).\\
\bot &\leftarrow aux\_first(I_1,I_2,T_1,E1), aux\_aux\_second(I_1,I_2,T_2),\\
&\quad\; T_1 \leq T_2, T_2 < E_1.\\
\bot &\leftarrow aux\_aux\_first(I_1,I_2,T1), aux\_second(I_1,I_2,T_2,E_2),\\
&\quad\; T_2 \leq T_1, T_1 < E_2.
\end{aligned}$$

The constraint to ensure that each task is assigned a time point is the same as the original encoding.

Grid-Cell Problem Encodings

Sudoku We used the encoding from DLV group in ASPCOMP09 with slight modifications. The guessing of the assignment of numbers to the free cells is written as

$$\begin{aligned}
\{sol(X,Y,N) : num(N)\} &\leftarrow notoccupied(X,Y), row(X), column(Y).\\
hasNum(X,Y) &\leftarrow sol(X,Y,N).\\
\bot &\leftarrow not\ hasNum(X,Y), row(X), column(Y).
\end{aligned}$$

The constraints of assigning one symbol per column and one symbol per row are the same as in the original encoding, but with standardizing apart over the sorts *row* and *column*.

$$\begin{aligned}
\bot &\leftarrow sol(X,Y_1,M), sol(X_2,Y_2,M), X = X_2, Y_1 < Y_2.\\
\bot &\leftarrow sol(X_1,Y,M), sol(X_2,Y_2,M), X_1 < X_2, Y = Y_2.
\end{aligned}$$

For the constraint of assigning one symbol per subregion, standardizing apart the original rules caused to have relations with many argument, thus we converted them into the rules

$$\bot \leftarrow sol(X_1,Y_1,M), sol(X_2,Y_2,M),$$

$$
\begin{aligned}
& \mathit{sameSubSquareLessThan}(X_1, Y_1, X_2, Y_2). \\
\mathit{sameSubSquareLessThan}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{sameSubSquare}(X_1, Y_1, X_2, Y_2), X_1 < X_2. \\
\mathit{sameSubSquareLessThan}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{sameSubSquare}(X_1, Y_1, X_2, Y_2), Y_1 < Y_2. \\
\mathit{sameSubSquare}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{subrangeR}(X_1, M), \mathit{subrangeR}(X_2, M), \\
& \mathit{subrangeC}(Y_1, R), \mathit{subrangeC}(Y_2, R).
\end{aligned}
$$

with the hardcoded facts $\mathit{subrangeR}(X, M)$ and $\mathit{subrangeC}(Y, R)$ for subregions w.r.t. rows and columns, respectively.

Knight's Tour We used the encoding from ASPCOMP11 [2] with slight modifications. At most one *move* atom is made for each *valid* movement among the cells.

$$\{\mathit{move}(X_1, Y_1, X_2, Y_2)\}1 \leftarrow \mathit{valid}(X_1, Y_1, X_2, Y_2).$$

In the original encoding, the valid cells computations were done using rules of the form

$$\mathit{valid}(X_1, Y_1, X_2, Y_2) \leftarrow \mathit{point}(X_1, Y_1), \mathit{point}(X_2, Y_2), X_1 = X_2 + 2, Y_1 = Y_2 + 1.$$

which are modified as

$$
\begin{aligned}
\mathit{validcell}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{dist1}(X_1, X_2), \mathit{dist2}(Y_1, Y_2). \\
\mathit{validcell}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{dist2}(X_1, X_2), \mathit{dist1}(Y_1, Y_2). \\
\mathit{valid}(X_1, Y_1, X_2, Y_2) &\leftarrow \mathit{validcell}(X_1, Y_1, X_2, Y_2), \mathit{point}(X_1, Y_1), \mathit{point}(X_2, Y_2).
\end{aligned}
$$

where the auxiliary facts $\mathit{dist1}(X_1, X_2), \mathit{dist2}(X_1, X_2)$ represent the arithmetic operations $X_1 = X_2 + 2, Y_1 = Y_2 + 1$.

The constraints to ensure that exactly one entering/leaving movement is made for each cell is the same as the original encoding. Having each cell visited is ensured by the following rules

$$
\begin{aligned}
\mathit{reached}(X, Y) &\leftarrow \mathit{move}(X_1, Y_1, X, Y), \mathit{start}(X_1, Y_1). \\
\mathit{reached}(X_2, Y_2) &\leftarrow \mathit{reached}(X_1, Y_1), \mathit{move}(X_1, Y_1, X_2, Y_2). \\
\bot &\leftarrow \mathit{point}(X, Y), \mathit{not}\ \mathit{reached}(X, Y), \mathit{row}(X), \mathit{column}(Y).
\end{aligned}
$$

where the atom $\mathit{start}(X, Y)$ is used to show the starting point, instead of having in the rule the atom $\mathit{move}(1, 1, X, Y)$ as it is originally. This change makes treating the program more convenient, as the rules do not contain constants that need to mapped to different abstract constants depending on the mapping.

[2] https://www.mat.unical.it/aspcomp2011/files/KnightTour/knight_tour.enc.asp

Visitall We encoded a planning problem following the guidelines shown in Section 2.2.1 on representing actions and change. We considered $go(X, Y, T)$ actions that can move horizontally/vertically to a cell X, Y. For such an action, we have to ensure that the action does not pass through an obstacle or a previously visited cell, and that all the passed cells become visited.

A common way of encoding this is to have auxiliary atoms that keep track of the cells that are in between such as

$$\begin{aligned}
aux\_passed(X, Y_2, T+1) &\leftarrow rAt(X, Y, T), go(X, Y_1, T), Y < Y_2, Y_2 \leq Y_1.\\
aux\_passed(X, Y_2, T+1) &\leftarrow rAt(X, Y, T), go(X, Y_1, T), Y_1 < Y_2, Y_2 \leq Y.\\
aux\_passed(X_2, Y, T+1) &\leftarrow rAt(X, Y, T), go(X_1, Y, T), X < X_2, X_2 \leq X_1.\\
aux\_passed(X_2, Y, T+1) &\leftarrow rAt(X, Y, T), go(X_1, Y, T), X_1 < X_2, X_2 \leq X.\\
passed(X, Y) &\leftarrow aux\_passed(X, Y, T).
\end{aligned}$$

which are then used to ensure the above conditions.

$$\begin{aligned}
\perp &\leftarrow passed(X, Y), obsAt(X, Y).\\
visited(X, Y, T) &\leftarrow aux\_passed(X, Y, T).\\
\perp &\leftarrow aux\_passed(X, Y, T+1), visited(X, Y, T).
\end{aligned}$$

We follow the remark in Section 4.6 on handling different abstraction levels on variables in a rule. For example, for the first rule, in addition to the standardizing apart the rule as

$$aux\_passed(X, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y_1, T), X = X_1, Y < Y_2, Y_2 \leq Y_1.$$

we add the additional rule

$$aux\_passed(X_1, Y_2, T+1) \leftarrow rAt(X, Y, T), go(X_1, Y_1, T), X = X_1, Y < Y_2, Y_2 \leq Y_1.$$

Similar is done with the remaining rules.

Furthermore, knowing that the action $go(X_1, Y_1, T)$ will only be picked in a horizontal (resp. vertical) direction of $rAt(X, Y, T)$, we drop the additional relation $X = X_1$ (resp. $Y = Y_1$) from the rule body in order to reduce the amount of relations in a rule.

APPENDIX B

Further Computational Details

In this appendix, we take a look at further details of the computational complexity for omission abstraction.

We consider the computation of put-back sets, which is needed for the elimination of spurious answer sets. To describe the complexity, we use some complexity classes for search problems, which generalize decision problems in that for a given input, some (possibly different or none) output values (or solutions) might be computed. Specifically, $\mathbf{FP}^{\mathsf{NP}}$ consists of the search problems for which a solution can be computed in polynomial time with an NPoracle, and $\mathbf{FP}^{\mathsf{NP}}_{\parallel}$ is analogous but under the restriction that all oracle calls have to be made at once in parallel. The class $\mathbf{FP}^{\Sigma^P_k}[log, wit]$, for $k \geq 1$, contains all search problems that can be solved in polynomial time with a witness oracle for Σ^p_k [BKT93]; a *witness* oracle for Σ^p_k returns in case of a yes-answer to an instance a polynomial size witness string that can be checked with an Σ^p_{k-1} oracle in polynomial time. In particular, for $k = 1$, i.e., for $\mathbf{FP}^{\mathsf{NP}}[log, wit]$, one can use a SAT oracle and the witness is a satisfying assignment to a given SAT instance, cf. [JM16].

While an arbitrary put-back set $PB \subseteq A$ is can be trivially obtained (just set $PB = A$), computing a minimal put-back set is more involved. Specifically, we have:

Theorem B.1. *Given a program* Π*, a set* A *of atoms, and a spurious answer set* I *of* $omit(\Pi, A)$*, computing (i) some* $\subseteq$*-minimal put-back set* S *resp. (ii) some smallest size put-back set* S *for* I *is in case (i) feasible in* $\mathbf{FP}^{\mathsf{NP}}$ *and* $\mathbf{FP}^{\mathsf{NP}}_{\parallel}$*-hard resp. is in case (ii)* $\mathbf{FP}^{\Sigma^P_2}[log, wit]$*-complete.*

Note that few $\mathbf{FP}^{\Sigma^P_2}[log, wit]$-complete problems are known. The notions of hardness and completeness are here with respect to a natural polynomial-time reduction between two problems P_1 and P_2: there are polynomial-time functions f_1 and f_2 such that (i) for every instance x_1 of P_1, $x_2 = f_1(x_1)$ is an instance of P_2, such that x_2 has solutions iff x_1 has, and (ii) from every solution s_1 of x_2, some solution $s_1 = f_2(x_1, s_2)$ is obtainable.

Proof of Theorem B.1. As for membership in (i), we can compute such a set S by an elimination procedure as follows. Starting with $A' = \emptyset$, we repeatedly pick some atom $\alpha \in A \setminus A'$ and test the following condition:

(+) for $A'' = A' \cup \{\alpha\}$, the program $omit(\Pi, A'')$ has no answer set $\widehat{I''}$ such that $\widehat{I''}|_{\overline{A}} = \hat{I}$.

If (+) holds, we set $A' := A''$ and make the next pick from $A \setminus A'$. Upon termination, $S = A \setminus A'$ is a minimal put-back set. The correctness of this procedure follows from Proposition 4.8, by which the elimination of spurious answer sets is anti-monotonic in the set A of atoms to omit. As for the effort, the test (+) can be done in polynomial time with an NPoracle; from this, membership in in $\mathbf{FP}^{\mathsf{NP}}$ follows.

The hardness for $\mathbf{FP}^{\mathsf{NP}}_{\parallel}$ is shown by a reduction from computing, given normal logic programs $\Pi_1, \ldots, \Pi_n$ on disjoint sets $X_1, \ldots, X_n$ of atoms, the answers $q_1, \ldots, q_n$ to whether Π_i has some answer set ($q_i = 1$) or not ($q_i = 0$).

To this end, we use fresh atoms a_i and b_i and construct

$$\begin{array}{lll}
\Pi'_i = \{ & a_i \leftarrow not\, b_i & \\
 & b_i \leftarrow not\, a_i & \\
 & \bot \leftarrow not\, b_i & \\
 & H(r) \leftarrow B(r), a_i & r \in \Pi_i \\
 & y \leftarrow x, not\, x & x, y \in X_i \\
 & a_i \leftarrow x, not\, x & x \in X_i \\
 & b_i \leftarrow x, not\, x & x \in X_i \;\; \}
\end{array}$$

Clearly, $\{a_i\}$ is an answer set of $omit(\Pi', X_i \cup \{b_i\})$, as the rule $a_i \leftarrow not\, b_i$ is turned into a choice; it is spurious, as only this rule in Π can derive a_i. However, this violates the constraint $\bot \leftarrow not\, b_i$.

Assuming w.l.o.g. that Π_i includes no constraints, for every set PB of atoms such that $X_i \not\subseteq PB$, the program $omit(\Pi'_i, (X_i \cup \{b_i\}) \setminus PB)$ has some answer set containing a_i, thanks to the abstraction of the rules with $x, not\, x$ in the body; thus $PB = X_i$ is the minimal candidate for being a put-back set. Furthermore, if Π_i has no answer set, then $\emptyset$ is the single answer set of $omit(\Pi'_i, \{b_i\})$ while if Π_i has some answer set S, then $omit(\Pi'_i, \{b_i\})$ has the answer set $S \cup \{a_i\}$. That is, X_i is the (unique) $\subseteq$-minimal put-back set iff Π_i has no answer set.

We construct the final program as $\Pi' = \bigcup_{i=1}^{n} \Pi'_i$. Then, $\hat{I} = \{a_1, \ldots, a_n\}$ is a spurious answer set of $omit(\Pi', \bigcup_{i=1}^{n} X_i \cup \{b_i\})$, and every minimal put-back set PB for $\hat{I}$ satisfies $b_i \in PB$ iff Π_i is satisfiable; this proves $\mathbf{FP}^{\mathsf{NP}}_{\parallel}$-hardness.

$$\begin{aligned}
&x_i. \qquad \overline{x_i}. && i = 1\ldots,n && \text{(B.2)}\\
&sat \leftarrow x_i, not\, x_i, \overline{x_i}, not\, \overline{x_i}. && i = 1\ldots,n && \text{(B.3)}\\
&z_i \leftarrow not\, \overline{z_i}, x_i, not\, \overline{x_i}. && i = 1\ldots,n && \text{(B.4)}\\
&\overline{z_i} \leftarrow not\, z_i, \overline{x_i}, not\, x_i. && i = 1\ldots,n && \text{(B.5)}\\
&y_j \leftarrow not\, \overline{y_j}, not\, sat. && j = 1,\ldots,m && \text{(B.6)}\\
&\overline{y_j} \leftarrow not\, y_j, not\, sat. && j = 1,\ldots,m && \text{(B.7)}\\
&sat \leftarrow l^\circ_{i_1}, \ldots l^\circ_{i_{n_i}}. && i = 1,\ldots,k && \text{(B.8)}\\
&sat \leftarrow y_j, not\, y_j. && j = 1,\ldots,m && \text{(B.9)}\\
&sat \leftarrow \overline{y_j}, not\, \overline{y_j}. && j = 1,\ldots,m && \text{(B.10)}\\
&sat \leftarrow z_i, not\, z_i. && i = 1\ldots,n && \text{(B.11)}\\
&sat \leftarrow \overline{z_i}, not\, \overline{z_i}. && i = 1\ldots,n && \text{(B.12)}
\end{aligned}$$

Figure B.1: Program rules for the proof of Theorem B.1-(ii), first part

As for (ii), the membership in $\mathbf{FP}^{\Sigma_2^P}[log, wit]$ holds as we can decide the problem by a binary search for a put-back set of bounded size using a Σ_2^p witness oracle, where the finally obtained put-back set is output.

The $\mathbf{FP}^{\Sigma_2^P}[log, wit]$ hardness is shown by a reduction from the following problem. Given a QBF $\Phi = \exists X \forall Y\, E(X, Y)$, compute a smallest size truth assignment σ to X such that $\forall Y\, E(\sigma(X), Y)$ evaluates to true, knowing that some σ with this property exists, where the size of σ is the number of atoms set to true.

More specifically, we assume that $E(X,Y) = \bigvee_{i=1}^{k} D_i$ is a DNF where $D_i = l_{i_1} \wedge \cdots \wedge l_{i_{n_i}}$ is a conjunction of literals over $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ similar as in the proof of Theorem 4.12 that contains some literal over Y, moreover without loss of generality that $E(X, Y)$ is a tautology if all literals over X are removed from it. To verify the latter assumption, we may rewrite Φ to

$$\exists X \forall Y \bigvee_{x_i \in X} (x \wedge \neg x_i \wedge y_j) \vee (x \wedge \neg x_i \wedge \neg y_j) \vee E(X, Y), \qquad \text{(B.1)}$$

for an arbitrary $y_j \in Y$, which has the desired property.

We set up a program Π with rules shown in Figure B.1, where $\overline{X} = \{\overline{x}_i \mid x_i \in X\}$, $Z = \{z_1, \ldots, z_n\}$ and $\overline{Z} = \{\overline{z}_i \mid z_i \in Z\}$ are copies of X and $\overline{Y} = \{\overline{y}_j \mid y_j \in Y\}$ is a copy of Y, and l° maps a literal l over $X \cup Y$ to default literals over $Y \cup \overline{Y} \cup Z \cup \overline{Z}$ as follows:

$$l^\circ = \begin{cases} not\, z_i, & \text{if } l = \neg x_i, \\ not\, \overline{z}_i, & \text{if } l = x_i, \\ y_j, & \text{if } l = y_j, \\ \overline{y}_j & \text{if } l = \neg y_j. \end{cases}$$

We note that Π has no answer set: due to the facts x_i and $\overline{x}_i$, none of the rules (B.3)–(B.5) is applicable and $z_i, \overline{z}_i$ must be false in every answer set of Π. This in turn implies that in (B.8) all $not\, z_i$, $not\, \overline{z}_i$ literals are true. Now if we assume that *sat* would be true in an answer set of Π, then no rule in (B.6) or (B.7) would be applicable to derive y_j resp. $\overline{y}_j$, and then by the assumption on $E(X, Y)$ no rule (B.8) is applicable; this means that *sat* is not reproducible and thus not in the answer set, which is a contradiction. If on the other hand *sat* would be false in an answer set, then the rules (B.6) and (B.7) would guess a truth assignment to Y; by the tautology assumption on $E(X, Y)$, some rule (B.8) is applicable and derives that *sat* is true, which is again a contradiction.

We then set $A = \mathcal{A}$ and $\hat{I} = \emptyset$. Then, clearly $\hat{I} = \emptyset$ is trivially a spurious answer set of $omit(\Pi, A)$.

The idea behind this construction is as follows. As long as we do not put back *sat*, the abstraction program $omit(\Pi, L')$ will have some answer set. Furthermore, if we do not put back either x_i or $\overline{x_i}$, or both z_i and $\overline{z_i}$ and all y_j, $\overline{y_j}$, then we can guess by (B.3) resp. (B.9) – (B.12) that *sat* is true, which again means that some answer set exists. The rules (B.4) – (B.5) serve then to provide with z_i and $\overline{z_i}$ access to x_i and its negation $\neg x_i$, respectively.

The rules (B.6) – (B.7) serve to guess an assignment μ to Y (but this only works if *sat* is false). The rule (B.6) checks whether upon a combined assignment $\sigma \cup \mu$, the formula $E(\sigma(X), \mu(Y))$ evaluates to true; if this is the case, *sat* is concluded which then however blocks the guessing in (B.6) – (B.7). It holds that some putback set of size $k = |X| + 2|X| + 2|Y| + 1$, which is the smallest possible here, exists iff Φ evaluates to true. On the other hand, if we put back a single further atom, for some $x_i, \overline{x_i}$ is in the database, and thus by the special form of $E(X, Y)$ in (B.1) for $Y' = Y$ one can derive *sat* again. Thus the closest putback set has either size k or $k + 1$. In order to discriminate among different $\sigma(X)$ and select the smallest one, we add further rules:

$$sat \leftarrow not\, \overline{z_i}, c_i \tag{B.13}$$

$$sat \leftarrow not\, \overline{z_i}, not\, z_i, d_1, \ldots, d_l \tag{B.14}$$

where all c_i and d_j are fresh atoms. Intuitively, when x_i is put in, then $\neg z_i$ evaluates to true and c_i must be omitted in order to avoid guessing on *sat*. Furthermore, if both x_i and $\overline{x_i}$ are put back, then all $d_1, \ldots, d_n$ must be put back as well. If $\sigma(X)$ makes $\forall Y E(\sigma(X), Y)$ true, then the closest putback set has size $k + 1 + |\sigma|$; if we let l be large enough, then putting both x_i and $\overline{x_i}$ back is more expensive than putting a proper assignment, Overall, the desired smallest $\sigma(X)$ is obtained. □

We remark that the problem is solvable in polynomial time, if the smallest putback set S has a size bounded by a constant k. Indeed, in this case we can explore all S of that size, and find all answer sets $\widehat{I'}$ of $omit(\Pi, \overline{A \cup S})$ that coincide with I on $\overline{L}$ in polynomial time.

We finally consider the problem of computing some refinement-safe abstraction that does not remove a given set A_0 of atoms.

Theorem B.2. *Given a set $A_0 \subseteq \mathcal{A}$, computing (i) some $\subseteq$-maximal set $A \subseteq \mathcal{A} \setminus A_0$ resp. (ii) some $A \subseteq \mathcal{A} \setminus A_0$ of largest size such that $omit(\Pi, A)$ is a refinement-safe faithful abstraction is in case (i) in $\mathbf{FP}^{\mathsf{NP}}$ and $\mathbf{FP}^{\mathsf{NP}}_{\parallel}$-hard and in case (ii) $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-complete, with $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-hardness even if $A_0 = \emptyset$.*

Proof. (i) One sets $A := \emptyset$ and $S := \mathcal{A} \setminus A_0$ initially and then picks an atom α from S and sets $S := S \setminus \{\alpha\}$. One tests whether (*) omitting $A' \cup \{\alpha\}$, for every subset $A' \subseteq A$, is a faithful abstraction; if so, then one sets $A := A \cup \{\alpha\}$. Then a next atom α is picked from S etc. When this process terminates, we have a largest set A such that omitting A from Π is a faithful abstraction. Indeed, by construction the final set A fulfills that for each $A' \subseteq A$, $omit(\Pi, A')$ is faithful, and thus A is refinement-safe; furthermore A is maximal: if a larger set $A' \supset A$ would exist, then at the point when $\alpha \in A' \setminus A$ was considered in constructing A the test (*) would not have failed and $\alpha \in A$ would hold.

Notably, (*) can be tested with an NP oracle: the conditions fails iff for some A', the program $omit(\Pi, A' \cup \alpha)$ has a spurious answer set $\hat{I}$. In principle, the spurious check for $\hat{I}$ is difficult (a coNP-complete problem, by our results), but we can take advantage of knowing that $omit(\Pi, A')$ is faithful: so we only need to check whether an extension of $\hat{I}$ is an answer set of $omit(\Pi, A')$, and not of Π itself; i.e., we only need to check $\hat{I} \notin AS(omit(\Pi, A'))$ and $\hat{I} \cup \{\alpha\} \notin AS(omit(\Pi, A'))$.

(ii) The proof of $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-completeness is similar as above for Theorem B.1. First, we note that to decide whether some refinement-safe faithful $A \subseteq \mathcal{A} \setminus A_0$ of size $|A| \geq k$ exists is in Σ_2^p: a nondeterministic variant of the algorithm for item (i), that picks α always nondeterministically and finally checks that $|A| \geq k$ holds establishes this. We then can run a binary search, using a Σ_2^p witness oracle, to find a refinement-safe faithful abstraction A of largest size. This shows $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-membership.

As for the $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-hardness part, in the proof of $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-hardness for Theorem B.1-(ii) each put-back set PB for the spurious answer set $\hat{I} = \emptyset$ for $A = \emptyset$ satisfies $omit(\Pi, \mathcal{A} \setminus PB) = \emptyset$, and is thus by Proposition 4.9 refinement-safe faithful. As the smallest size PB sets correspond to the maximum size $A' = \mathcal{A} \setminus PB$ sets, the $\mathbf{FP}^{\Sigma_2^P}[log, wit]$-hardness follows, even for $A_0 = \emptyset$. $\square$

We remark that without refinement safety, the problem is likely to be more complex: deciding whether an abstraction is faithful is Π_2^p-complete, and this question is trivially reducible to this problem.