FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reasoning with Forest Logic Programs

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der technischen Wissenschaften

eingereicht von

Cristina Maria Feier
Matrikelnummer 0928183

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter

Diese Dissertation haben begutachtet:

(Prof. Dr. Thomas Eiter)	(Prof. Dr. Torsten Schaub)

Wien, 15.08.2014

(Cristina Maria Feier)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Reasoning with Forest Logic Programs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der technischen Wissenschaften

by

Cristina Maria Feier

Registration Number 0928183

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. Dr. Thomas Eiter

The dissertation has been reviewed by:

(Prof. Dr. Thomas Eiter) (Prof. Dr. Torsten Schaub)

Wien, 15.08.2014

(Cristina Maria Feier)

Erklärung zur Verfassung der Arbeit

Cristina Maria Feier

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

(Ort, Datum)	(Unterschrift Cristina Maria Feier)

Acknowledgements

I would like to start by thanking the two persons who were the most influential as concerns the development of this thesis: Prof. Thomas Eiter and Dr. Stijn Heymans. Stijn introduced me to the topic and also gave me lots of helpful advice in the first part of my PhD journey. I am deeply indebted to Thomas who supported and guided my work after Stijn left to US. His professional advice was very helpful in shaping up this thesis and his integrity and calm were and are always an inspiration. I would also like to thank Thomas for the careful proofreading of the thesis and the extremely detailed and useful comments.

To my mother who had infinite patience with me even when I could not be beside her when she needed me most. And to my father who unfortunately is no longer here with us, but whose smile still shines through. His death, while unexpected and tragic, also served as a reminder that life is there for the taking and motivated me to go further.

Many thanks go to my colleagues and friends: Katrin, Guohui, Michael, Minh, TK, Sebastian, Peter, Patrik, Joerg, Magdalena, Dasha, Mantas, whose pleasant and warm company I enjoyed many times. Special thanks go to Eva and Matthias who were always very helpful with solving various administrative and technical issues. My thoughts go also to my out-of-work friends Mihai, Edi, Franka, Alina,... Thank you guys for your friendship and time! And to the Vienna flickr group who gave me the opportunity to immerse a bit in the local Viennese scene and practice my shabby German; I regret I did not have time to join more of your photo walks, but who knows, we might still meet sometime in the future...

Special thanks go to my former colleague and friend Uwe Keller who selflessly helped me edit and translate the abstract of a project proposal which eventually funded part of my research. And of course to the EU Comission and the FWF funding agency who funded my work.

Finally, these acknowledgements would not be complete without mentioning ic hir; while our world views were sometimes dissonant, he helped me grow, get to know myself better, and actively encouraged me to pursue my goals. He also helped with the German translation of the abstract of the thesis which was then further refined by Thomas.

Abstract

Answer Set Programming (ASP) is a popular rule-based formalism for describing and solving combinatorial search problems. Open Answer Set Programming (OASP) extends Answer Set Programming with an open domain semantics: programs are interpreted with respect to arbitrary domains which might contain anonymous individuals, i.e. individuals which do not occur explicitly in the program. This makes it possible to state generic knowledge using OASP. At the same time, OASP inherits from ASP the non-monotonic treatment of negation under the stable model semantics. As such, OASP bridges two important knowledge representation paradigms: the classical First Order Logic (FOL) world and the non-monotonic rules world.

In the general case, OASP is undecidable. With the purpose of achieving decidability, several fragments have been defined by means of syntactical restrictions on the shape of rules. Some such fragments are Conceptual Logic Programs (CoLPs) – a fragment which disallows the occurrence of constants and which exhibits the tree model property –, and Forest Logic Programs (FoLPs) – a fragment which has the forest model property. CoLPs can simulate reasoning in the Description Logic $\mathcal{SHQ}$ and they were proved to be decidable by a reduction to two-way alternating tree automata with a parity condition. FoLPs can simulate reasoning in the Description Logic $\mathcal{SHOQ}$ and their decidability status was not known at the beginning of this work. Decidability had been previously established only for the fragments of acyclic and local FoLPs in which the usage of recursion is heavily restricted.

Hence the reasoning support for CoLPs and FoLPs is rather limited. The forest model property of FoLPs makes them a good candidate for being tackled by means of tableau-based procedures which construct forest shaped models. Unlike reductions to automata, goal-oriented procedures like tableau algorithms, are usually not worst-case optimal, but are amenable to optimizations and they behave well in practice. As such the objective of this thesis is devising **tableau-based algorithms to reason with FoLPs**. We choose to focus on FoLPs as they are quite an expressive fragment which also subsume the fragment of CoLPs.

One of the main challenges when designing tableau algorithms for FoLPs consists in ensuring that every atom in a constructed model is well-supported, i.e. there is a reason for the presence of the atom in the model and this reason is not circular or infinite. Standard blocking mechanisms employed by tableau algorithms enhanced with such well-supportedness checks are no longer enough to guarantee termination. As such, our algorithms rely heavily on techniques to achieve the finite model property for FoLPs which are then transposed in runtime termination conditions. Based on two such techniques, we devise algorithms for satisfiability checking of unary predicates with respect to FoLPs which run in the worst-case in non-deterministic

double and non-deterministic single exponential time, respectively. The second class of algorithms, while exhibiting a better worst-case runtime behaviour, requires extensive bookkeeping. We identify a new fragment of FoLPs, called *simple Forest Logic Programs*, which generalizes acyclic FoLPs and for which a straightforward non-deterministic algorithm can be devised.

The above-mentioned algorithms show that satisfiability checking of unary predicates with respect to FoLPs can be decided in NEXPTIME. From the fact that FoLPs are able to simulate reasoning within the DL $\mathcal{SHOQ}$, we know that reasoning with respect to FoLPs is EXPTIME-hard. In a quest to obtain worst-case optimal algorithms, we investigate the determinisation of the above-mentioned algorithms by means of AND/OR data structures. We obtain two worst-case optimal algorithms to reason with the fragments of CoLPs and simple FoLPs, respectively.

Kurzfassung

Answer Set Programming (ASP) ist ein beliebter, regelbasierter Formalismus zur Beschreibung und Lösung von kombinatorischen Suchproblemen. Open Answer Set Programming (OASP) erweitert Answer Set Programming um eine Semantik mit offenen Domänen: Programme erhalten eine Interpretation in beliebigen Dömanen, die anonyme Individuen enthalten kann, d.h. Individuen, die nicht explizit im Programm aufscheinen. Dies ermöglicht es mittels OASP generisches Wissen zu formulieren. Gleichzeitig erbt OASP von ASP die nicht-monotone Behandlung der Negation unter der stabilen Modell-Semantik (Stable Model Semantics). OASP verbindet auf seine Art zwei wichtige Paradigmen der Wissensrepräsentation: die klassische Prädikatenlogik erster Stufe (First-Order Logic, FOL) und die Welt der nicht-monotonen Regeln.

Im allgemeinen Fall ist OASP unentscheidbar. Um Entscheidbarkeit zu erreichen wurden deshalb verschiedene Fragmente von OASP durch deshalb syntaktische Einschränkungen der Form der Regeln definiert. Beispiele solcher Fragmente sind Conceptual Logic Programs (CoLPs) – ein Fragment, das die Verwendung von Konstanten verbietet und die Tree-Modell Eigenschaft erfüllt –, und Forest Logic Programs (FoLPs) – ein Fragment mit der Forest-Modell Eigenschaft. CoLPs können Schließen (engl. Reasoning) in der Beschreibungslogik (engl. Description Logic) $\mathcal{SHQ}$ simulieren und es wurde durch eine Reduktion auf two-way alternating tree automata gezeigt, dass sie entscheidbar sind. FoLPs hingegen können Reasoning in der Beschreibungslogik $\mathcal{SHOQ}$ simulieren, aber es war vor dieser Arbeit nicht bekannt, ob sie entscheidbar sind oder nicht. Entscheidbarkeit war vorher nur für Fragmente von azyklischen und lokalen FOLPs bewiesen, in denen die Verwendung von Rekursion sehr stark eingeschränkt ist. Deswegen ist der Reasoning-Support für CoLPs und FoLPs ziemlich begrenzt.

Die Forest-Modell Eigenschaft von FOLPs eröffnet die Möglichkeit, sie mit tableau-basierten Methoden zu untersuchen, die Modelle mit Forest Form erzeugen. Im Gegensatz zur Reduktion auf Automaten, sind zielorientierte Prozeduren wie Tableau-Algorithmen nicht worst-case optimal, aber sie eignen sich für Optimierungen und verhalten sich in der Praxis gut. Daher ist das Ziel der Arbeit, tableau-basierte Algorithmen für das Reasoning mit FoLPs zu entwickeln. Die Wahl von FoLPs erfolgte, weil sie ein sehr ausdrucksstarkes Fragment von OASP sind, das auch CoLPs umfasst.

Eine der größten Herausforderungen bei der Entwicklung von Tableau-Algorithmen für FoLPs besteht darin, für jedes Atom im konstruierten Modell sicher zu stellen, dass es „well-supported“ ist, d.h. dass es eine Begründung für das Atom im Modell gibt, die weder zirkulär noch unendlich ist. Die Erweiterung des Standard Blocking, wie es von Tableau Algorithmen angewandt wird, um die Überprüfung von Well-Supportedness ist aber nicht ausreichend, um die Termination des Verfahrens zu garantieren. Deswegen stützen sich unsere Algorithmen stark auf Techniken mit denen für FoLPs die Finite-Modell Eigenschaft erreichtwerden kann; diese werden dann in Laufzeit-Endbedingungen umgewandelt.

Basierend auf zwei solche Techniken entwickeln wir Algorithmen für die Überprüfung der Erfüllbarkeit (engl. Satisfiability) für unäre Prädikate bezüglich FoLPs, die im schlechtesten Fall nichtdeterministisch doppelt exponentielle bzw. nichtdeterministisch einfach exponentielle Laufzeit haben. Während die zweite Klasse von Algorithmen ein besseres Laufzeitverhalten im schlechtesten Fall aufweist, erfordert sie ausführliche Aufzeichnungen (engl. Bookkeeping). Wir identifizieren ein neues Fragment von FoLPs, *simple Forest Logic Programs*, das azyklische FoLPs verallgemeinert und für die man einen unkomplizierten nicht-deterministischen Algorithmus entwickeln kann. Die oben erwähnten Algorithmen zeigen, dass Satisfiability-Testen von unären Prädikaten in FoLPs in nichtdeterministische einfach exponentieller Zeit entschieden werden kann. Aufgrund der Tatsache, dass FoLPs Reasoning in der Description Logic $\mathcal{SHOQ}$ simulieren können, wissen wir, dass Reasoning in FolPs EXPTIME hart ist.

Auf der Suche nach worst-case optimalen Algorithmen untersuchen wir die Determinisierung der oben erwähnten Algorithmen mit Hilfe von AND/OR Datenstrukturen. Wir erhalten zwei worst-case optimale Algorithmen für das Reasoning mit Fragmenten von CoLPs und einfachen FoLPs.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The Semantic Web envisions to increase automation of knowledge-related tasks on the Web by making knowledge more formal [Berners-Lee et al., 2001]. Motivated by this vision there has been a lot of interest in logic-based languages which have a clear well-understood semantics and desirable representational properties. Two main classes of languages that have been identified as suitable for this purpose are the so-called *ontological* languages, which are logically grounded in the family of Description Logics [Baader et al., 2003a], and the *rule-based* languages, which are logically grounded in the Logic Programming paradigm [Lloyd, 1987, Baral and Gelfond, 1994].

Description Logics (DLs) are decidable subsets of First Order Logic (FOL) which originated from frame-based systems [Minsky, 1985]. They are the basis of the Semantic Web standards Web Ontology Language (OWL) [Dean and Schreiber, 2004], OWL 2 [OWL 2, 2009] and the OWL 2 profiles languages [Motik et al., 2009a]. The main modelling entities in DLs are concepts and roles; different sets of constructors allow the description of complex concepts and roles. Due to their ability to capture domain language at a sufficient level of detail for practical purposes, DL knowledge bases are commonly referred to as ontologies.

On the other hand, rule-based languages have a very simple if-then syntax: a rule consists in a head and a body, where the body is usually a conjunction of literals seen as the premise of the rule, and the head is usually a disjunction of literals seen as the consequence of the rule. Unlike ontological languages which are rooted in DLs, and hence in FOL, rule-based languages typically adopt a *closed-world assumption*: everything which is not known to be true is false. The treatment of negation under this assumption is commonly referred to as *negation-as-failure*. The two most prominent semantics for rules with a negation-as-failure operator are the stable model semantics [Gelfond and Lifschitz, 1988] and the well-founded semantics [Gelder et al., 1991]. The World Wide Web Consortium (W3C) developed the Rule Interchange Format (RIF) standard [Kifer and Boley, 2010] as an inter-lingua to facilitate exchange between different rule-based languages [Kifer, 2008].

The complementary features of the two classes of languages, ontological and rule-based, led to various attempts to integrate them. One scenario featuring the need for integration of both types of knowledge is business rule specification and execution: typically, business specific knowledge is represented in the form of rules which use concepts that belong to a separate domain ontology. Making use of the operational knowledge requires a combined execution engine for both rules and ontologies [Berrueta et al., 2011]. Among the approaches which combine rule-based knowledge and ontological knowledge we mention: the Semantic Web Rule Language [Horrocks and Patel-Schneider, 2004], Description Logic Programs [Grosof et al., 2003], DL-safe rules [Motik et al., 2005], $\mathcal{DL}$+log [Rosati, 2006], dl-programs [Eiter et al., 2008], Description Logic Rules [Krötzsch et al., 2008a], and continuing with more recent approaches like Datalog$\pm$ [Calì et al., 2009], MKNF$^+$ knowledge bases [Motik and Rosati, 2010], Description Graph Logic Programs [Magka et al., 2012] and Nonmonotonic Existential Rules [Magka et al., 2013].

Open Answer Set Programming (OASP) [Heymans et al., 2008] is a rule-based formalism which at the same time has an ontological flavour: it is an extension of Answer Set Programming (ASP) [Gelfond and Lifschitz, 1988] with an open domain semantics. ASP is simply the incarnation of Logic Programming under the stable model semantics. The stable model semantics distinguishes between alternative models when the knowledge base contains a certain amount of non-determinism and as such ASP is a suitable language to describe and reason about combinatorial search problems. Returning to OASP, its syntax is identical to the ASP syntax, while its semantics is a generalization of the stable model semantics. Like in the case of ASP, the negation operator is interpreted as negation-as-failure under the stable model semantics; however, unlike there, OASP programs are grounded with respect to *open domains*, i.e., non-empty arbitrary domains which extend the Herbrand universe. As such, in the context of OASP, one speaks about *open interpretations* – these are interpretations in which the domain of interpretation is an open domain which occurs as an explicit parameter. The open domain semantics makes it possible to state generic knowledge using OASP without the need to mention actual constants.

In the general case, OASP is undecidable. With the purpose of achieving decidability, several fragments of OASP have been defined by syntactically restricting the language in a similar vein as decidable fragments of First Order Logic have been defined in the past [Grädel, 2003]. These fragments are:

- *Conceptual Logic Programs (CoLPs) under the Inverted World Assumption (IWA)* [Heymans et al., 2006], a fragment which allows only for unary and binary predicate symbols and which disallows the presence of constants in programs. The Inverted World Assumption refers to the fact that special binary predicate symbols of the form f^i are allowed to occur in CoLPs and they are to be interpreted as the inverse of their counterpart predicate symbol f, that is, $f^i(x, y)$ amounts to $f(y, x)$.

  CoLPs also impose some constraints on the shape of rules. One type of rules which can occur in CoLPs are *tree-shaped rules*: every such rule has one distinguished variable ('root' variable) which is connected to the other variables ('successor' variables) via a positive body literal (atom). For example,

$$a(X) \leftarrow f(X, Y), b(Y), g(X, Z), not\ c(Z)$$

is a tree-shaped CoLP rule in which X is a root variable, while Y and Z are successor variables. The other type of rules which can occur in a CoLP are so-called *free rules* of type:

$$a(X) \vee not\ a(X) \leftarrow$$

or

$$f(X, Y) \vee not\, f(X, Y) \leftarrow$$

These are unsafe rules, i.e. rules which contain variables that do not occur in atoms in the body of the rule, in this case the bodies of the rules being empty. This property in conjunction with the open domain semantics enables the free inclusion in the model of atoms having as arguments anonymous individuals.

The syntactical restrictions lend to CoLPs the *tree model property*: if a unary predicate is satisfiable with respect to a CoLP, then it is satisfied by a tree-shaped model of the program. A tree-shaped model is a model which can be represented as a labeled tree, where nodes are labeled with sets of unary predicates and arcs are labeled with sets of binary predicates. The domain of the model is exactly the set of nodes of the tree, while the set of unary and binary atoms which make part of the model can be reconstructed from the labels of nodes and arcs of the tree and the nodes and the arcs themselves, respectively.

CoLPs under the IWA can simulate reasoning in the expressive Description Logic $\mathcal{SHIQ}$. They were proved to be decidable; more precisely, deciding satisfiability of unary predicates with respect to a CoLP has been reduced to non-emptiness checking of two-way alternating tree automata with a parity condition [Vardi, 1998]. When the IWA is dropped, CoLPs can simulate reasoning in the DL $\mathcal{SHQ}$.

- *Forest Logic Programs (FoLPs)* [Heymans et al., 2007], a fragment which has the forest model property. FoLPs generalize CoLPs in the sense that they allow also for the presence of constants. In all other respects, the syntactical restrictions on rules are similar to the ones for CoLPs. The fragment has the *forest model property*: if a unary predicate is satisfiable then it is satisfied by a forest-shaped model. The forest in such a model contains a tree for each constant in the program having as root the respective constant, and possibly an additional tree having as root an anonymous individual.

  While believed to be decidable, decidability had been previously shown only for the so-called *acyclic and local FoLPs* in which the usage of recursion is restricted. Both fragments are quite inexpressive compared to the whole FoLP fragment. For example, local FoLPs disallow the presence of unary atoms containing a successor variable in tree-shaped rules. The two restricted fragments have the *finite bounded model property*, i.e. if a unary predicate is satisfiable than it is satisfied by a model with a universe size bounded by a double exponential in the size of the program, and as such reasoning can be reduced to the classical ASP case.

- *(Loosely) Guarded Programs* [Heymans et al., 2008] are, as their name suggests, a guarded fragment which can be translated to the decidable fragment of (loosely) guarded Fixed Point Logic [Grädel and Walukiewicz, 1999].

As the summary above attests, the reasoning support for these fragments is quite limited: with the exception of local and acyclic FoLPs, decidability had been shown via reductions to automata or to other logics. While automata are a useful tool to prove decidability and to obtain worst-case upper bounds for various logics, they are not a practical approach for actual reasoning. They are general purpose devices, in which the structure of the problem is lost, and as such they are not amenable to optimizations related to the structure of the problem at hand [Eiter et al., 2012]. Also, their best case running time is in the same complexity class as their worst-case running time [Baader et al., 2003b]. Finally, the reductions to ASP provided for reasoning with local and acyclic FoLPs are computationally expensive: the worst-case runtime complexity of both ASP-based algorithms is $2\textsc{ExpTime}^{\Sigma_2^P}$. As such, in the case of OASP, there was a need for more practical algorithms to reason with these fragments.

The forest model property of FoLPs (and CoLPs) recommends them as good candidates for being tackled by means of tableau algorithms. The latter [Baader et al., 2003a] are goal-oriented procedures which construct models with predefined structures, e.g. tree-shaped models or forest-shaped models, in a step-by-step fashion. They start with a skeleton model and an initial set of constraints (the goal), e.g. a concept/predicate checked to be satisfiable, and they extend the model to satisfy the existing constraints. The model extension leads to new constraints which have to be satisfied and so on. Termination is usually achieved by employing so-called *blocking* techniques which check for repetitions in the constructed structures, e.g. whether two nodes on a branch of a tree/forest have identical labels. Unlike reductions to automata, tableau algorithms are usually not worst-case optimal, but are amenable to optimizations and they behave well in practice.

In this thesis we *investigate the application of tableau-based techniques to reason with FoLPs*. We devise several such tableau-based algorithms for reasoning with FoLPs and, in some cases, some of its fragments. The tableau algorithms we provide establish among others the decidability of FoLPs. A main difficulty which we had to overcome when designing such algorithms was the combination of the open domain assumption and the stable model semantics. The stable model semantics requires that every atom in an open answer set is well-supported, i.e. there has to be a reason for the presence of the atom in the model and this reason should not be circular or infinite. On the other hand, the open domain semantics imposes the need for some termination mechanism. Standard blocking techniques employed by tableau algorithms based on comparison of node labels are no longer enough to guarantee termination when enhanced with well-supportedness checks. As such, our algorithms rely heavily on techniques to achieve the finite model property for FoLPs which are then transposed in runtime termination conditions. Besides successful termination by means of enhanced blocking we also introduce rules for unsuccessful termination, due not to some inconsistency in the constructed model, but due to some redundancy: if there is a model, there must be a model with a simpler structure than the one in construction.

We choose to focus on FoLPs as they are quite an expressive fragment which also subsumes the fragment of CoLPs. In [Heymans, 2006] it had been shown how satisfiability checking in the DL $\mathcal{ALCHOQ}$ can be reduced to reasoning with acyclic FoLPs. It is well known that reasoning[1] with $\mathcal{SHOQ}$ KBs can be reduced to reasoning with $\mathcal{ALCHOQ}$ KBs via a transitivity

[1]We refer here to satisfiability checking and related tasks, not to conjunctive query answering.

elimination transformation [Kazakov, 2008]. Thus, it is actually possible to simulate reasoning with $\mathcal{SHOQ}$ KBs within the fragment of acyclic FoLPs.

Furthermore, in [Feier and Heymans, 2013] we showed how reasoning with $\mathcal{SHOQ}$ KBs can be directly simulated within the fragment of FoLPs. This simulation made it possible to combine $\mathcal{SHOQ}$ KBs and FoLPs into a hybrid formalism called *f-hybrid KBs*. An f-hybrid KB is a pair of a $\mathcal{SHOQ}$ KB and a FoLP: the interaction between the two components is captured via shared interpretations, which means the formalism falls in the category of so-called *tightly-coupled approaches* for combining rules and ontologies [de Bruijn, 2009]. The f-hybrid KBs distinguish themselves among other tightly-coupled approaches by the fact that they impose no restrictions on the interaction between the signatures of the two components; other approaches typically restrict the occurrence of DL predicates, i.e. predicates which occur in the ontological component, in the rule-based component. Such restrictions prevent the need for reasoning with unknown individuals in the rule component. As the underlying technical foundation of f-hybrid KBs is the simulation of $\mathcal{SHOQ}$ KBs within FoLPs, no such restriction is needed. While the formal treatment of f-hybrid KBs is not part of the objective of this thesis, we note that any algorithm developed for reasoning with FoLPs can be used to reason with f-hybrid KBs as well.

Conceptual modeling using FoLPs is not restricted to simulating reasoning with $\mathcal{SHOQ}$ KBs: it is also possible to translate *object-role modeling (ORM)* models as sets of FoLP rules. In [Heymans, 2006] a translation of a particular ORM model to a CoLP (thus, also a FoLP) is provided. While a formal translation from ORM models to CoLPs/FoLPs is not provided there, the example translation shows how CoLP satisfiability checking can be used to verify that the various ORM object types can be populated, that some derived properties do (not) hold, etc.

1.2 Main Results

As mentioned in Section 1.1, the main objective of this thesis was developing tableau algorithms for reasoning with FoLPs and/or sub-fragments thereof. One of the main challenges in designing such algorithms has been ensuring that the constructed models are minimal. In chronological order of their development, the results are as follows:

- **A tableau algorithm for satisfiability checking of unary predicates with respect to FoLPs which runs in the worst-case in non-deterministic double exponential time**. The algorithm exploits the forest model property of the fragment: it tries to construct a model by evolving a forest-shaped data structure called "completion structure" using so-called expansion rules. The algorithm – which we will refer to as $\mathcal{A}_1$– was the first to deal with the whole fragment of FoLPs and as such established the decidability of the language.

  For termination, $\mathcal{A}_1$ employs a rather unusual blocking condition which besides being subset-based [Baader et al., 2003a], it also checks that the model obtained by unravelling the completion is minimal, i.e. there are no cyclic dependencies between atoms in the model. To this purpose, the algorithm maintains a dependency graph of the atoms in the constructed model. For termination a new rule called *redundancy rule* is introduced which sets a threshold on the depth of explored paths in a completion structure which is exponential in the size of the input program.

The algorithm is sound and complete, and as such it also establishes the fact that FoLPs have the finite bounded model property: if a unary predicate is satisfiable, it has to be satisfied by a model with finite bounded size. This opens the way for standard ASP reasoning with FoLPs.

- **A knowledge compilation technique for reasoning with FoLPs**. The technique consists in pre-computing all possible building blocks of forest models in the form of trees of depth one. Such trees are called *unit completion structures* (UCSs) and they can be computed using the expansion rules of $\mathcal{A}_1$. A new algorithm, called $\mathcal{A}_2$, is provided which tries to construct a forest model by a simple match-and-append process of UCSs using the original termination conditions of $\mathcal{A}_1$. We also define a notion of *redundant UCSs* which describes structures that are strictly more constraining than others, i.e. they are harder to be matched against, when constructing a model as opposed to other UCSs. When constructing a model, these redundant UCSs can be disregarded without losing completeness.

  Algorithm $\mathcal{A}_2$ has the same worst-case runtime complexity as $\mathcal{A}_1$, i.e. non-deterministic double exponential in the size of the original program. For both algorithms, the runtime complexity is determined by the maximal number of nodes that have to be expanded in order to ensure completeness; as both $\mathcal{A}_1$ and $\mathcal{A}_2$ employ a form of ancestor blocking, i.e. a node can be blocked only by one of its ancestors, and as the length of a branch in a completion structure can be exponential in the size of the program, this number is double exponential in the size of the input program.

- **An optimized algorithm for reasoning with FoLPs which runs in the worst-case in non-deterministic exponential time**. The algorithm, referred to as $\mathcal{A}_3$, reuses the knowledge compilation technique $\mathcal{A}_2$, but it employs different termination conditions. In particular, it uses a mechanism for reusing computation across branches which leads to a decrease in the worst-case running time complexity by one exponential level. Usually, in tableau parlance, such reuse of computation across branches is referred to as 'anywhere blocking'; this is due to the fact that the conditions for (ancestor) blocking and anywhere blocking are very similar. However, in our case the two sets of conditions are rather asymmetric; as such we will refer to this mechanism with the more general term of *caching*.

  Algorithm $\mathcal{A}_3$ uses also a much more efficient redundancy rule to curtail the expansion of paths in the cases where blocking does not suffice. The improvements of $\mathcal{A}_3$ over $\mathcal{A}_1$ and $\mathcal{A}_2$ are made possible by *a new strategy to reduce a (potentially infinite) model to a finite bounded size one* which is part of the completeness proof of the algorithm.

- **A new fragment of FoLPs, simple FoLPs, together with a reasoning procedure**. The termination rules employed by $\mathcal{A}_3$ require extensive bookkeeping regarding dependencies between atoms in the constructed model and checking complex conditions over such dependencies. We define a new fragment of Forest Logic Programs, called **simple Forest Logic Programs**, for which the blocking and caching conditions of $\mathcal{A}_3$ collapse into a simple subset-based anywhere blocking condition. The fragment generalizes acyclic FoLPs. The algorithm for reasoning with simple FoLPs can be seen as a simplified variant of $\mathcal{A}_3$ and is denoted as $\mathcal{A}_3^s$.

- **Worst-case optimal procedures for reasoning with CoLPs and simple FoLPs**. The procedures, referred to as $\mathcal{A}_{3,c}^{det}$ and $\mathcal{A}_{3,s}^{det}$, respectively, can be seen as deterministic versions of $\mathcal{A}_3$ and $\mathcal{A}_3^s$, respectively. They construct AND/OR completion structures in which all possible choices for a particular successor of a node are recorded: as such, AND/OR completion structures capture the space of all tree/forest models of certain size.

  In order not to incur an exponential blow-up when all choices regarding the size of the AND/OR completion structures are made explicit, we employ similar termination mechanisms as in the non-deterministic case; however, here reuse of computation across branches of an AND/OR structure implies reuse of computation across models.

  In this new scenario a tableau algorithm has two phases: a construction phase, in which an AND/OR completion structure is constructed by matching and appending UCSs; and an evaluation phase, in which by means of a complex truth value propagation procedure, it is decided whether the constructed AND/OR completion structure embeds an actual model. Unfortunately the technique does not generalize to the case of full FoLPs.

As the next chapters will show, constructing tableau algorithms for reasoning with FoLPs is a highly non-trivial task. Furthermore, improving complexity bounds for our algorithms comes at the price of either maintaining ever-increasing complex data structures or reducing the language to a fragment in which recursion is restricted. From a theoretical point of view, the results are interesting as they shed light on the interaction between the stable model semantics and the open world assumption. However, due to the complex blocking conditions and extensive bookkeeping they require, it is not clear how well the algorithms would behave in practice; it is subject to future work to establish this.

1.3 Structure of the Thesis

The thesis is organised as follows:

- Chapter 2 describes some preliminaries. Section 2.1 introduces notations concerning data structures like trees, forests and graphs. ASP is introduced in Section 2.2, followed by OASP in Section 2.3, and CoLPs and FoLPs in Section 2.4.

- Chapter 3 introduces $\mathcal{A}_1$, the non-deterministic double exponential algorithm for reasoning with FoLPs. An extended example which traces the running of $\mathcal{A}_1$ on a particular FoLP is provided in Section 3.6. Section 3.7 discusses the relation between the new algorithm and tableau procedures in the DL realm, on the one hand, and ASP solvers – tableau or non-tableau based –, on the other hand.

- The knowledge compilation technique for reasoning with FoLPs is described in Chapter 4. In particular, Section 4.3 formalizes the notion of redundant UCSs, while Section 4.4

describes the actual algorithm which uses for the construction of a model only the set of non-redundant UCSs. Finally, Section 4.5 discusses the pros and cons of the technique and some related work.

- Algorithm $\mathcal{A}_3$, the optimized tableau algorithm for reasoning with FoLPs, is described in Chapter 5. In the same chapter, in Section 5.8 we introduce the fragment of simple FoLPs and describe the simplified algorithm $\mathcal{A}_3^s$.

- The worst-case optimal algorithms for reasoning with CoLPs and simple FoLPs are described in Chapter 6 in Section 6.2 and Section 6.3, respectively. Section 6.4 discusses why the deterministic approach does not scale in the case of (full) FoLPs and presents some related work.

- Some further related work is discussed in Section 7.1. This is followed by a summary of the thesis in Section 7.2 and a discussion concerning future work in Section 7.3.

1.4 Publications

Some parts of this thesis have been published in a preliminary form in articles occurring in journals, conferences and workshops.

The first algorithm to deal with FoLPs, $\mathcal{A}_1$, has been first described in [Feier and Heymans, 2009]. This has been extended to a journal publication containing full proofs and further details in [Feier and Heymans, 2013].

The knowledge compilation technique which underlies $\mathcal{A}_2$ has been first introduced in [Feier and Heymans, 2010] only for the case of CoLPs.

A fragment similar to simple FoLPs, named simple CoLPs has been described in [Feier and Heymans, 2008] and in [Heymans et al., 2009]. Simple CoLPs is a fragment of FoLPs which allows to simulate reasoning in the DL $\mathcal{ALCH}$. The fragment of simple FoLPs has been described in [Feier and Heymans, 2013], albeit with a different treatment as concerns reasoning – instead of using the knowledge compilation technique introduced in Chapter 4, there, a simplified version of $\mathcal{A}_1$ has been employed to reason with the fragment. Finally, the results concerning $\mathcal{A}_3$ are described in [Feier, 2012].

CHAPTER 2

Preliminaries

2.1 Trees, Forests, Graphs

This section introduces some notations for trees which extend those in [Vardi, 1998] and their generalizations to forests introduced in [Heymans et al., 2007].

Let $\cdot$ be a concatenation operator between sequences of constants or natural numbers. A *tree* T with root c, also denoted as T_c, where c is a specially designated constant, is a set of nodes, where each node is a sequence of the form $c \cdot s$, where s is a (possibly empty) sequence of positive integers formed with the help of the concatenation operator and which has the property that for every $x \cdot d \in T_c$, $d \in \mathbb{N}_{>0}$, where $\mathbb{N}_{>0}$ is the set of positive integers, it must be the case that $x \in T_c$. When the root of the tree is irrelevant, we will simply refer to the tree as T. For every such sequence of concatenated positive integers s, we denote with $||s||$ its length, i.e. the number of positive integers which are concatenated: thus, for a node x in a tree T, $||x||$ represents the depth at which x occurs in T. The set of all such sequences of natural numbers formed using the concatenation operator is denoted by $\langle \mathbb{N}_{>0} \rangle$.

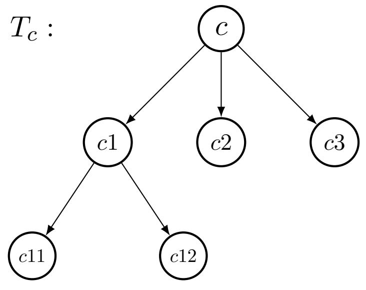


Figure 2.1: A simple tree

Example 1. A tree with root c, where c has three direct successors and the first of these successors has in turn two direct successors, will be denoted by $T_c = \{c, c \cdot 1, c \cdot 2, c \cdot 3, c \cdot 1 \cdot 1, c \cdot 1 \cdot 2\}$ or $T_c = \{c, c1, c2, c3, c11, c12\}$ [1]. Figure 2.1 depicts such a tree.

The set of arcs of a tree T, A_T, is defined as follows: $A_T = \{(x, y) \mid x, y \in T, \exists n \in \mathbb{N}_{>0}.y = x \cdot n\}$. For $x, y \in T$, one says that $x \leqslant_T y$ ($x \geqslant_T y$) if x is a prefix (suffix) of y (with the relations being strict when $x \neq y$). The successors of a node x in a tree T, $succ_T(x)$, is the set of nodes defined as follows: $succ_T(x) = \{y \in T \mid y = x \cdot i, i \in N_{>0}\}$. The predecessor of a node x in a tree T is denoted by $prec_T(x)$ and it is the node y such that there exists $i \in \mathbb{N}_{>0}$ such that $x = y \cdot i$.

The lowest common ancestor of two nodes x and y in a tree T, denoted by $lca_T(x, y)$, is the node z such that $z <_T x$, $z <_T y$, and for every node $z' \in T$ such that $z' <_T x$, and $z' <_T y$, it is the case that $z' \leqslant_T z$. A node $x \in T$ is said to be to the right of a node $y \in T$ and is denoted by $right_T(x, y)$ if there exists a node $z \in T$, $i, j \in \mathbb{N}_{>0}$, and $s_1, s_2 \in \langle\mathbb{N}_{>0}\rangle$, such that $x = z \cdot i \cdot s_1$, $y = z \cdot j \cdot s_2$, and $i > j$.

The subtree of T at y, denoted by $T[y]$, is the set $\{x \mid x \in T, x = y \cdot s, s \in \langle\mathbb{N}_{>0}\rangle\}$. A path in a tree T from x to y is denoted by $path_T(x, y)$ and is defined as $path_T(x, y) = \{z \mid x \leqslant z \leqslant y\}$. By frontier of a tree T, $fr(T)$, we understand the set of nodes in T which have no successors: $fr(T) = \{x \in T \mid succ_T(x) = \emptyset\}$.

Example 2. Consider the tree T_c in Figure 2.1. The lowest common ancestor of $c11$ and $c2$ in T_c, $lca_{T_c}(c11, c2)$, is c, and it is also the case that: $right_{T_c}(c3, c2)$ and $right_{T_c}(c2, c12)$. The frontier of T_c is the set $\{c11, c12, c2, c3\}$.

An extended tree ET is a tuple (T, ES), where T is a tree, and $ES \subseteq T \times T$ – thus, it is technically a graph overlaid on the skeleton of a given tree. The set of nodes of ET, N_{ET}, is exactly T, while the set of its arcs, A_{ET}, is defined as follows: $A_{ET} = A_T \cup ES$. The set of extra arcs, ES, extends the successor relation between nodes in T:

$$succ_{ET}(x) = succ_T(x) \cup \{y \mid (x, y) \in ES\}.$$

A *forest* F is a set of trees $\{T_c \mid c \in C\}$, where C is a finite set of arbitrary constants. The set of nodes, N_F, and the set of arcs, A_F, of a forest F are defined as follows: $N_F = \cup_{T \in F} T$, and $A_F = \cup_{T \in F} A_T$, respectively. For a node $x \in N_F$, let $succ_F(x) = succ_T(x)$, where $x \in T$ and $T \in F$, be the set of successors of x in F. Also, similarly to trees, a strict partial order relationship $<_F$ on the set of nodes N_F of a forest F is defined, where $x <_F y$ if $x <_T y$ for some tree T in F.

An *extended forest* EF is a tuple (F, ES), where $F = \{T_c \mid c \in C\}$ is a forest and $ES \subseteq N_F \times N_F$. The sets of nodes N_{EF} and arcs A_{EF} of an extended forest EF are defined as follows: $N_{EF} = N_F$, and $A_{EF} = A_F \cup ES$, respectively. As in the case of extended trees, an extended forest is basically a graph overlaid on a forest structure: ES relates arbitrary pairs of nodes in the forest and extends the successor relation between nodes as follows:

$$succ_{EF}(x) = succ_F(x) \cup \{y \mid (x, y) \in ES\}.$$

[1]By abuse of notation, it is assumed that there are at most 9 successors for every node, so $a \cdot b$ will be abbreviated with ab

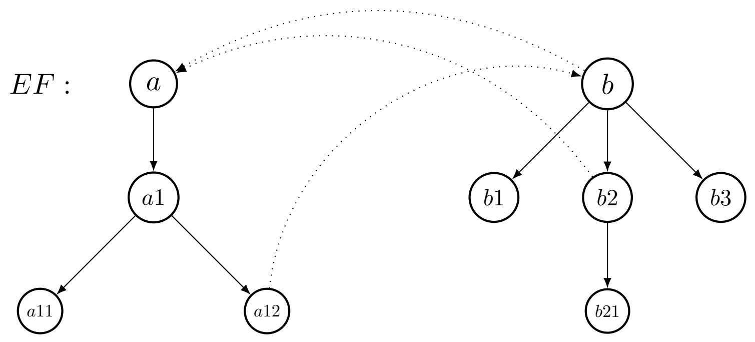


Figure 2.2: An extended/interconnected forest

When an extended forest $EF = (F, ES)$ is such that $ES \subseteq N_F \times C$, in other words ES relates arbitrary nodes in the forest to roots of trees in the forest, then EF is said to be an *interconnected forest*.

Example 3. Figure 2.2 depicts an extended forest EF which is an interconnected forest. The underlying forest is composed of two trees, $T_a = \{a, a1, a11, a12\}$ and $T_b = \{b, b1, b2, b3, b21\}$, and the set of extra arcs ES is $\{(a12, b), (b2, a), (b, a)\}$.

When an interconnected forest $EF = (F, ES)$ is such that F is a set of trees $\{T_c \mid c \in C\}$, where C is a finite set of arbitrary constants, and there exists $d \in C$ such that $T_c = \{c\}$, for every $c \in C \backslash \{d\}$, and $ES \subseteq T_d \times C$, we call the forest an *interconnected tree with root d with respect to C*: all trees but one (T_d) are single-node trees. The depth of an interconnected tree is the depth of its distinguished tree T_d.

In general, for an interconnected forest $EF = (F, ES)$, the trees in the underlying forest F together with the additional arcs in ES give rise to interconnected trees in EF, where such a tree is one of $T_c \in F$, extended with the arcs $\{(x, y) \mid (x, y) \in ES, x \in T_c\}$ and with the nodes $\{y \mid (x, y) \in ES, x \in T_c\}$. The interconnected tree obtained as an extension of T_c in EF is denoted by T_c^{EF}.

Example 4. The interconnected tree T_a^{EF} induced by T_a in EF in Example 3 is the extension of T_a with the extra arc $(a12, b)$, and the interconnected tree T_b^{EF} induced by T_b in EF in the same example is the extension of T_b with the extra arcs (b, a) and $(b2, a)$.

A labelled (extended/interconnected) forest/tree is a tuple (FT, ft) where FT is an (extended/interconnected) forest/tree and $ft : N_{FT} \rightarrow \Sigma$ is a labelling function, with Σ being a set of arbitrary symbols; sometimes we will refer to such an object as ft.

Example 5. Figure 2.3 depicts an extended labelled forest (a labeled version of the extended forest from Figure 2.2).

As was the case for (simple) interconnected forests, a labeled interconnected forest (EF, ef), with $EF = (F, ES)$ and $F = \{T_c \mid c \in C\}$, induces a set of labeled interconnected trees

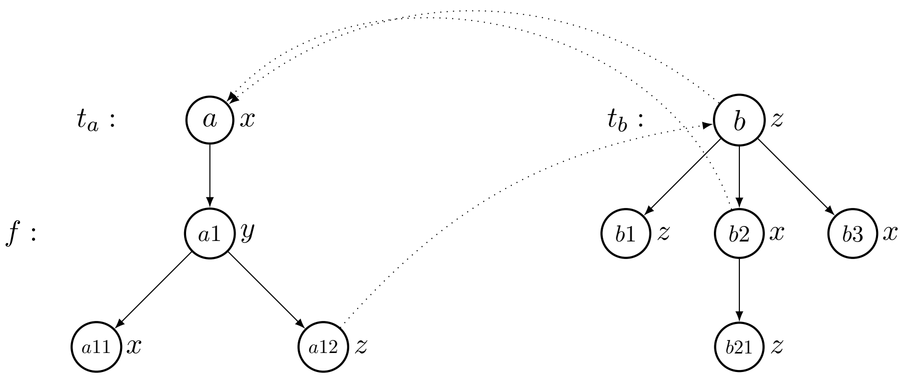


Figure 2.3: An Extended Labeled Forest

$\{(T_c^{ef}, t_c^{ef}) \mid c \in C\}$, with $t_c^{ef} : N_{T_c^{ef}} \to \Sigma$ defined as follows: $t_c^{ef}(x) = ef(x)$, for any $x \in T_c^{ef}$. The definition of interconnected subtree generalizes straightforwardly also to the one of labeled interconnected subtree.

We introduce the operation of replacing in a labeled interconnected forest ef a labeled interconnected subtree $t^{ef}[x]$ with another labeled interconnected subtree $t^{ef}[y]$, where both x and y are from N_{EF}, and denote this operation with $replace_{ef}(x, y)$.

Example 6. Figure 2.4 describes the result of applying the replace operation on the extended forest from Figure 2.3 with two different sets of arguments. In the first case, $t_b^{ef}[b2]$ is replaced with $t_a^{ef}[a1]$, while in the second case $t_a^{ef}[a1]$ is replaced with $t_a^{ef}[a12]$.

Note that the names of nodes in the replaced subtree are not substituted with names of the nodes from the replacing subtree, but new names are generated for the new nodes in concordance with the naming scheme for nodes of that tree. As a result of applying the first replacement operation, one of the extra arcs of t_b, $(b2, a)$, is dropped (it was part of the replaced extended subtree) and a new extra arc is introduced, $(b22, b)$, which mirrors the arc $(a12, b)$ from the replacing extending subtree. Similarly, as a result of the second transformation, $(a12, b)$ is dropped and $(a1, b)$ is introduced.

Finally, a directed graph G is defined as usual by its sets of nodes V and arcs A. We introduce some graph-related notations: $paths_G$ denotes the set of finite paths in G, where each path is represented as a tuple of nodes from V:

$$paths_G = \{(x_1, \ldots, x_n) \mid ((x_i, x_{i+1}) \in A)_{1 \leqslant i < n}\}.$$

The set of finite paths in G from x to y is denoted by $paths_G(x, y)$:

$$paths_G(x, y) = \{(x_1, \ldots, x_n) \in paths_G \mid x_1 = x, x_n = y\},$$

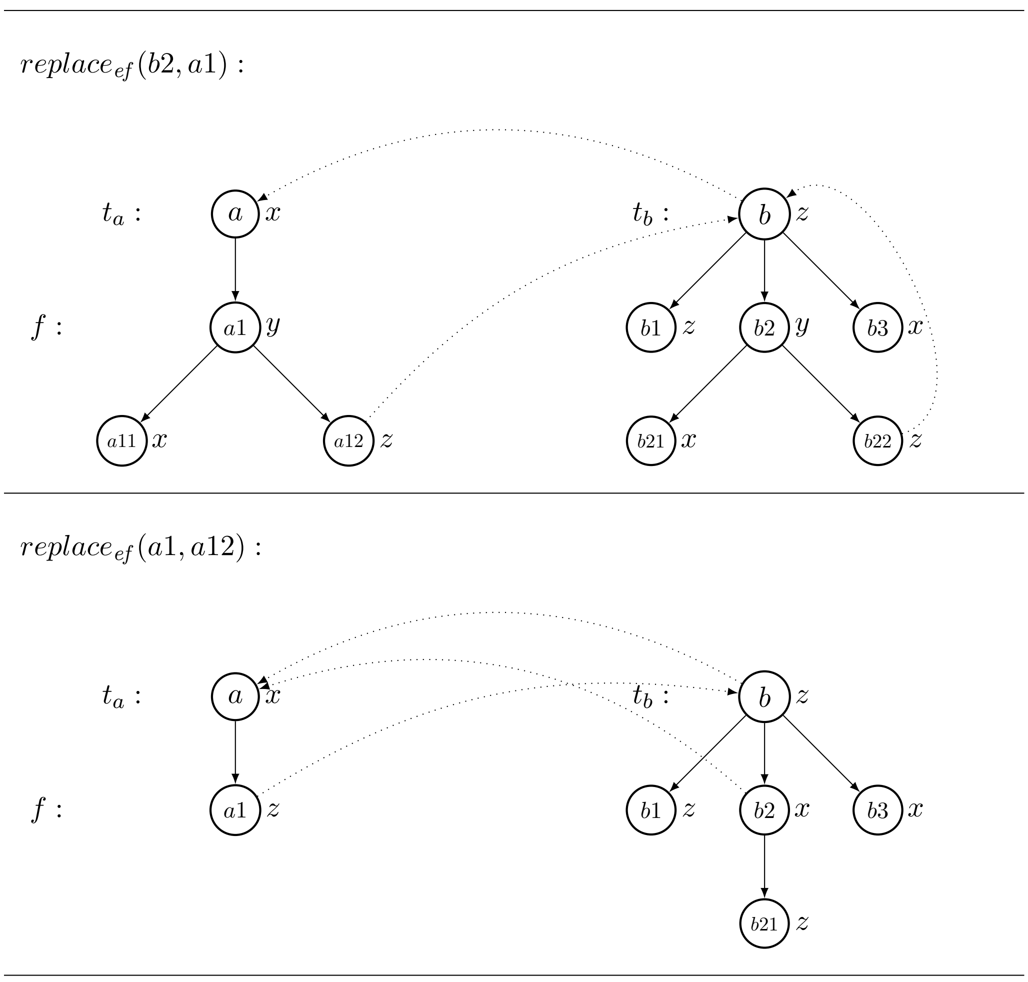


Figure 2.4: Two applications of the replace operator on ef

while $conn_G$ denotes the set of pairs of connected nodes from V:

$$conn_G = \{(x_1, x_n) \mid (x_1, \ldots, x_n) \in paths_G\}.$$

Cycles and elementary cycles in directed graphs are defined as usually.

In order to operate with paths in directed graphs we also introduce some tuple operators: the concatenation of two tuples $T_1 = (x_1, \ldots, x_n)$, and $T_2 = (y_1, \ldots, y_m)$, denoted by $T_1 \hat{} T_2$, is the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$. A tuple T_1 is part of another tuple T_2: $T_1 \subseteq T_2$, if there exist two (possibly empty) tuples T_3 and T_4 such that $T_2 = T_3 \hat{} T_1 \hat{} T_4$.

As extended/interconnected trees/forests are particular types of directed graphs, all graph related notations introduced in this section apply to these entities.

2.2 Answer Set Programming

Answer Set Programming [Lifschitz, 2008] is a declarative programming language which has a rule-based syntax and adopts a Closed World Assumption [Reiter, 1978]: the intended meaning of its programs is captured by the stable model semantics [Gelfond and Lifschitz, 1988, Marek and Truszczynski, 1989, Marek, 1999]. Every rule has a body, which represents a conjunction of possibly negated atoms (literals), and a head which represents a disjunction of possibly negated atoms (literals). Intuitively, at least one literal in the head of such a rule should be true when all literals in the body of the rule are true as well. Rules which have empty bodies and whose heads contain a single atom are called *facts*.

Example 7. Consider the following rules:

$$\begin{array}{rrcl} r_1: & hiking \lor tennis & \leftarrow & sunny \\ r_2: & hiking \lor tennis & \leftarrow & not\ raining \\ r_3: & sunny & \leftarrow & \end{array}$$

Note that rule r_3 is a fact. When considering rule r_1 and the fact r_3, it must the case that either $hiking$ is true, $tennis$ is true or both activities are true at the same time (as we will see, the semantics for ASP disallows the last possibility). According to rule r_2, same conclusions should be drawn if it is not raining. Then, when considering both r_2 and the fact r_3 can one infer that either $hiking$ or $tennis$ is true? Or in other words, when is it not raining? Should that piece of information be explicitly listed in our knowledge base or simply its omission from the knowledge base is enough? This particular example, but also the way humans tend to make decisions based only on available information, suggest the latter: it is enough not to know (to be able to derive) that it rains.

As the example suggests, new atoms should be derivable also based on absence of information. This is exactly what the closed world assumption is about: every atom which is not explicitly true or derivable by means of rules is assumed to be false. As such, the form of negation which occurs in answer set programs is commonly denominated as *negation-as-failure*: an atom is not true if its derivation fails. While it is obvious that by allowing more than one literal in the heads of rules, some non-determinism is introduced with respect to the intended meaning of a program, non-determinism is present as well even when rules have just one literal in their head. This is due to the interaction between negation as failure and circular derivation paths via negative literals.

Example 8. Consider the following rules:

$$\begin{array}{rrcl} r_1: & p & \leftarrow & not\ q \\ r_2: & q & \leftarrow & not\ p \end{array}$$

When considering only r_1, it seems that p should be derived based on the absence of q. When considering only r_2, it seems that q should be derived based on the absence of p. What happens when both rules are considered at the same time? It is clear that p and q cannot hold at the same time, as then, they cannot be derived from the corresponding rules and only facts that are either derivable or explicitly true in the program hold.

The answer to the question in Example 8 is that the program composed of both r_1 and r_2 has two meanings: one in which p and only p holds, the other in which q and only q holds. Technically, this is enforced via a minimal model semantics. First, a guess is made regarding an intended meaning of the program – a candidate model, and the rules of the program are simplified using the hypothetical model such that all sources of non-determinism are removed: all literals in the heads of rules which are not satisfied by the candidate model are removed, all negative literals in the bodies of rules which are not satisfied by the candidate model are removed, and finally all rules whose bodies or heads are obviously unsatisfiable are removed as well. Then, it is checked whether the candidate model is the minimal model of the reduced program. If that is the case, the candidate model is a *stable model* of the original program, also called an *answer set*. The semantics is called the *stable model semantics* and the simplification step is commonly known as the *Gelfond-Lifschitz reduct* [Gelfond and Lifschitz, 1988]. An alternative simplification procedure is the so-called FLP reduct [Faber et al., 2004]. For the rest of this work, we will consider always only the first type of reduct. Initially [Gelfond and Lifschitz, 1988], the stable model semantics has been defined only for normal logic programs, i.e. programs without disjunction in the head, but subsequently it has been extended to disjunctive logic programming [Przymusinski, 1991, Gelfond and Lifschitz, 1991, Lifschitz and Woo, 1992].

Example 9. Let P be the following answer set program:

$$\begin{array}{rcll} r_1: & p & \leftarrow & not\ q, r \\ r_2: & q & \leftarrow & not\ p, s \\ r_3: & p & \leftarrow & \end{array}$$

Also, let $\{p, r\}$ be a candidate model for P. To compute the Gelfond-Lifschitz reduct of P with respect to the candidate model we remove the literal $not\ q$ from the body of r_1 (as the literal holds with respect to the model) and we also remove rule r_2 (as the literal $not\ p$ in its body does not hold). The reduct P' is the following program:

$$\begin{array}{rcll} {r'}_1: & p & \leftarrow & r \\ r_3: & r & \leftarrow & \end{array}$$

It can be seen that $\{p, r\}$ is indeed the minimal model of P' in a classical sense, and thus it also an answer set of P.

By repeating the procedure with another candidate model $\{q, s\}$, the Gelfond-Lifschitz reduct of P with respect to $\{q, s\}$, P', is the program:

$$\begin{array}{rcll} {r'}_1: & q & \leftarrow & s \\ r_3: & r & \leftarrow & \end{array}$$

In this case, the candidate model is not a minimal model of P': neither q, nor s, can be derived from facts occurring in P'. Thus, $\{q, s\}$ is not an answer set of P.

Due to its ability to describe multiple intended meanings, ASP is an important device to represent problems and reason in areas like search [Niemelä, 1999, Marek, 1999], planning [Dimopoulos et al., 1997, Subrahmanian and Zaniolo, 1995, Eiter et al., 2004], diagnosis [Nogueira et al., 2001], product configuration [Tiihonen et al., 2003, Soininen and Niemelä, 1999], etc.

In the following we will proceed to the formal treatment of the language.

2.2.1 Syntax

We assume the presence of countable infinite sets of constants $a, b, c, \ldots$, *variables* $X, Y, \ldots$, and predicates $q, p, \ldots$, which do not overlap. Every predicate p has associated a certain arity $n \in \mathbb{N}$: when p has arity n, we say that p is n-ary. A *term* is a constant or a variable. A *literal* is an atom L or a negated atom $not\ L$.

When a predicate p occurs either as it is or negated ($not\ p$) in some context, but it is not known a priori which is the case, we will refer to it as $\pm p$; it is assumed that its form does not change throughout the context, thus $\pm p$ will refer to the same entity in the given context.

For a literal L, $pred(L)$, and $args(L)$ denote the (possibly negated) predicate, and the (tuple of) arguments of L, respectively.[2] By $arg_i(L)$ we denote the i-th argument of L. For a set S of literals or (possibly negated) predicates, $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid not\ a \in S\}$. For a set S of atoms, $not\ S = \{not\ a \mid a \in S\}$.

An *answer set program* is a countable set of rules $\alpha \leftarrow \beta$, with α and β being finite sets of literals. The set α is the *head* and represents a disjunction, while β is the *body* and represents a conjunction. Rules can also be named, as in $r : \alpha \leftarrow \beta$, where r is the name of the rule. If $\alpha = \emptyset$, the rule is called a *constraint*.

Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program R, let $cts(R)$ be the constants in R, $vars(R)$ its variables, and $preds(R)$ its predicates. For P and U an answer set program and a set of constants, respectively, we denote with P_U the ground program obtained from P by substituting every variable in P by every element in U. The set of atoms (literals) that can be formed from a ground program P using $preds(P)$ as predicate symbols and $cts(P)$ as terms, is denoted by $\mathcal{B}_P$ ($\mathcal{L}_P$).

For a term t, the *exact replacement* of ground term x with ground term y in t, denoted by $t_{x|y}$, is defined as follows:

$$t_{x|y} = \begin{cases} y, & \text{if } t = x; \\ t, & \text{otherwise.} \end{cases}$$

The notation is extended to tuples of terms, literals, rules, and programs. For a tuple of terms $T = (t_1, \ldots, t_n)$, $T_{x|y} = ((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a regular literal $L = (not)p(t_1, \ldots, t_n)$, $L_{x|y} = (not)\ p((t_1)_{x|y}, \ldots, (t_n)_{x|y})$. For a set of literals S, $S_{x|y} = \{L_{x|y} \mid L \in S\}$. For a named rule $r : \alpha \leftarrow \beta$, its image under the exact replacement of x with y is $r_{x|y} : \alpha_{x|y} \leftarrow \beta_{x|y}$ (where $r_{x|y}$ is the new name of the rule, and does not involve any term replacement). For a ground program P, its image under the exact replacement of x with y is $P_{x|y} = \{r_{x|y} \mid r \in P\}$.

Example 10. The exact replacement of c with b in x, $x_{c|b}$, is x, while the exact replacement of c with b in c, $c_{c|b}$, is b. Consequently, $(p(x, c))_{c|b} = p(x, b)$.

[2]If the literal L has just one argument, $args(L)$ will return the argument itself.

2.2.2 Semantics

As mentioned already, the semantics of Answer Set Programming is the so-called stable model semantics. An *interpretation* I of a ground answer set program P is a subset of $\mathcal{B}_P$. We say that I satisfies a ground atom $p(t_1, \ldots, t_n)$ and write $I \models p(t_1, \ldots, t_n)$ if $p(t_1, \ldots, t_n) \in I$. Also, $I \models not\ p(t_1, \ldots, t_n)$ if $I \not\models p(t_1, \ldots, t_n)$. For a set of ground literals L, $I \models L$ if $I \models l$ for every $l \in L$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* with respect to I, denoted by $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$.

For a positive ground program P, i.e. a program without not, an interpretation I of P is a *model* of P, if I satisfies every rule in P; it is an *answer set* of P, if it is a subset-minimal model of P. For ground programs P containing not, the *GL-reduct* [Gelfond and Lifschitz, 1988] with respect to I is denoted by P^I, and it contains all rules of the type $\alpha^+ \leftarrow \beta^+$, where there exists a rule $\alpha \leftarrow \beta$ in P such that:

- $I \models not\ \beta^-$, and
- $I \models \alpha^-$.

For examples of the application of the GL-reduct see Example 9.

An interpretation I is an *answer set* of a ground answer set program P if I is an answer set of the reduct P^I.

Some typical reasoning tasks in ASP are:

- *Consistency checking*: an answer set program P is said to be consistent if it admits at least one answer set.
- *Brave entailment*: a program P *bravely entails* a ground atom a if there exists an answer set I of P such that $I \models a$.

  This reasoning task can be reduced to consistency checking: P *bravely entails* a if $P \cup \{a \leftarrow\}$ is consistent.
- *Skeptical entailment*: a program P *skeptically entails* a ground atom a if for every answer set I of P, it is the case that $I \models a$.

  Like its brave counterpart, this reasoning task can be reduced to consistency checking: P *skeptically entails* a if $P \cup \{\leftarrow a\}$ is not consistent.

Note that some treatments of ASP [Gelfond and Lifschitz, 1991] also include a form of classical negation, called strong negation. Typically, this construct is denoted by one of the symbols neg or $\neg$. Its intended meaning is that a strongly negated fact holds only when it occurs in a negated form explicitly in the program or its negation can be derived from the program. We will omit this type of negation throughout this dissertation as it can be seen as 'syntactic sugar' [Gelfond and Lifschitz, 1991]: any program which contains strong negation can be rewritten into a program without strong negation which is equivalent to the original program with respect to all major reasoning tasks.

2.3 Open Answer Set Programming

Open Answer Set Programming (OASP) [Heymans et al., 2008] extends Answer Set Programming (ASP) by opening up the domain of interpretation for rules under the stable model semantics: the semantics is still defined in a declarative fashion using the Gelfond-Lifschitz reduct, but the grounding of the programs is done with respect to arbitrary universes which are non-empty supersets of the set of constants which occur in the program. As such, the language has a first-order flavour while at the same time preserves the minimal model semantics of ASP.

Syntactically, OASP rules are very similar to ASP ones. The only difference is that we allow for *equality* and *inequality literals* of the form $s = t$ and $s \neq t$, respectively, where s and t are terms. A literal that is not an inequality literal will be called a *regular literal*. Again, when dealing with OASP we do not consider strong negation explicitly as it can be seen as syntactic sugar on top of the basic language.

Semantically, interpretations are parametric with respect to so-called universes where:

Definition 1 (after [Heymans, 2006])**.** A *universe* U for a program P is a non-empty countable superset of the constants in P: $cts(P) \subseteq U$. We call P_U the ground program obtained from P by substituting every variable in P by every possible element from U.

A program is assumed to be a finite set of rules; infinite programs only appear as by-products of grounding with an infinite universe.

Definition 2 (after [Heymans, 2006])**.** An *open interpretation* of an open answer set program P is a pair (U, M), where U is a universe for P and M is an interpretation of P_U.

Definition 3 (after [Heymans, 2006])**.** An *open answer set* of P is an open interpretation (U, M) such that M is an answer set of P_U.

Example 11. Consider the following open answer set program P:

$$\begin{array}{rcl} p(X) & \leftarrow & not\ q(X), r(X) \\ q(a) & \leftarrow & \\ r(X) \vee not\ r(X) & \leftarrow & \\ q(X) & \leftarrow & q(Y), Y \neq a \end{array}$$

Note that the program contains a single constant a. Under regular ASP semantics, it has two answer sets: $\{q(a)\}$ and $\{q(a), r(a)\}$. They correspond to the open answer sets $(\{a\}, \{q(a)\})$ and $(\{a\}, \{q(a), r(a)\})$. Note that under the regular answer set semantics the predicate p is not satisfiable.

Next, we consider a universe $\{a, x\}$: that is, besides the constants which appear in the program, the universe contains an anonymous individual x. This gives rise to the following ground program $P_{\{a,x\}}$:

$$
\begin{array}{rcl}
p(a) & \leftarrow & not\ q(a), r(a) \\
q(a) & \leftarrow & \\
r(a) \vee not\ r(a) & \leftarrow & \\
q(a) & \leftarrow & q(a), a \neq a \\
q(x) & \leftarrow & q(a), a \neq a \\
p(x) & \leftarrow & not\ q(x), r(x) \\
r(x) \vee not\ r(x) & \leftarrow & \\
q(a) & \leftarrow & q(x), x \neq a \\
q(x) & \leftarrow & q(x), x \neq a
\end{array}
$$

Its answer sets are:

- $M_1 = \{q(a)\}$,
- $M_2 = \{q(a), r(a)\}$,
- $M_3 = \{q(a), r(x), p(x)\}$, and
- $M_4 = \{q(a), r(a), r(x), p(x)\}$.

Each of these answer sets induces an open answer set: $OA_i = (\{a, x\}, M_i)$, where $1 \leqslant i \leqslant 4$. Note that the predicate p is satisfiable in this setting: $p(x)$ occurs in both M_3 and M_4. Thus, the introduction of new domain elements in the universe, gave rise to new open answer sets in which some previously unsatisfiable predicates become satisfiable.

Rules like $r(X) \vee not\ r(X) \leftarrow$ in P present a particular interest: they justify the presence or absence from the model of the atom $r(x)$, for every element x in the universe. As we will see in Section 2.4, such rules will have a special status and will be called *free rules*.

It is worthy to mention that in a regular ASP setting, the open answer sets OA_i, for $1 \leqslant i \leqslant 4$, could be retrieved by means of a domain predicate which introduces x as a new available constant. However, in the general case it is hard to know a priori how many such fresh constants need to be introduced to render a predicate satisfiable.

The reasoning tasks we are interested in when dealing with OASP programs are firstly the same ones we described for ASP: brave and skeptical entailment of ground atoms and consistency checking. Their definition is similar to the ASP case, except that here, answer sets are replaced with open answer sets.

Additionally, we consider a reasoning task which caters more to the open flavoured semantics of OASP. The new task is:

- *satisfiability checking of a predicate p with respect to an OASP P*: if p has arity n, we say that p is satisfiable with respect to P if there exist an open answer set (U, M) of P and some terms $t_1, \ldots, t_n \in U$ such that $p(t_1, \ldots, t_n) \in M$.

Consistency checking of open answer set programs, and hence also checking of brave and skeptical entailment of ground atoms, can be reduced to satisfiability checking:

Theorem 1 (after [Heymans, 2006]). Let P be an OASP. P is consistent iff p is satisfiable with respect to $P \cup \{p(X) \vee not\ p(X) \leftarrow\}$, where p is a unary predicate not appearing in P.

In [Heymans, 2006], it has been shown that, in its unrestricted form, OASP is undecidable, by means of a reduction from the origin constrained domino problem [Wang, 1961].

2.4 Conceptual and Extended/Local Forest Logic Programs

As already anticipated in the Introduction, *Conceptual Logic Programs (CoLPs)* and *Forest Logic Programs (FoLPs)* are two fragments of OASP which have the tree model property and the forest model property, respectively. They were introduced in [Heymans, 2006].

2.4.1 CoLPs and FoLPs

CoLPs disallow the presence of constants in programs. They also impose some constraints on the shape of rules: unary and binary rules are tree-shaped rules which have as head a single unary atom and binary atom, respectively. The tree-like structure of rules refers to the chaining pattern of rule variables: one variable can be seen as the root of a tree and the others as successors of the root such that for every arc in the tree there is a positive binary literal in the body which connects the two corresponding variables. Inequalities between 'successor' variables can also appear in the body of such a rule; we will refer to the set of literals in the body of a rule formed only with the help of the 'root' variable as the 'local part' of the rule and to the remaining part of the rule body as the 'successor part' of the rule. Constraints, i.e. rules with empty head, are also allowed, but their body also has to be tree-shaped, so that they can be simulated via unary rules. Another type of rules which can appear in CoLPs are so-called *free rules* which have one of the following shapes: $a(X) \vee not\ a(X) \leftarrow$ or $f(X, Y) \vee not\, f(X, Y) \leftarrow$, where a is a unary predicate and f is a binary predicate.

FoLPs allow the same type of rules as CoLPs, with the addition that they also allow the presence of constants in the programs. An exception is made also in what concerns tree-shaped rules in which constants occur in the successor part of the rule: the 'root' term does not necessarily have to be linked to such a constant via a binary atom.

Formally:

Definition 4. A *forest logic program (FoLP)* is an open answer set program with only unary and binary predicates, and such that a rule is either:

- a *free rule*:
$$a(s) \vee not\ a(s) \leftarrow\ , \tag{2.1}$$
or
$$f(s, t) \vee not\, f(s, t) \leftarrow \tag{2.2}$$

- a *unary rule*:
$$a(s) \leftarrow \beta(s), (\gamma_m(s, t_m), \delta_m(t_m))_{1 \leqslant m \leqslant k}, \psi \tag{2.3}$$
with $\psi \subseteq \bigcup_{1 \leqslant i \neq j \leqslant k} \{t_i \neq t_j\}$ and $k \in \mathbb{N}$,

- a *binary rule*:

$$f(s,t) \leftarrow \beta(s), \gamma(s,t), \delta(t), \tag{2.4}$$

- or a *constraint*:

$$\leftarrow a(s) \text{ or } \leftarrow f(s,t), \tag{2.5}$$

  where in each rule above:

  - a is a unary predicate,
  - f is a binary predicate,
  - s, t, and $(t_m)_{1 \leqslant m \leqslant k}$ are distinct terms,
  - β, δ, and $(\delta_m)_{1 \leqslant m \leqslant k}$ are sets of (possibly negated) unary predicates, and
  - γ, and $(\gamma_m)_{1 \leqslant m \leqslant k}$ are sets of (possibly negated) binary predicates,

  and

  - equality and inequality do not appear in any γ:
    - $\{=, \neq\} \cap \gamma_m = \emptyset$, for $1 \leqslant m \leqslant k$, and
    - $\{=, \neq\} \cap \gamma = \emptyset$;
  - there is a positive atom that connects the head term s with any successor term which is a variable:
    - $\gamma_m^+ \neq \emptyset$, if t_m is a variable, for $1 \leqslant m \leqslant k$, and
    - $\gamma^+ \neq \emptyset$, if t is a variable.

Conceptual Logic Programs result from FoLPs by disallowing the occurrence of constants in such programs:

Definition 5. A *conceptual logic program (CoLP)* is a forest logic program in which all terms are variables.

A predicate q in a FoLP/CoLP P is said to be *free* if it occurs in a free rule in P.

Example 12. The following program P is a FoLP which describes the fact that somebody is happy if he meets a friend who is happy or an enemy who is unhappy, and somebody is unhappy if he meets an enemy who is happy or a friend who is not happy. This is expressed by means of four unary rules (rules r_1-r_4). Each of these unary rules have X as the root variable and Y as a successor variable for X.

Furthermore, somebody is happy if he has at least two friends: this is captured by another unary rule, r_5, which has X as the root variable and Y and Z as distinct successor variables (expressed by the inequality in the body of the rule).

The binary predicates $sees$, $friend$, and $enemy$ are free predicates, i.e. they are defined only via free rules (rules r_6-r_8). Some constraints enforce that somebody cannot be at the same time both friends and enemies with the same person (rule r_9), and that somebody cannot be at the same time both happy and unhappy (rule $r_1 0$) .

Additionally, there is an individual j who is unhappy when he is hungry (rule r_{11}) and who is indeed hungry (fact r_{12}).

$$\begin{array}{rrcl}
r_1: & happy(X) & \leftarrow & sees(X,Y), friend(X,Y), happy(Y) \\
r_2: & happy(X) & \leftarrow & sees(X,Y), enemy(X,Y), unhappy(Y) \\
r_3: & unhappy(X) & \leftarrow & sees(X,Y), friend(X,Y), not\ happy(Y) \\
r_4: & unhappy(X) & \leftarrow & sees(X,Y), enemy(X,Y), happy(Y) \\
r_5: & happy(X) & \leftarrow & friend(X,Y), friend(X,Z), Y \neq Z \\
r_6: & sees(X,Y) \vee not\ sees(X,Y) & \leftarrow & \\
r_7: & friend(X,Y) \vee not\ friend(X,Y) & \leftarrow & \\
r_8: & enemy(X,Y) \vee not\ enemy(X,Y) & \leftarrow & \\
r_9: & & \leftarrow & happy(X), unhappy(X) \\
r_{10}: & & \leftarrow & friend(X,Y), enemy(X,Y) \\
r_{11}: & unhappy(j) & \leftarrow & hungry(j) \\
r_{12}: & hungry(j) & \leftarrow &
\end{array}$$

The program obtained by deleting the last two rules from the FoLP P described above is a CoLP: it contains no longer any reference to the only constant occurring in P, j.

In the following, for a FoLP/CoLP P, we will denote with $upreds(P)$ and $bpreds(P)$ the sets of unary and binary predicates, respectively, which occur in P. For S being a set of (possibly negated) unary predicates and t being a term, let $S(t) = \{p(t) \mid p \in S\}$. When S is a set of (possibly negated) binary predicates and t and v are terms: $S(t,v) = \{f(t,v) \mid f \in S\}$. Also, for every non-free predicate q and FoLP/CoLP P, let P_q be the set of rules of P that have q as a head predicate.

For a unary rule r of type (2.3), the degree of r, denoted by $degree(r)$, is the number (k) of successor variables which appear in the rule. Intuitively, the degree of a rule indicates the maximum number of successor individuals needed to make the body of the rule true. The degree of a free rule is 0. Also the degree of a constraint is 0.[3]

For a unary predicate p:

$$degree(p) = max\{degree(r) \mid p \in head(r)\}.$$

Finally, the degree of a CoLP/FoLP P is defined as:

$$degree(P) = \sum_{p \in upreds(P)} degree(p).$$

The degree of a CoLP/FoLP P is an over-approximation of the maximum number of successor individuals needed to satisfy all atoms of the form $p(x)$, where $p \in upreds(P)$, for a given individual x.

[3]Intuitively, a violated constraint can never be 'made safe' by introducing more constraints, in particular new successor individuals.

As anticipated, constraints can be left out of both fragments without losing expressivity. Indeed, a constraint:

$$\leftarrow body$$

can be replaced by a rule of the form

$$constr(x) \leftarrow not\ constr(x), body,$$

where $constr$ is a new predicate. As the elimination of constraints in this manner potentially changes the rank of the CoLP/FoLP under consideration, whenever we refer to the rank of a CoLP/FoLP we consider the rank of the program before constraint elimination.

Example 13. Consider again rule r_5 of the FoLP described in Example 12:

$$r_5 : happy(X) \leftarrow friend(X, Y), friend(X, Z), Y \neq Z$$

The rule is a unary rule with head term X, and $k = 2$, i.e., there are two successor terms, variables Y and Z. In this case $\beta = \emptyset$, $\gamma_1 = \gamma_2 = \{friend\}$, $\delta_1 = \delta_2 = \emptyset$, and $\psi = \{Y \neq Z\}$. There is an atom which links X with each of the successor terms Y and Z: $friend(X, Y)$ and $friend(X, Z)$, respectively.

The degree of rule r_5 is 2. Besides r_5, the predicate $happy$ occurs also in the head of rules r_1 and r_2. However, as r_1 and r_2 contain each just one successor variable, the degree of $happy$ is the same as the degree of r_5: 2. The only other predicate with degree different from 0 is $unhappy$: $degree(unhappy) = 1$. As such, the rank of P is 3.

The main reasoning task which will be investigated in this thesis is:

- **Satisfiability checking of unary predicates with respect to a FoLP/CoLP**: as mentioned in Section 2.3 this enables us to check consistency, brave entailment and skeptical entailment of ground atoms as well. It is also trivial to see that satisfiability checking of a binary predicate f with respect to a CoLP/FoLP P can be reduced to satisfiability checking of a unary predicate p with respect to the program:

  $$P \cup \{p(X) \leftarrow f(X, Y)\},$$

  where p does not occur in P.

In [Heymans et al., 2007] satisfiability checking of unary predicates with respect to CoLPs under the IWA (a form of CoLPs augmented with so-called inverted predicates) has been reduced to checking emptiness of two-way alternating tree automata with a parity condition. Also there, it has been shown that satisfiability checking with respect to $\mathcal{SHIQ}$ KBs can be reduced to reasoning with CoLPs and as such that reasoning with CoLPs under the IWA is an EXPTIME-complete problem. It is easy to see how in the absence of inverse predicates the reduction holds for the DL $\mathcal{SHQ}$ for which satisfiability checking is also an EXPTIME-complete problem [Schild, 1991]. Thus:

Proposition 1. Satisfiability checking of unary predicates with respect to CoLPs is EXPTIME-complete.

2.4.2 Local and Acyclic FoLPs

When this work started, the decidability status of satisfiability checking of unary predicates with respect to full FoLPs was not known. Decidability had been shown only in the case of the restricted fragments: *local FoLPs* and *acyclic FoLPs*. These fragments are formally defined as follows:

Definition 6 (after [Heymans, 2006])**.** A FoLP is *local* if:

- for every rule of type 2.3 as in Definition 4, the set $\delta_m^+(t_m)$ is empty whenever t_m is not a constant, and
- for every rule of type 2.4 as in Definition 4 such that t is not a constant, the set δ^+ is empty.

Example 14. The FoLP introduced in Example 12 can be 'adapted' into a local FoLP as follows:

$$\begin{array}{rrcl}
r_1: & happy(X) & \leftarrow & sees(X,Y), friend(X,Y), \\
 & & & not\ unhappy(Y) \\
r_2: & happy(X) & \leftarrow & sees(X,Y), enemy(X,Y), \\
 & & & not\ happy(Y) \\
r_3: & unhappy(X) & \leftarrow & sees(X,Y), friend(X,Y), \\
 & & & not\ happy(Y) \\
r_4: & unhappy(X) & \leftarrow & sees(X,Y), enemy(X,Y), \\
 & & & not\ unhappy(Y) \\
r_5: & happy(X) & \leftarrow & friend(X,Y), friend(X,Z), \\
 & & & Y \neq Z \\
r_6: & sees(X,Y) \vee not\ sees(X,Y) & \leftarrow & \\
r_7: & friend(X,Y) \vee not\ friend(X,Y) & \leftarrow & \\
r_8: & enemy(X,Y) \vee not\ enemy(X,Y) & \leftarrow & \\
r_9: & & \leftarrow & happy(X), unhappy(X) \\
r_{10}: & & \leftarrow & friend(X,Y), enemy(X,Y) \\
r_{11}: & unhappy(j) & \leftarrow & hungry(j) \\
r_{12}: & hungry(j) & \leftarrow &
\end{array}$$

Some atoms in the successor part of rules r_1, r_2, and r_4 have been replaced with negated literals, e.g. atom $happy(Y)$ has been replaced with literal $not\ unhappy(Y)$. Note that the two programs, the original FoLP and the local FoLP, are not equivalent: for example, the infinite universe $\{x_1, x_2, x_3, \ldots\}$ and the infinite interpretation $\{happy(x_1), friend(x_1, x_2), sees(x_1, x_2), happy(x_2), friend(x_2, x_3), sees(x_2, x_3), \ldots\}$ form an open answer set of the local FoLP, but they do not form an open answer set of the FoLP in Example 12.

In order to introduce the fragment of acyclic FoLPs we introduce first the notion of positive predicate dependency graph of an OASP P:

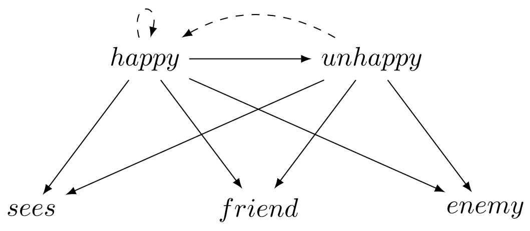


Figure 2.5: The positive dependency graph

Definition 7. Let P be an OASP. Its positive predicate dependency graph $PDG(P) = (V, A)$ is the following graph:

- $V = preds(P)$,
- $(p, q) \in A$ if there exists a rule $\alpha \leftarrow \beta \in P$ and some atoms $a_1 \in \alpha$ and $a_2 \in \beta$ such that $pred(a_1) = p$ and $pred(a_2) = q$.

Definition 8 (after [Heymans, 2006])**.** A FoLP is *acyclic* if its positive predicate dependency graph $PDG(P)$ is acyclic.

Example 15. The program P' obtained by dropping rules r_1 and r_4 from the FoLP P introduced in Example 12 is acyclic. Its positive predicate dependency graph is depicted in Figure 2.5: the solid lines in the figure represent the arcs of this graph.

However, by adding either rule r_1 or r_4 to P' (see the dotted lines in Figure 2.5), the positive predicate dependency graph of P' is no longer acyclic, and as such P' is no longer acyclic either.

In [Heymans, 2006] it has been shown that acyclic FoLPs can be rewritten into programs which do not contain any positive unary atoms in the bodies of rules, thus into local FoLPs. Also there, it has been shown that satisfiability checking of concepts in the DL $\mathcal{ALCHOQ}$ can be reduced to satisfiability checking of unary predicates in an acyclic FoLPs. A corollary of this is that:

Proposition 2. Satisfiability checking of unary predicates with respect to acyclic/local FoLPs is EXPTIME-hard.

As concerns upper bounds for reasoning with acyclic/local FoLPs, the following has been established in [Heymans, 2006]:

Proposition 3. Satisfiability checking of unary predicates with respect to acyclic/local FoLPs is in 2EXPTIME$^{\Sigma_2^P}$.

2.4.3 Tree and Forest Model Property

For an open answer set program, the tree model property is as follows: *if a unary predicate p is satisfiable, then there exists a model which satisfies p that can be seen as a labelled tree which contains p in the label of the root of the tree*. Formally:

Definition 9. Let P be an open answer set program. A predicate $p \in \mathit{upreds}(P)$ is *tree satisfiable* with respect to P if there exist an open answer set (U, M) of P; a tree T_ε, with ε being an anonymous individual; and a labelling function $\mathcal{L} : T_\varepsilon \cup A_{T_\varepsilon} \to 2^{\mathit{preds}(P)}$ such that:

- $p \in \mathcal{L}(\varepsilon)$,
- $U = T_\varepsilon$, and
- $\mathcal{L}(x) \in 2^{\mathit{upreds}(P)}$, when $x \in T_\varepsilon$,
- $\mathcal{L}(x) \in 2^{\mathit{bpreds}(P)}$, when $x \in A_{T_\varepsilon}$,
- $M = \{p(x) \mid x \in T_\varepsilon, p \in \mathcal{L}(x)\} \cup \{f(x, y) \mid (x, y) \in A_{T_\varepsilon}, f \in \mathcal{L}(x, y)\}$, and
- for every $(z, z \cdot i) \in A_{T_\varepsilon}$: $\mathcal{L}(z, z \cdot i) \neq \emptyset$.

We call such a (U, M) a *tree model*

Definition 10. An OASP P has the *tree model property* if the following property holds:

- if a unary predicate p is satisfiable with respect to P, then p is tree satisfiable with respect to P.

Proposition 4 ([Heymans et al., 2007])**.** CoLPs have the tree model property.

Example 16. Let P' be the CoLP obtained from the FoLP P introduced in Example 12 by deletion of the last two rules, r_{11} and r_{12}. Then the interpretation (U, M) with:

$$U = \{\varepsilon, \varepsilon 1, \varepsilon 11, \varepsilon 12\}, \text{ and}$$

$$M = \{\mathit{unhappy}(\varepsilon), \mathit{sees}(\varepsilon, \varepsilon 1), \mathit{enemy}(\varepsilon, \varepsilon 1), \mathit{happy}(\varepsilon 1), \mathit{friend}(\varepsilon 1, \varepsilon 11), \mathit{friend}(\varepsilon 1, \varepsilon 11)\}$$

is a tree-shaped open answer set for P'. Figure 2.6 depicts (U, M) as a labelled tree: note that the tree has no arcs with empty labels, i.e. every root term is connected to a successor term via a binary atom.

As it can be seen from the figure, the unary predicate $\mathit{unhappy}$ is satisfied by (U, M), as it occurs in the label of ε and as such it is tree-satisfiable.

For an open answer set program, the forest model property is as follows: *if a unary predicate p is satisfiable, then there exists a model which satisfies p that can be seen as an interconnected forest*. The forest contains for each constant in the program a tree having the constant as root, and possibly an additional tree with an anonymous root. The predicate checked to be satisfiable, p, belongs to the label of one of the root nodes. Besides the trees corresponding to constants which occur in the program, an anonymous root tree is also introduced, as p might be satisfied only in conjunction with an anonymous individual, and not a constant.

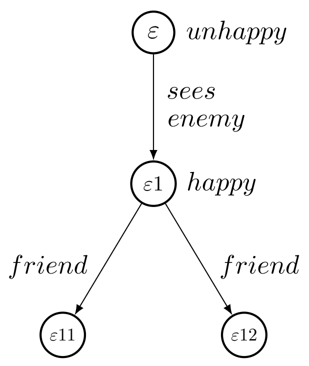


Figure 2.6: A tree model

Example 17. Consider the FoLP P which consists in the following two rules:

$$\begin{array}{rcl} q(a) & \leftarrow & p(a), not\ q(a) \\ p(X) \vee not\ p(X) & \leftarrow & \end{array}$$

While p is satisfiable with respect to P, $p(a)$ does not appear in any open answer set of P.

Formally, the notion of forest satisfiability is defined as follows:

Definition 11. Let P be a program. A predicate $p \in upreds(P)$ is *forest satisfiable* with respect to P if there exist an open answer set (U, M) of P; an interconnected forest $EF \equiv (\{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES)$, where ε is a constant, possibly one of the constants occurring in P; and a labelling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$ such that:

- $p \in \mathcal{L}(\varepsilon)$,
- $U = N_{EF}$,
- $\mathcal{L}(x) \in 2^{upreds(P)}$, when $x \in T_\varepsilon \cup \{T_a \mid a \in cts(P)\}$,
- $\mathcal{L}(x) \in 2^{bpreds(P)}$, when $x \in A_{T_\varepsilon}$,
- $M = \{\mathcal{L}(x)(x) \mid x \in N_{EF}\} \cup \{\mathcal{L}(x, y)(x, y) \mid (x, y) \in A_{EF}\}$, and
- for every $(z, z \cdot i) \in A_{EF}$: $\mathcal{L}(z, z \cdot i) \neq \emptyset$.

We call such a (U, M) a *forest model* and a program P has the *forest model property* if the following property holds:

If a unary predicate p is satisfiable with respect to P then p is forest satisfiable with respect to P.

Proposition 5 ([Heymans et al., 2007]). FoLPs have the forest model property.

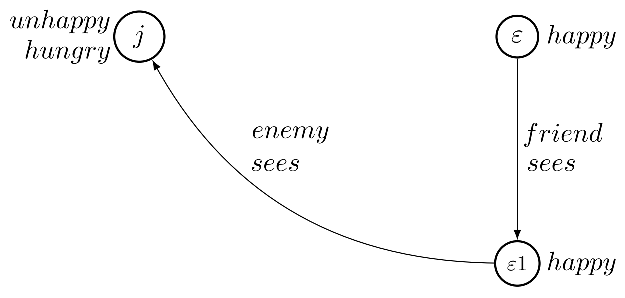


Figure 2.7: A forest model

Example 18. Let P be the FoLP introduced in Example 12. The interpretation (U, M), with: $U = \{j, \varepsilon, \varepsilon 1\}$ and $M = \{unhappy(j), hungry(j), happy(\varepsilon), sees(\varepsilon, \varepsilon 1), friend(\varepsilon, \varepsilon 1), happy(\varepsilon 1), enemy(\varepsilon 1, j), sees(\varepsilon 1, j)\}$, is an open answer set for P which satisfies the unary predicate $happy$: $happy \in \mathcal{L}(x)$. Furthermore, as it can be seen in Figure 2.7, (U, M) is a forest model of P: the figure depicts an interconnected labelled forest EF composed of two trees, one with root j, the constant appearing in the program, and the other one with root ε, where ε is an anonymous individual:

$$EF = (\{T_\varepsilon, T_j\}, \{(\varepsilon 1, j)\}).$$

The set of nodes of the forest coincides with U and every predicate symbol corresponding to some atom in M is in the label of the argument of the atom, e.g. $unhappy \in \mathcal{L}(j)$. The converse also holds: every node/arc of the interconnected forest in conjunction with every predicate symbol in its label forms an atom which is part of the interpretation.

In the following chapters, many times terms which occur in tree/forest models of CoLPs/FoLPs are nodes in some interconnected/extended forest, and as such they are sequences formed with the $\cdot$ operator. Taking into account the structure of such terms, a finer grained (ground) term replacement operator is introduced which replaces the prefix of a term with another term. The *replacement* of x with y in t, denoted by $t_{x||y}$, is defined as:

$$t_{x||y} = \begin{cases} y \cdot z, & \text{if } t = x \cdot z; \\ t, & \text{otherwise} \end{cases}.$$

Similarly to the exact replacement operator introduced in Section 2.2, the current notion of replacement is extended to (sets of) literals, tuples, rules, and programs.

Example 19. The *replacement* of $a \cdot 1$ with b in $a \cdot 1 \cdot 2$, denoted by $(a \cdot 1 \cdot 2)_{a \cdot 1||b}$ is the term $b \cdot 2$. The *replacement* of $a \cdot 1$ with b in $a \cdot 2$, denoted by $(a \cdot 2)_{a \cdot 1||b}$ is the term $a \cdot 2$. Consequently, the *replacement* of $a \cdot 1$ with b in $p(a \cdot 1 \cdot 2, a \cdot 2)$ is $p(b \cdot 2, a \cdot 2)$.

CHAPTER 3

Tableau Algorithm for Reasoning with Forest Logic Programs

In this chapter we describe a sound, complete, and terminating tableau algorithm for satisfiability checking with respect to FoLPs. While not worst-case optimal, the algorithm was the first to deal with the whole fragment of FoLPs, and, as such, established the decidability of the fragment: as mentioned in the introduction, decidability had been previously shown only for restricted fragments of FoLPs [Heymans et al., 2007]. It also serves as the basis for a knowledge compilation technique which will be described in next chapter. Throughout this section, and also in the rest of this work, we will refer to the algorithm as $\mathcal{A}_1$.

The algorithm exploits the forest model property of Forest Logic Programs: given a FoLP P and a unary predicate p, it tries to construct a forest model of P which satisfies p.

Section 3.1 introduces the data structure underlying the algorithm, called $\mathcal{A}_1$-completion structure, and describes how such a structure is initialized when checking satisfiability of a distinguished unary predicate p with respect to a FoLP P.

Section 3.2 describes how the structure is evolved by means of so-called expansion rules which justify the presence or absence of certain atoms in the model by asserting the presence or absence of other atoms in the model, as well.

Some other rules are employed to specify an expansion strategy: such rules prescribe which atoms should be considered first when applying the expansion rules; when a node on a branch is terminal, i.e. the unary literals having as argument the node should no longer be expanded; or when the expansion of the completion structure should be aborted due to an encountered inconsistency. These rules are called applicability rules and are described in Section 3.3.

Section 3.4 describes when the expansion is complete and successful, i.e. the completion structure can be unravelled to an actual open answer set.

The proofs for termination, soundness, and completeness of $\mathcal{A}_1$ are provided in Section 3.5. In the same section we also provide a complexity analysis for the worst-case running time of the algorithm and show as a corollary of the results obtained in this chapter that Forest Logic Programs have the finite bounded model property.

An extended example which shows how the algorithm works is provided in Section 3.6.

Finally, Section 3.7 discusses the results obtained in this chapter and relates them to existing work in the areas of tableau algorithms for Description Logics and proof systems for Answer Set Programming.

3.1 Completion Structures

The basic data structure used by the algorithm $\mathcal{A}_1$ is a so-called $\mathcal{A}_1$-*completion structure*. An $\mathcal{A}_1$-completion structure describes a forest model in construction. As such, the main components of the structure are an interconnected forest EF, whose set of nodes constitutes the forest-shaped universe of the open answer set in construction, and a labeling function $\textsc{ct}$, which assigns to every node, resp. arc of EF, a set of possibly negated unary, resp. binary predicates, called the *content* of the given node/arc.

The presence of a predicate symbol/negated predicate symbol in the content of some node or arc indicates the presence/absence in the forest model in construction of the atom formed using that predicate and having the current node or arc as argument. Note that unlike the labeling function $\mathcal{L}$ in Definition 11, which describes which atoms are in the forest model, the labeling function $\textsc{ct}$ keeps track also of which atoms are not in the forest model. This is needed as the forest model is updated by justifying the presence or absence of each atom in the Herbrand base of the program in the open answer set.

The presence (absence) of an atom in a forest model in construction is justified by imposing that the body of at least one ground rule which has the respective atom in the head is satisfied (no body of a rule which has the respective atom in the head is satisfied). In order to keep track which (possibly negated) predicate symbols in the content of some node or arc have already been justified, a so-called *status* function is introduced. The status function $\textsc{st}$ assigns the value unexp to pairs of nodes/arcs and possibly negated unary/binary predicates which have not yet been 'expanded', i.e. justified, and the value exp to such pairs which have already been considered.

Furthermore, in order to ensure that no atom in the model is circularly justified (does not depend on itself) or infinitely justified (does not depend on an infinite chain of other atoms), a graph G which keeps track of dependencies between atoms in the model is maintained.

Definition 12. An $\mathcal{A}_1$-*completion structure for a FoLP P* is a tuple $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$ where:

- $EF = \langle F, ES\rangle$ is an interconnected forest, its set of nodes being the universe of the forest model in construction,
- $\textsc{ct} : N_{EF} \cup A_{EF} \to 2^{preds(P) \cup not\ (preds(P))}$ is the 'content' function which maps a node of the interconnected forest to a set of (possibly negated) unary predicates and an arc of the interconnected forest to a set of (possibly negated) binary predicates such that:
  - $\textsc{ct}(x) \subseteq upreds(P) \cup not(upreds(P))$ if $x \in N_{EF}$, and
  - $\textsc{ct}(x) \subseteq bpreds(P) \cup not(bpreds(P))$ if $x \in A_{EF}$,

- $\text{ST} : \{(x, \pm q) \mid \pm q \in \text{CT}(x), x \in N_{EF} \cup A_{EF}\} \to \{exp, unexp\}$ is the 'status' function which indicates which predicates in the content of some node/arc are already justified, and which are not,
- $G = (V, A)$ is a directed graph with:
  - $V \subseteq atoms(P_{N_{EF}})$, and
  - $A \subseteq atoms(P_{N_{EF}}) \times atoms(P_{N_{EF}})$,

For checking satisfiability of a unary predicate p with respect to a FoLP P, a skeleton of an $\mathcal{A}_1$-completion structure called $\mathcal{A}_1$-*initial completion structure* is created as follows: the interconnected forest EF is initialized with the set of single-node trees having as root a constant appearing in P and possibly a new single-node tree with an anonymous root.

The root ε of one of the single-node trees in EF is (non-deterministically) set to be distinguished and its content is initialized with $\{p\}$, the predicate checked to be satisfiable. The contents of the other nodes (roots) are initialized with $\emptyset$. G is initialized to the graph with a single node $p(\varepsilon)$.

Definition 13. An $\mathcal{A}_1$-*initial completion structure* for checking satisfiability of a unary predicate p with respect to a FoLP P is an $\mathcal{A}_1$-completion structure $\langle EF, \text{CT}, \text{ST}, G\rangle$ where:

- $EF = \langle F, ES\rangle$,
- $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where:
  - ε is a constant, possibly in $cts(P)$,
  - $T_x = \{x\}$, for every $x \in cts(P) \cup \{\varepsilon\}$,
  - $ES = \emptyset$,
- $G = \langle V, A\rangle$, $V = \{p(\varepsilon)\}$, $A = \emptyset$,
- $\text{CT}(\varepsilon) = \{p\}$, and
- $\text{ST}(\varepsilon, p) = unexp$.

3.2 Expansion Rules

Expansion rules update a completion structure by making explicit constraints which are necessary to hold for a certain literal to be part of a forest model.

Every atom which is part of an open answer set has to be 'supported', i.e. there must be a ground rule which has the atom in the head, and whose body is satisfied by the model. The *Expand-Unary/Binary-Positive* rules enforce this condition for every positive unary/binary atom in the label of a node/arc of the constructed model. In the process, the (i) Expand-Unary-Positive rule might introduce new domain elements as successors of the current node and both rules assert new constraints in the model which are needed to satisfy the bodies of the corresponding ground rules by means of updating the contents of existing and/or newly created nodes and arcs.

Conversely, for an atom not to be part of an open answer set, it must be the case that the bodies of all ground rules which have as head the atom are not satisfied by the model. The expansion rules which check/enforce this condition are the *Expand-Unary/Binary-Negative* rules. Note that in the case of unary atoms, the condition can be fully checked only when all possible successors of the argument of the considered atom are known. As long as new successors are introduced new groundings can be obtained which might lead to a ground rule whose body is satisfied and whose head is the given atom. As such, there is an interaction between the (iii) Expand-Unary-Negative rule and other expansion rules and an explicit order of the application of these rules must be enforced.

Newly introduced domain elements might lead to ground rules whose head and body literals are unrelated to the constraints in the partially constructed model, but which nonetheless might render the program inconsistent. In order to be sure that the partially constructed model can be extended to an actual model, every ground atom which can be formed with the given domain elements and unary/binary predicates in the given program has to either be part or not to be part of the forest model. If a ground atom is not constrained in this respect, a random choice is made. The expansion rules which make such choices are the *Choose-Unary/Binary* rules.

Before formally introducing the expansion rules, we introduce a sequence of operations which all rules will make extensive use of. This refers to the operations needed to enforce the presence of a literal $\pm p(z)$ in the open answer set in construction (where z is a term in case $p \in \mathit{upreds}(P)$, and a pair of terms in case $p \in \mathit{bpreds}(P)$) as part of supporting the presence of another literal l. Informally, these consist in (1) inserting $\pm p$ in the content of z and mark it as unexpanded, in case the predicate symbol is not already there, (2) in case $\pm p(z)$ is an atom, ensuring that it is a node in G, and (3) if l is also an atom, creating a new arc from l to $\pm p(z)$ to capture the dependencies between the two elements of the forest model. Formally:

1. let $\textsc{ct}(z) := \textsc{ct}(z) \cup \{\pm p\}$ and $\textsc{st}(z, \pm p) := \mathit{unexp}$,
2. if $\pm p = p$, then let $V := V \cup \{p(z)\}$,
3. if $l \in \mathit{atoms}(P_{N_{EF}})$ and $\pm p = p$, then let $A := A \cup \{(l, p(z))\}$.

As a shorthand, we denote this sequence of operations as $\mathit{update}(l, \pm p, z)$; more general, $\mathit{update}(l, \beta, z)$ for a set of (possibly negated) predicates β, denotes the set of operations[1]: $\{\mathit{update}(l, \pm a, z) \mid \pm a \in \beta\}$.

In the following, for an $\mathcal{A}_1$-completion structure $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$, let $x \in N_{EF}$ and $(x, y) \in A_{EF}$ be the node, respectively arc, under consideration.

3.2.1 Expanding a Unary Positive Predicate

This rule is employed to support the presence of a unary atom $p(x)$ in the tentative open answer set, or in other words, of the predicate symbol p in the content of node x.

Rule. ***(i) Expand-Unary-Positive.*** Let $p \in \textsc{ct}(x)$ be a unary positive predicate such that $\textsc{st}(x, p) = \mathit{unexp}$. If p is not a free predicate symbol:

[1]In any future context, the order of application of the operations in the set is irrelevant.

- non-deterministically choose a rule $r \in P_p$ of the form (2.3) such that s matches x and do all of the following:
- $\mathit{update}(p(x), \beta, x)$,
- for every $1 \leqslant m \leqslant k$, non-deterministically choose or introduce a successor y_m for x such that:
  - for every $1 \leqslant (i, j) \leqslant k$ such that $t_i \neq t_j \in \psi$, it holds that $y_i \neq y_j$

  and for every $1 \leqslant m \leqslant k$ one of the following holds:
  - $y_m \in \mathit{succ}_{EF}(x)$, or
  - y_m is defined as a new successor of x in the tree T_c, where $x \in T_c$: $y_m := x \cdot n$, where $n \in \mathbb{N}_{>0}$ such that $x \cdot n \notin \mathit{succ}_{EF}(x)$, and $T_c := T_c \cup \{y_m\}$, or
  - y_m is defined as a new successor of x in EF in the form of a constant: $y_m := a$, where a is a constant from $\mathit{cts}(P)$ such that $a \notin \mathit{succ}_{EF}(x)$. In this case also add (x, a) to ES: $ES := ES \cup \{(x, a)\}$,
- for every successor y_m introduced at the previous step do all of the following:
  - $\mathit{update}(p(x), \gamma_m, (x, y_m))$, and
  - $\mathit{update}(p(x), \delta_m, y_m)$.

Set $\text{ST}(x, p) := \mathit{exp}$.

If p is free, its status in the content of x is simply marked as expanded, as the presence of $p(x)$ in the forest model in construction is trivially justified by the free rule which defines p grounded with x.

3.2.2 Choosing a Unary Predicate

This rule makes a choice for a unary predicate symbol p, whether it appears in positive form (p) or in negated form ($\mathit{not}\ p$) in the content of node x.

Rule. ***(ii) Choose-Unary.*** If there exists a unary predicate $p \in \mathit{upreds}(P)$ such that:

- $p \notin \text{CT}(x)$,
- $\mathit{not}\ p \notin \text{CT}(x)$,
- for every (positive) unary predicate $q \in \text{CT}(x)$: $\text{ST}(x, q) = \mathit{exp}$, and
- for every arc $(x, y) \in A_{EF}$ and for every (possibly negated) binary predicate $\pm f \in \text{CT}(x, y)$ (both positive and negative predicates): $\text{ST}((x, y), \pm f) = \mathit{exp}$,

then do one of the following:

- let $\text{CT}(x) := \text{CT}(x) \cup \{p\}$ and $\text{ST}(x, p) := unexp$, or
- let $\text{CT}(x) := \text{CT}(x) \cup \{not\ p\}$ and $\text{ST}(x, not\ p) = unexp$.

In other words, if there are still unary predicates which do not appear in $\text{CT}(x)$ (either in a positive or a negated form), and all positive predicates in the content of x have been justified, as well as all positive or negative predicates in the content of one of the arcs starting in x have been justified, one has to non-deterministically choose such a unary predicate symbol p and inject either p or $not\ p$ in $\text{CT}(x)$.

As mentioned in the introduction to this section, this rule is needed in order to ensure that the partially constructed forest model is part of an actual model: as a result of introducing new domain elements in the process of constructing a forest model, there might be ground rules whose heads are not relevant per se for the satisfiability task at hand, but which are not satisfiable in any total extension of the partial forest model.

We try to effectively construct such an extension of the partial model by making a random choice for unconstrained ground atoms regarding their membership to model.

Example 20. Let P be the following FoLP:

$$\begin{array}{rcl} a(X) \lor not\ a(X) & \leftarrow & \\ b(X) & \leftarrow & not\ b(X). \end{array}$$

Suppose we want to check whether a is satisfiable: an initial completion structure will be created with only one tree with anonymous root ε. While it is trivial to see that $a(\varepsilon)$ is justified by the first rule of P, the program has no open answer sets due to the inconsistency introduced by the second rule. This will be tracked down by the algorithm by trying to prove successively both $b(\varepsilon)$ and $not\ b(\varepsilon)$ (after each of them is inserted in the content of ε as a result of applying the ii) Choose-Unary rule, and failing in each case.

For reasons described in the next subsection, this rule has priority over the rule which describes the expansion of unary negative predicates.

3.2.3 Expanding a Unary Negative Predicate

In general, for justifying that a negative unary literal $not\ p$ belongs to the content of a node x (or in other words, the absence of $p(x)$ in the constructed forest model), one has to refute the body of every ground (non-free) rule with head atom $p(x)$. Let $r \in P_p$ and

$$r' : p(x) \leftarrow \beta(x), (\gamma_m(x, y_m), \delta_m(y_m))_{1 \leqslant m \leqslant k}, \psi, \text{ with}$$

$\psi \subseteq \bigcup_{1 \leqslant i \neq j \leqslant k} \{y_i \neq y_j\}$, and $k \in \mathbb{N}$, be a ground version of r. The body of r' can be either:

- (i) 'locally' refuted: by refutation of a literal from $\beta(x)$. For this, one has to enforce that there is a $\pm q \in \beta$ which does not appear in $\text{CT}(x)$, or in other words: $\mp q \in \text{CT}(x)$; note that this refutes all ground versions of r where the head variable is substituted with x.

- (ii) refuted in the 'successor' part of the rule: by refutation of a literal from one of $(\gamma_m(x, y_m))_{1 \leqslant m \leqslant k}$ or $(\delta_m(y_m)))_{1 \leqslant m \leqslant k}$, or by impossibility to satisfy ψ.

  In a forest model, all groundings of r, in which one of the successor terms has been substituted with y, where y is a node in the forest which is not a direct successor of x, are refuted: there is no arc which links x to y, and as such there are no literals of the form $f(x, y)$ with $f \in bpreds(P)$ in the constructed open answer set.

  Thus, one has to consider only groundings in which $(y_m)_{1 \leqslant m \leqslant k}$ are successors of x in EF: $(y_m = x \cdot z_m)_{1 \leqslant m \leqslant k}$, and which satisfy ψ. For such ground rules, their body can be refuted by enforcing that there exists a (negated) predicate symbol $\pm f \in \delta_m$ which does not appear in $\text{CT}(x, x \cdot z_m)$ (this is the same as $\mp f \in \text{CT}(x, x \cdot z_m)$) or that there exists a (negated) predicate symbol $\pm q \in \gamma_m$ which does not appear in $\text{CT}(x \cdot z_m)$ (this is the same as $\mp q \in \text{CT}(x \cdot z_m)$), for some $1 \leqslant m \leqslant k$.

As we want to refute the bodies of all ground versions of r, we either apply (i) once, or apply (ii) for every assignment of successor terms in r with successors of x in EF which satisfies ψ. As ψ imposes a minimum bound on the number of distinct successor terms, if the number of successors of x in EF is smaller than this bound, there is no such assignment which satisfies ψ. In this case, all bodies of ground versions of r are refuted.

Formally:

Rule. ***(iii) Expand-Unary-Negative.*** Let $not\ p \in \text{CT}(x)$ be a unary negative predicate for which $\text{ST}(x, not\ p) = unexp$ and let $y_1, \dots, y_n$ be the successors of x in EF. If:

- for all $p \in upreds(P)$, $p \in \text{CT}(x)$ or $not\ p \in \text{CT}(x)$, and
- for all $p \in \text{CT}(x)$, $\text{ST}(p, x) := exp$,

then for every rule $r \in P_p$ of the form (2.3) such that x matches s, do one of the following:

- non-deterministically choose $\pm q \in \beta$ and $update(not\ p(x), \mp q, x)$, or
- for all $y_{i_1}, \dots, y_{i_k}$ s. t. $(1 \leqslant i_j \leqslant n)_{1 \leqslant j \leqslant k}$: if for all $1 \leqslant j, l \leqslant k$, $t_j \neq t_l \in \psi \Rightarrow y_{i_j} \neq y_{i_l}$, do one of the following:
  - for some m, $1 \leqslant m \leqslant k$, non-deterministically choose $\pm f \in \delta_m$ and $update(not\ p(x), \mp f, (x, y_{i_m}))$, or
  - for some m, $1 \leqslant m \leqslant k$, non-deterministically choose $\pm q \in \gamma_m$ and $update(not\ p(x), \mp q, y_{i_m})$.

Set $\text{ST}(x, not\ p) := exp$.

Note that the introduction of new successors of x gives rise to new ground unary rules with head $p(x)$. Such successors are introduced in the process of expanding positive unary predicates. In order to ensure that $p(x)$ is indeed refuted, this rule should be applied only when all successors of x have been introduced, i.e., when there is no possibility to further introduce and expand a positive unary predicate. Hence, the precondition of the rule which checks that neither the (i) Expand-Unary-Positive rule nor the (ii) Choose-Unary rule can be further applied with respect to a certain node x.

3.2.4 Expanding a Binary Positive Predicate

Similarly to the case of unary positive predicates in the content of nodes, the presence of binary positive predicates in the content of arcs has to be justified by means of binary rules having the respective predicates as head predicates. Formally:

Rule. ***(iv) Expand-Binary-Positive.*** Let f be a binary positive predicate symbol such that $\text{ST}((x,y),f) = \mathit{unexp}$. If f is not free, non-deterministically choose a rule $r \in P_f$ of the form (2.4) such that x matches s and y matches t and update the completion as follows:

- $\mathit{update}(p(x,y),\beta,x)$,
- $\mathit{update}(p(x,y),\gamma,(x,y))$, and
- and $\mathit{update}(p(x,y),\delta,y)$.

Set $\text{ST}((x,y),f) := \mathit{exp}$ (also in case f is free).

3.2.5 Expanding a Binary Negative Predicate

Again, the intuition for this expansion rule is similar to the intuition for the (iii) Expand-Unary-Negative rule described in Section 3.2.3. However, unlike its unary counterpart, when refuting all ground versions of rules which have the given predicate in the head, when constructing such groundings, the new rule for expanding binary negative predicates does not have to consider all successors of x, just y, the successor of x in the given arc. As such, there are no complex interactions between this rule and the (iv) Expand-Binary-Positive one.

Rule. ***(v) Expand-Binary-Negative.*** Let $\mathit{not}\ f \in \text{CT}(x,y)$ be a binary negative predicate symbol such that $\text{ST}((x,y),\mathit{not}\ f) = \mathit{unexp}$. Then, for every rule $r \in P_f$ of the form (2.4) such that x matches s and y matches t do one of the following:

- non-deterministically choose a $\pm p$ from β and $\mathit{update}(\mathit{not}\ f(x,y),\mp p,x)$, or
- non-deterministically choose a $\pm g$ from γ and $\mathit{update}(\mathit{not}\ f(x,y),\mp g,(x,y))$, or
- non-deterministically choose a $\pm q$ from δ and $\mathit{update}(\mathit{not}\ f(x,y),\mp q,y))$.

Set $\text{ST}((x,y),\mathit{not}\ f) := \mathit{exp}$.

3.2.6 Choosing a Binary Predicate

Binary rules might introduce inconsistencies in a FoLP in a similar manner as the unary rule in Example 20. As such, for each binary atom it has to be decided, too, whether the atom is part or not of the open answer set in construction.

Rule. ***(vi) Choose-Binary***. If there exists a binary predicate $f \in \mathit{bpreds}(P)$ such that:

- $f \notin \text{CT}(x,y)$,

- $not\ f \notin \textsc{ct}(x, y)$,
- for every (possibly negated) unary predicate $\pm a \in \textsc{ct}(x)$, $\pm a$ cannot be expanded according to expansion rules (i) and (iii), and
- for every arc $(x, y) \in A_{EF}$, and (possibly negated) binary predicate $\pm f \in \textsc{ct}(x, y)$, $\pm f$ cannot be expanded according to rules (iv) and (v),

then do one of the following:

- add f to $\textsc{ct}(x, y)$ and let $\textsc{st}((x, y), p) := unexp$, or
- add $not\ f$ to $\textsc{ct}(x, y)$ and let $\textsc{st}((x, y), not\ p) := unexp$.

3.3 Applicability Rules

While expansion rules ensure that an atom is part of the constructed model iff there exists a ground rule whose body is satisfied as well in the model, a second set of rules called *applicability rules* governs the expansion of a completion structure by specifying the circumstances in which these rules are actually applicable. In particular, they ensure the (successful or unsuccessful) termination of the algorithm.

3.3.1 Saturation

This rule imposes an order on the expansion of unary/binary predicates in the contents of nodes/arcs of the interconnected forest. An expansion rule should be applied with respect to a (negated) unary/binary predicate symbol in the content of a node/arc iff all possible expansion rules have been applied with respect to (negated) unary/binary predicates in the contents of all nodes/arcs which are ‘above’ the current node/arc in the structure. A node is said to be ‘saturated’ if this is the case for the node itself and all of its outgoing arcs:

Definition 14. A node $x \in N_{EF}$ is *saturated* iff:

- for all $p \in upreds(P)$, it is the case that $p \in \textsc{ct}(x)$ or $not\ p \in \textsc{ct}(x)$,
- for all $\pm p \in \textsc{ct}(x)$, $\textsc{st}(x, p) = exp$,
- for all $(x, y) \in A_{T_c}$, where T_c is a tree in EF, and $f \in bpreds(P)$, it is the case that either $f \in \textsc{ct}(x, y)$ or $not\ f \in \textsc{ct}(x, y)$, and
- for all $\pm f \in \textsc{ct}(x, y)$, $\textsc{st}((x, y), p) = exp$.

Rule. ***(vii) Saturation***. An expansion rule is applicable with respect to a given node/arc $x \in N_{EF}$/$(x, y) \in A_{EF}$, and (possibly negated) unary/binary predicate symbol $p/f \in \textsc{ct}(x)$/$\textsc{ct}(x, y)$, only if for every $y <_F x$, y is saturated.

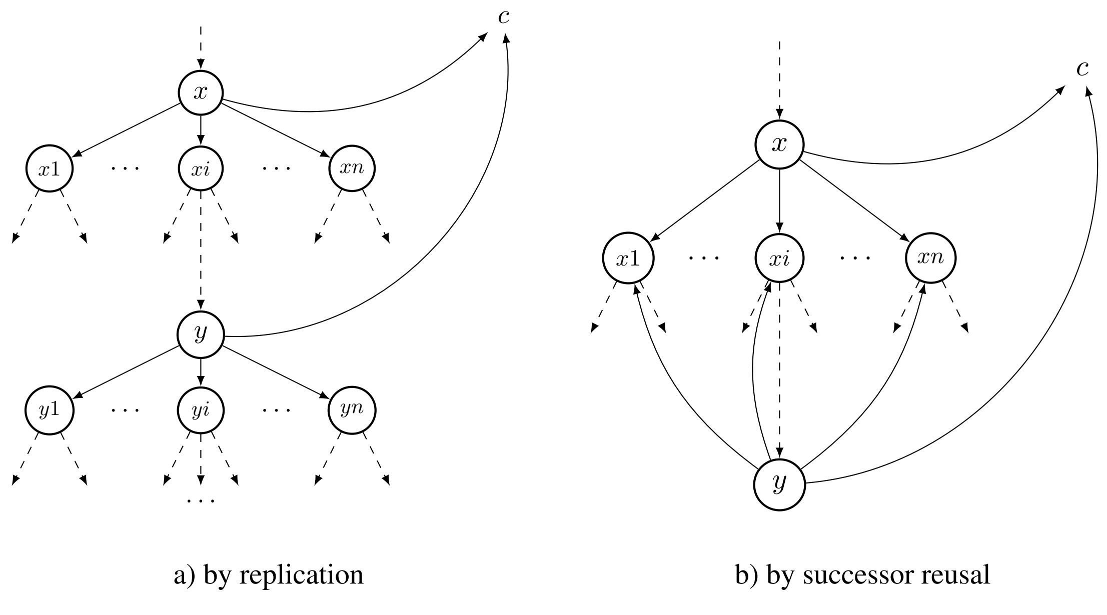


Figure 3.1: Justifying the content of a blocked node y in a similar manner to the content of its corresponding blocking node x

3.3.2 Blocking

As forest models can potentially have infinite sizes, as usually with tableau algorithms, a mechanism to identify repetitions in the model is employed.

Definition 15. A node $x \in N_{EF}$ is *blocked* iff there exists an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, such that:

- $\text{CT}(x) \subseteq \text{CT}(y)$, and
- the set $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in conn_G \wedge q$ is not free$\}$ is empty.

We say that (y, x) is a *blocking pair* and y is a *blocking node*.

A blocked node is no longer expanded (it is expanded 'by default' due to its status):

Rule. ***(viii) Blocking***. Let x be a blocked node. Then, set $\text{ST}(x) := exp$.

The blocking mechanism which we employ uses subset blocking: if there exists an ancestor y of x which is not a constant, whose content includes the content of x, it is possible to extend the partial interpretation such that the contents of x and its outgoing arcs are identical to the contents of y and its outgoing arcs. The newly introduced atoms which have x as an argument will be justified in a similar way as their counterpart atoms which have y as an argument. This can be done by either:

1. reusing the successors of y as successors of x: this consists in the introduction of 'backward' arcs in the interconnected forest from the leaf node x to the said successors. The contents of these backward arcs will replicate the content of their counterpart arcs from y to its successors. The interpretation thus obtained is no longer a forest shaped one. This is the approach we consider for proving the soundness of the algorithm, it is depicted in Figure 3.1 a), and further exemplified in Section 3.5.2.

2. or introducing new successors for x which are similar to the successors of y and which at their turn will be justified similarly to the successors of y, and so on. In this case, one obtains an infinite forest interpretation. This approach is depicted in Figure 3.1 b) and further exemplified in Section 3.6.

However, in order for the interpretation constructed in one of the above ways to be a forest model, the subset blocking condition is not sufficient: if some atom formed with the blocked node depends on an atom formed with the blocking node (according to the dependency graph G), by applying one of the techniques described above, G might contain either a cycle or a path of infinite length, in which case the constructed model is not minimal, and thus not an open answer set. As such, the blocking condition is enhanced with a check on dependencies between such atoms: there should be no path in G between any $p(y)$ and any $q(x)$.

3.3.3 Redundancy

The condition in the blocking rule might never be fulfilled during the expansion of a completion structure, even in the cases where an infinite structure is constructed as a result of applying the expansion rules. As such, some other mechanism is needed to ensure termination. This comes in the form of a rule which basically stops the expansion of a certain structure when a certain number of nodes with identical content, all in the same branch, has been encountered.

Definition 16. A node $x \in N_{EF}$ is *redundant* iff:

- it is saturated;
- it is not blocked;
- there exist k ancestors of x in F: $y_i <_F x$, for $1 \leqslant i \leqslant k$, where $k = 2^n(2^{n^2} - 1) + 2$, and $n = |\mathit{upreds}(P)|$, such that $\textsc{ct}(x) = \textsc{ct}(y_i)$.

Rule. ***(ix) Redundancy***. A redundant node is no longer expanded.

In other words, a node is redundant if there are other k nodes on the same branch with the current node which all have content equal to the content of the current node. The presence of a redundant node stops the expansion process.

The completeness proof in Section 3.5.3 shows that any forest model of a FoLP P which satisfies p can be reduced to another forest model which satisfies p and has at most $k + 1$ nodes with equal content in any branch of a tree from the forest model, and furthermore the $(k + 1)$-st node, in case it exists, is blocked. Thus, it is possible to search for forest models only of the latter type. This rule exploits that result: a redundant node signals that the limit for the model size has been reached, and thus the current model can be discarded.

Note that the number of nodes with distinct contents in a completion structure (and thus, on a branch of a completion structure) is finite and bounded: there are at most 2^n such nodes, with $n = |\mathit{upreds}(P)|$. Thus, we can rephrase the rule by imposing that the expansion is aborted when a certain depth $nk + 1$ (number of nodes, not necessarily with identical content) has been reached while expanding a certain branch. While this condition is more transparent, it would obviously lead to a less efficient algorithm than the original condition in the redundancy rule.

3.3.4 Contradictory Completion Structures

During the expansion process it is possible to introduce contradictions, in the form of pairs of unary/binary predicates and their negation in the contents of nodes/arcs. The contradiction rule takes care of this possibility.

Definition 17. An $\mathcal{A}_1$-completion structure is said to be *contradictory* iff one of the following holds:

- there exists a node $x \in N_{EF}$ and a unary predicate $a \in \mathit{upreds}(P)$ such that: $\{a, \mathit{not}\ a\} \subseteq \textsc{ct}(x)$, or
- there exists an arc $(x, y) \in A_{EF}$ and a binary predicate $f \in \mathit{bpreds}(P)$ such that: $\{f, \mathit{not}\ f\} \subseteq \textsc{ct}(x, y)$.

Rule. ***(x) Contradiction***. An $\mathcal{A}_1$-contradictory completion structure is no longer expanded.

3.3.5 Circular Completion Structures

While the blocking condition ensures that no infinite paths/cycles in G are introduced as a result of applying the blocking technique, cycles in G might still occur during the expansion process between atoms which have as their first argument the same node x (so-called 'local cycles') or between atoms from different trees in the forest (so-called 'constant cycles').

Example 21. Consider a simple program P_1 containing only the rule $p(X) \leftarrow p(X)$. It is easy to see that when constructing an $\mathcal{A}_1$-completion structure to check satisfiability of p with respect to P_1, a cycle in G will be created between $p(\varepsilon)$ and $p(\varepsilon)$, where ε is the (anonymous) root of the only tree in the completion structure (in this case as there are no constants in the program and the program has only one rule, the construction will be deterministic).

Further on, let P_2 be the following FoLP:

$$\begin{aligned} r_1 &: p(b) \leftarrow f(b, X), q(X) \\ r_2 &: q(c) \leftarrow f(c, X), p(X) \end{aligned}$$

Figure 3.2 depicts an $\mathcal{A}_1$-completion structure for P_2 for which the interconnected forest contains two trees, with roots b and c, respectively. These are also the only nodes of the completion: from the figure it can be seen that $p(b)$ and $q(c)$ hold in the tentative model depicted by the

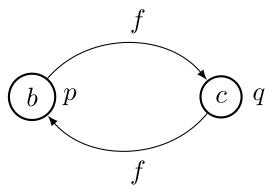


Figure 3.2: A circular completion structure for P_2

completion. We assume this is the case as the presence of the two atoms has been justified using the expand unary positive rule w.r.t. rules r_1 and r_2, where the successor variable was instantiated with c and b, respectively. Thus, in order for $p(b)$ to be satisfied it is necessary that $q(c)$ is satisfied. At the same time, in order for $q(c)$ to be satisfied it is necessary that $p(b)$ is satisfied. In other words, there is a cyclic dependency between $p(b)$ and $q(c)$: the dependency graph associated with the completion structure will contain both arcs $(p(b), q(c))$ and $(q(c), p(b))$, thus it contains a cycle. As the cycle contains arcs between atoms having as arguments nodes from different trees (and thus mandatory, constants) we call such a cycle a constant cycle.

Definition 18. An $\mathcal{A}_1$-completion structure is said to be *circular* iff G contains a cycle.

As previously explained such cycles are undesirable and for this reason:

Rule. ***(xi) Circularity***. A circular $\mathcal{A}_1$-completion structure is no longer expanded.

3.4 Overview of $\mathcal{A}_1$

First we describe when the expansion process of an $\mathcal{A}_1$-completion structure is complete, and furthermore when it is successful, i.e. it can be unravelled to an actual model.

Definition 19. A *complete $\mathcal{A}_1$-completion structure* for a FoLP P and a $p \in \mathit{upreds}(P)$ is an $\mathcal{A}_1$-completion structure that results from the application of the expansion rules (i-vi) to an $\mathcal{A}_1$-initial completion structure for p and P, taking into account the applicability rules (vii-xi), such that no expansion rule can be further applied.

As usually for tableau algorithms, conditions which clearly violate the successful construction of a completion are referred to as clashes. The following definition captures the cases in which such clashes do not occur:

Definition 20. An $\mathcal{A}_1$-completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ is *clash-free* iff:

1. for every unary/binary (possibly negated) predicate $\pm p$ in the content of some node/arc x: $\textsc{st}(p, x) = \mathit{exp}$,
2. CS is neither contradictory, nor circular,

3. EF does not contain redundant nodes.

An overview of the algorithm $\mathcal{A}_1$ which tries to construct a clash-free complete $\mathcal{A}_1$-completion structure is given by Algorithm 3.1. Note that the algorithm is non-deterministic, thus the failure of a particular run does not mean that there exists no such structure. However, a successful run means that a clash-free complete $\mathcal{A}_1$-completion structure has been constructed.

Next section will show that a predicate p is satisfiable with respect to a FoLP P iff there exists a clash-free complete $\mathcal{A}_1$-completion structure for p with respect to P.

3.5 Termination, Soundness, and Completeness

This section shows that:

1. an initial $\mathcal{A}_1$-completion structure for a unary predicate p and a FoLP P can always be expanded to a complete $\mathcal{A}_1$-completion structure – *termination* (Section 3.5.1),

2. whenever there exists a clash-free complete $\mathcal{A}_1$-completion structure for p with respect to P, p is satisfiable with respect to P – *soundness* (Section 3.5.2),

3. if p is satisfiable with respect to P, there exists a clash-free complete $\mathcal{A}_1$-completion structure – *completeness* (Section 3.5.3),

4. the algorithm runs in the worst case in non-deterministic double exponential time (Section 3.5.4), and that

5. Forest Logic Programs have the finite bounded model property (Section 3.5.5).

3.5.1 Termination of $\mathcal{A}_1$

Proposition 6. Let P be a FoLP, p be a unary predicate in P, and IC be an initial $\mathcal{A}_1$-completion structure for p with respect to P. Then the algorithm $\mathcal{A}_1$ terminates: any application of the expansion rules (i)-(vi) taking into account the applicability (vii)-(xi) rules to IC is finite.

Proof. Assume that there exists an infinite sequence of expansion rules and applicability rules that can be applied to IC. Every expansion rule can be applied at most once with respect to a unary/binary predicated symbol and a node/arc and each applicability rule can be applied at most once after every expansion rule.

Thus, every infinite sequence of expansion/applicability rules would lead to an infinite size $\mathcal{A}_1$-completion structure. Clearly, if one has a finite $\mathcal{A}_1$-completion structure that is not complete, a finite application of expansion rules would complete it, unless new successors are introduced. However, it is not possible to introduce infinitely many successors: every infinite path in the interconnected forest will eventually contain $|k+1|$ saturated nodes with equal content, where k is as defined in (ix) Redundancy, and thus either a blocked or a redundant node, which is not further expanded.

Algorithm 3.1: Overview of $\mathcal{A}_1$

```
input  : FoLP $P$, unary predicate $p$;
output: checks satisfiability of $p$ with respect to$P$;
Construct an initial $\mathcal{A}_1$-completion structure $CS$ for $p$ with respect to $P$ as in Definition 13;
$S = N_{EF}$;
repeat
    Pick up a node $x \in S$ such that $x \in cts(P) \cup \varepsilon$ or its ancestor $y$ in $EF$ is saturated,
    and there exists $p \in \mathrm{CT}(x)$ such that $\mathrm{ST}(p, x) = unexp$;
    $S = S - \{x\}$;
    if there is an ancestor $y$ of $x$: $y <_F x$, $y \notin cts(P)$, s. t.:
      $\mathrm{CT}(x) \subseteq \mathrm{CT}(y)$, and
      $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty
    then
        $x$ is blocked;
    else
        Non-deterministically apply the expansion rules (i)-(iii) with respect to $x$
        and the expansion rules (iv)-(vi) with respect to arcs $(x, y)$, where
        $y \in succ_{EF}(x)$, taking into account the applicability rules (x)
        Contradiction and (xi) Circularity, until no expansion rule can be further
        applied.
        Check for clashes:
            i) if $x$ is not saturated then
                return false;
            end
            ii) if $CS$ is circular or contradictory then
                return false;
            end
            iii) if there are $k$ ancestors $y_i$ of $x$ in $F$, where $k$ is as defined in the
            (ix) Redundancy rule, $1 \leqslant i \leqslant k$, $y_i \notin cts(P)$, s. t.: $\mathrm{CT}(x) = \mathrm{CT}(y_i)$,
            for every $1 \leqslant i \leqslant k$ then
                $x$ is redundant: return false;
            end
    end
until $S = \emptyset$;
return true;
```

Furthermore, the arity of the trees in the $\mathcal{A}_1$-completion structure is bounded by the number of successor variables in unary rules, more precisely by $rank(P)$, where P is the FoLP under consideration. $\square$

3.5.2 Soundness of $\mathcal{A}_1$

Proposition 7. Let P be a FoLP and $p \in upreds(P)$. If there exists a complete clash-free $\mathcal{A}_1$-completion structure for p with respect to P, then p is satisfiable with respect to P.

Proof. Let $CS = \langle EF, \text{CT}, \text{ST}, G \rangle$ be a clash-free complete $\mathcal{A}_1$-completion structure for P with respect to P, where $EF = \langle F, ES \rangle$ is the corresponding interconnected forest, and $G = (V, A)$ is the corresponding dependency graph, and let bl be the set of blocking pairs[2] corresponding to the completion.

The proof is structured as follows: first, we construct an open interpretation (U, M) from CS (step (1) below) and then we show that this interpretation is indeed an open answer set of P that satisfies p in two steps: we show that M is a model of P_U (step (2) below) and we show that M is minimal (step (3) below).

1. *Construction of open interpretation.*

   As discussed in Section 3.3.2, one way to construct such an open interpretation, is by unraveling the $\mathcal{A}_1$-completion structure to an infinite structure corresponding to a forest-shaped open answer set with an infinite universe and an infinite interpretation. For simplicity, in this proof we choose the alternative approach: from a forest-shaped $\mathcal{A}_1$-completion structure, a finite size graph-shaped open answer set is generated by extending the content of blocked nodes such that they are identical to the content of the corresponding blocking nodes and by introducing additional arcs from blocked nodes to successors of blocking nodes which mirror the arcs from the blocking nodes themselves to their successors (thus, also inheriting their content).

   Intuitively, the atoms having as arguments non-blocked nodes are justified by the way the completion structure was constructed, while atoms having a blocked node as one of the arguments are justified in a similar way to their counterparts, where the counterpart atom of an atom $p(x)/f(x, y)$, where x is a blocked node, is the atom $p(z)/f(z, y)$, where $(z, x) \in bl$.

   The universe of the open interpretation is the set of nodes of the new graph (identical to the set of nodes of the interconnected forest), while the interpretation is the set of atoms having as arguments nodes/arcs of the graph and as predicate symbols predicates in the content of these nodes/arcs.

   Formally, let $G_{ext} = (V_{ext}, A_{ext})$ be the graph obtained by extending G in the following way: first, let $V_{ext} = V$ and $A_{ext} = A$, and then for every pair $(x, y) \in bl$ do the following:

[2] A blocking pair is a pair of nodes (x, y), $x, y \in N_{EF}$, where y is a blocked node and x is its corresponding blocking node.

a) for every p such that $p(x) \in V$: $V_{ext} = V_{ext} \cup \{p(y)\}$;
b) for every f and z such that $f(x,z) \in V$: $V_{ext} = V_{ext} \cup \{f(y,z)\}$;
c) for every p, q such that $(p(x), q(x)) \in A_{ext}$: $A_{ext} = A_{ext} \cup \{(p(y), q(y))\}$;
d) for every p, q, z such that $(p(x), q(z)) \in A_{ext}$, and $z \neq x$: $A_{ext} = A_{ext} \cup \{(p(y), q(z))\}$;
e) for every p, f, z such that $(p(x), f(x,z)) \in A_{ext}$: $A_{ext} = A_{ext} \cup \{(p(y), f(y,z))\}$;
f) for every f, q, z such that $(f(x,z), q(x)) \in A_{ext}$: $A_{ext} = A_{ext} \cup \{(f(y,z), q(y))\}$;
g) for every f, q, z such that $(f(x,z), q(z)) \in A_{ext}$: $A_{ext} = A_{ext} \cup \{(f(y,z), q(z))\}$;
h) for every f, g, z such that $(f(x,z), g(x,z)) \in A_{ext}$: $A_{ext} = A_{ext} \cup \{(f(y,z), g(y,z))\}$;

Also, let (U, M) be the following open interpretation:

- $U = N_{EF}$, i.e., the universe is the set of nodes in the interconnected forest, and
- $M = V_{ext}$, i.e., the interpretation corresponds to the set of nodes in the extended graph.

2. *M is a model of P_U^M*. We show that each type of ground rule in P_U^M is satisfied.

All free rules are trivially satisfied.

Take a ground unary rule from P_U^M:

$$r' : a(x) \leftarrow \beta^+(x), (\gamma_m^+(x, y_m), \delta_m^+(y_m))_{1 \leqslant m \leqslant k},$$

originating from:

$$r : a(s) \leftarrow \beta(s), (\gamma_m(s, t_m), \delta_m(t_m))_{1 \leqslant m \leqslant k}, \psi,$$

with:

- $\beta^-(x) \nsubseteq M$,
- $\gamma_m{}^-(x, y_m) \nsubseteq M$ and $\delta_m{}^-(y_m) \nsubseteq M$, for all $1 \leqslant m \leqslant k$, and
- $y_i \neq y_j$, for all $t_i \neq t_j \in \psi$.

Assume that: $M \models \beta^+(x) \cup \bigcup_{1 \leqslant m \leqslant k} \gamma_m^+(x, y_m) \cup \bigcup_{1 \leqslant m \leqslant k} \delta_m^+(y_m)$. Together with the assumptions about the negative part of the rule, this amounts to:

$$M \models \beta(x) \cup \bigcup_{1 \leqslant m < \leqslant k} \gamma_m(x, y_m) \cup \bigcup_{1 \leqslant m \leqslant k} \delta_m(y_m) \cup \psi \text{ and } a(x) \notin M,$$

or in other words rule r' is not satisfied.

We show that this leads to a contradiction. Depending on x there are two cases:

- x is not a blocked node. Then $not\ a \in \text{CT}(x)$, x is saturated, and no expansion rules can be further applied to $not\ a$. This means that for every ground rule derived from a rule $r \in P_a$ with head $a(x)$, the (iii) Expand-Unary-Negative rule has been applied. Such a rule is r'. The application of the (iii) Expand-Unary-Negative rule to $not\ a \in \text{CT}(x)$ and r' leads to one of the following situations:

  - there exists a unary predicate symbol $\pm q \in \beta$, such that $\mp q \in \textsc{ct}(x)$ (the result of $\mathit{update}(\mathit{not}\ a(x), \mp q, x)$), or in other words, $\mp q(x) \in M$. This is in contradiction with $M \models \beta(x)$.
  - there are two successors of x, y_i and y_j, such that $y_i = y_j$ and $t_i \neq t_j \in \psi$. This contradicts the assumption that for all $t_i \neq t_j \in \psi$: $y_i \neq y_j$.
  - for some $1 \leqslant m \leqslant k$, there exists a binary predicate symbol $\pm f \in \gamma_m$ such that $\mp f \in \textsc{ct}(x, y_m)$ (as a result of applying $\mathit{update}(\ \mathit{not}\ a(x), \mp f, (x, y_m)))$. In other words, $\mp f(x, y_m) \in M$. This contradicts with $M \models \gamma_m(x, y_m)$.
  - for some $1 \leqslant m \leqslant k$, there exists a unary predicate symbol $\pm q \in \delta_m$ such that $\mp q \in \textsc{ct}(y_m)$ (as a result of applying the operation $\mathit{update}(\mathit{not}\ a(x), \mp q, y_m))$. In other words, $\mp q(y_m) \in M$. This contradicts with $M \models \delta_m(y_m)$.

- x is a blocked node. Let y be such that $(y, x) \in \mathit{bl}$: by replacing x with y in r', one obtains a ground rule r'' which again should not be satisfied because due to the construction of M:

$$M \models \beta(x) \cup \bigcup_{1 \leqslant m < \leqslant k} \gamma_m(x, y_m) \cup \bigcup_{1 \leqslant m \leqslant k} \delta_m(y_m) \cup \psi \text{ implies}$$

$$M \models \beta(y) \cup \bigcup_{1 \leqslant m < \leqslant k} \gamma_m(y, y_m) \cup \bigcup_{1 \leqslant m \leqslant k} \delta_m(y_m) \cup \psi \text{ and}$$

  $a(x) \notin M$ implies $a(y) \notin M$.

  Thus, this case is reduced to the previous one.

Both cases lead to a contradiction, thus the original assumption that rule r' is not satisfied by M was false. Thus, every unary rule is satisfied by M.

The proof for the satisfiability of binary rules is similar.

3. *M is a minimal model of P_U^M.*

Before proceeding with the actual proof we introduce a notation and some lemmas which will prove useful in the following.

Let $EF^{ext} = (F, ES^{ext})$ be the extended forest obtained from EF as follows:

$$ES^{ext} = ES \cup \{(y, z) \mid \exists x.(x, y) \in \mathit{bl} \wedge z \in \mathit{succ}_{EF}(x)\}.$$

The new extended forest captures in a more accurate way the structure of M: blocked nodes are connected to successors of the corresponding blocking nodes, as their contents is justified similarly to the content of blocking nodes.

Lemma 1. For every $x, y \in N_{EF}$, if there exists a path

$$Pt_1 = (p(x), \ldots, l_1) \in \mathit{paths}_G / \mathit{paths}_{G_{ext}},$$

with $l_1 = q(y)$ for some $q \in \mathit{upreds}(P)$ or $l_1 = g(y,z)$ for some $g \in \mathit{bpreds}(P)$, and $x \neq y$, then there exists a path

$$Pt_2 = (x, \dots, y) \in paths_{EF}/paths_{EF^{ext}}$$

such that for every $t \in Pt_2$ there exists a unary atom $l_2 \in Pt_1$ with $args(l_2) = t$.

Proof.

Let $S = (x_1 = x, x_2, \dots, x_n)$ be a tuple of nodes from $\mathit{EF}/\mathit{EF}^{ext}$ constructed as follows: consider each element l of Pt_1 at a time: if $args(l) = y$ and y is not already part of the tuple, add y to the end of the tuple. We show that $S \in paths_{EF}/paths_{EF^{ext}}$ and furthermore that $x_n = y$.

For every two consecutive elements of S, x_i and x_{i+1}, with $1 \leqslant i < n$, there must exist two unary atoms l' and l'' in Pt, with $args(l') = x_i$ and $args(l'') = x_{i+1}$, respectively, such that there exists no other unary atom l in the sub-path of Pt_1: $(l', \dots, l'')$. It is easy to see that such a sub-path has the form:

$$(l' = r(x_i), f_1(x_i, x_{i+1}), \dots, f_m(x_i, x_{i+1}), l'' = s(x_{i+1})),$$

with $r, s \in \mathit{upreds}(P)$, and $f_1, \dots f_m \in \mathit{bpreds}(P)$, and thus $(x_i, x_{i+1}) \in A/A'$ for every $1 \leqslant i < n$: $(x_1, \dots, x_n)$ is a path in $\mathit{EF}/\mathit{EF}^{ext}$.

To see that $x_n = y$, consider the opposite: $x_n \neq y$. Then, there exists a unary atom $l = r(x_n)$ in Pt_1 with $args(l) = x_n$, such that there exists no other unary atom in the sub-path of Pt_1: $(r(x_n), \dots, g(y,z))$. Then, the sub-path has the form:

$$(r(x_n), f_1(x_n, t), \dots, f_m(x_n, t), g(y,z)),$$

where t is some successor of x_n in $\mathit{EF}/\mathit{EF}^{ext}$: $(x_n, t) \in A/A'$. But there exists no arc of the form $(f_m(x_n, t), g(y,z))$ in A/A' with $x_n \neq y$, so we obtain a contradiction. $\square$

Lemma 2. Let $C = (l_1, l_2, \dots, l_n = l_1)$ be a cycle in G_{ext}. If one of the following holds:

- (i) there exists no unary atom in C and for every $l_i = f_i(x_i, y_i)$, $1 \leqslant i \leqslant n$, x_i is not blocked, or
- (ii) there exists at least one unary atom in C and for every unary atom $l_j = a_j(x_j)$ in C, x_j is not a blocked node in CS.

then C is a cycle in G.

Proof. From the construction of G_{ext} it can be seen that any arc which is added to G is of the form $(p(x), l)$ or $(f(x,y), l)$, where p is some unary predicate, f is some binary predicate, and x is a blocked node. It is clear that when condition (i) or condition (ii) holds there exists no arc of the first form in C. As concerns arcs of the latter type, it is again obvious that there exist no such arcs if condition (i) is fulfilled. In case condition

(ii) holds, assume there exists an arc $(f(x,y), l)$ where x is a blocked node. We know that there must be at least one unary atom in the cycle. Let $p(z)$ be such a unary atom. In this case there exists a path in G (and also in G_{ext}) from $p(z)$ to $f(x,y)$, and z is different from x by virtue of (ii). According to Lemma 1 the path must contain a unary atom with argument x (as any path in EF from z to x contains x). However this is in contradiction with condition (ii) which says that there exist no such atom in C. $\square$

Lemma 3. Let $C = (l_1, l_2, \ldots, l_n = l_1)$ be a cycle in G_{ext}. If one of the following holds:

- (i) there exists no unary atom in C and for some $l_i = f_i(x_i, y_i)$, $1 \leqslant i \leqslant n$, x_i is blocked, or
- (ii) there exists at least one unary atom in C and all unary atoms have the same argument y and y is blocked,

then G contains a cycle.

Proof. We will treat the two cases separately:

(i) First, notice that in this case (when there exists no unary atom in the cycle),

$$args(l_1) = args(l_2) = \ldots = args(l_n) = (x, y),$$

as there exists no arc in A_{ext} from a binary atom $f(x,y)$ to another binary atom $g(z,t)$, with $x \neq z$ or $y \neq t$ (by construction of G_{ext}). So the cycle C has the following form:

$$C = (f_1(x,y), f_2(x,y), \ldots, f_n(x,y) = f_1(x,y)),$$

where $f_i \in bpreds(P)$, for every $1 \leqslant i \leqslant n$. Let z be the blocking node corresponding to x: $(z,x) \in bl$. As $(f_i(x,y), f_{i+1}(x,y)) \in A_{ext}$, for every $1 \leqslant i < n$, it follows that $(f_i(z,y), f_{i+1}(z,y)) \in A$, for every $1 \leqslant i < n$, so the following path in G:

$$C' = (f_1(z,y), f_2(z,y), \ldots, f_n(z,y) = f_1(z,y))$$

is actually a cycle in G.

(ii) Let $p_1(y), p_2(y), \ldots, p_n(y)$ be the unary atoms in C with y being a blocked node. Without loss of generality we consider $p_n = p_1$. Then the cycle C has the following form:

$$(p_1(y), f_{11}(y,z_1), \ldots, f_{1m_1}(y,z_1), p_2(y), f_{21}(y,z_2), \ldots, f_{2m_2}(y,z_2)), \ldots p_n(y) = p_1(y)),$$

where $f_{ij} \in bpreds(P)$, for every $1 \leqslant i < n$, and $1 \leqslant j \leqslant m_i$, and $(y, z_i) \in A'$, for every $1 \leqslant i < n$ (as the only binary atoms reachable from $p(y)$ are of the form $f(y,z)$, where $(y,z) \in A'$). Similarly to the previous case it can be shown that the following path in G:

$$(p_1(x), f_{11}(x,z_1), \ldots, f_{1m_1}(x,z_1), p_2(x), f_{21}(x,z_2), \ldots, f_{2m_2}(x,z_2)), \ldots p_n(x) = p_1(x)),$$

where x is the corresponding blocking node for y: $(x,y) \in bl$ is actually a cycle in G. $\square$

Lemma 4. Let $C = (l_1, l_2, \ldots, l_n = l_1)$ be a cycle in G_{ext}. If there exist $1 \leqslant i, j \leqslant n$ such that $pred(l_i), pred(l_j) \in upreds(P)$, $args(l_i) \neq args(l_j)$, and $args(l_i)$ is a blocked node, then there exists $1 \leqslant k \leqslant n$ such that $(args(l_k), args(l_i)) \in bl$ and $(l_k, l_i) \in paths_G$.

Proof.

Let $1 \leqslant i, j \leqslant n$ be some indices as in the premise of the lemma. From Lemma 1 it follows that: $(args(l_i), args(l_j)) \in paths_{EF^{ext}}$ and $(args(l_j), args(l_i)) \in paths_{EF^{ext}}$. Thus, $args(l_i)$ and $args(l_j)$ are part of a cycle in EF^{ext}.

Let $y = args(l_i)$. From the construction of EF^{ext} it follows that any cycle in EF^{ext} which involves $y \in T_c$ contains $path_{T_c}(z, y)$, where $z \in succ_{T_c}(x)$ and $z \in path_{T_c}(x, y)$, where x is the corresponding blocking node for y: $(x, y) \in bl$. There are two kinds of cycles in EF^{ext}:

- cycles which contain x, z, and y (these cycles will contain also elements from other trees than T_c): in this case there exists a unary atom l_k with argument x in C - thus the claim in the lemma is satisfied;
- cycles which contain z, and y, but do not contain x (actually, there exists a unique such cycle which has all elements from $path_{T_c}(z, y)$): in this case, there exists $1 \leqslant t \leqslant n$ such that $args(l_t) = z$, such that the path induced by C in G_{ext} from l_i to l_t has the form: $l_i, f_1(y, z), \ldots, f_n(y, z), l_t$. Due to the construction of G_{ext}, this implies that there is a path $((l_i)_{y|x}, f_1(x, z), \ldots, f_n(x, z), l_t)$ in G. At the same time note that there exists a path in G from l_t to l_i. So, $((l_i)_{y|x}, l_t) \in paths_G$ and $(l_t, l_i) \in paths_G$, thus $((l_i)_{y|x}, l_i) \in paths_G$. $\square$

Now we can proceed to the actual proof of the minimality of M. We start by noticing that:

- premise (i) from Lemma 2 and premise (i) from Lemma 3 cover all potential cycles in G_{ext} which contain only binary atoms,
- premise (ii) from Lemma 2 covers the case where a potential cycle in G_{ext} contains no unary atoms with blocked arguments,
- premise (ii) from Lemma 3 covers the case where there might be unary atoms in the cycle with blocked arguments, but all such arguments have to be identical, and
- the premise of Lemma 4 covers the case where a cycle in G_{ext} contains at least two unary atoms with distinct arguments, at least one of which is blocked.

Thus, altogether the premises of Lemma 2, Lemma 3, and Lemma 2 cover all possible types of cycles C in G_{ext}. The conclusions of the lemmas, that:

- there exists a cycle in G, in the case of Lemma 2 and Lemma 3, and
- that $connpr_G(x, y) \neq \emptyset$ for some pair $(x, y) \in bl$, in the case of Lemma 2

are in contradiction with the fact that $\langle EF, \text{CT}, \text{ST}, G, bl\rangle$ is a complete clash-free completion structure. Thus, a corollary of the three lemmas is that:

Corollary 1. G_{ext} is acyclic.

Assume now that there exists a model $M' \subset M$ of $Q = P_U^M$. Then, there must be an atom $l_1 \in M$ such that $l_1 \notin M'$. Consider a rule $r_1 \in Q$ of the form $l_1 \leftarrow \beta_1$ such that $M \models \beta_1$; note that such a rule always exists by construction of M and the (i) Expand-Unary-Positive rule. If $M' \models \beta_1$, then $M' \models l_1$ (as M' is a model), which is a contradiction. Thus, $M' \not\models \beta_1$, and then there must be the case that some $l_2 \in \beta_1$ exists such that $l_2 \notin M'$.

Continuing with the same line of reasoning, one obtains an infinite sequence $l_1, l_2, \ldots$ with $(l_i \in M)_{1 \leqslant i}$ and $(l_i \notin M')_{1 \leqslant i}$. M is finite (the complete clash-free completion structure has been constructed in a finite number of steps, and when constructing M (V_{ext}) we added only a finite number of atoms to the ones already existing in V), thus there exists a pair of indices $1 \leqslant (i, j)$, $i \neq j$, such that $l_i = l_j$.

We observe that $(l_i, l_{i+1})_{1 \leqslant i} \in E_{ext}$ by construction of E_{ext} and the (i) Expand-Unary-Positive rule, so our assumption leads to the existence of a cycle in G_{ext}. But this is in contradiction with Corollary 1. Thus, our initial assumption was false and there exists no model M' of $Q = P_U^M$ such that $M' \subset M$; M is minimal.

3.5.3 Completeness of $\mathcal{A}_1$

In this section we prove that $\mathcal{A}_1$ is complete:

Proposition 8. Let P be a FoLP and $p \in \mathit{upreds}(P)$. If p is satisfiable with respect to P, then there exists a clash-free complete $\mathcal{A}_1$-completion structure for p with respect to P.

Proof. If p is satisfiable with respect to P then p is forest-satisfiable with respect to P (Proposition 5). We construct a clash-free complete $\mathcal{A}_1$-completion structure for p with respect to P, by guiding the non-deterministic application of the expansion rules with the help of a forest model of P which satisfies p and by taking the constraints imposed by the saturation, blocking, and redundancy rules into account. The proof is inspired by completeness proofs in DL for tableaux procedures, for example in [Horrocks et al., 1999], but requires additional mechanisms to eliminate redundant parts from Open Answer Sets.

There are two main stages in the proof:

1. In the first stage, a so-called *complete clash-free relaxed $\mathcal{A}_1$-completion structure* is constructed with the help of a forest model of P which satisfies p. Such a structure is defined/constructed similarly as a classical $\mathcal{A}_1$-completion structure apart from the fact that the redundancy rule is not employed. Accordingly, for a relaxed $\mathcal{A}_1$-completion structure to be clash-free, the condition regarding the absence of redundant nodes is not relevant.

2. The second stage consists in transforming such a complete clash-free relaxed $\mathcal{A}_1$-completion structure into a complete clash-free $\mathcal{A}_1$-completion structure. The transformation consists in several successive steps, each step 'shrinking' the structure, in such a way that the new structure is still a complete clash-free relaxed completion structure. It is shown that the result of this transformation is a structure in which every branch has at most k nodes with equal content, with k as defined in the redundancy rule, and thus, it is a complete clash-free completion structure.

We introduce the notion of *relaxed $\mathcal{A}_1$-completion structure* which is a tuple $\langle EF$, $\textsc{ct}$, $\textsc{st}$, $G, bl\rangle$, where EF is an interconnected forest, and G, $\textsc{ct}$, and st represent the same kind of entities as the respective components of a regular completion structure. We will keep track explicitly also of the blocking pairs in such a structure: bl is the set of all such pairs. An *initial relaxed $\mathcal{A}_1$-completion structure for checking satisfiability of a unary predicate p with respect to a FoLP P* is defined similarly to an initial $\mathcal{A}_1$-completion structure for checking satisfiability of p with respect to P.

A relaxed $\mathcal{A}_1$-completion structure is evolved using the expansion rules (i)-(vi) and the applicability rules (vii)-(viii) and (x)-(xi). Note that the *redundancy* rule is left out. A *complete $\mathcal{A}_1$-relaxed completion structure* is a relaxed $\mathcal{A}_1$-completion structure evolved from an initial relaxed completion structure to which none of the expansion rules (i)-(vi) can be further applied.

A *clash-free relaxed $\mathcal{A}_1$-completion structure* is a relaxed $\mathcal{A}_1$-completion structure evolved from an initial relaxed $\mathcal{A}_1$-completion structure for p and P, such that: the structure is neither contradictory nor circular, and for which every node in the structure is marked as expanded.

First, we construct a complete clash-free relaxed $\mathcal{A}_1$-completion structure starting from a forest model of a FoLP P which satisfies p. Note that in the general case, constructing a complete clash-free relaxed $\mathcal{A}_1$-completion structure might be a non-terminating process (the termination for the construction of complete clash-free $\mathcal{A}_1$-completion structures was based on the application of the redundancy rule), but as we will see in the following, the process does terminate when a forest model is used as a guidance.

Let (U, M) be a forest model of a FoLP P which satisfies p. Then, there exist:

- an interconnected forest $EF = \langle \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES\rangle$, where ε is a constant, possibly one of the constants appearing in P, and
- a labeling function $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$,

which fulfil the conditions from Definition 11.

We define an initial relaxed $\mathcal{A}_1$-completion structure $CS_0 = \langle EF'$, $\textsc{ct}$, $\textsc{st}$, G, $bl\rangle$ for p and P such that:

- $EF' = \langle F', ES'\rangle$,
- $F' = \{T'_\varepsilon\} \cup \{T'_a \mid a \in cts(P)\}$, where:
  - ε is the same ε used to define EF,
  - $T_x = \{x\}$, for every $x \in cts(P) \cup \{\varepsilon\}$,

  - $ES' = \emptyset$,

- $G = \langle V, A \rangle$, $V = \{p(\varepsilon)\}$, $A = \emptyset$,
- $\textsc{ct}(\varepsilon) = \{p\}$,
- $\textsc{st}(\varepsilon, p) = \mathit{unexp}$,
- $bl = \emptyset$.

We evolve CS_0 using the expansion rules (i)-(vi) and the applicability rules (vii)-(viii) and (x)-(xi). Whenever a non-deterministic choice has to be made, we will use (U, M) as a guidance. In fact, we will show that under these circumstances the applicability rules (x)-(xi) are never applicable, thus the evolved relaxed $\mathcal{A}_1$-completion structure is always clash-free. To this purpose, we define inductively a function $\pi : N_{EF'} \cup A_{EF'} \to U \cup (U \times U)$ that relates nodes/arcs in the relaxed $\mathcal{A}_1$-completion structure to nodes/arcs in the forest model such that at any point during the expansion process the following holds:

$$(\ddagger) \text{ for every } z \in N_{EF'} \cup A_{EF'}, \begin{cases} \{q \mid q \in \textsc{ct}(z)\} \subseteq \mathcal{L}(\pi(z)) \\ \{q \mid \mathit{not}\ q \in \textsc{ct}(z)\} \cap \mathcal{L}(\pi(z)) = \emptyset \end{cases}$$

Intuitively, the positive content of a node/arc in the $\mathcal{A}_1$-completion structure is contained in the label of the corresponding forest model node/arc, and the negative content of a node/arc in the $\mathcal{A}_1$-completion structure cannot occur in the label of the corresponding forest model node/arc.

We start by setting $\pi(x) = x$, for every $x \in \mathit{cts}(P) \cup \{\varepsilon\}$ (the roots of the trees in the relaxed $\mathcal{A}_1$-completion structure correspond to the roots of the trees in the forest model). It is clear that ($\ddagger$) is satisfied for CS_0. By induction, let CS be a relaxed $\mathcal{A}_1$-completion structure derived from CS_0 and π a function that satisfies ($\ddagger$) with respect to the nodes in CS. We show how CS can be expanded such that ($\ddagger$) is still satisfied: We consider the expansion rules (i)-(vi) and the applicability rules (vii)-(viii) and (x)-(xi):

1. *(i) Expand-Unary-Positive*. Assume x is a node in CS and $q \in \textsc{ct}(x)$ such that $\textsc{st}(q, x) = \mathit{unexp}$. We have, by the induction hypothesis, that $q \in \mathcal{L}(\pi(x))$. Since M is a minimal model, there exists a rule $r \in P_q$ of the form (2.3) and a ground version of r:

$$r' : q(\pi(x)) \leftarrow \beta^+(\pi(x)), (\gamma_m^+(\pi(x), z_m))_{1 \leqslant m \leqslant k}, (\delta_m^+(z_m))_{1 \leqslant m \leqslant k} \in (P_q)_U^M$$

   such that:

$$M \models \beta^+(\pi(x)) \cup (\gamma_m^+(\pi(x), z_m))_{1 \leqslant m \leqslant k} \cup (\delta_m^+(z_m))_{1 \leqslant m \leqslant k}.$$

   First apply $\mathit{update}(q(x), \beta, x)$, and then, for each $1 \leqslant m \leqslant k$:

   - if $z_m = \pi(z)$ for some z already in EF':
     - let $y_m = z$;
     - if $z \in \mathit{cts}(P)$ and $(x, z) \notin ES'$ then $ES' = ES' \cup \{(x, z)\}$.

- if $z_m = \pi(x) \cdot s$ and z_m is not yet the image of π of some node in EF', then:
  - add $x \cdot s$ as a new successor of x in F': $T'_c = T'_c \cup \{x \cdot s\}$, where $x \in T'_c$,
  - set $\pi(x \cdot s) = \pi(x) \cdot s$, and
  - set $\pi(x, x \cdot s) = (\pi(x), \pi(x) \cdot s)$.
- $update(q(x), \gamma_m, (x, y_m))$,
- $update(q(x), \delta_m, y_m)$.

In other words we removed the non-determinism from the (i) Expand-Unary-Positive rule, by choosing the rule r and the successors corresponding to the open answer set (U, M). It can be verified that (‡) still holds for π.

2. The expansion rules (ii)-(vi) can be dealt with in a similar way, making the non-deterministic choices in accordance with (U, M).

3. (vii) *Saturation.* No expansion rule can be applied on a node from EF' which is not a constant until its predecessor is saturated. This rule is independent of the particular open answer set which guides the construction, so it is applied as usual.

4. (viii) *Blocking.* Consider a node $x \in N_{EF'}$ which is selected for expansion. If there exists a saturated node $y \in N_{EF'}$ which is not a constant, $y <_{T_c} x$, where $T_c \in F'$, $\text{CT}(x) \subseteq \text{CT}(y)$, and $connpr_G(y, x) = \emptyset$, mark x as a blocked node in CS and y as its blocking counterpart: $bl = bl \cup \{(x, y)\}$.

   Furthermore, we impose that if there are more nodes y which satisfy the condition we will consider as the blocking node for x the one which is closest to the root of the tree T_c (the tree to which x belongs), so the node y for which there exists no node z such that $z <_{T_c} y$, $\text{CT}(x) \subseteq \text{CT}(z)$, and $connpr_G(z, y) = \emptyset$. This choice over possible blocking nodes is relevant for the next stage of the proof, where a complete clash-free relaxed $\mathcal{A}_1$-completion structure is transformed into a complete clash-free completion structure. The condition (‡) still holds for π as we have not modified the content of nodes or arcs, but just removed some unexpanded nodes.

5. (x) *Contradiction.* Assume that for some unary/binary predicate symbol p and node/arc x in EF' it is the case that $not\ p, p \in \text{CT}(x)$. Then, from the induction hypothesis it follows that $p \in \mathcal{L}(\pi(x))$ and $p \notin \mathcal{L}(\pi(x))$, or in other words, $p(x)$ is at the same time in and out of the open answer set (U, M) – contradiction. Thus, this rule is never applicable when using (U, M) as a guidance.

6. (xi) *Circularity.* Assume that there exists a cycle in G. Then, due to the fact that CS has been evolved using (U, M) as a guidance and from the induction hypothesis, it follows that there exists a cyclic dependency between atoms in M. Thus, M is not minimal – contradiction with the fact that (U, M) is an open answer set. Thus, this rule is never applicable when using (U, M) as a guidance.

We show that when evolving CS_0 in the manner described above:

1. *the process terminates*: assume the process does not terminate. Then, for every $x, y \in N_{EF'}$ such that $x <_{F'} y$ and $\text{CT}(x) = \text{CT}(y)$, the blocking rule cannot be applied, so there exists a path from a $p(x)$ to some $q(y)$. This suggests the existence of an infinite path in G (as on any infinite branch in a tree from F' there would be an infinite number of nodes with equal content - there is a finite amount of values for the content of a node), which contradicts with the fact that any atom in an open answer set is justified in a finite number of steps [Heymans et al., 2006, Theorem 2].

2. *the resulted complete relaxed $\mathcal{A}_1$-completion structure is clash-free*: follows directly from the fact that the evolved structure is neither contradictory nor circular (as the corresponding applicability rules are not applicable) and from the fact that every (possibly negated) predicate symbol $\pm p$ in the content of some node/arc x of the structure in construction such that $\text{ST}(\pm p, x) = \mathit{unexp}$, can be expanded eventually using one of the expansion rules (i), (iii), (iv) or (v).

At this point we have constructed a complete clash-free relaxed $\mathcal{A}_1$-completion structure CS for p w.r.t P starting from a forest open answer set for P which satisfies p.

The preference relation over different blocking nodes choices in the construction above has several consequences described by the following results:

Lemma 5. Let $CS = \langle EF,\ \text{CT},\ \text{ST},\ G\ bl\rangle$ be a complete clash-free relaxed $\mathcal{A}_1$-completion structure constructed in the manner described above, with ($EF = \langle F, ES\rangle$). Then, for every node $x \in N_{EF}$ such that there exists a node $y \in N_{EF}$, with $(x, y) \in bl$ (x is a blocking node in CS), there exists no node $z <_{T_c} x$, $T_c \in F$ such that $\text{CT}(z) = \text{CT}(x)$.

Proof. Assume by contradiction that x is a blocking node in CS, i.e. there exists a node y such that $(x, y) \in bl$, and that there exists also $z <_{T_c} x$, $T_c \in F$ such that $\text{CT}(z) = \text{CT}(x)$. Observe that: $(p(z), q(y)) \in conn_G(z, y)$ implies that there exists a unary predicate $r \in \text{CT}(x)$ such that $(p(z), r(x)) \in conn_G(z, x)$ and $(r(x), q(y)) \in conn_G(x, y)$ (according to Lemma 1 the existence of a path from a $p(z)$ to a $q(y)$ in G implies the existence of a path from z to y in EF; all paths from z to y in EF include the path from z to y in T_c and consequently, x; then, according to the same lemma there must be an atom in the initial path in G with argument x: $r(x)$ in this case).

But $connpr_G(x, y) = \emptyset$ as $(x, y) \in bl$, so $connpr_G(z, y) = \emptyset$. Additionally, $\text{CT}(z) = \text{CT}(x) \supseteq \text{CT}(y)$, so the existence of z is in contradiction with the preference condition over potential blocking nodes. □

Corollary 2. Let $CS = \langle EF,\ \text{CT},\ \text{ST},\ G, bl\rangle$ be a complete clash-free relaxed $\mathcal{A}_1$-completion structure constructed in the manner described above ($EF = \langle E, ES\rangle$) and B a branch of a tree T_c from F. Then, there are at most 2^p distinct blocking nodes in B where $p = |upreds(P)|$.

Proof. The result follows from the fact that there cannot be two blocking nodes with equal content on the same path in a tree according to the previous lemma and the finite number of values for the content of a node which is given by the cardinality of the power set of $upreds(P)$. □

The next step is to transform a relaxed clash-free complete $\mathcal{A}_1$-completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G, bl \rangle$, where $EF = \langle F, ES \rangle$, into a complete clash-free $\mathcal{A}_1$-completion structure, that is, a complete clash-free relaxed $\mathcal{A}_1$-completion structure which has no redundant nodes. This is done by applying a series of successive transformations on the relaxed $\mathcal{A}_1$-completion structure. Each transformation "shrinks" the $\mathcal{A}_1$-completion structure in the sense that the transformed relaxed $\mathcal{A}_1$-completion structure has a smaller number of nodes than the original one and is still complete and clash-free. The result of applying the transformation is a relaxed clash-free $\mathcal{A}_1$-complete completion structure which has a bound on the number of nodes on any branch which matches the bound k from the redundancy condition, which is thus a complete clash-free $\mathcal{A}_1$-completion structure.

A way to shrink a (relaxed) $\mathcal{A}_1$-completion structure is that whenever two distinct nodes u and v in a tree T_c from F are on the same path ($u <_{T_c} v$), and they have equal content ($\textsc{ct}(u) = \textsc{ct}(v)$), then the subtree $T_c[u]$ is replaced with the subtree $T_c[v]$. We call such a transformation $collapse_{CS}(u, v)$; its result is a new relaxed $\mathcal{A}_1$-completion structure $CS' = \langle EF'$, $\textsc{ct}'$, $\textsc{st}'$, G', $bl' \rangle$, whose elements are defined as follows:

- let $ef : N_{EF} \to C$ be a labeled interconnected forest which associates to every node of EF a label from a set of distinguished constants C such that $ef(x) \neq ef(y)$ for every x and y in N_{EF} such that $x \neq y$. Then, let $ef' = replace_{ef}(u, v)$ be a new labeled interconnected forest and EF' be its corresponding unlabeled interconnected forest.

  For every $x \in EF'$ let $\overline{x}$ be the counterpart of x in EF in the sense that: $ef'(x) = ef(\overline{x})$. Note that for every $x \in EF'$ there is a unique such counterpart in EF. For simplicity we also introduce the notation $\overline{S}$ to refer to the counterpart tuple (the tuple of counterpart nodes) corresponding to the tuple of nodes from S from T' . Formally, $\overline{(x_1, \ldots, x_n)} = (\overline{x_1}, \ldots, \overline{x_n})$. With the help of this notion of counterpart nodes we will also define the other components of the resulting completion structure.

- $G' = (V', A')$. The set of nodes V' of the new graph G' contains all atoms l for which there is an atom in V formed with the same predicate symbol as l and having as arguments the counterpart of the arguments of l. Additionally, V' contains binary atoms which connect the predecessor of u (it is the same both in EF and EF') with the new node $\overline{u}$ which were also present in V - this is necessarily as $\overline{u} = v$, so otherwise these connections would be lost:

$$\begin{aligned} V' = &\{l_1 \mid \exists l_2 \in V. pred(l_1) = pred(l_2) \land args(l_1) = \overline{args(l_2)}\} \cup \\ &\{f(z, u) \mid z \in T' \land f(z, u) \in V\}. \end{aligned}$$

  The set of arcs A' of the new graph G' contains all pair of atoms (l_1, l_2) for which there is a corresponding pair in A, (l_3, l_4), such that l_3 and l_4 have the same predicate symbols as l_1 and l_2, respectively, and their argument tuples are the counterpart of the argument tuples of l_1, and l_2, respectively. Additionally, A' contains arcs from A which connect atoms whose arguments include the predecessor of u (it is the same both in T and T') with atoms whose arguments include the new node $\overline{u}$ - this is necessary as $\overline{u} = v$, so otherwise these connections would be lost:

$$\begin{aligned} A' =&\{(l_1,l_2) \mid \exists(l_3,l_4) \in A.pred(l_1) = pred(l_3) \wedge pred(l_2) = pred(l_4) \\ &\wedge args(l_1) = \overline{args(l_3)} \wedge args(l_2) = \overline{args(l_4)}\} \cup \\ &\{(l_1,l_2) \mid (l_1,l_2) \in E \wedge u \in arg(l_2) \wedge z \in arg(l_1) \wedge z < u\}. \end{aligned}$$

- $\text{CT}'(x) = \text{CT}(\overline{x})$, for every $x \in N_{EF'} \cup A_{EF'}$;
- $\text{ST}'(x,p) = \text{ST}(\overline{x},p)$, for every $x \in N_{EF'} \cup A_{EF'}$ and $\pm p \in \text{CT}(\overline{x})$;
- $bl' = \{(x,y) \mid (\overline{x},\overline{y}) \in bl \wedge connpr_{G'}(x,y) = \emptyset\}$. We maintain those blocking pairs whose counterparts in EF formed a blocking pair, and which furthermore still fulfill the blocking condition.

Note that the result of applying the transformation on a complete clash-free relaxed $\mathcal{A}_1$-completion structure might be an incomplete clash-free relaxed $\mathcal{A}_1$-completion structure. If completeness of the original structure was achieved by applying among others the blocking rule, the transformation might leave some branches “unfinished”. This is the case when the blocking node is eliminated or when two nodes who formed a blocking pair are still found in the new structure, but do no longer fulfil the blocking condition.

We introduce two lemmas which describe cases in which the transformation can be applied without losing the completeness of the resulted structure. Before that, however, we need to state a general result which will prove useful in the demonstration of the two lemmas. The result states that if as a result of applying the *collapse* transformation on a complete clash-free relaxed $\mathcal{A}_1$-completion structure one obtains an $\mathcal{A}_1$-completion structure in which the path between a blocking pair of nodes remains untouched (every node in the original path is the counterpart of some node in the new structure), then the nodes which have as counterparts the nodes of the blocking pair form a blocking pair in the new completion structure.

Lemma 6. Let $CS = \langle EF, \text{CT}, \text{ST}, G, bl\rangle$, with $EF = \langle F, ES\rangle$, be a complete clash-free relaxed $\mathcal{A}_1$-completion structure, and let $u, v \in T_c$, where $T_c \in EF$, such that: $\text{CT}(u) = \text{CT}(v)$ and $u <_{T_c} v$. Furthermore, let $CS' = \langle EF', \text{CT}', \text{ST}', G', bl'\rangle$ be the result returned by $collapse_{CS(u,v)}$. Then, for every $(x,y) \in bl$, with $x, y \in T_c$:

- if for every $z \in path_{T_c}(x,y)$, there exists some $z' \in EF'$ such that $\overline{z'} = z$, then $(x',y') \in bl'$, where $x', y' \in EF'$, $\overline{x'} = x$ and $\overline{y'} = y$.

Proof. Let EF, EF', x, y, x', and y' be as above. The conditions for the two nodes x' and y' from EF' to form a blocking pair: $(x',y') \in bl'$, are that $(\overline{x},\overline{y}) \in bl$ and $connpr_{G'}(x',y') = \emptyset$. The first condition is part of the prerequisites of the lemma, so it remains to be proved that $connpr_{G'}(x',y') = \emptyset$.

Assume by contradiction that there exists a path in G' from a $p(x')$ to a $q(y')$. Then, according to Lemma 1, there exists a path Pt in EF' from x' to y' such that for every $z \in P$ there exists a unary atom with argument z in the path in G' from $p(x')$ to $q(y')$. Any path in EF' from x' to y' includes the path in T'_c (the tree to which both x' to y' belong) from x' to y'.

Assume $path_{T_c'}(x', y') = (x_1' = x', x_2', \ldots, x_n' = y')$: then Pt contains the unary atoms $l_1', l_2', \ldots, l_n'$ with $args(l_i') = x_i'$, for $1 \leqslant i \leqslant n$ such that $(l_i', l_{i+1}') \in connpr_{G'}$, for every $1 \leqslant i < n$. Let $\overline{x_i'} = x_i$. As every node on the path $path_{T_c}(x, y)$ is the counterpart of some node in $path_{T_c'}(x', y')$ and every node in $path_{T_c'}(x', y')$ has the some counterpart in $path_{T_c}(x, y)$, we can conclude that $path_{T_c}(x, y) = (x_1, x_2, \ldots, x_n)$. Also, from the definition of $collapse$ it can be seen that the presence of unary atoms l'_i with $args(l'_i) = x_i'$ in Pt/G' implies the presence of atoms l_i with $args(l_i) = x_i$ and $pred(l_i) = pred(l_i')$ in G, for every $1 \leqslant i \leqslant n$. Furthermore, $(l'_i, l_{i+1}') \in connpr_{G'}$ implies $(l_i, l_{i+1}) \in connpr_G$, for every $1 \leqslant i < n$. The latter implies that: $(l_1, l_n) \in connpr_G$ with $args(l_1) = x_1 = \overline{x_1'} = x$ and $args(l_n) = x_n = \overline{x_n'} = y$, or in other words to $(pred(l_1), pred(l_n)) \in connpr_G(x, y)$. This contradicts with the fact that $(x, y) \in bl$, and thus $connpr_G(x, y) = \emptyset$. $\square$

Finally, the first transformation which preserves the completeness of the structure is as follows:

Lemma 7. Let $CS = \langle EF, \text{CT}, \text{ST}, G, bl \rangle$ be a complete clash-free relaxed $\mathcal{A}_1$-completion structure and let u and v be two nodes in a tree T_c in F such that:

- $u <_{T_c} v$,
- $\text{CT}(u) = \text{CT}(v)$, and
- there is no blocking node x' such that: $x' <_{T_c} v$.

Then, $collapse_{CS}(u, v)$ returns a complete clash-free relaxed $\mathcal{A}_1$-completion structure.

Proof. We have to show that $CS' = collapse_{CS}(u, v)$ is complete, that is, no expansion rule further applies to this completion structure. We will consider every leaf node x of EF' and show that no rule can be applied to further expand such a node. There are three possible cases as concerns the counterpart $\overline{x}$ of x in EF, (which in turn is a leaf node in EF):

- $\overline{x}$ is a blocked node in CS, and it does not belong to the tree T_c to which u and v belong. Assume $\overline{x} \in T_d$ (with $d \neq c$): then there is a node $y' \in T_d$ such that $(y', \overline{x}) \in bl$. No node was eliminated from T_d as a result of the transformation so for every $z \in path_{T_c}(\overline{x}, y')$, there exists $z' \in N_{EF'}$ such that $\overline{z'} = z$. Thus Lemma 6 can be applied and we conclude $(x, y) \in bl'$, where y is the node in EF' for which $\overline{y} = y'$. So x is a blocked node in CS.

- $\overline{x}$ is a blocked node in CS which belongs to the same tree T_c to which u and v belong: then, there is a node $y' \in T_c$ such that $(y', \overline{x}) \in bl$. Depending on the location of y' in T_c we distinguish between the following situations:

  - $y' \not>_{T_c} u$ (Figure 3.3 a)): in this case y' is on a branch which does not contain u and v (as it is also the case that $y' \not< u$ due to the fact that there is no blocking node x' such that $\varepsilon \leqslant x' < v$) and it is not eliminated as a result of applying the transformation, so the path from $\overline{x}$ to y' in T_c is preserved as a result of the transformation. Lemma 6 can be applied with the result that $(y, x) \in bl$ where y is the node in EF' for which $\overline{y} = y'$.

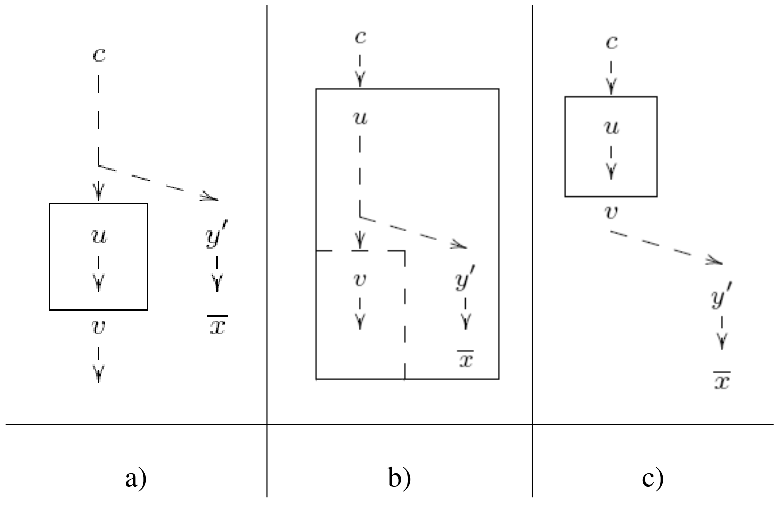


Figure 3.3: Shrinking a completion structure by eliminating a subtree with a root above any blocking node (the eliminated part is highlighted with continuous line; the part highlighted with dashed line is still kept in)

– $y' \geqslant_{T_c} u$ and $y' \ngeqslant v$ (Figure 3.3 b)): in this case y' is eliminated as a result of applying the transformation, but $\overline{x}$ is also eliminated which contradicts the existence of x in CS'. To see why $\overline{x}$ is also eliminated notice that $y' \nleqslant v$ (as again this would contradict with the fact that there is no blocking node x' such that $\varepsilon \leqslant x' < v$) and $\overline{x} > y'$. This implies that $\overline{x} > u$ and $\overline{x} \nleqslant v$, and thus $\overline{x}$ is one of the eliminated nodes, too.

– $y' \geqslant v$ (Figure 3.3 c)): in this case y' is not eliminated as a result of applying the transformation, so the path from $\overline{x}$ to y' in T_c is preserved as a result of the transformation. Lemma 6 can be applied with the result that $(x, y) \in bl$ where y is the node in EF' for which $\overline{y} = y'$.

So the conclusion of the analysis above is the existence of a node $y \in T'$ such that $(\overline{y}, \overline{x}) \in bl$. As $connpr_G(\overline{y}, \overline{x}) = \emptyset$, $connpr_{G'}(y, x) = \emptyset$ as the subtree $T[\overline{y}]$ can be found in T' intact in the form of the subtree $T'[y]$: the eliminated nodes were not part of this subtree as, again, there is no blocking node x' in T, such that $\varepsilon \leqslant x' < v$.

- $\overline{x}$ is not a blocked node in CS; as CS is complete, no expansion rule can be applied to $\overline{x}$ in CS and, by transfer neither to x in CS' (as they are two nodes which have equal contents which are justified in a similar way).

The second transformation which preserves the completeness of the structure is as follows:

Lemma 8. Let $CS = \langle EF, \textsc{ct}, \textsc{st}, G, bl\rangle$ be a complete clash-free relaxed $\mathcal{A}_1$-completion structure and let u and v be two nodes in some tree T_c in EF such that:

- $u <_{T_c} v$,
- $\textsc{ct}(u) = \textsc{ct}(v)$, and
- there exists a node $z <_{T_c} u$ such that:
  - there is no blocking node x' such that $z <_{T_c} x' <_{T_c} v$, and
  - $connpr_G(z, u) \subseteq connpr_G(z, v)$.

Then, $collapse_{CS}(u, v)$ returns a complete clash-free relaxed $\mathcal{A}_1$-completion structure.

Proof. Similarly to the proof for Lemma 7, we show that any leaf node in the completion structure $CS' = collapse_{CS}(u, v)$ (or more precisely in the corresponding tree T_c') cannot be further expanded. Let x be such a leaf node. We distinguish between three cases as concerns the counterpart of x in T_c, $\overline{x}$:

- $\overline{x}$ is a blocked node in CS, which belongs to the same tree T_c to which u and v belong. This case can be treated similarly to the first case in Lemma 7.
- $\overline{x}$ is a blocked node in CS which makes part from the same tree T_c from which u and v make part: then there is a node $y' \in T_c$ such that $(y', \overline{x}) \in bl$. Using a similar argument as for the previous lemma one concludes that there is a node $y \in T'$ such that $y' = \overline{y}$, or in other words y' has not been eliminated as a result of applying the transformation. In the following we will show that $(y, x) \in bl'$ and x is not further expanded. We will do this on a case-by-case basis considering different locations of $\overline{y}$ and $\overline{x}$ in T_c with respect to the nodes z, u, an v (we consider only those cases in which after the transformation both $\overline{y}$ and $\overline{x}$ are maintained in the structure):
  - $\overline{y} \leqslant_{T_c} z$ and there is a node z' such that $z' <_{T_c} u$, $z' \geqslant_{T_c} \overline{y}$, and $\overline{x} >_{T_c} z'$ (Figure 3.4 a)): in this case the transformation does not remove any node from $path_{T_c}(\overline{y}, \overline{x})$ so Lemma 6 can be applied: then, $(y, x) \in bl'$.
  - $\overline{y} >_{T_c} v$ (Figure 3.4 b)): in this case no nodes from the subtree $T_c[\overline{y}]$ are removed during the transformation, so using the same argument as above we obtain that $(y, x) \in bl'$.
  - $\overline{y} \not>_{T_c} z$ and $\overline{y} \not\leqslant_{T_c} z$ (Figure 3.4 c)): in this case $\overline{y}$ is not on the same path as z, u, and v and again the subtree $T_c[\overline{y}]$ is copied intact into T_c', so $(y, x) \in bl'$.
  - $\overline{y} \leqslant_{T_c} z$ and $\overline{x} \geqslant_{T_c} v$ (Figure 3.4 d)):: in this case $\overline{y}$, z, u, v and $\overline{x}$ are all on the same path in T_c. Assume by contradiction that $connpr_{G'}(y, x) \neq \emptyset$, or in other words there is a path in G' from a $p(y)$ to some $q(x)$. From Lemma 1 it follows that there must be a path $Pt \in pathsP_{EF'}(y, x)$: note that every such path contains $path_{T_c'}(y, x)$. From the same lemma and the previous observation it follows that there must be a set of unary atoms $l_1, l_2, \ldots, l_n$ in G' with arguments $x_1, x_2, \ldots x_n$,

such that $path_{T'_c}(y,x) = (x_1 = y, x_2, \dots x_n = x)$, where $(l_i, l_{i+1}) \in connpr_{G'}$, for every $1 \leqslant i < n$. Note that $(l_i, l_{i+1}) \in connpr_{G'}$, for $1 \leqslant i < n$ implies that $(l_i, l_j) \in connpr_{G'}$, for every pair $1 \leqslant i < j \leqslant n$.

In this case the counterpart of z from T_c in T'_c is still z, and the counterpart of v from T_c in T'_c is u, or in other words $\overline{z} = z$ and $\overline{u} = v$. Thus:

* $z, u \in path_{T'_c}(y,x)$, and
* there exist $1 \leqslant j < k \leqslant n$ such that $x_j = z$ and $x_k = u$.

As (l_1, l_j), (l_j, l_k), $(l_k, l_n) \in connpr_{G'}$, it follows that:

* $(pred(l_1), pred(l_j)) \in connpr_{G'}(y,z)$,
* $(pred(l_j), pred(l_k)) \in connpr_{G'}(z,u)$, and
* $(pred(l_k), pred(l_n)) \in connpr_{G'}(u,x)$.

We know from the definition of $collapse$ that:

* $connpr_{G'}(y,z) = connpr_G(\overline{y}, z)$,
* $connpr_{G'}(z,u) = connpr_G(z,u)$, and
* $connpr_{G'}(u,x) = connpr_G(v, \overline{x})$.

Thus, it is the case that:

* $(pred(l_1), pred(l_j)) \in connpr_G(\overline{y}, z)$ (1),
* $(pred(l_j), pred(l_k)) \in connpr_G(z,u)$, and
* $(pred(l_k), pred(l_n)) \in connpr_G(v, \overline{x})$ (2).

One of the premises of the current lemma is that: $connpr_G(z,u) \subseteq connpr_G(z,v)$, thus: $(pred(l_j), pred(l_k)) \in connpr_G(z,v)$. Finally, from $(pred(l_1), pred(l_j)) \in connpr_G(\overline{y}, z)$, $(pred(l_j), pred(l_k)) \in connpr_G(z,v)$, and $(pred(l_k), pred(l_n)) \in connpr_G(v, \overline{x})$, it follows that: $(pred(l_1), pred(l_n)) \in connpr_G(\overline{y}, \overline{x})$, which is in contradiction with the fact that $connpr_G(\overline{y}, \overline{x}) = \emptyset$ as $(\overline{y}, \overline{x}) \in bl$. Thus, the initial assumption was false: $connpr_{G'}(y,x) = \emptyset$, and $(y,x) \in bl'$.

- $\overline{x}$ is not a blocked node in CS (Figure 3.4 d)); using a similar argument as for the previous lemma, it can be shown that no expansion rule applies to x in CS'.

The transformations described in Lemma 7 and Lemma 8 can be applied to successively shrink a relaxed clash-free complete $\mathcal{A}_1$-completion structure. For such a structure CS, we will denote with $shrunk(CS)$ the result of applying the transformations described in Lemma 7 and Lemma 8 until it is no longer possible to apply any further transformation (the order in which the transformations are applied is irrelevant). A direct corollary of Lemma 7 and Lemma 8 is the following:

Corollary 3. Let CS be a relaxed clash-free complete $\mathcal{A}_1$-completion structure. Then, it is the case that $shrunk(CS)$ is a relaxed clash-free complete $\mathcal{A}_1$-completion structure, as well.

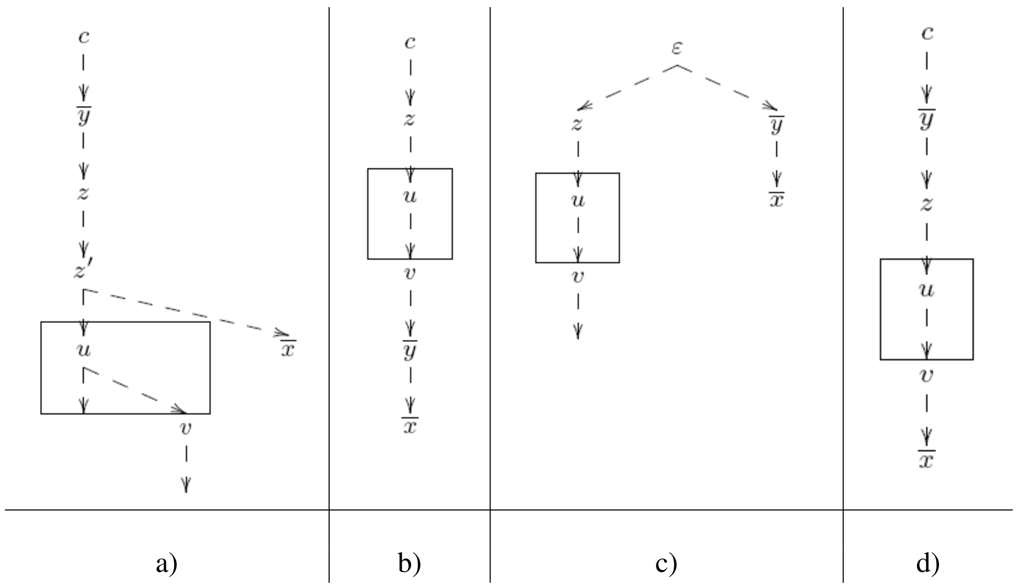


Figure 3.4: Shrinking an $\mathcal{A}_1$-completion structure by eliminating a subtree with a root below a blocking node (the eliminated part is highlighted)

It still remains to be shown that $shrunk(CS)$ is a regular $\mathcal{A}_1$-completion structure, i.e. it contains no redundant nodes. We will do this by showing that every branch of CS has at most $k = 2^p(2^{p^2} - 1) + 3$ nodes with $p = |upreds(P)|$.

We start with a lemma which counts the number of nodes with given content on different portions of a generic branch in the structure.

Lemma 9. Let CS be a relaxed clash-free complete $\mathcal{A}_1$-completion structure. Furthermore, let $shrunk(CS) = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ and let B be a branch in a tree T_c in EF which contains n blocking nodes $x_1, x_2, \ldots x_n$ and for which the terminal node is denoted with end (Figure 3.5). Then, for any set $S \in 2^{upreds(P)}$, it holds that:

- for every $1 \leqslant i < n$, there are at most 2^{p^2} nodes in $path_{T_c}(x_i, x_{i+1})$ with content equal to S, in case there is no node $x \in T_c$ such that $c <_{T_c} x \leqslant_{T_c} x_i$ and $\textsc{ct}(x) = S$,
- for every $1 \leqslant i < n$, there are at most $2^{p^2} - 1$ nodes in $path_{T_c}(x_i, x_{i+1})$ with content equal to S, except for x_i, in case there is a node $x \in T_c$ such that $c <_{T_c} x \leqslant_{T_c} x_i$ and $\textsc{ct}(x) = S$,
- there are at most 2^{p^2} nodes in $path_{T_c}(x_n, end)$ with content equal to S, except for x_n.

Proof.

We prove that for every $1 \leqslant i < n$, there are at most 2^{p^2} nodes in $path_{T_c}(x_i, x_{i+1})$ with content equal to S in case there is no node $x \in T_c$ such that $c <_{T_c} x \leqslant_{T_c} x_i$ and

$$\begin{tikzcd} c \arrow[r, dashed] & x_1 \arrow[r, dashed] & x_2 \arrow[r, dashed] & \cdots \arrow[r, dashed] & x_n \arrow[r, dashed] & end \end{tikzcd}$$

Figure 3.5: A random branch B in a complete clash-free relaxed completion structure: $x_1, \ldots,$ x_n are blocking nodes

$\text{CT}(x) = S$. Assume by contradiction, that there are at least $2^{p^2} + 1$ nodes in $path_{T_c}(x_i, x_{i+1})$ with content equal to S. Let these nodes be $y_1, y_2, \ldots, y_m$, where $m > 2^{p^2}$. It is necessary that $connpr_G(y_1, y_i) \supset connpr_G(y_1, y_{i+1})$ for every $1 < i < m$, otherwise a transformation as described in Lemma 8 could be further applied to CS. As $connpr_G(x, y) \subseteq upreds(P) \times upreds(P)$ and $|2^{upreds(P) \times upreds(P)}| = 2^{p^2}$, and there at least 2^{p^2} distinct values for $connpr_G(y_1, y_i)$, when $1 < i < m$, there must be an $1 < i < m$ such that $connpr_G(y_1, y_i) = \emptyset$. But in this case $(y_1, y_i) \in bl$ (as the two nodes also have equal content) which contradicts with the fact that $y_i \neq end$. The other cases are proved similarly. □

We proceed next to the actual counting of the number of nodes in an arbitrary branch B with given content $s \in 2^{upreds(P)}$.

Lemma 10. Let CS be a relaxed clash-free complete $\mathcal{A}_1$-completion structure. Furthermore, let $shrunk(CS) = \langle EF, \text{CT}, \text{ST}, G \rangle$ and let B be a branch in a tree T_c in EF. Then, for every set $S \in 2^{upreds(P)}$, B contains at most $k = 2^p(2^{p^2} - 1) + 3$ nodes x such that $\text{CT}(x) = S$, where $p = |upreds(P)|$.

Proof.
Without loss of generality, assume B has the form depicted in Figure 3.5. When counting the maximum number of nodes with content S in B, we distinguish between the following three different cases as regards S:

- there is no node $x \in T_c$ with $c <_{T_c} x <_{T_c} x_1$ such that $\text{CT}(x) = S$, and there is no $1 \leqslant i \leqslant n$ such that $\text{CT}(x_i) = S$. In this case there is at most one node with content equal to S in $path_{T_c}(c, x_1)$ (the root), there are at most 2^{p^2} nodes in each $path_{T_c}(x_i, x_i + 1)$ and at most 2^{p^2} nodes in $path_{T_c}(x_n, end)$ (according to Lemma 9); for the last path there cannot be $2^{p^2} + 1$ nodes as that would mean that end is a blocked node with content equal to S, so there would be a blocking node with content equal to S, which contradicts with the hypothesis that there is no blocking node with content equal to S. Also there are at most $2^p - 1$ blocking nodes: if there would be 2^p such nodes, (the maximum number by Corollary 2) there would remain no valid value for S. Summing all up, in this case there are at most $2^{p^2}(2^p - 1) + 1$ nodes with content equal to S.

- there is no node x such that $c <_{T_c} x <_{T_c} x_1$ such that $\text{CT}(x) = S$ but there is a node x_i, $1 \leqslant i \leqslant n$ such that $\text{CT}(x_i) = S$. In this case, there is no node x such that $c <_{T_c} x <_{T_c} x_i$ which has content equal to S (Lemma 5), and thus $path_{T_c}(c, x_1)$ has at most 1 node with content equal to S (the root). $path_{T_c}(x_i, x_{i+1})$ has at most 2^{p^2} nodes, every path

(x_j, x_{j+1}), where $i < j < n$, has at most $2^{p^2} - 1$ nodes, and the path (x_n, end) has at most 2^{p^2} nodes (according to lemma 9). Summing all up, in this case there are at most $(2^{p^2} - 1)(n - i + 1) + 3$ nodes with content equal to S, where n is the number of blocking nodes. There are at most 2^p blocking nodes (corollary 2), and the maximum of the expression is met when $i = 1$ and $n = 2^p$, and is $2^p(2^{p^2} - 1) + 3$.

- there is a node x such that $c <_{T_c} x <_{T_c} x_1$ and $\textsc{ct}(x) = S$. In this case, $\textsc{ct}(x_i) \neq S$, for every $1 \leqslant i \leqslant n$ (lemma 5). The counting is as follows: $path_{T_c}(c, x_1)$ has at most one node with content equal to S (x), otherwise a transformation as described in Lemma 8 could be applied, $path_{T_c}(x_i, x_{i+1})$ has maximum $2^{p^2} - 1$ nodes, $1 \leqslant i < n$ and the path (x_n, end) has at most 2^{p^2} nodes (according to Lemma 9). Also, there are at most $2^p - 1$ blocking nodes (if there would be 2^p such nodes, the maximum indicated by corollary 2 there would remain no valid value for S). Summing all up, in this case there are at most $(2^{p^2} - 1)(2^p - 1) + 1$ nodes with content equal to S.

From the three cases the maximum of number of nodes with content equal to a given set S in any branch B of a tree $T_c \in F$ is bounded by $2^p(2^{p^2} - 1) + 3$, which is exactly k from the applicability rule (ix) Redundancy. $\square$

A direct corollary of Lemma 10 and the applicability rule (ix) Redundancy is:

Corollary 4. Let CS be a relaxed clash-free complete $\mathcal{A}_1$-completion structure. Then, $shrunk(CS)$ is a complete clash-free $\mathcal{A}_1$-completion structure for p with respect to P.

3.5.4 Complexity Analysis

Proposition 9. $\mathcal{A}_1$ runs in the worst case in non-deterministic double exponential time in the size of the program.

Proof.
We bound the number of steps needed to expand an initial completion structure for checking satisfiability of a unary predicate p with respect to a FoLP P to a complete completion structure for p with respect to P. We split this task in two orthogonal tasks:

1. *Counting the maximum number of nodes in a completion structure*: an interconnected forest in a completion structure has in the worst case a double exponential number of nodes in the size of the program:
   - there are at most $k + 1$ nodes with equal content on any branch of a tree in the completion, where $k = 2^n(2^{n^2} - 1) + 2$, and $n = |upreds(P)|$,
   - there are 2^n different possible configurations for the content of a unary node,
   - the number of trees in the interconnected forest is bounded by $|cts(P)| + 1$, and
   - the arity of any such tree is bounded by $r = rank(P)$.

Thus, the bound on the number of nodes is $b = (|cts(P)| + 1)r^{2^{2n+n^2} - 2^{2n} + 2^{n+1}}$, which is double exponential in the size of P.

2. *Counting the maximum number of steps needed to saturate the content of a given node*: first of all, we notice that it takes polynomial time to justify the presence of a unary predicate in the content of a node and the presence of a (possibly negated) binary predicate in the content of an arc. Justifying the presence of a negated unary predicate in the content of a node takes exponential time (all groundings of certain unary rules have to be considered, and, in general, there is an exponential number of such groundings). As such, justifying the content of a node takes exponential time, while justifying the content of an arc takes polynomial time, and saturating a node takes exponential time.

By combining 1) and 2) above, we obtain that a complete completion structure for p with respect to P will be evolved in at most double exponential number of steps in the size of P, and as the algorithm is non-deterministic, it runs in the worst case in non-deterministic double exponential time.

3.5.5 FoLPs Have the Bounded Finite Model Property

Proposition 10. FoLPs have the *bounded finite model property*: if for a certain FoLP P and unary predicate p, p is satisfiable with respect to P, then there exists an open answer set of P which satisfies p with a finite bounded size: the size of its universe is at most double exponential in the size of the program P.

Proof. The property follows as a corollary of the forest model property, the soundness and the completeness results: Proposition 5 shows that if p is satisfiable with respect to P, then it is satisfied by a forest-shaped open answer set of P. The proof of Proposition 8 (Completeness) shows that from an arbitrary forest-shaped open answer set of a FoLP P which satisfies a unary predicate p, it is possible to construct a clash-free complete completion structure for p with respect to P with at most b nodes, where b is defined as in the proof of Proposition 9 (Complexity).

At the same time, the construction of the open interpretation in the proof of Proposition 7 (Soundness) shows that from any clash-free complete structure for p with respect to P, it is possible to construct an open answer set of P which satisfies p and whose universe is exactly the set of nodes of the completion.

Thus, if p is satisfiable, it is satisfied by an open answer set of P with size bounded by b, which is double exponential in the size of P. $\square$

3.5.6 Reduction to Answer Set Programming Using the Bounded Finite Model Property

In this section we show how the result obtained in the previous section opens the way for standard Answer Set Programming with FoLP.

Let P be a FoLP and $p \in \mathit{upreds}(P)$. We define the program P_k to be a new program obtained from P by addition of a constraint

$$\leftarrow \mathit{not}\ p(x_1), \ldots, \mathit{not}\ p(x_k), \mathit{not}\ p(c_1), \ldots, \mathit{not}\ p(c_m)\ ,$$

where k is a natural number, $1 \leqslant k \leqslant b - |\mathit{cts}(P)|$, with b being the bound on the size of the universe established in Section 3.5.5, x_1, ..., x_k are some newly introduced individuals, and $\mathit{cts}(P) = \{c_1, \ldots, c_m\}$. To check whether p is satisfiable with respect to P, we can simply check answer set existence for the programs P, $P_1, \ldots, P_{b-|\mathit{cts}(P)|}$. Once an answer set is found for any of these programs it can be concluded that p is satisfiable and the procedure is curtailed. From the bounded finite model property, we know that if p is satisfiable, it is satisfied by a model with maximal size b. Thus, if none of the programs P_k is consistent, p is not satisfiable.

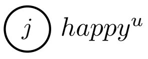
As b is double exponential in the size of P, $b - |\mathit{cts}(P)|$ is also double exponential in the size of P. It follows that constructing the programs P_1, ..., $P_{b-|\mathit{cts}(P)|}$ starting from P takes also time double exponential in the size of P (one has to add to P in each case a new rule with $1, 2,$..., $b - |\mathit{cts}(P)|$ atoms). Each P_i, for $1 \leqslant i \leqslant b - |\mathit{cts}(P)|$, is a non-ground program with bounded predicate arities, so according to [Eiter et al., 2007], checking its answer set existence is in $\mathrm{NP}^{\mathrm{NP}} (= \Sigma_2^p)$. As the maximum size of each P_i is double exponential in the size of P, the algorithm runs in the worst case in non-deterministic double exponential time with an oracle in NP, which is thought to be worse than non-deterministic double exponential time, the running time of our algorithm.

3.6 Illustration of the Algorithm

Consider a slightly modified version of the FoLP program described in Example 12, in which the constraints have been replaced by unary rules as described in Section 2.4, and the last rule has been removed. We will refer to this program as P.

$$\begin{array}{rrcl}
r_1: & \mathit{happy}(X) & \leftarrow & \mathit{sees}(X, Y), \mathit{friend}(X, Y), \mathit{happy}(Y) \\
r_2: & \mathit{happy}(X) & \leftarrow & \mathit{sees}(X, Y), \mathit{enemy}(X, Y), \mathit{unhappy}(Y) \\
r_3: & \mathit{unhappy}(X) & \leftarrow & \mathit{sees}(X, Y), \mathit{friend}(X, Y), \mathit{not}\ \mathit{happy}(Y) \\
r_4: & \mathit{unhappy}(X) & \leftarrow & \mathit{sees}(X, Y), \mathit{enemy}(X, Y), \mathit{happy}(Y) \\
r_5: & \mathit{happy}(X) & \leftarrow & \mathit{friend}(X, Y), \mathit{friend}(X, Z), Y \neq Z \\
r_6: & \mathit{sees}(X, Y) \vee \mathit{not}\ \mathit{sees}(X, Y) & \leftarrow & \\
r_7: & \mathit{friend}(X, Y) \vee \mathit{not}\ \mathit{friend}(X, Y) & \leftarrow & \\
r_8: & \mathit{enemy}(X, Y) \vee \mathit{not}\ \mathit{enemy}(X, Y) & \leftarrow & \\
r_9: & c(X) & \leftarrow & \mathit{not}\ c(X), \mathit{happy}(X), \mathit{unhappy}(X) \\
r_{10}: & d(X, Y) & \leftarrow & \mathit{not}\ d(X, Y), \mathit{friend}(X, Y), \mathit{enemy}(X, Y) \\
r_{11}: & \mathit{unhappy}(j) & \leftarrow & \mathit{hungry}(j)
\end{array}$$

We want to check the satisfiability of the predicate *happy* w.r.t. P. For this purpose, we first define an initial completion structure for *happy* w.r.t. P: $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$. There is one constant in P, j, so there will be a tree with root j, T_j, in EF; further on, we choose not to include a tree with anonymous root in EF, and thus the only choice for placing the initial constraint $happy$ is in the content of node j. The initial status of $happy$ in this node is unexpanded, so the status function is updated accordingly. The graph $G = (V, A)$ which keeps track of dependencies between atoms in the model in construction is initialized such that $V = \{happy(j)\}$, and $A = \emptyset$. The picture below depicts the initial completion structure for $happy$ w.r.t. P. Note that the fact that the status of $happy$ is unexpanded is marked by appending the superscript u to $happy$.



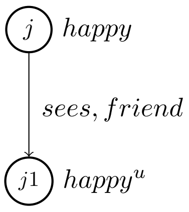
According to the (i) Expand-Unary-Positive rule, the presence of the unexpanded predicate $happy$ in the content of a node j, or in other words of $happy(j)$ in the corresponding tentative open answer set, has to be justified by means of a unary rule with head predicate $happy$ and head term which matches j. We apply (i) Expand-Unary-Positive by choosing the unary rule r_1 to justify $happy(j)$:

$$r_1 : happy(X) \leftarrow sees(X, Y), friend(X, Y), happy(Y).$$

A new successor $j1$ is created for j in T_j and the predicates $sees$ and $friend$ are inserted in the content of the arc $(j, j1)$, and the predicate $happy$ is inserted in the content of $j1$. G is also updated as follows:

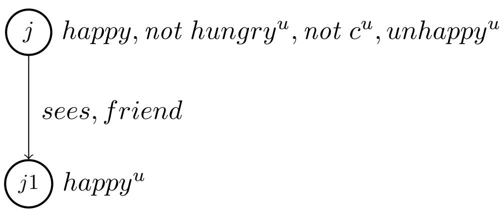
$$V = V \cup \{happy(j1), sees(j, j1), friend(j, j1)\},$$
$$A = A \cup \{(happy(j), sees(j, j1)), (happy(j), friend(j, j1)), (happy(j), happy(j1))\}.$$

In other words, $happy(j)$ is in the model in construction if there is an individual $j1$ such that $sees(j, j1)$, $friend(j, j1)$, and $happy(j1)$ are all present in the open answer set in construction. Next figure depicts the situation after the application of this expansion operation for $happy(j)$. The predicate $happy$ in the content of $j1$ is marked as unexpanded. The other predicates are either expanded ($happy$ in the content of j) or free predicates ($sees$ and $friend$ in the content of $(j, j1)$), and as such they are not superscripted.



Once again, the only unexpanded predicate is $happy$, only this time the occurrence in the content of $j1$. However, we cannot proceed to its expansion since j is not saturated: there are predicates which do not appear either in a positive or a negative form in the contents of j and its outgoing arcs. Remember that according to applicability rule *(vii) Saturation* no expansions can be performed on a node which is not a constant until its predecessor is saturated.

We pick the predicate $hungry$ and apply the (ii) Choose-Unary rule by inserting $not\ hungry$ in the content of j. It is not possible to apply the (iii) Expand-Unary-Negative rule with respect to $not\ hungry$ in the content of j, as one can still apply the (ii) Choose-Unary rule: for example, neither c nor $not\ c$ occur in the content of j. We apply the (ii) Choose-Unary rule to c and choose to insert $not\ c$ in the content of j. Once again, j is not saturated and the (ii) Choose-Unary rule can be applied with respect to $unhappy$; we choose to insert $unhappy$ in the content of j:

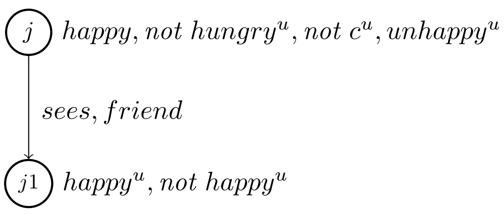


Note that c (which is used to simulate a constraint) does not appear in the head of any rule other than r_9 and, thus, is never satisfiable. As c does not appear in the body of any other rule other than r_9, the (ii) Choose-Unary rule has to be applied with respect to c and every node in the model in construction in order to saturate the content of the respective node. To simplify the exposition, in the following we will always choose to insert $not\ c$ in the content of every node (it is easy to see that any other choice would lead to a contradiction). The same reasoning applies to d: for every arc, there has to be an application of the (vi) Choose-Binary rule with respect to d and the respective arc and the choice in each case will be to insert $not\ d$ in the content of the arc.

Moving forward, among the unexpanded predicates in the content of j, we pick $unhappy$ as the next candidate for expansion, as the (i) Expand-Unary-Positive rule has priority over the (iii) Expand-Unary-Negative rule. Among the rules with head predicate $unhappy$ and head term which matches j, we pick the following for justifying the presence of $unhappy(j)$ in the model in construction:

$$r_3 : unhappy(X) \leftarrow sees(X, Y), friend(X, Y), not\ happy(Y).$$

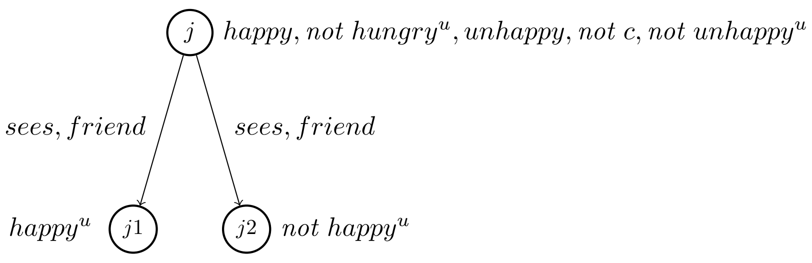
Either the successor of j, $j1$, is reused or a new one is introduced to satisfy the non-local part of the rule. Suppose we pick up the already existing successor, $j1$. In this case, $sees$ and $friend$ are inserted into the content of the arc $(j, j1)$ (they are already there), while $not\ happy$ is inserted into the content of $j1$: this leads to a contradiction as now both $not\ happy$ and $happy$ are in the content of $j1$.



The algorithm backtracks and introduces a new successor for j, $j2$: $sees$ and $friend$ are inserted into the content of the arc $(j, j2)$, and $not\ happy$ is inserted in the content of the node $j2$. Now the predicate $unhappy$ in the content of node j can be marked as expanded, and we proceed further with the expansion process. Suppose we pick $not\ c$ for expansion. There is a single ground rule which defines $c(j)$:

$$c(j) \leftarrow not\ c(j), happy(j), unhappy(j).$$

According to the (iii) Expand-Unary-Negative rule, the body of this ground rule has to be refuted. There are three possible choices for doing this: inserting c, $not\ happy$, or $not\ unhappy$ into the content of j. Each of the three choices leads to a contradiction. The figure below depicts the case when $not\ unhappy$ was chosen to refute the body of the rule.



The algorithm backtracks to the previous choice, which was the choice of the unary rule to justify $unhappy$ in the content of j. There are still two more unary rules in P whose head atoms match $unhappy(j)$: these are r_4 and r_{11}. However, from the previous developments, one can see that even if $unhappy$ is satisfied in some other way, one will eventually reach a contradiction due to the presence of $happy$, $unhappy$, and $not\ c$ in the content of j. As such, we skip the remaining two choices as concerns rules to justify $unhappy(j)$. Backtracking further, one has to retract $unhappy$ from the content of j, and insert $not\ unhappy$ instead, and mark it as unexpanded.

In the next step, $not\ unhappy$ is selected for expansion. According to the (iii) Expand-Unary-Negative rule, every ground rule which defines $unhappy(j)$ has to be considered and its body has to be refuted. There is one instantiation for each rule whose head matches $unhappy(j)$:

- r_3: $unhappy(j) \leftarrow sees(j,j1), friend(j,j1), not\ happy(j1)$. The body of this rule has to be refuted: the atoms $sees(j,j1)$ and $friend(j,j1)$ are already part of the tentative open answer set so they cannot be refuted. The only remaining choice is to refute the literal $not\ happy(j1)$, thus to insert the predicate $happy$ into the content of node $j1$.

- r_4: $unhappy(j) \leftarrow sees(j,j1), enemy(j,j1), happy(j1)$. Here the only choice which does not lead to contradiction is inserting the negated predicate symbol $not\ enemy$ into the content of $j1$. The predicate $enemy$ is a free predicate, defined only by a free rule, so it is trivially expanded.

- r_{11}: $unhappy(j) \leftarrow hungry(j)$. The body of this rule is already refuted by the presence of $not\ hungry$ into the content of node j.

Finally, in order to saturate j, the (vi) Choose-Binary rule is applied and $not\ d$ is inserted into the content of arc $(j, j1)$. Then, $not\ d$ is expanded using the (v) Expand-Binary-Negative rule. It can be observed that the body of the only ground rule derived from r_{10}:

$$d(j,j1) \leftarrow not\ d(j,j1), friend(j,j1), enemy(j,j1)$$

is already refuted by the presence of $not\ enemy$ in the content of $(j, j1)$:

$$\begin{tikzcd}
j \quad happy, not\ hungry, not\ unhappy, not\ c \arrow[d, "sees, friend, not\ enemy, not\ d"] \\
j1 \quad happy^u
\end{tikzcd}$$

At this moment, j is saturated and by means of applicability rule *(vii) Saturation* it is possible to proceed with the expansion of its successor $j1$. One can see that the content of $j1$ is included in the content of j, so according to rule *(viii) Blocking*, $(j, j1)$ is a candidate blocking pair. However G contains the arc $(happy(j), happy(j1))$, so $connpr_G(j, j1) \neq \emptyset$, and the second condition of the blocking rule is not fulfilled. Intuitively, if one would justify the content of $j1$ in a similar manner as the content of j, an infinite chain of the type $happy(j), happy(j1), \ldots$ would be present in the model in construction, each atom in the set being supported by the next one in the set, thus there would be atoms in the model which are not finitely supported.

Thus, $j1$ cannot be blocked, and its content has to be expanded. In order to justify the presence of $happy(j1)$ in the tentative open answer set, rule r_5 is picked up. To satisfy the body of some ground version of the rule, two distinct successors of $j1$, $j11$ and $j12$, are created leading to the ground rule:

$$happy(j) \leftarrow friend(j,j1), friend(j,j2), j1 \neq j2.$$

The (i) Expand-Unary-Positive rule is applied and $friend$ is asserted to the content of both $(j1, j11)$ and $(j1, j12)$. Arcs $(happy(j1), friend(j1, j11))$, and $(happy(j1), friend(j1, j12))$ are added to A in G to capture the new dependencies between atoms in the tentative open answer set.

$$\begin{tikzcd}
 & j \quad happy, not\ hungry, not\ unhappy, not\ c \arrow[d, "sees^u, friend^u, not\ enemy, not\ d"] & \\
 & j1 \quad happy \arrow[dl, "friend^u"'] \arrow[dr, "friend^u"] & \\
j11 & & j12
\end{tikzcd}$$

Now we proceed to saturate node $j1$ by choosing to add $not\ c$, $not\ hungry$, and $not\ unhappy$ to the content of $j1$ by repeated applications of the (iii) Expand-Unary-Negative rule. The first two additions are expanded in a similar manner as their counterparts in the content of j. As concerns *not unhappy*, all possible groundings of the three rules which define the predicate *unhappy* have to be considered again. The justification with respect to rule r_{11} is similar as in the case of node j, as the rule is a local rule. There are two successors of $j1$, $j11$ and $j12$, so there are two ground versions of r_3:

$$unhappy(j1) \leftarrow sees(j1, j11), friend(j1, j11), not\ happy(j11), \text{ and}$$

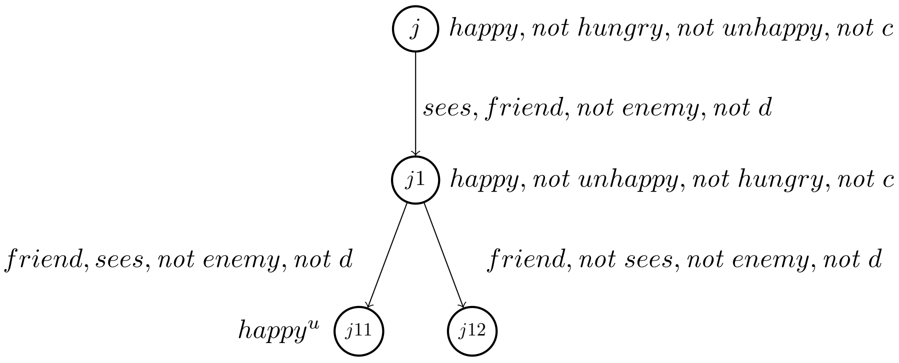
$$unhappy(j1) \leftarrow sees(j1, j12), friend(j1, j12), not\ happy(j12),$$

and two ground versions of rule r_4:

$$unhappy(j1) \leftarrow sees(j1, j11), enemy(j1, j11), happy(j11), \text{ and}$$

$$unhappy(j1) \leftarrow sees(j1, j12), enemy(j1, j12), happy(j12).$$

The bodies of all these four ground rules have to be refuted. In order to do this, *happy* is added to the content of node $j11$, *not sees* is added to the content of arc $(j1, j12)$, and *not enemy* is added to both the contents of arc $(j1, j11)$ and arc $(j1, j12)$. Finally, we saturate node $j1$ by completing the contents of the arcs $(j1, j11)$ and $(j1, j12)$ in a similar manner as we did for the arc $(j, j1)$.


j
happy, not hungry, not unhappy, not c
sees, friend, not enemy, not d
j1
happy, not unhappy, not hungry, not c
friend, sees, not enemy, not d
friend, not sees, not enemy, not d
happy^u
j11
j12

At this moment, $j1$ is also saturated and we observe that the contents of both its successors are included in its own content. Unlike the case where $\text{CT}(j1) \subset \text{CT}(j)$, but $connpr_G(j, j1) \neq \emptyset$, both $connpr_G(j1, j11) = \emptyset$, and $connpr_G(j1, j12) = \emptyset$, thus both $(j1, j11)$ and $(j1, j12)$ are blocking pairs. Thus, the completion structure depicted in the figure above is a complete clash-free completion structure and p is satisfiable with respect to P.

As anticipated already in the applicability rule (vii) Blocking in Section 3.3, a complete clash-free completion structure can be unravelled to a forest-shaped open answer set. In the current case, this would be achieved by making the contents of $j11$ and $j12$ to be identical to the content of $j1$ and also justifying them similarly to the content of $j1$. This will give rise to two new successors for both $j11$ and $j12$, which again will be justified in the same manner, etc. The obtained forest model is depicted in Figure 3.6.

The corresponding forest-shaped open answer set which satisfies *happy* is (U, M), with:

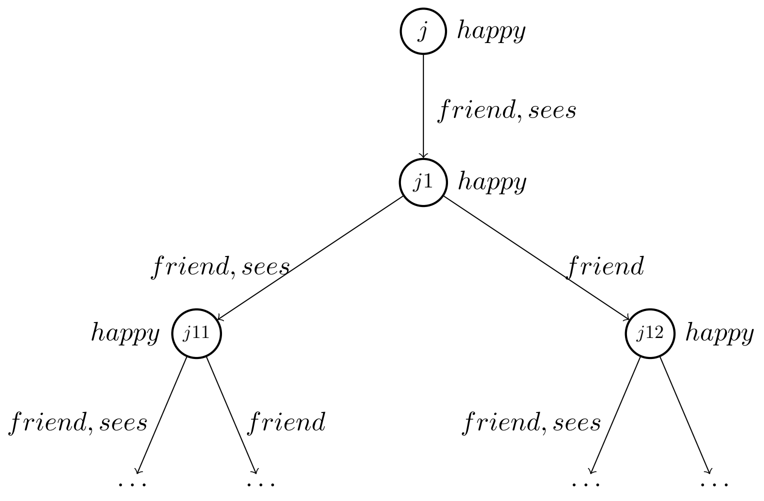


Figure 3.6: Unravelling the clash-free completion structure to a tree-shaped model

$$U = \{j, j1, j11, j12, j111, j112, \ldots\}$$

$$\begin{aligned} M = \{happy(j)\} &\cup \{happy(js) \mid s = 1, 11, 12, 111, 112, \ldots\} \\ &\cup \{friend(js, js1) \mid s = 1, 11, 12, 111, 112, \ldots\} \\ &\cup \{friend(js, js2) \mid s = 1, 11, 12, 111, 112, \ldots\} \\ &\cup \{sees(js, js1) \mid s = 1, 11, 12, 111, 112, \ldots\} \end{aligned}$$

Alternatively, the clash-free complete completion structure can be unravelled to a graph-shaped open answer set as described in the soundness proof in Section 3.5. The model is depicted by Figure 3.7.

In this case, the model (U, M) is finite. More precisely:

$$U = \{j, j1, j11, j12\}, \text{ and}$$

$$M = \{happy(j), friend(j, j1), sees(j, j1), happy(j1), friend(j1, j11), \ldots, \}.$$

3.7 Discussion and Related Work

3.7.1 Connection with DL Tableau Algorithms

This chapter introduced $\mathcal{A}_1$, a tableau algorithm for satisfiability checking of unary predicates with respect to FoLPs which runs in the worst-case in non-deterministic double exponential time in the size of the program. Note that such a high complexity is typical when dealing with

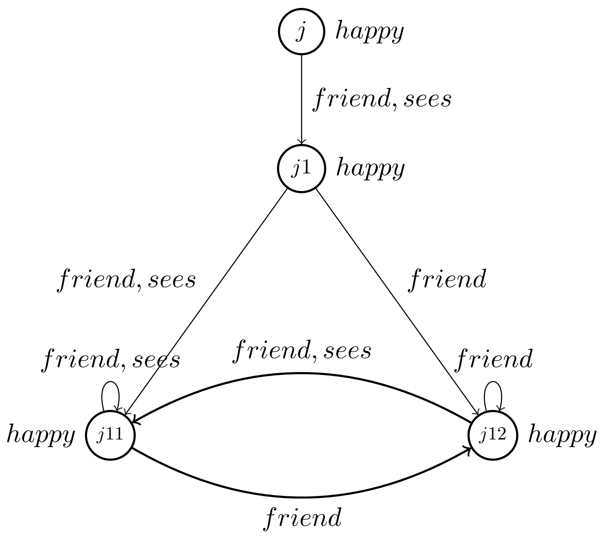


Figure 3.7: Unravelling the clash-free completion structure to a graph-shaped model

tableau-like algorithms. For example in Description Logics, although satisfiability checking in $\mathcal{SHIQ}$ or $\mathcal{SHOQ}$ is EXPTIME-complete, practical algorithms run in non-deterministic double exponential time [Horrocks and Sattler, 2001, Tobies, 2001, Horrocks et al., 2000]. Even in the case of the optimized hyper-tableau reasoner for $\mathcal{SHOIQ}^+$ HermiT [Motik et al., 2009b], which uses a form of anywhere blocking, the worst-case running time behaviour remains non-deterministic double exponential.

As we will see in Chapter 5, the introduction of a form of caching/anywhere blocking will greatly improve our algorithm, leading to a NEXPTIME procedure. However, these improvements come at the price of a much more complicated underlying data structure and blocking/redundancy conditions.

It is worthy to note that $\mathcal{A}_1$ uses a form of single subset blocking, similar to the one used in the case of the tableau algorithm for $\mathcal{SHOQ}$ [Horrocks and Sattler, 2001]. This is not surprising, as, as mentioned in the Introduction, reasoning with $\mathcal{SHOQ}$ KBs (consistency checking, concept satisfiability, etc.) can be reduced to reasoning with FoLPs. However, unlike the typical case for tableau algorithms for DLs [Baader et al., 2003a], label comparison, i.e. subset blocking in this case, is not enough for pruning the expansion of a branch in a completion forest. As we explained throughout this chapter, relying just on that condition could potentially lead to circular motivations of atoms in the model, which is not allowed in a non-monotonic setting.

There are several extensions of DL which adopt a minimal-style semantics like autoepistemic [Donini et al., 2002], default [Baader and Hollunder, 1995] and circumscriptive DL [Bonatti et al., 2006, Grimm and Hitzler, 2008, Grimm and Hitzler, 2009]. The first two are restricted to reasoning with explicitly named individuals, while [Grimm and Hitzler, 2008, Grimm and Hitzler, 2009] allow for defeats to be based on the existence of unknown individuals. A tableau-based method for reasoning with the DL $\mathcal{ALCO}$ in the circumscriptive case has been introduced in [Grimm and Hitzler, 2007]. A special preference clash condition is introduced there to distinguish between minimal and non-minimal models which is based on constructing a new classical DL knowledge base and checking its satisfiability.

As a further analogy to the DL world, we mention the connection between the Choose-Unary/Binary rules introduced in this chapter and the so-called 'internalization' of TBox performed by DL tableau algorithms when checking concept satisfiability [Horrocks et al., 1999]. Internalization consists in reducing reasoning with respect to the TBox to satisfiability checking of a new concept which is constructed by taking into account all axioms in the TBox and not only those on which the initial concept checked to be satisfiable depends on. The effect of our Choose-Unary/Binary rules is similar: as every possible atom is asserted as either being part or not being part of the constructed model; every rule in the program grounded with the universe of the model in construction is satisfiable in the respective model.

3.7.2 Reflection on Using Standard ASP Reasoning vs. the Tableau Method

In Section 3.5.6 we showed how the bounded finite model property established in this chapter opens the way also for standard Answer Set Programming reasoning with FoLPs. This provides an algorithm which runs in the worst case in non-deterministic double exponential time with an oracle in NP, which is thought to be worse than non-deterministic double exponential time, the running time of our algorithm.

Aside from worst-case running time complexity considerations, one advantage of the tableau method presented in this chapter compared to the standard ASP based approach, is that it offers more guidance concerning the size of the universe needed to make a predicate p satisfiable. At any step of the algorithm, we might find out that, in order to motivate the presence of a predicate in the open answer set, the current node has to have a minimum number of successors which can be greater than one (due to successor variables inequalities in unary rule bodies). While in the tableau approach we simply introduce the desired number of successors, in the standard ASP based approach, one has to check also programs with intermediary sized universes which do not satisfy p.

It is also worthy to mention that the algorithm we described in this chapter was actually the device that allowed us to establish a maximal bound on the universe size needed to make a predicate satisfiable.

3.7.3 Connection with ASP Reasoning Procedures

While being a tableau-based algorithm, $\mathcal{A}_1$ is connected to other procedures used to construct models for formalisms with open domains, at the same time it is tightly connected to procedures for reasoning about the stable model semantics.

Supported, Well-supported Models, and Clark's Completion

An alternative characterization for the stable model semantics was provided in [Fages, 1991]. There, answer sets for normal logic programs are equated to so-called *well-supported interpretations*. An interpretation is well-supported iff (1) every atom in the interpretation is *supported*, i.e. there exists a rule whose body is satisfied in the interpretation, and (2) every such atom is *well-supported*, i.e. if one considers the dependency graph capturing the relations between atoms in the interpretation induced by the supportedness relation, the graph contains no infinite path or cycle. It is easy to see that such a characterization holds also for our setting where disjunction is used sparingly, only in free rules. In fact, if one considers the expansion rules used by $\mathcal{A}_1$, it is obvious that:

- the (i) Expand-Unary-Positive and the (iv) Expand-Binary-Positive rules ensure that every atom which is part of the constructed model is supported,
- the (iii) Expand-Unary-Negative and the (v) Expand-Binary-Negative rules ensure that every atom which is not part of the constructed model is not supported, and
- the check on the dependency graph G which is part of the blocking condition, ensures that every support is finite, i.e. every atom in the constructed model depends on a finite number of atoms in the model.

In [Fages, 1994] it has been also shown that for a given logic program, its supported models coincide with the (Herbrand) models of its Clark's completion. For a ground logic program P, Clark's completion [Clark, 1978] is a propositional theory obtained from P as follows: for every atom p and rules of the type $p \leftarrow \beta_i$ in P, for $1 \leqslant i \leqslant n$, the completion contains the following equivalence: $p \equiv \bigvee_{1 \leqslant i \leqslant n} \beta_i$. The notion can be extended straightforwardly to the case of non-ground programs. Thus, by dropping the checks on the dependency graph G which are part of the blocking condition, our algorithm would check set satisfiability with respect to supported models, i.e. models of the program's Clark completion (with respect to a given universe). Note that in this case the blocking condition becomes effectively a subset-blocking condition and as such it is enough to achieve termination: there is no need to further employ the redundancy rule.

The connection between the stable model semantics and Clark's completion has been exploited also in [Lin and Zhao, 2002] and [Lin and Zhao, 2004]: there, it is shown how normal logic programs under the stable model semantics can be translated into propositional theories. The translation consists in the addition to the Clark's completion for the given program of a set of so-called 'loop formulas'. The role of these formulas is to filter from the set of models of Clark's completion, i.e. the set of supported models, those which are not well-supported.

A loop is a cycle in the atom dependency graph of a given program. In order to be well-supported, any atom which is part of a loop cannot use as its support a rule whose body contains an atom which is also part of the loop. The work lies at the basis of the answer set solver `ASSAT` and has been extended to the case of disjunctive logic programs in [Lee and Lifschitz, 2003]. In view of this, the checks on the dependency graph G performed by our tableau algorithm have as purpose the elimination of models which are not well-supported, i.e. models which contain loops and/or infinite length dependency paths.

Goal Rewriting System for Brave Reasoning with ASP

In the area of proof systems for Answer Set Programming, [Lin and You, 2002] describes a goal rewrite system for brave reasoning under the stable model semantics which is sound and complete only for partial stable models. If the program has no odd loops (cycles via negation in the predicate dependency graph of the program), its partial stable models and its stable models coincide. Note that such programs cannot express constraints as these are just syntactic sugar for rules in which a predicate depends negatively on itself. The problem with constraints is that they can render the program inconsistent, and thus, the rewriting, even if it is successful, is no longer valid.

FoLPs allow for odd loops and $\mathcal{A}_1$ deals with such loops by exploring/asserting new constraints to the model beyond the dependencies generated by the predicate checked to be satisfiable: a complete answer set is constructed by taking care that the content of every node in the completion structure is saturated. As already explained in Section 3.2.2 and in Section 3.2.6, this is taken care of by the (ii) Choose-Unary and the (vi) Choose-Binary rules.

As concerns termination, [Lin and You, 2002] distinguishes between positive, negative, odd, and even loops in the predicate dependency graph and deals with them accordingly. However, there is no blocking mechanism and, as such, also no check for infinite length dependency chains: for achieving termination, [Lin and You, 2002] considers only "domain restricted programs", which can be instantiated only on domain predicates over variables which do not appear in the head. Note that FoLPs do not have such a restriction: there are FoLPs (actually, even CoLPs) in which no constant occurs and which still have infinite groundings. As such, we need the rather complicated blocking mechanism for ensuring that there are no atoms with infinite justifications in the open answer set.

Transition-Based Framework For Computing Supported and Stable Models of Ground Logical Programs

An extension of an abstract framework for executing DPLL [Nieuwenhuis et al., 2006] which computes supported models and stable models of a ground logical program is described in [Lierler, 2008]. The framework employs a graph structure for encoding different computation paths. Models are constructed in a bottom-up fashion: transition rules prescribe how new atoms are derived. The following transition rules are employed to derive supported models:

- *Unit Propagation*: a supported atom, i.e. an atom for which there exists a body which is satisfied in the partial model, becomes part of the model - this rule is similar to our (i) Expand-Unary-Positive and (iv) Expand-Binary-Positive rules, but it acts in the reverse direction.

- *Decide*: a literal is inserted as part of the model and is marked as a decision literal. The rule is similar to our (ii) Choose-Unary and (vi) Choose-Binary expansion rules. However, unlike in our case, here there is no check whether the counterpart atom is already part of the model in construction.

- *Fail*: when the constructed model is inconsistent and it contains no decision literals, i.e. there is no possibility to backtrack to a choice point, the derivation fails.

- *Backtrack*: when the constructed model is inconsistent and it contains some decision literal, the system backtracks to a choice point, where a literal has been introduced using the *Decide* rule and the literal is flipped. In our algorithm, being non-deterministic, the backtracking mechanism is implicit.

For simulating the behaviour of the answer set solver SMODELS [Simons et al., 2002], [Lierler, 2008] introduces an extra derivation rule whose role is to filter models which are not well-supported. The rule is called *Unfounded* and it prescribes that every atom which is part of some unfounded set with respect to the partial constructed model is false, i.e. it is not part of the model. An unfounded set S with respect to a set of literals M is a counterpart of the notion of loop: it consists of atoms which are potentially supported (with respect to M) only by rules whose bodies contain at least one atom from the set S itself. By prescribing that this derivation rule is applicable only when the model in construction is a total model (every atom is either part or not part of the model), the authors describe the behaviour of another solver called SUP. This solver computes first a supported model and then, in a last stage, applies the *Unfounded* rule which triggers an inconsistency in the model in case the model is not well-supported.

Tableau Calculi For Answer Set Programming

In an effort to systematize different approaches taken by ASP reasoners, [Gebser and Schaub, 2006] introduces a family of tableau calculi for answer set programming. Like in the case of [Lierler, 2008], computations of answer sets are seen as derivations in an inference system. However, here the objects which make up a computation state are signed (true or false) atoms and rule bodies (this is different from [Lierler, 2008], where such a state was a set of signed atoms, i.e. literals). An exhaustive set of derivation rules is provided which describe both bottom-up and top-down computation and as well mechanisms to ensure well-supportedness via both unfounded sets and loop formulas. As such, they allow the simulation of the behaviour of different answer set solvers like ASSAT [Lin and Zhao, 2004], cmodels [Giunchiglia et al., 2006], DLV [Leone et al., 2006], nomore++ [Anger et al., 2005], and smodels [Simons et al., 2002]. From the set of derivation rules, we mention those which are closer to our expansion rules:

- *Backward True Atom*: if an atom is true and the bodies of all rules which have the atom in the head but one are false, the body of the respective rule is true;
- *Backward True Body*: if the body of a rule is true, all the literals in the body of the rule are true as well. This rule together with the previous rule is similar to our (iii) Expand-Unary-Positive and (v) Expand-Binary-Positive rules.
- *Backward False Atom*: if an atom is false, the body of any rule which has the atom in the head is false;
- *Backward False Body*: if the body of a rule is false and all literals in the body but one are true, the remaining literal is false. This rule together with the previous rule is similar to our (iii) Expand-Unary-Negative and (v) (iii) Expand-Binary-Negative rules.
- *Cut*: the rule prescribes a choice point between an atom and its negation. This rule is obviously related to our (ii) Choose-Unary and (vi) Choose-Binary rules.

The derivation rules described above are finer-grained than our expansion rules, as they take into account also signed bodies of rules in the program. Except for the cut rule, they are also deterministic. This is possible due to the propositional setting for such tableau algorithms.

In the case of FoLPs we do not know a priori the size of the universe needed to satisfy a certain unary predicate symbol and as such, we cannot perform an initial grounding (unless we take the incremental approach described in Section 3.5.6, where any of the above mentioned strategies for computing answer sets can be employed). The (i) Expand-Unary-Positive rule, besides offering support for a unary atom, expands also the domain of the interpretation, by introducing a certain number of successors. This latter task is inherently non-deterministic.

However, expansion rules could be possibly improved to make them less non-deterministic. Actually, an optimized version of the (iii) Expand-Unary-Negative rule is provided in the next chapter. Furthermore, in the same chapter, we provide a knowledge compilation method which pre-computes all building blocks of a model, in the form of interconnected trees of depth 1, which are essentially the parts of the model where all computation related to constructing supported models takes place. As concerns well-supportedness, we cannot check global properties like unfounded sets or loops. In the presence of an ever-expanding universe, we have to use blocking to ensure that the algorithm terminates.

Resolution Calculus for ASP Extended with Function Symbols

A resolution-based calculus for credulous reasoning in ASP extended with function symbols is introduced in [Bonatti et al., 2008]. The calculus is sound for ground order-consistent programs, i.e. programs which do not allow for any cycles in their atom dependency graph (where both positive and negative dependencies are considered, i.e. $a \leftarrow not\ b$ induces a negative dependency between a and b), and complete for ground finite recursive programs, i.e. programs for which every ground atom depends only on a finite number of other ground atoms.

The calculus is extended to the non-ground case, where it is proved to be sound for programs whose ground versions are order consistent, and complete for finitely recursive, odd-cycle free programs. In particular, the calculus is neither sound nor complete for programs which have odd cycles, which are needed for simulating constraints. As already mentioned, FoLPs allow the presence of constraints and/or can simulate such constraints. The calculus is also not complete for programs which are not finitely recursive. On the other hand, there are FoLPs which are not finitely recursive: consider a FoLP which contains the rule $a(X) \leftarrow f(X, Y), a(Y)$; grounding the program with an infinite universe leads to an infinite path in its atom dependency graph of the form $a(x_1), a(x_2), \ldots$.

CHAPTER 4

Knowledge Compilation Technique for Reasoning with FoLPs

This chapter describes a knowledge compilation technique for reasoning with FoLPs which builds on the algorithm presented in Chapter 3. The main idea is to capture all possible local computations, which are typically performed over and over again in the process of saturating the content of a node of a completion structure, by pre-computing all possible completion structures of depth one. Every such pre-computed structure will contain a node with saturated content (either corresponding to a constant or to an anonymous individual), and some successor nodes for this node which contain the constraints necessary to hold in order for the content of the saturated node to hold (which were introduced in the process of expanding the saturated node).

Such structures will be called *unit completion structures* and the node which is saturated will be referred to as the *root* node of the structure. Unit completion structures (UCSs) are defined constructively, as the result of applying the expansion rules from algorithm $\mathcal{A}_1$ introduced in Chapter 3, governed by a certain strategy that enforces that only structures containing interconnected trees up to depth one are constructed.

As the expansion rules of $\mathcal{A}_1$ are non-deterministic, it can be shown that constructing a single UCS using that algorithm takes in worst-case non-deterministic exponential time. An attempt to determinize the algorithm leads to an exponential blow-up. As such we provide an alternative constructive characterization of UCSs which allows us to compute the set of all UCSs in worst-case deterministic exponential time. Section 4.1 formally introduces the notion of unit completion structure, while Section 4.2 describes how the set of all UCSs can be computed in the worst case in deterministic exponential time.

Once the set of UCSs has been computed, they can be used to reason with Forest Logic Programs in a tableau-like manner similar to the tableau algorithm described in Chapter 3: a completion structure is constructed following the usual applicability rules. However, justifying the content of a node/arc (expanding the corresponding negated predicates in the content of some node/arc) can be done now by simply picking up a unit completion structure whose content of the root node can be matched against the content of the node in question, and by replacing the unsaturated node with the unit completion structure. By matching contents we understand that every unexpanded predicate symbol in the content of the unsaturated node occurs in the same form (negated or positive) in the content of the root of the UCS.

When a unit completion structure is used to construct a forest model by appending it to the completion structure in construction, its (successor) nodes have unexpanded content, which in turn will have to be saturated at some point in time during the expansion process. These unexpanded predicate symbols can be seen as constraints which will have to be further satisfied by appending other UCSs. Thus, unit completion structures with identical roots can impose different constraints on the model in construction. Besides the constraints which we already mentioned regarding the contents of successor (non-blocked) nodes, another type of constraints which a UCS poses on a model are the paths from an atom formed with the root node of the UCS to an atom formed with a successor node in the UCS – the more such paths there are, the harder blocking becomes when using the UCS in question.

We define a notion of *redundant unit completion structure* which captures the idea of a structure which is comparable and strictly more constraining than another unit completion structure. Once the set of all such structures is identified, they can be discarded as they can be seen as redundant building blocks of a model. Section 4.3 formalizes the notion of redundant UCS, while Section 4.4 describes the actual algorithm which uses for the construction of a model only the set of non-redundant UCSs. Finally, Section 4.5 discusses the pros and cons of the technique and some related work.

4.1 Unit Completion Structures

As mentioned in the introduction of this chapter, the intention is to obtain all completion structures whose interconnected forests consist in interconnected trees with maximal depth one that can be used as building blocks in our algorithm. We call such structures *unit completion structures*. In order to construct a unit completion structure one starts with a skeleton, an *initial unit completion structure* which is similar to an $\mathcal{A}_1$-initial completion structure for checking the satisfiability of a unary predicate p with respect to a FoLP P. An initial unit completion structure has the same interconnected forest skeleton as an $\mathcal{A}_1$-initial completion structure, but it does not impose any constraints regarding membership of predicates to nodes/arcs. This is because UCSs have to be more generic if they are to be reused as building blocks of the model. So, an initial unit completion structure is actually an $\mathcal{A}_1$-completion structure.

Definition 21. An *initial unit completion structure with root* ε for a FoLP P is an $\mathcal{A}_1$-completion structure $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$ where:

- $EF = (F, ES)$, $F = \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}$, where ε is a constant, possibly in $cts(P)$, and $ES = \emptyset$;
- $T_x = \{x\}$, for every $x \in \{\varepsilon\} \cup cts(P)$;
- $\textsc{ct}(x) = \emptyset$, for every $x \in \{\varepsilon\} \cup cts(P)$; and
- $G = \langle V, A\rangle$, $V = \emptyset$, $A = \emptyset$.

Next we specify when a unit completion structure is 'fully' expanded. As previously mentioned, the root node is saturated, while every predicate which appears in the content of a successor node is unexpanded. Furthermore, we also impose that a constant node in a UCS has non-empty content only when it is the root node itself or it is a successor node, i.e. it is linked via an arc to the root of the structure. Finally, only the root node can have successors: this ensures that the interconnected forest is indeed an interconnected tree of depth 1.

Definition 22. A *unit completion structure* $\langle EF, \textsc{ct}, \textsc{st}, G\rangle$ with root ε for a FoLP P, with $EF = (F, ES)$, is an $\mathcal{A}_1$-completion structure derived from an initial $\mathcal{A}_1$-unit completion structure with root ε for P by application of the expansion rules (i)-(vi) from Section 3.2, taking into account the applicability rules (vii)-(xi) from Section 3.3, which has the following properties:

- for all $p \in \mathit{upreds}(P)$, either:
  - $p \in \textsc{ct}(\varepsilon)$ and $\textsc{st}(p, \varepsilon) = \mathit{exp}$, or
  - $\mathit{not}\ p \in \textsc{ct}(\varepsilon)$ and $\textsc{st}(\mathit{not}\ p, \varepsilon) = \mathit{exp}$;
- for all v_1, v_2 such that $(v_1, v_2) \in A_{EF}$:
  - $v_1 = \varepsilon$,
  - for all $f \in \mathit{bpreds}(P)$, either:
    * $f \in \textsc{ct}(v_1, v_2)$ and $\textsc{st}(p, (v_1, v_2)) = \mathit{exp}$, or
    * $\mathit{not}\ f \in \textsc{ct}(v_1, v_2)$ and $\textsc{st}(\mathit{not}\ p, (v_1, v_2)) = \mathit{exp}$;
  - for all $\pm p \in \textsc{ct}(v_2)$, $\textsc{st}(\pm p, v_2) = \mathit{unexp}$;
- for all $c \in \mathit{cts}(P)$ such that $\textsc{ct}(c) \neq \emptyset$: $c = \varepsilon$ or there exists an arc $(\varepsilon, c) \in ES$,
- the structure is neither contradictory, nor circular.

As the status function is relevant only in the definition/construction of a unit completion structure, but not in the context of using such structures, in the rest of this thesis we will refer to a unit completion structure as a triple $\langle EF, \textsc{ct}, G\rangle$.

Example 22. Consider the following CoLP P:

$$\begin{array}{lrcl}
r_1: & p(X) & \leftarrow & \mathit{not}\ p(X)\\
r_2: & p(X) & \leftarrow & f(X,Y), \mathit{not}\ q(Y)\\
r_3: & p(X) & \leftarrow & f(X,Y), p(Y)\\
r_4: & p(X) & \leftarrow & f(X,Y), \mathit{not}\ q(Y), p(Y)\\
r_5: & q(X) & \leftarrow & f(X,Y), \mathit{not}\ p(Y)\\
r_6: & f(X,Y) \vee \mathit{not}\ f(X,Y) & \leftarrow &
\end{array}$$

Figure 4.1 depicts three unit completion structures for P: as P is a CoLP, and thus it contains no constants, the UCSs are all trees of depth 1. They all have the same content for the root node: $\{p, \mathit{not}\ q\}$, but they differ in the way the atom $p(\varepsilon)$ has been justified. The presence of p in the

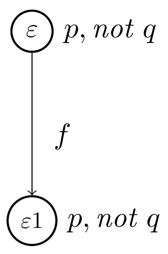
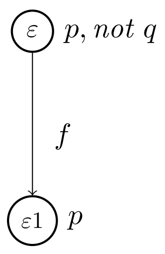
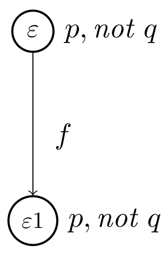
UC_1 :	UC_2 :	UC_3 :
		
$G_1 = (V_1, A_1)$	$G_2 = (V_2, A_2)$	$G_3 = (V_3, A_3)$
$V_1 : \{p(\varepsilon), p(\varepsilon 1), f(\varepsilon, \varepsilon 1)\}$	$V_2 : \{p(\varepsilon), p(\varepsilon 1), f(\varepsilon, \varepsilon 1)\}$	$V_3 : \{p(\varepsilon), p(\varepsilon 1), f(\varepsilon, \varepsilon 1)\}$
$A_1 : \begin{matrix}\{p(\varepsilon) \to f(\varepsilon, \varepsilon 1), \\ p(\varepsilon) \to p(\varepsilon 1)\}\end{matrix}$	$A_2 : \begin{matrix}\{p(\varepsilon) \to f(\varepsilon, \varepsilon 1), \\ p(\varepsilon) \to p(\varepsilon 1)\}\end{matrix}$	$A_3 : \{p(\varepsilon) \to f(\varepsilon, \varepsilon 1)\}$

Figure 4.1: Three unit completion structures for Pr: UC_1, UC_2, and UC_3.

content of the root node has been justified in the first structure by means of rule r_4, in the second structure by means of rule r_3, and in the third structure by means of rule r_2. The different ways to justify p lead to different sets of arcs in the dependency graphs belonging to each structure.

Note that r_1 is not even considered for justifying the presence of p in the root node as it acts as a constraint – it enforces that if there exists an open answer set (U, M), M should contain an atom $p(x)$, for every $x \in U$, but does not support the presence of such an atom.

On the other hand, to motivate that $not\ q$ is in the content of the root node, in each case it was shown that the body of r_5 grounded such that X is instantiated as the root node and Y as the successor node, is not satisfied, or more concretely the presence of p in the content of the successor node was enforced in each case ($not\ f$ could not be used to invalidate the triggering of the rule as f was already present in the content of the arc from the root node to the successor node in each case).

Definition 23. A unit completion structure is *final* iff all its successor nodes are blocked, or they have empty contents.

For CoLPs (but not for FoLPs) it is the case that:

Proposition 11. A final unit completion structure for a CoLP P is a complete clash-free $\mathcal{A}_1$-completion structure for P.

Obviously, one can use such final unit completion structures to derive information about the satisfiability of various unary predicates in a given CoLP: all unary predicates appearing in some node of a final UCS of a CoLP are satisfiable.

Example 23. Consider again the three unit completion structures depicted in Figure 4.1. One can notice that while the content of the successor node is included in the content of the root node in each of the cases, only for UC_3, the two nodes form a blocking pair as $paths_{G_3}(c, c1) = \emptyset$.

As the program P from Example 22 is a CoLP, UC_3 is a final unit completion structure for P, and thus, it is also a complete clash-free $\mathcal{A}_1$-completion structure. Thus, the unary predicate p is satisfiable with respect to P.

4.2 Computing the Set of Unit Completion Structures: Complexity Considerations

Note that the notion of UCS is defined constructively using the non-deterministic expansion rules (i)-(vi) of $\mathcal{A}_1$. This gives us a direct algorithm to compute a UCS which runs in the worst case in non-deterministic exponential time as an exponential number of ground rules have to be refuted to justify the presence of a negative predicate symbol in the content of the root node. In order to compute the set of all unit completion structures for a certain FoLP P, a brute force determinisation of this algorithm can be employed where instead of making a non-deterministic choice one iterates over all possible choices. As the following analysis shows, such a procedure runs in double exponential time:

1. there are at most 2^p different values for the content of a saturated node, in this case for the content of the root of a unit completion structure, where $p = |upreds(P)|$;

2. for every positive predicate symbol p in the content of some node, there is an exponential number of ways to justify it by applying the (i) Expand-Unary-Positive rule (an exponential number of possible groundings for every rule), and the actual expansion can be done in polynomial time;

3. when justifying the presence of a negated predicate symbol $not\ p$ in the content of a node using the (iii) Expand-Unary-Negative rule, one has to consider all ground rules which have $p(x)$ in the head, where x is the node under consideration. The number of such rules is exponential in the size of the program. For each such rule, exactly one literal in its body has to be refuted. Thus, the number of possible choices to refute all ground rules is double exponential in the size of the program. Once such a choice has been made, the actual refutation can be done in deterministic exponential time;

4. there are at most $degree(P)$ outgoing arcs from the root node of a UCS and at most 2^f different values for the content of an arc, where $f = |bpreds(P)|$;

5. for every positive predicate symbol f in the content of some node, there is an exponential number of ways to justify it by applying the (i) Expand-Binary-Positive rule (an exponen-

tial number of possible groundings for every rule), and the actual expansion can be done in polynomial time;

6. for every negated predicate symbol $not\ f$ in the content of some arc, the bodies of all ground rules with $f(x, y)$ in the head, where (x, y) is the arc under consideration have to be refuted. There is a linear number of such ground rules, one for each (non-ground) rule $r \in P_f$. Thus, there is an exponential number of choices to perform this task, and for each choice the actual refutation can be done in linear time.

From the analysis above, one can observe that justifying the presence of a negated predicate symbol in the content of the root node using the original (iii) Expand-Unary-Negative rule introduced in Section 3.2 takes in the worst-case double exponential time (when one considers all possible options). However, there is only an exponential number of choices for the contents of the successor nodes in a unit completion structure. This suggests that one can devise a procedure for constructing the set of unit completion structures which runs in the worst case in single deterministic exponential time.

In the following we introduce an alternative notion of UCS, defined again constructively, but this time using a different rule to expand negative predicates in the content of the root node of the UCS: the new expansion rule replaces the (iii) Expand unary negative rule and while it has the same effect, i.e. its application in conjunction with the other original expansion rules will produce alternative UCSs which can be mapped one-to-one to the original UCSs, it is more efficient. The new rule keeps track which segments in a ground unary rule have been refuted. Such a segment is either the local part of the rule or is characterized by a successor variable in the original (unground) unary rule and by the successor individual in the UCS in construction with which the variable has been replaced in the ground rule. Note that such segments will occur in different groundings of the same unary rule, so once such a segment is refuted all ground rules which contain it will be refuted as well.

In order to implement this idea, the notion of completion structure is extended with a partial function that marks every segment in a ground rule which has been refuted. Formally:

Definition 24. An $\mathcal{A}_1'$*-completion structure for a FoLP* P is a tuple $\langle EF, \text{CT}, \text{ST}, \text{REF}, G\rangle$ where EF, CT, ST, and G are as in Definition 12, and

- $\text{REF} : P \times \{0, 1, \ldots, k\} \times N_{EF} \to \{yes\}$ is a partial refutation function which marks which segments in ground unary rules where the root variable is instantiated with the root node in the completion and the successor variables are instantiated with successor nodes in the completion are already refuted. The first argument is the (non-ground) unary rule from which the ground rule has been derived, the second argument denotes the segment which is refuted, where $k = max_{r \in P_p, p \in upreds(P)}(degree(r))$, while the third argument is the node in the UCS in construction used for grounding that particular segment: for $k = 0$ the node is the root node of the UCS, while for $k \geqslant 1$ the node is one of the successor nodes in the UCS.

An $\mathcal{A}_1'$*-completion structure for a FoLP* P is defined similarly to an $\mathcal{A}_1$*-completion structure for a FoLP* P: the function REF is simply undefined for any input.

The new expansion rule which replaces the (iii) Expand-Unary-Negative rule introduced in Section 3.2 and which works on $\mathcal{A}_1'$-completion structures is defined as follows:

Rule. ***(iii′) Expand-Unary-Negative.*** Let $\mathit{not}\ p \in \textsc{ct}(x)$ be a unary negative predicate for which $\textsc{st}(x, \mathit{not}\ p) = \mathit{unexp}$ and let $y_1, \ldots, y_n$ be the successors of x in EF. If:

- for all $p \in \mathit{upreds}(P)$, $p \in \textsc{ct}(x)$ or $\mathit{not}\ p \in \textsc{ct}(x)$, and
- for all $p \in \textsc{ct}(x)$, $\textsc{st}(p, x) := \mathit{exp}$,

then for every rule $r \in P_p$ of the form (2.3) such that x matches s (s is the term from the head of the rule), if $\textsc{ref}(r, 0, x)$ is undefined do one of the following:

1. refute the rule 'locally':
   - non-deterministically choose $\pm q \in \beta$,
   - $\mathit{update}(\mathit{not}\ p(x), \mp q, x)$, and
   - set $\textsc{ref}(r, 0, x) = \mathit{yes}$,

   or

2. refute all ground versions of the rule in their non-local part:
   for all $y_{i_1}, \ldots, y_{i_k}$ s. t. $(1 \leqslant i_j \leqslant n)_{1 \leqslant j \leqslant k}$, if:
   - for all $1 \leqslant j, l \leqslant k$, $t_j \neq t_l \in \psi \Rightarrow y_{i_j} \neq y_{i_l}$, and
   - $\textsc{ref}(r, l, y_{i_l})$ is undefined for every l with $0 < l \leqslant k$,

   then:
   - choose a segment $1 \leqslant m \leqslant k$,
   - non-deterministically choose $\pm f \in \delta_m/\gamma_m$ and $\mathit{update}(\mathit{not}\ p(x), \mp f, (x, y_{i_m})/y_{i_m})$, and
   - set $\textsc{ref}(r, m, y_{i_m}) = \mathit{yes}$.

Set $\textsc{st}(\mathit{not}\ p, x) := \mathit{exp}$.

Given the new expansion rule, we can define constructively the new notion of UCS:

Definition 25. An *alternative unit completion structure* $\langle EF, \textsc{ct}, \textsc{st}, \textsc{ref}, G\rangle$ with root ε for a FoLP P, with $EF = (F, ES)$, is an $\mathcal{A}_1'$-completion structure derived from an initial $\mathcal{A}_1'$-unit completion structure with root ε for P by application of the expansion rules (i)-(ii) and (iv)-(vi) from Section 3.2 and of expansion rule (iii′) introduced in this section, taking into account the applicability rules (vii)-(xi) from Section 3.3, which has the following properties:

- for all $p \in \mathit{upreds}(P)$, either:
  - $p \in \textsc{ct}(\varepsilon)$ and $\textsc{st}(p, \varepsilon) = \mathit{exp}$, or

- $not\ p \in \text{CT}(\varepsilon)$ and $\text{ST}(not\ p, \varepsilon) = exp$;

- for all v_1, v_2 such that $(v_1, v_2) \in A_{EF}$:
  - $v_1 = \varepsilon$,
  - for all $f \in bpreds(P)$, either:
    * $f \in \text{CT}(v_1, v_2)$ and $\text{ST}(p, (v_1, v_2)) = exp$, or
    * $not\ f \in \text{CT}(v_1, v_2)$ and $\text{ST}(not\ p, (v_1, v_2)) = exp$;
  - for all $\pm p \in \text{CT}(v_2)$, $\text{ST}(\pm p, v_2) = unexp$;

- for all $c \in cts(P)$ such that $\text{CT}(c) \neq \emptyset$: $c = \varepsilon$ or there exists an arc $(\varepsilon, c) \in ES$,

- the structure is neither contradictory, nor circular.

Likewise we did for regular UCSs, we will drop the status and the refutation functions when referring to alternative unit completion structures as these are relevant only in the definition phase of such a structure. As such, we will refer to an alternative unit completion structure as a triple $\langle EF, \text{CT}, G \rangle$.

As anticipated, it is the case that:

Proposition 12. Let P be a FoLP. Then there exists a UCS with root ε: $\langle EF, \text{CT}, G \rangle$ for P iff there exists an alternative UCS with root ε: $\langle EF, \text{CT}, G \rangle$ for P (where ε, EF, CT, and G denote the same entities for both structures).

Thus, instead of constructing the set of UCSs for P one can construct the set of alternative UCSs for P. An analysis of the expansion rule (iii′) introduced in this section reveals that a determinization of this rule will lead in the worst case to deterministic exponential time behavior: this is due to the fact that there exists a polynomial number of segments (s, y) to be refuted with respect to any unary rule. For each such segment there exists a polynomial number of choices for the actual refutation. Thus:

Proposition 13. The set of all alternative unit completion structures for a FoLP P can be computed in the worst-case scenario in exponential time in the size of P.

A corollary of Proposition 12 and of Proposition 13 is that:

Corollary 5. The set of all unit completion structures for a FoLP P can be computed in the worst-case scenario in exponential time in the size of P.

4.3 Redundant Unit Completion Structures

As seen in Example 22, there exist unit completion structures with roots which have equal content, but possibly different topologies, different contents of the successor nodes and/or possibly different dependency graphs. As discussed in the introduction to this chapter, it is worthwhile to identify structures which are strictly more constraining than others, in the sense that while they have identical roots, they impose more constraints on the content of the successor nodes of the structure and introduce more paths in the dependency graph than other structures. As for such structures, there always exists a more general structure which can be used as a building block of a model, they can be discarded. In other words, they are redundant structures. The following definition formalizes this notion of redundant unit completion structure.

Definition 26. Let $\mathcal{UCS}_P$ be the set of all unit completion structures of a FoLP P. A *unit completion structure* for $UC_1 \in \mathcal{UCS}_P$, with root ε_1, where $UC_1 = \langle EF_1, \textsc{ct}_1, G_1 \rangle$, is said to be *redundant* iff there exists another unit completion structure $UC_2 \in \mathcal{UCS}_P$ for P with root roo_2, where $UC_2 = \langle EF_2, \textsc{ct}_2, G_2 \rangle$, such that:

- if $\varepsilon_2 \in cts(P)$, then $\varepsilon_2 = \varepsilon_1$;
- $\textsc{ct}(\varepsilon_1) = \textsc{ct}(\varepsilon_2)$;
- if $\varepsilon_2 \cdot s_1, \ldots, \varepsilon_2 \cdot s_l$ are the non-blocked successors of ε_2, there exist l distinct successors $\varepsilon_1 \cdot t_1, \ldots, \varepsilon_1 \cdot t_l$ of ε_1 such that:
  - $\textsc{ct}(\varepsilon_2 \cdot s_i) \subseteq \textsc{ct}(\varepsilon_1 \cdot t_i)$, for every $1 \leqslant i \leqslant l$, and
  - $connpr_{G_2}(\varepsilon_2, \varepsilon_2 \cdot s_i) \subseteq connpr_{G_1}(\varepsilon_1, \varepsilon_1 \cdot t_i)$, for every $1 \leqslant i \leqslant l$,

  with at least one inclusion being strict.

The intuition is that the content of the successor nodes of a simpler structure can always be expanded in a similar way to the content of the corresponding successor nodes of the more complex structure, while the fact that there are fewer paths between atoms formed with the root node and atoms formed with successor nodes guarantees that no blocking conditions are violated, and even more, blocking might occur earlier than when using the more complex structure.

When comparing two structures, one supposedly redundant and the other supposedly a redundancy witness, we ignore the blocked successor nodes of the redundancy witness. For such blocked nodes, even if there exist counterpart nodes in the 'redundant' structure which are less constrained, by definition of UCSs and blocking, those counterpart nodes will be blocked as well. In practical terms, comparing blocked nodes does not make a difference as such nodes do not have to be further expanded.

Example 24. Consider the three UCS-s introduced in Example 22 and depicted in Figure 4.1. They are all comparable: one can see that UC_1 is strictly more constraining than both UC_2 UC_3, and UC_2 is strictly more constraining than UC_3. Thus, UC_1 and UC_2 are redundant structures and the only UCS which is non-redundant is UC_3.

Note that, as previously discussed, UC_3 is a final, thus complete clash-free completion structure: its only successor node is a blocked node. Thus, the condition regarding successor nodes from Definition 26 is trivially fulfilled.

Intuitively, the redundancy of UC_1 and UC_2 does not come as a surprise, as when one looks at the rules which used to support p in the the two UCSs, there is a certain degree of redundancy in these rules themselves: the body of r_4, the rule used to support p in the case of UC_1 is strictly less general/more constraining than the body of r_4, the rule used to support p in the case of UC_3.

At the same time rule r_3, which is used to support P in UC_2 is redundant in itself, in the sense that it merely prescribes that $p(x)$ is satisfiable if there exists an f-successor of x, where p is again satisfiable and so on.

Proposition 14. Computing the set of non-redundant unit completion structures for a FoLP P can be performed in the worst case in exponential time in the size of P.

Proof. From Proposition 5 it follows that there exist at most an exponential number of unit completion structures, or in other words $|\mathcal{UCS}_P|$ is exponential in the size of P.

Comparing whether any two given UCSs are such that one is redundant and the other is its redundancy witness can be done in non-deterministic polynomial time (by guessing which one is redundant and then guessing the counterpart successor nodes to every non-blocked successor node in the supposedly redundant structure). Thus, to filter out the redundant UCSs, one needs at most $|\mathcal{UCS}_P|(|\mathcal{UCS}_P| - 1)/2$ comparisons, where each can be performed in non-deterministic polynomial time. $\square$

4.4 Reasoning with FoLPs Using Unit Completion Structures

This section describes a new algorithm which uses the set of pre-computed (non-redundant) completion structures. We call this algorithm $\mathcal{A}_2$.

As was the case with the previous algorithm, $\mathcal{A}_2$ starts with an initial $\mathcal{A}_2$-completion structure for checking satisfiability of a unary predicate p with respect to a FoLP P and expands this to a so-called $\mathcal{A}_2$-completion structure.

An *$\mathcal{A}_2$-completion structure* $\langle EF, \text{CT}, \text{ST}, G\rangle$ is defined similarly as an $\mathcal{A}_1$-completion structure, but the *status* function has a different domain, namely the set of nodes of the forest:

$$\text{ST} : N_{EF} \to \{exp, unexp\}.$$

An *initial $\mathcal{A}_2$-completion structure for a unary predicate p and FoLP P* is defined similarly as an initial $\mathcal{A}_1$-completion structure for p and P, the only difference being that every node in the extended forest is marked as unexpanded: $\text{ST}(x) = unexp$, for every $x \in N_{EF}$.

The difference in the definition of an $\mathcal{A}_2$-completion structure compared to its $\mathcal{A}_1$ homonym is that in this scenario nodes and not predicates are expanded; this is done by matching their contents with existing unit completion structures. We make the notion of matching the content of a node with a unit completion structure for a FoLP P explicit by introducing a notion of *local satisfiability*:

Definition 27. A (non-redundant) unit completion structure $UC = \langle EF, \text{CT}, G\rangle$, with $EF = (F, ES)$, *locally satisfies* a (possibly negated) unary predicate $\pm p$ iff $\pm p \in \text{CT}(\varepsilon)$. Similarly, UC locally satisfies a set S of (possibly) negated unary predicates iff $S \subseteq \text{CT}(\varepsilon)$.

Example 25. All three unit completions in Figure 4.1 locally satisfy the set $\{p, not\ q\}$.

It is easy to observe that if a unary predicate p is not locally satisfied by any unit completion structure UC for a FoLP P (or equivalently $not\ p$ is locally satisfied by every unit completion structure), then p is not satisfiable with respect to P.

However, local satisfiability of a unary predicate p in every unit completion structure for a FoLP P does not guarantee 'global' satisfiability of p with respect to P.

Example 26. Consider the program P' containing only rule r_3 from the CoLP P introduced in Example 22. The only completion structure for P' is the UCS UC_2 from Figure 4.1. While UC_2 locally satisfies p, it is clear that p is not satisfiable.

In the process of building for a FoLP P an $\mathcal{A}_2$-completion structure $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$, with $G = (V, A)$, by using unit completion structures as building blocks an operation commonly appears: the expansion of a node $x \in N_{EF}$ by addition of a unit completion structure $UC = \langle EF^{'}, \text{CT}^{'}, G'\rangle$, with $EF^{'} = (F', ES^{'})$ and $G^{'} = (V^{'}, A^{'})$, which locally satisfies $\text{CT}(x)$ at x, given that its root matches with x. Note that an anonymous individual behaves like a variable: it matches with any term, while a constant matches only with itself. Thus, unit completion structures with roots constants can only be used as initial building blocks for the trees with non-anonymous roots in the structure. We denote this operation as: $expand_{CS}(x, UC)$. Formally, its application updates CS as follows:

- $\text{ST}(x) = exp$,
- $N_{EF} = N_{EF} \cup \{x \cdot s \mid \varepsilon \cdot s \in N_{EF'}\}$,
- $A_{EF} = A_{EF} \cup \{(x, x \cdot s) \mid (\varepsilon, \varepsilon \cdot s) \in A_{EF'}\}$,
- $\text{CT}(x) = \text{CT}(\varepsilon)$,
- for all s such that $\varepsilon \cdot s \in N_{EF'}$, $\text{CT}(x \cdot s) = \text{CT}(\varepsilon \cdot s)$,
- $V = V \cup \{p(x) \mid p \in \text{CT}(\varepsilon)\} \cup \{p(x \cdot s) \mid p \in \text{CT}(\varepsilon \cdot s)\}$,
- $A = A \cup \{(p(\overline{z}), q(\overline{y})) \mid (p(z), q(y)) \in A^{'}\}$, where $\overline{\varepsilon} = x$, and $\overline{\varepsilon \cdot s} = x \cdot s$.

The new algorithm $\mathcal{A}_2$ has a new rule compared with the original algorithm $\mathcal{A}_1$ which we call *Match*:

Rule. ***(xii) Match***. For a node $x \in N_{EF}$: if $\text{ST}(x) = unexp$, non-deterministically choose a non-redundant unit completion structure UC with root matching x which satisfies $\text{CT}(x)$ and perform $expand_{CS}(x, UC)$.

The Match rule replaces the expansion rules (i)-(vi) and the applicability rules (vii) Saturation and (x) Contradiction from the original algorithm. In this variant of the algorithm we still employ the applicability rules (viii) Blocking and (ix) Redundancy introduced in Section 3.3. Note that the local clash conditions regarding contradictory structures or structures which have local cycles in the dependency graph G are no longer relevant and as such the applicability

Algorithm 4.1: Overview of $\mathcal{A}_2$.

```
input  : FoLP $P$, unary predicate $p$;
output: checks satisfiability of $p$ with respect to$P$;
1) Construct the set of non-redundant Unit Completion Structures (UCSs) for $P$ (if not
available already);
2) Construct an $\mathcal{A}_2$-initial completion structure $CS$ for $p$ with respect to $P$ as in
Definition 21;
3) $S = N_{EF}$;
repeat
    Pick up a node $x \in S$ such that $\textsc{st}(x) = unexp$;
    $S = S - \{x\}$;
        a) if there is an ancestor $y$ of $x$: $y <_F x$, $y \notin cts(P)$, s. t.:
         $\textsc{ct}(x) \subseteq \textsc{ct}(y)$, and
         $connpr_G(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_G \wedge q$ is not free$\}$ is empty
        then
            $x$ is blocked;
            $\textsc{st}(x) = exp$;
        end
        b) if $\textsc{st}(x) = unexp$ then
            non-deterministically choose a unit completion structure $UC$ which
            matches $x$ and perform $expand_{CS}(x, UC)$;
            i) if $\textsc{st}(x) = unexp$ then
                return false;
            end
            ii) if $G$ contains cycles then
                return false;
            end
            iii) if $\textsc{st}(x) = exp$ and there are $k$ ancestors $y_i$ of $x$ in $F$, $1 \leqslant i \leqslant k$,
            $y \notin cts(P)$, such that: $\textsc{ct}(x) = \textsc{ct}(y)$, for every $1 \leqslant i \leqslant k$ then
                $x$ is redundant: return false;
            end
        end
until $S = \emptyset$;
return true;
```

rule (x) Contradiction is dropped. The same for the applicability rule (vii) Saturation. However, non-local, i.e. constant cycles in G are still relevant, thus rule (xi) Circularity is still employed.

Algorithm 4.1 provides an overview of $\mathcal{A}_2$.

In the following we make precise what is the result returned by $\mathcal{A}_2$, i.e. when an $\mathcal{A}_2$-completion structure is *complete*.

Definition 28. A *complete $\mathcal{A}_2$-completion structure* for a unary predicate p with respect to a FoLP P, is an $\mathcal{A}_2$-completion structure that results from applying the rule (xii) Match to an initial $\mathcal{A}_2$-completion structure for p with respect to P, taking into account the applicability rules (viii)-(ix) and (xi), such that no other rules can be further applied.

The termination of $\mathcal{A}_2$ follows immediately from the usage of the blocking and of the redundancy rule:

Proposition 15. An initial $\mathcal{A}_2$-completion structure for a unary predicate p and a FoLP P will be expanded to a complete $\mathcal{A}_2$-completion structure by a finite number of applications of the rule *Match*, taking into account the applicability rules (viii)-(ix) and (xi).

The following definition singles out the cases where the algorithms stops unsuccessfully, i.e. when a *clash* occurs:

Definition 29. An $\mathcal{A}_2$-completion structure for a unary predicate p with respect to a FoLP P, $CS = \langle EF, \text{CT}, \text{ST}, G\rangle$, is *clash-free* iff:

1. there is no node $x \in N_{EF}$ such that $st(x) = unexp$,
2. CS is not circular, and
3. EF does not contain redundant nodes.

The algorithm $\mathcal{A}_2$ is sound and complete:

Proposition 16. A unary predicate p is satisfiable with respect to a FoLP P iff there is a complete clash-free $\mathcal{A}_2$-completion structure for p with respect to P.

Proof. The soundness of $\mathcal{A}_2$ follows from the soundness of $\mathcal{A}_1$: any completion structure computed using $\mathcal{A}_2$ could have actually been computed using $\mathcal{A}_1$ by replacing every usage of the *Match* rule with the corresponding rule application sequence used by $\mathcal{A}_1$ to derive the unit completion structure which is currently appended to the structure.

The completeness of $\mathcal{A}_2$ follows again from the completeness of $\mathcal{A}_1$: any clash-free complete $\mathcal{A}_1$-completion structure can actually be seen as a complete clash-free $\mathcal{A}_2$-completion structure. It is essential here that the discarded unit completion structures were strictly more constraining than some other (preserved) unit completion structures. Whenever the expansion of a node in the complete clash-free $\mathcal{A}_1$-completion structure has been performed by a sequence of rules captured by a redundant unit completion structure, it is possible to construct a complete clash-free $\mathcal{A}_2$-completion structure by using the simpler non-redundant unit completion structure instead. □

As we still employ the redundancy rule in this version of the algorithm, a complete $\mathcal{A}_2$-completion structure has in the worst case a double exponential number of nodes in the size of the program. Consequently:

Proposition 17. $\mathcal{A}_2$ runs in the worst-case in non-deterministic double exponential time.

4.5 Discussion and Related Work

$\mathcal{A}_2$, the algorithm we described in this chapter, has the same worst-case running time complexity as $\mathcal{A}_1$. The high complexity does not come as a surprise as the scope of the knowledge compilation technique, is saving time by avoiding redundant local computations. The worst-case running complexity of the algorithm depends on the depth of the trees which have to be explored in order to ensure completeness of the algorithm: as neither of the two algorithms, $\mathcal{A}_1$ or $\mathcal{A}_2$, employs anywhere blocking, the size of the completion is exponential in the maximal depth of the trees which compose it.

However, it is reasonable to expect $\mathcal{A}_2$ will perform considerably better than the original algorithm in returning positive answers to satisfiability checking queries, while it might still take considerable time in the cases where a predicate is not satisfiable.

Especially problematic are cases where there exists a unit completion structure which locally satisfies the predicate checked to be satisfiable, but the predicate is actually unsatisfiable. One can deal with such situations by employing heuristics like establishing a limit on the depth of the explored structures: in practice it is highly improbable that if there exists a solution, it can be found only in an open answer set of a considerable size: actually, it is quite hard to come up with examples of such situations.

In the next chapter we will describe an algorithm for reasoning with FoLPs which together with a new redundancy condition, employs also a form of caching across branches, i.e. a node may reuse the computation performed to justify the content of a node on a different branch, which will lead to a decrease in the worst-case running complexity of one exponential level. Note that the form of caching employed there is not anywhere blocking, as the conditions for blocking and caching are different. The algorithm still uses the technique introduced in this chapter. The same chapter will describe also an algorithm for reasoning with a restricted fragment of Forest Logic Programs, called simple Forest Logic Programs, which again will use the knowledge compilation technique.

4.5.1 Related Work

$\mathbb{FDNC}$ *Programs*

A formalism related to FoLPs is $\mathbb{FDNC}$ [Šimkus and Eiter, 2007]. $\mathbb{FDNC}$ is an extension of a fragment of ASP with function symbols, which as FoLPs has the forest model property. $\mathbb{FDNC}$ rules are required to be safe unlike FoLP ones: every variable which occurs in the head of a rule must occur in a positive literal in the body as well. The complexity for standard reasoning tasks for $\mathbb{FDNC}$ is EXPTIME-complete.

The reasoning technique for $\mathbb{FDNC}$ programs introduced in [Šimkus and Eiter, 2007] is similar to the knowledge compilation technique we described in this chapter, in the sense that it uses blocks in the form of trees of depth 1 called *knots* to build models. While knots are structurally similar to unit completion structures, they have different semantic properties.

The content of the root node of a knot serves as a justification for the contents of the successor nodes in the knot; in a unit completion structure, the contents of successor nodes are constraints which have to be fulfilled in order for the content of the root node to hold. As such, in the case of $\mathbb{FDNC}$, any chaining of knots which contains a knot for every constant in the program is well-founded. The main property which makes $\mathbb{FDNC}$ programs amenable to such a bottom-up reasoning technique is safeness.

In an alternative work [Bonatti, 2011], the decidability of consistency checking for $\mathbb{FDNC}$ programs has been reformulated in terms of regular splitting sequences [Baral, 2002, Lifschitz and Turner, 1994] on finitely recursive programs. As mentioned in the related work section of Chapter 3, FoLPs are not finitely recursive, thus their expressiveness lies outside that of $\mathbb{FDNC}$ programs. In particular, the top-down nature of FoLP rules makes them a suitable device for reasoning about the past, while the bottom-up structure of $\mathbb{FDNC}$ programs makes them a suitable device for reasoning about the future.

To overcome this limitation of $\mathbb{FDNC}$ – that new atoms cannot be derived from structurally more complex ones, and thus it is not possible to reason about the past, the fragment has been extended to the case of bidirectional ASP programs with function symbols (BD-programs) [Eiter and Šimkus, 2009]. BD-programs allow one to talk both about the past and about the future. The algorithms for reasoning with BD-programs are automata-based and as such their best-case behaviour coincides with their worst-case behaviour. In terms of complexity, reasoning with BD-programs is quite hard: already brave reasoning in the case where disjunction is disallowed is 2EXPTIME-hard.

Knowledge Compilation Technique for $\mathcal{ALC}$ KBs

A knowledge compilation technique for reasoning with the Description Logic $\mathcal{ALC}$ is described in [Furbach et al., 2009]. The pre-compilation technique consists of two steps. In the first step, all possible sub-concepts of a concept which are conjunctions of simple concepts and role restrictions are computed. These sub-concepts are captured by so-called *paths* which are sets of simple concepts and role restrictions. Paths which contain contradictory concepts are removed (these are called *links*), as well as paths which are super-sets of other paths. Note that this step is similar to our knowledge compilation method as concerns removing local contradictions and redundancy. However our way of removing redundancy is much more sophisticated as we also consider redundancies in the set of dependencies between atoms in the model.

In the second step, role restrictions are considered: all links for ‘potentially reachable’ concepts from the original concept are removed and a so-called *linkless graph* is obtained. The method explores the linkless graph for checking concept consistency and answering subsumption queries. Reachability is defined as the transitive closure of the relation between a concept and each of its role restriction fillers. Unlike there, our knowledge compilation method only constructs structures of depth one – we consider that pre-computing structures with higher depth would be an overkill.

Pre-processing for DL Tableau Algorithms

In the area of tableau algorithms for DL, several pre-processing techniques were employed successfully so far, like *normalization* and *absorption* [Horrocks, 2003]. Our method is closest to normalization, which seeks to eliminate local contradictions and tautologies, and as well to simplify some concepts.

Abstract Tableau Systems and Their Connection to Automata-Based Procedures

Baader et al. [2003b] investigated the relationship between automata and tableau-based inference procedures for description logics. An abstract notion of tableau system is introduced together with algorithms which convert such a system in automata-based algorithms.

Tableau systems are schematic tableau algorithms in which termination is abstracted away. They prescribe how to construct (possibly infinite) models. Such a system manipulates tree-shaped structures called *patterns* which can be transformed into other patterns by means of completion rules. Patterns are also used to capture clash conditions and they may contain beside a labelled tree some additional information called global memory elements. The depth of trees in patterns is bounded by a parameter of the tableau system, called *pattern depth*.

The notion of unit completion structure which we introduced in this chapter is similar to the notion of a pattern of depth 1, where the dependency graph associated to each such structure is the global memory stored in the pattern. However, in our case clashes cannot be captured by patterns of depth 1. One type of clash is the presence of redundant nodes which can be placed at arbitrary distances from each other along a branch in a completion structure. If we want to capture clashes by means of tree structures, we have to consider unary trees (paths) of exponential depth.

Baader et al. [2003b] also defined a notion of pattern inclusion: a pattern is a sub-pattern of another pattern iff it can be homomorphically embedded in the latter and its global memory elements are a subset of the global memory elements of the latter. Maintaining the comparison with our notion of unit completion structures, a UCS is redundant iff the associated pattern is a super-pattern of a pattern corresponding to another UCS.

In an effort to identify tableau systems for which the underlying reasoning task can be reduced to checking emptiness of looping tree automata, the class of EXPTIME admissible tableau systems is defined. A tableau system is EXPTIME- admissible iff certain assumptions hold regarding the time it takes to compute various parts of a tableau system, like labels of nodes and arcs, global memory elements, whether a pattern is equivalent to another pattern or whether a pattern constitutes a clash. For tableau systems which fall into this category, the translation to looping tree automata provides automatically an EXPTIME upper bound for reasoning within the DL for which the tableau system was designed. Note that an underlying assumption of the method is that patterns have finite bounded depth. While our tableau algorithm would fulfil all the conditions for EXPTIME-admissibility, it does not fulfil this underlying assumption – we need patterns of arbitrary depth to capture clashes. Thus, the automata method does not work for our algorithm.

Finally, some sufficient conditions are provided for tableau systems to be translatable into actual tableau algorithms which use blocking as a termination mechanism: in this case the tableau elements mentioned above have to be only effectively computable. The blocking mechanism is based on pattern-matching: two nodes on the same branch are in a blocking relationship iff the pattern having as root the descendant node is a sub-pattern of the pattern having as root the ancestor node. Again, one underlying assumption is the possibility to capture clashes by means of patterns – thus, unsurprisingly this particular blocking mechanism is not applicable to our algorithm. The abstract framework does however provide an insight into our blocking condition: the check on the dependency graph G can be seen as a check to exclude clashes which occur in arbitrarily sized patterns.

CHAPTER 5

Optimized Tableau Algorithm for Reasoning with Forest Logic Programs

This chapter describes $\mathcal{A}_3$, an optimized tableau algorithm for reasoning with FoLPs, which runs in the worst-case in non-deterministic exponential time, one exponential level lower than its predecessors $\mathcal{A}_1$ and $\mathcal{A}_2$. Like in the case of $\mathcal{A}_2$, a completion structure is constructed by matching and appending UCSs, but a different strategy is employed for termination.

In particular, the algorithm redefines the notion of redundant nodes: in $\mathcal{A}_3$ redundant nodes are typically identified much earlier than their counterparts in $\mathcal{A}_1$. There is no exponential threshold regarding the depth at which such nodes can occur on a path. The new algorithm identifies as well when some computation on a branch can be reused during the expansion of another branch: if a node which is currently selected for expansion is similar to a non-ancestor node which has been already expanded, the justification of the latter is reused when dealing with the original node. The new rule which deals with this is called *caching*. $\mathcal{A}_3$ improves also on the condition used for blocking by $\mathcal{A}_1$ and $\mathcal{A}_2$, by requiring less bookkeeping than the previous blocking rule.

In keeping in line with the previous chapters, we call the structure which is evolved by the algorithm an $\mathcal{A}_3$-completion structure. The data structures which are part of such an $\mathcal{A}_3$-completion structure are the same as the ones employed by an $\mathcal{A}_2$-completion structure, and as such we will not provide formal definitions for the notions of (initial) $\mathcal{A}_3$-completion structure. The difference between $\mathcal{A}_3$-completion structures and $\mathcal{A}_2$-completion structures lies in their construction: they are evolved using different applicability rules.

Section 5.1 describes the new blocking condition together with some notation which will be needed for the subsequent rules. The new notion of redundant nodes is introduced in Section 5.2, while the new caching rule is described in Section 5.3.

Section 5.4 describes when an $\mathcal{A}_3$-completion structure is complete and clash-free, and it provides an overview of the algorithm. Section 5.5 shows that $\mathcal{A}_3$ terminates by computing an upper bound on the size of an $\mathcal{A}_3$-completion structure: any such structure has a number of nodes which is at most exponential in the size of the input program. This leads to a worst-case running time behaviour of the algorithm which is non-deterministic exponential in the size of the input program.

The usage of the caching rule has improved the worst case running time of the algorithm by one exponential. The new applicability rules are at a first glance not that different from previous applicability rules. However they rely on different proof strategies, especially on a different strategy to reduce a (potentially infinite) model to one of a finite bounded size. As was the case with $\mathcal{A}_1$ and $\mathcal{A}_2$, such a reduction is part of the completeness proof. The soundness proof and the completeness proof of $\mathcal{A}_3$ are provided in Sections 5.6 and Section 5.7, respectively.

A cursory look at the applicability rules shows that they require extensive bookkeeping and checking complex conditions regarding dependencies of atoms in the atom dependency graph. Section 5.8 introduces a restricted fragment of FoLPs, called simple FoLPs, which generalizes the fragment of acyclic FoLPs, for which the blocking and caching conditions collapse into a simple subset anywhere blocking condition.

Finally, in Section 5.9 we discuss the results obtained in this chapter and relate them to existing work.

5.1 New Blocking Rule

The intuition for blocking is as in the case of $\mathcal{A}_1$ and $\mathcal{A}_2$, to identify pairs of nodes on the same branch of a completion structure which have similar content, and between which there are no dependencies according to the dependency graph G. We observe that when checking for such dependencies between atoms formed with nodes which are in a potential blocking relationship, it is enough to check just a certain part of the dependency graph G, namely its projection over the current tree T; that is, a graph whose nodes have arguments only from T.

Formally, for a given $\mathcal{A}_3$-completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ with $EF = (F, ES)$ and $G = (V, A)$, for every tree $T \in F$, we define G_T to be the graph (V_T, A_T), where:

- $V_T = \{v \in V \mid arg_1(v) \in T\}$, and
- $A_T = A \cap (V_T \times V_T)$.

The new definition for blocked/blocking nodes becomes:

Definition 30. A node $x \in N_{EF}$ is *blocked* iff there is an ancestor y of x in some tree T in F, $y <_T x$, $y \notin cts(P)$, such that:

- $\textsc{ct}(x) \subseteq \textsc{ct}(y)$, and
- the set $connpr_{G_T}(y, x) = \{(p, q) \mid (p(y), q(x)) \in paths_{G_T} \wedge q \text{ is not free}\}$ is empty.

We call (y, x) a *blocking pair* and y is said to be a *blocking node*.

The blocking rule remains unchanged: a blocked node is marked as being expanded and no expansions can be performed on such a node.

Intuitively, the new weaker blocking condition is sufficient as every path in G from some $p(y)$ to some $q(x)$ has to contain a path in G_T from some $r(y)$ to $q(x)$: for example, the path

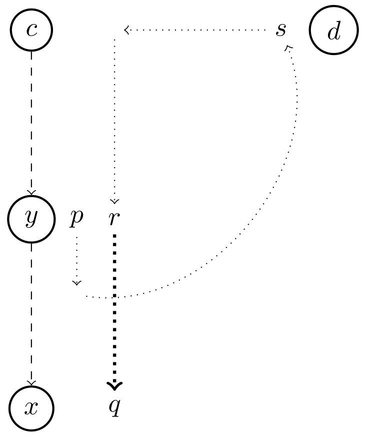


Figure 5.1: Any path from $p(x)$ to $q(y)$ in G contains a path from $r(x)$ to $q(y)$ in G_T

from $p(y)$ to $q(x)$ in Figure 5.1, via $s(d)$ (the dashed path), contains the path from $r(y)$ to $q(x)$ (the dashed bold path), which is a local path, a path with nodes only from T_c.

5.2 Revisiting Redundancy

As discussed in Section 3.3.3, due to its complexity, the blocking condition might never be fulfilled while exploring a finite number of nodes on any given branch. $\mathcal{A}_1$ and $\mathcal{A}_2$ use as an extra condition to ensure termination a redundancy rule that consist in aborting the expansion of a branch once a certain number of nodes with equal content has been encountered on the branch. In this section we introduce a more refined strategy for aborting the expansion of a branch which requires some extra bookkeeping regarding the dependencies between atoms in the partially constructed model.

In particular, for each unary atom we keep track of the set of oldest paths in the dependency graph G in which the atom occurs, where by 'oldest' path we understand a path containing a unary atom whose argument has the smallest depth in the tree/interconnected forest among all the unary atoms occurring in such paths. By intersection of a set of paths with a node x, we understand the maximal set of unary predicate symbols S such that for every unary predicate $p \in S$, it is the case that $p \in \textsc{ct}(x)$ and $p(x)$ occurs in one of the paths in the set.

The general idea is that, at any point during the construction of a forest model, the intersection of the set of oldest paths running along a branch of the forest with subsequent nodes on the branch with equal content should be minimized. Whenever two nodes x and y, $x < y$, on the same branch of an $\mathcal{A}_3$-completion structure, have equal content, and the intersection of the set of oldest paths with x (the node 'above') is included in the intersection of the set of oldest paths with y (the node 'below'), y said to be redundant and the computation is aborted, i.e. the presence of such a node constitutes a clash. Nodes with identical content are allowed on the same branch, only if every subsequent occurrence of such a node shrinks the set of oldest paths. While before failure was detected only when reaching a node of exponential depth, the new strategy potentially identifies failure much earlier.

In order to implement the new strategy we introduce some notations: by *rank of a unary atom* $a = p(x)$ whose argument x belongs to a tree T we understand the minimum between the depth of x in T and the smallest depth of a node $y \in T$ such that there exists a unary predicate q with $(q(y), p(x)) \in conn_{G_T}$, where G_T is as defined in Section 5.1. Formally:

$$rank(p(x)) = min(\{||x||\} \cup \{rank(a) \mid (a, p(x)) \in conn_{G_T}\})$$

The notation is extended to nodes: *the rank of a node* x is the minimum among the ranks of the unary atoms formed with predicates in the label of the node and the node itself:

$$rank(x) = \min_{p \in \text{CT}(x)} rank(p(x))$$

We also denote with $isp(k, x)$ the set of intersections of paths which start at level k with the content of a node x (presuming $||x|| \geqslant k$):

$$isp(k, x) = \{p \mid rank(p, x) = k, p \in \text{CT}(x)\}.$$

Example 27. Let P be the following FoLP:

$$\begin{array}{lrcl}
r_1: & smember(X) & \leftarrow & supportedBy(X, Y), smember(Y) \\
r_2: & smember(X) & \leftarrow & supportedBy(X, Y), rmember(Y), \\
 & & & supportedBy(X, Z), rmember(Z), \\
 & & & Y \neq Z \\
r_3: & supportedBy(X, Y) \vee not\ supportedBy(X, Y) & \leftarrow & \\
r_4: & & \leftarrow & smember(X), rmember(X) \\
r_5: & rmember(X) & \leftarrow & involvedIn(X, Y), project(Y) \\
r_6: & involvedIn(X, Y) \vee not\ involvedIn(X, Y) & \leftarrow & \\
r_7: & project(j) & \leftarrow &
\end{array}$$

According to P, an individual is a special member of an organization (*smember*) if he has the support of another special member (*rule* r_1), or if he has the support of two regular members of the organization (*rmember*) (*rule* r_2). The binary predicate *supportedBy* which describes the 'has support' relationship is free (*rule* r_3). No individual can be at the same time both a special member and a regular member (*constraint* r_4). Somebody is a regular member if he is involved in some project (*rule* r_5). The binary predicate *involvedIn* which describes the 'involved in a project' relationship is free (*rule* r_6). Finally, there is a project j (*fact* r_7).

Figure 5.2 depicts a complete clash-free $\mathcal{A}_2$-completion structure for $smember$ with respect to P from which a forest model can been derived. For clarity, negative predicate symbols which occur in the content of nodes/arcs are not explicitly listed in the figure - however, it is assumed that every unary/binary predicate symbol which does not appear explicitly in its positive form in the content of some node/arc, is part of the respective content in its negated form. The solid arcs between positive predicate symbols represent the arcs in the dependency graph associated with the structure.

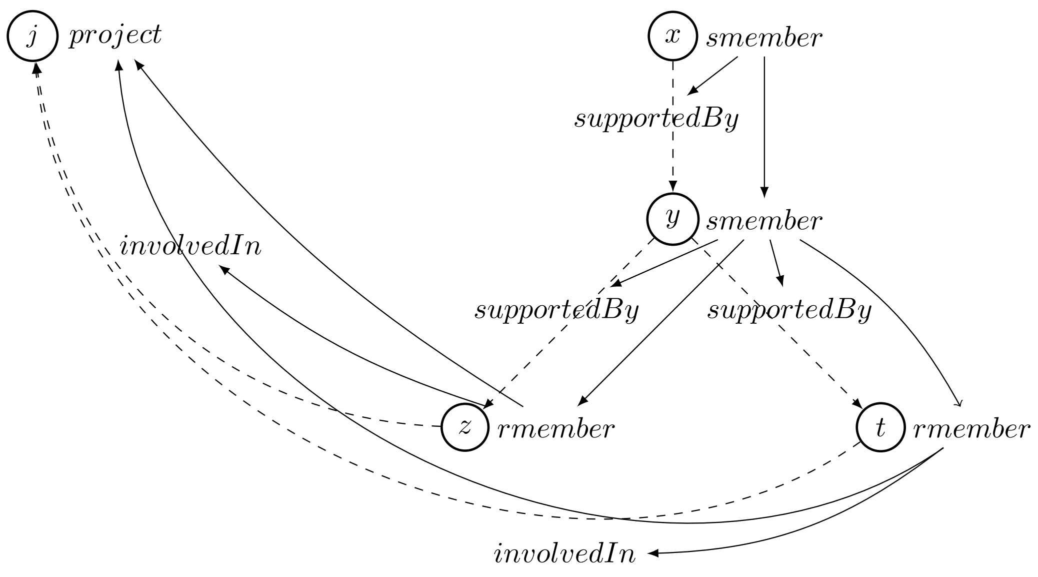


Figure 5.2: Complete clash-free $\mathcal{A}_2$-completion structure with explicit dependency arcs

From Figure 5.2, it can be seen that every unary atom in the model induced by the structure can be reached by a path starting with $smember(x)$ or $project(j)$. As x and j are roots of trees in the forest model, $smember(x)$ and $project(j)$ have rank 1. Thus, all unary atoms in the model have rank 1. It is also the case that $isp(1, x) = isp(1, y) = \{smember\}$ and $isp(2, t) = \emptyset$.

Given the new notations, the notion of redundant node becomes:

Definition 31. A node $x \in N_{EF}$ is *redundant* iff $\text{ST}(x) = exp$ and there is an ancestor y of x in F, $y <_F x$, $y \notin cts(P)$, such that:

- $\text{CT}(x) = \text{CT}(y)$,
- $rank(x) = rank(y) = r$, and
- $isp(r, x) \supseteq isp(r, y)$.

We call (y, x) a *redundancy pair* and say that y is the *redundancy witness* for x.

The redundancy rule remains unchanged: the presence of a redundant node constitutes a clash and as such the expansion process is stopped when such a node is identified.

Example 28. Consider the $\mathcal{A}_2$-completion structure depicted in Figure 5.2. The structure is not an $\mathcal{A}_3$-completion structure as y is a redundant node. The redundancy witness is node x:

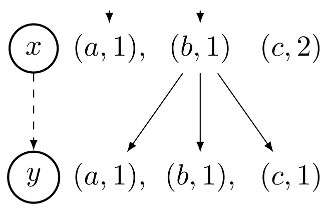


Figure 5.3: Redundancy: y is redundant $\text{CT}(x) = \text{CT}(y)$ and $isp(x,1) \subseteq isp(y,1)$

- $\text{CT}(x) = \text{CT}(y) = \{smember, not\ rmember, not\ project\}$,
- $rank(smember(x)) = rank(smember(y)) = 1$, and
- $isp(1,x) = isp(1,y) = \{smember\}$.

Example 29. Consider also a generic example: Figure 5.3 shows an extract from an $\mathcal{A}_3$-completion structure in which every unary predicate p in the content of a node x is augmented with the rank of $p(x)$. The arcs between predicates in the content of some nodes are arcs in the dependency graph: thus, G contains arcs from $b(x)$ to $a(y)$, $b(y)$, and $c(y)$, respectively. As $rank(b(x)) = 1$, we also have that: $rank(a(y)) = rank(b(y)) = rank(c(y)) = 1$. As a consequence: $isp(1,x) = \{a,b\}$, $isp(2,x) = \{c\}$, and $isp(1,y) = \{a,b,c\}$.

Clearly $\text{CT}(y) = \text{CT}(x) = \{a,b,c\}$ and $rank(x) = rank(y) = 1$. Furthermore, the set of oldest paths in G which traverse both x and y is expanding from x to y, i.e. $isp(1,y) \supseteq isp(1,x)$. Thus, y is redundant.

Intuition: Both strategies for identifying redundant nodes, i.e. the one used in the case of $\mathcal{A}_1$/$\mathcal{A}_2$ and the one used by $\mathcal{A}_3$, are related to techniques for reducing an infinite forest model to a finite one that are used in the completeness proofs of the algorithms. The general principle behind such a reduction is to consider nodes in the infinite model which are on the same infinite length branch and have equal content, and to collapse the two nodes by deleting the path between them (together with all the paths which start at nodes on this path). However, nodes with equal content cannot be indiscriminately collapsed: some extra conditions have to be met in order for the remaining structure to be a minimal model.

In the case of $\mathcal{A}_1$/$\mathcal{A}_2$, the technique used for reducing a model was to first identify for each infinite branch blocking pairs (nodes with equal content with no dependencies via paths in the dependency graph) and then collapse nodes with equal content found on the same branch, if the set of dependency paths in between a 'reference' node and the first node is included into the set of dependency paths between the reference node and the second node. Some extra conditions had to be met for collapsing the two nodes, like that there was no blocking node between them. By using this technique for reducing models, it is possible to obtain a bound on the branch size in forest models. However, the specific conditions necessary for safely collapsing nodes can only be checked at 'proof time', but not at 'construction time', as the set of blocking pairs has to be known a priori. For this reason at construction time it was possible to only use the bound established by this technique, but not the technique itself.

The new technique for reducing models does not use any fixed reference point when comparing nodes with equal content. Furthermore, except for checking subset inclusion of the intersections with the set of oldest paths which traverse the two nodes, no extra condition has to be met before collapsing the nodes. The information about $rank$ and isp is known at construction time, thus the technique can be replicated during the algorithm, too. Unlike the previous technique for establishing the finite bounded model property, the current one, while scanning a branch of a model, does not look for a blocking pair from the outset: it identifies and collapses redundant nodes, until eventually the branch ends a blocking pair is found. Intuitively, this is possible as for infinite branches, by always chasing the set of oldest paths in the dependency graph running along the branch (and exhausting them in a finite number of steps), we reach a point where there are no running paths between two nodes of the branch (within finite distance of each other). Due to fact that the branch is infinite we eventually reach two nodes with equal content having this property which are then naturally in a blocking relation. More details are provided in the completeness proof in Section 5.7.

5.3 Caching

Blocking can be generalized to a form of anywhere blocking which we call here *caching*, where a node reuses the justification/expansion of another node which is not on the same branch. Again, the typical condition regarding subset inclusion of the contents of the nodes has to be fulfilled. In addition, a condition regarding the intersections of different sets of paths in the dependency graph which start at some common ancestor of the two nodes with the nodes themselves has to be fulfilled. Formally:

Definition 32. A node $y \in T_c$ is said to be *cached* iff there is a node $x \in T_c$, $x \notin cts(P)$ such that:

- $\textsc{ct}(y) \subseteq \textsc{ct}(x)$,
- $isp(r, y) \subseteq isp(r, x)$, for every $1 \leqslant r \leqslant ||z||$, where z is the lowest common ancestor of x and y: $z = lca_{T_c}(x, y)$;
- if $rank(y) = 1$, then: $connpr_{G_{T_c}}(c, y) \subseteq connpr_{G_{T_c}}(c, x)$, and
- $right_{T_c}(y, x)$.

We call (x, y) a *caching pair* and x a *caching node.*

A cached node is no longer expanded (it is expanded ‘by default’ due to its status):

Rule. ***(xiii) Caching.*** Let x be a cached node. Then, set: $\textsc{st}(x) = exp$.

Intuition. As was the case with blocking, a cached node will be expanded in a similar way to its corresponding caching node, or in other words reuses the justification of the caching node.

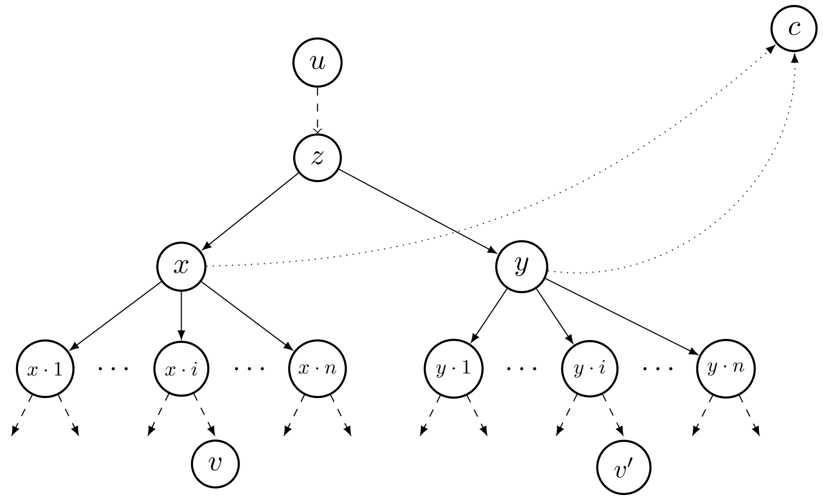


Figure 5.4: Justifying a cached node y by replicating the justification of its corresponding caching node x

One prerequisite for this is that the content of the cached node is a subset of the content of the caching node.

Like in the case of blocking, the content of a cached node y in a caching pair (x, y) can be justified in two different ways: either by copying the subtree T_x at y or by reusing the successors of x as successors of y. In the first case (depicted in Figure 5.4), it has to hold that (1) if (u, v) is a blocking pair, with v being a leaf node in T_x, and $v \geqslant_T z$, where $z = lca_T(x, y)$, then (u, v') is still a blocking pair, where v' is the copy of v in the new subtree T_y. In the second case, the obtained model is no longer forest-shaped and one has to check that (2) no cycles are introduced in G: this is the approach taken in the Soundness proof and it is described in Section 5.6. The second condition in Definition 32 ensures that (1) and (2) hold.

Furthermore, in case x and y have rank 1, i.e. there are paths in G_{T_c} from unary atoms having as argument the root c of the tree T_c to which x and y belong, to atoms having as arguments the two nodes in question, the condition regarding the intersection of the sets of these paths with the two nodes has to be stronger: it is no longer enough that $isp(1, y) \subseteq isp(1, x)$, but for every path from some $p(c)$ to some $q(y)$ in G_{T_c} there must be a path from $p(c)$ to $q(x)$ in G_{T_c}. This is needed in order to prevent the formation of cycles in G when connecting nodes from different trees in EF, including T_c, which contain cached nodes.

Example 30. Figure 5.5 depicts such a situation: x and y are nodes in a tree T_c and there is a path from $p(c)$ to $r(x)$, as well as from $q(c)$ to $r(y)$. Thus, $isp(1, y) = isp(1, x)$. However, x and y do not form a caching pair: by reusing the justification of the content of x when expanding the content of y a cycle would be created in the dependency graph. This is due to the existence of a path from $p(c)$ to $q(c)$ via an atom having as argument an external node d: this would translate into a path from $q(c)$ to $q(c)$ (a cycle) when replicating the justification of y for x. The extra

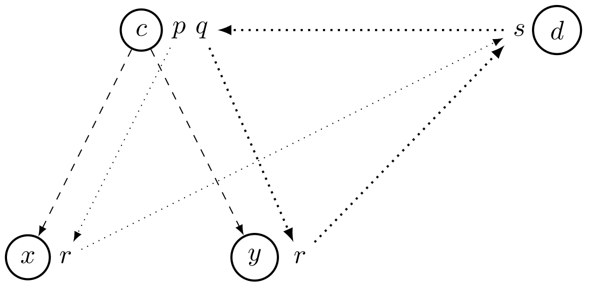


Figure 5.5: While $isp(1, y) \subseteq isp(1, x)$, if x and y form a blocking pair and y reuses the justification of x, a cycle is formed in G

caching condition takes care of this possibility: x and y can be cached iff $connpr(c, y)_{G_{T_c}} \subseteq connpr_{G_{T_c}}(c, x)$, in this case iff $\{(q, r)\} \subseteq \{(p, r)\}$, which is not the case.

Finally, the last condition in Definition 32 is imposed in order not to have nodes which mutually reuse each other's justification, where each node caches some ancestor of the other node. In conjunction with this condition, in order to ensure an exhaustive application of the caching rule (a node can be cached only when it is unexpanded), we also impose the following strategy for the expansion of a completion structure: "*a node $x \in T \in F$ can be expanded iff every node y such that $right_T(y, x)$ holds is expanded*". In other words, the completion should be expanded in a depth-first, left-to-right fashion. The *Match* rule is modified to reflect this new expansion strategy:

Rule. ***(xii′) Match***. For a node $x \in N_{EF}$: if $\text{ST}(x) = unexp$ and for every node y such that $right_T(y, x)$ holds, $\text{ST}(y) = exp$, non-deterministically choose a unit completion structure UC which matches x and perform $expand_{CS}(x, UC)$.

Example 31. Figure 5.6 depicts an $\mathcal{A}_3$-completion structure for $smember$ with respect to the FoLP P introduced in Example 27 in which every node except t is expanded: note that the completion structure contains no redundancy pair.

We have that:

- $y = lca_T(z, t)$,
- $\text{CT}(t) \subset \text{CT}(z)$, and
- $connpr_{G_{T_y}}(y, z) = connpr_{G_{T_y}}(y, t) = \{(smember, rmember)\}$.

Thus, z and t form a caching pair: t will be expanded similarly to z either by reusing the successors of z or by replicating the expansion of z. In this case, no matter which type of justification is used for the content of the caching node, the same result is obtained: the only successor of z is the constant j; thus, even if we choose to replicate the expansion of z, no new successor is introduced, but j is reused instead.

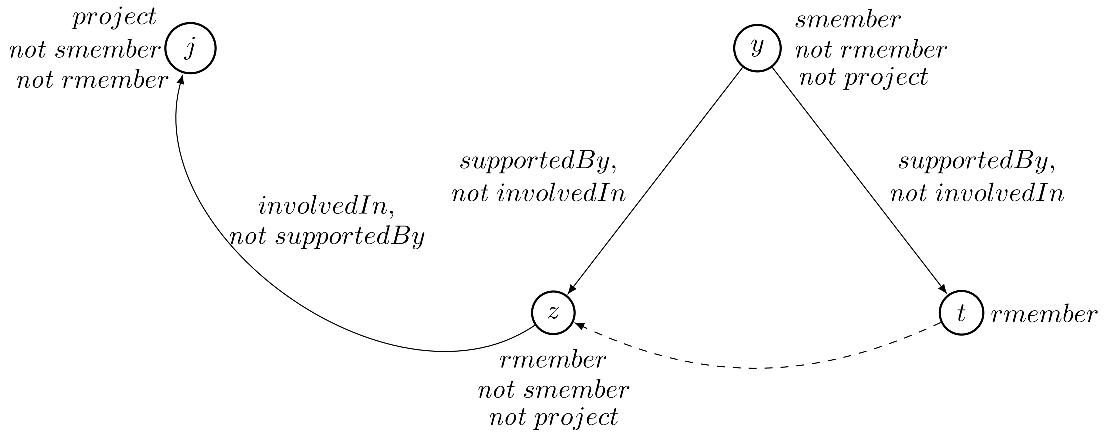


Figure 5.6: An $\mathcal{A}_3$-completion structure in which (z, t) is a caching pair

5.4 Overview of $\mathcal{A}_3$

We start as usually with the definitions of complete $\mathcal{A}_3$-completion structure and clash-free $\mathcal{A}_3$-completion structure:

Definition 33. An $\mathcal{A}_3$-completion structure for a unary predicate p with respect to a FoLP P, is *complete* if it results from the repeated application of rule (xii′) Match to an initial $\mathcal{A}_3$-completion structure for p with respect to P, taking into account the applicability rules (viii) *Blocking*, (ix) *Redundancy*, (xi) *Circularity*, and (xiii) *Caching* such that no rules can be further applied.

Note that in the definition above it is assumed that the blocking rule and the redundancy rule used the new definitions for blocked nodes and redundant nodes, respectively, that were introduced in this chapter.

As regards clash conditions, similarly to the previous version of the algorithm, the presence of redundant nodes and of cycles in G constitutes a clash. Also, every terminal node should be expanded (possibly by virtue of being a blocked or a cached node).

Definition 34. An $\mathcal{A}_3$-completion structure $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ is *clash-free* iff:

- it is not circular,
- it contains no redundant node, and
- for every $x \in N_{EF}$: $\textsc{st}(x) = exp$.

An overview of the algorithm $\mathcal{A}_3$ for checking satisfiability of p with respect to a FoLP P is provided by Algorithm 5.1.

The algorithm receives as input besides p, the predicate checked to be satisfiable, and P, the FoLP under consideration, also the set $\mathcal{UCS}(P)$ of pre-computed non-redundant unit completion

Algorithm 5.1: Overview of $\mathcal{A}_3$, an optimized non-deterministic algorithm to check satisfiability of unary predicates with respect to FoLPs.

```
input  : a FoLP $P$, a unary predicate $p$, the set $\mathcal{UCS}(P)$ of non-redundant UCSs for $P$;
output: yes, if $p$ is satisfiable with respect to $P$; false, otherwise;
1) Let $CS = \langle EF, \textsc{ct}, \textsc{st}, G\rangle$ be an initial $\mathcal{A}_3$-completion structure for $p$ w.r.t. $P$;
2) $S := N_{EF}$;
repeat
    Pick up a node $x \in S$ such that $\textsc{st}(x) = unexp$;
    $S := S - \{x\}$;
    Apply one or several of the followings rules (in decreasing order of priority):
        a) if there is an ancestor $y$ of $x$: $y <_F x$, $y \notin cts(P)$, such that
        $\textsc{ct}(x) \subseteq \textsc{ct}(y)$ and $connpr_{G_{T_c}}(y, x) = \emptyset$, where $x \in T_c$ then
            $x$ is blocked;
            $st(x) := exp$;
        end
        b) if there is a node $y \in T_c - cts(P)$ such that $\textsc{st}(y) = unexp$,
        $right_{T_c}(x, y)$, $\textsc{ct}(x) \subseteq \textsc{ct}(y)$, and for all $1 \leqslant r \leqslant ||z||$:
        $isp(x, r) \subseteq isp(y, r)$, where $z = lca_{T_c}(x, y)$, and if $rank(y) = 1$:
        $connpr_{G_{T_c}}(c, y) \subseteq connpr_{G_{T_c}}(c, x)$ then
            $x$ is cached;
            $\textsc{st}(x) = exp$;
        end
        c) if $\textsc{st}(x) = unexp$ and for every $y \in T_c$ such that $right_{T_c}(y, x)$:
        $\textsc{st}(y) = exp$ then
            non-deterministically choose a unit completion structure $UC$ which
            matches $x$ and perform $expand_{CS}(x, UC)$;
            i) if $\textsc{st}(x) = unexp$ then
                return false;
            end
            ii) if $G$ contains cycles then
                return false;
            end
            iii) if $\textsc{st}(x) = exp$ and there is a node $y <_F x$, $y \notin cts(P)$, s. t.:
             $\textsc{ct}(x) \subseteq \textsc{ct}(y)$,
             $rank(x) = rank(y) = r$, and
             $isp(r, x) \supseteq isp(r, y)$
            then
                $x$ is redundant: return false;
            end
        end
until $S = \emptyset$;
return true;
```

structures for P. Then, one by one, the nodes of the completion in construction are selected for expansion. As the blocking and caching conditions can be checked before a node is actually expanded and in case they are fulfilled the respective node should not be expanded, the blocking rule (step a)) and caching rule (step b)) have higher priority than the matching rule (step c)), which takes care of the expansion.

If a node is still unexpanded even after applying the matching rule (step c) i)), there is a clash, and thus the procedure returns without success (as the algorithm is non-deterministic it does not mean that the predicate checked to be satisfiable is actually not satisfiable, but rather that the current attempt to witness its satisfiability failed).

The subsequent steps of the algorithm check the remaining clash conditions: at step c) ii) it is checked whether the graph G is still acyclic (no cycles have been introduced during the expansion of the current node). Then, step c) iii) checks whether the newly expanded node is a redundant node using the conditions from the new redundancy rule.

Finally, if every node has been considered and it is either expanded, blocked, or cached, the algorithm is successful, i.e. the predicate checked to be satisfiable is indeed satisfiable.

5.5 Termination and Complexity

In this section we show that $\mathcal{A}_3$ terminates and that in the worst case it runs in non-deterministic exponential time: first, a bound is computed on the path length in any $\mathcal{A}_3$-completion structure, and then, using this result, on the total number of nodes in any $\mathcal{A}_3$-completion structure. Both bounds are exponential in the size of the input FoLP P. A necessary condition to obtain the latter result is the usage of the caching rule.

Proposition 18. Every branch in an $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.

Proof. We show that any branch has at most

$$(n2^n - 1)(2^n - 1) + n2^n = n2^{2n} - 2^n + 1 \text{ nodes},$$

where $n = |upreds(P)|$. Assume the opposite. Then, there exists a branch in a tree T with at least:

$$(n2^n - 1)(2^n - 1) + n2^n + 1 \text{ nodes}.$$

There is a finite number of nodes with different contents, viz. 2^n many, on any branch in the completion structure and in the completion structure itself. As such, there must be a set $S \in 2^{upreds(P)}$, and a sequence of non-terminal nodes $(x_i)_{1 \leqslant i \leqslant n2^n}$ belonging to the branch such that: $\text{CT}(x_i) = S$, for every $1 \leqslant i \leqslant n2^n$.

We use the following lemma:

Lemma 11. Let $x_1, \ldots, x_{n2^n}$ be a sequence of nodes as defined above. Then x_{n2^n} is either a redundant or a blocked node.

Proof. Let $r_1, \ldots, r_n$ be the ordered sequence of ranks of unary predicates in $\text{CT}(x_1)$ such that:

- $\bigcup_{1 \leqslant j \leqslant n} \{r_j\} = \{k \mid p \in \text{CT}(x_1) \wedge rank(p(x_1)) = k\}$;
- $r_j \geqslant r_{j+1}$, for every $1 \leqslant j < n$;
- if $l = |\{k \mid p \in \text{CT}(x_1) \wedge rank(p(x_1)) = k\}| < n$, then $r_i = max\{k \mid p \in \text{CT}(x_1) \wedge rank(p(x_1)) = k\}$, for every $i > l$.

We show by induction that in case x_{n2^n} is not redundant, for every $1 \leqslant j \leqslant n$: $rank(x_{j2^n}) > r_j$. Intuitively, this captures the fact that there is no path in G_T from an atom with rank less or equal to r_j to an atom having as argument x_{j2^n}.

Base case: $j = 1$. If (x_1, x_{2^n}) is a blocking pair the claim is obvious. We now prove that $rank(x_{2^n}) > r_1$ in the case where (x_1, x_{2^n}) is not a blocking pair. We have that $rank(x_i) \geqslant rank(x_1) = r_1$, for $1 \leqslant i \leqslant 2^n$, and (x_i, x_k) is neither a blocking nor a redundancy pair, for any $1 \leqslant i < k \leqslant 2^n$. Assume that $rank(x_{2^n}) = r_1$. Then $rank(x_i) = r_1$, for $1 \leqslant i \leqslant 2^n$, and $isp(x_i, r_1) \not\subseteq isp(x_k, r_1)$, for any $1 \leqslant i < k \leqslant 2^n$ (otherwise (x_k, x_i) would be a redundancy pair). But $|\{S \mid S = isp(x, r),$ for some $x \in N_{EF}$ and $r \in \mathbb{N}\}| = 2^n$, which contradicts the previous statement. Thus, the original assumption was false and $rank(x_{2^n}) > r_1$.

Induction case: if $rank(x_{j2^n}) > r_j$, for a certain $1 \leqslant j < n$, we show that $rank(x_{(j+1)2^n}) > r_{j+1}$. Again, we assume the opposite. Then, $rank(x_{j2^n+k}) = r_{j+1}$, for every $1 \leqslant k \leqslant 2^n$. Using a similar argument to the one from the base case, we obtain a contradiction.

Thus, $rank(x_{j2^n}) > r_j$, for every $1 \leqslant j \leqslant n$, and in particular, $rank(x_{n2^n}) > r_n$. As $rank(p, x_1) \leqslant r_n$, for every $p \in \text{CT}(x_1)$, it follows that $rank(p, x_1) < rank(x_{n2^n})$, for every $p \in \text{CT}(x_1)$. This translates into the fact that the set of oldest paths in G_T traversing x_{n2^n} started at a node below x_1, and thus there are no paths in G_T running between x_1 and x_{n2^n}. As $\text{CT}(x_1) = \text{CT}(x_{n2^n})$, this implies that (x_1, x_{n2^n}) is a blocking pair and thus x_{n2^n} is a blocked node. This proves the lemma. $\Box$

Every redundant or blocked node is a leaf node, thus Lemma 11 implies that x_{n2^n} has to be a leaf node as well. But this is a contradiction as every node x_i, for $1 \leqslant i \leqslant n2^n$, is non-terminal. Thus, the original assumption was false: a branch has at most $n2^{2n} - 2^n + 1$ nodes, where $n = |\mathit{upreds}(P)|$. $\Box$

Furthermore, it is possible to show that a complete $\mathcal{A}_3$-completion structure has at most an exponential number of nodes. Note that a complete $\mathcal{A}_2$-completion structure had in the worst case a double exponential number of nodes in the size of the program. The exponential drop in the bound on the number of nodes is due to the interplay between the caching and the redundancy rule:

Proposition 19. A complete $\mathcal{A}_3$-completion structure for a unary predicate p and a FoLP P has at most an exponential number of nodes in the size of P.

Proof.

We associate with every node in a complete $\mathcal{A}_3$-completion structure $x \in T_c$, $T_c \in F$, a function:

$$f_x : \mathit{upreds}(P) \to 2^{\mathit{upreds}(P)} \times \{r \mid 0 \leqslant r \leqslant n2^{2n} - 2^n + 1\},$$

where $n = |\mathit{upreds}(P)|$. The function is defined as follows:

$$f_x(p) = \begin{cases} (\emptyset, 0), & \text{if } p \notin \textsc{ct}(x); \\ (\emptyset, rank(p(x))), & \text{if } rank(p(x)) > 1; \\ (\{q \mid (q,p) \in connpr_{G_{T_c}}(c,x)\}, 1), & \text{if } rank(p(x)) = 1 \end{cases}$$

The number of such functions is:

$$\mathcal{F} = |2^{upreds(P)} \times \{r \mid 0 \leqslant r \leqslant n2^{2n} - 2^n + 1\}|^{|upreds(P)|} = (2^n(n2^{2n} - 2^n + 1))^n,$$

which is exponential in n.

In the following we show that every two nodes which are in the same tree T_c in CS, but on different branches in T_c, and which have associated identical functions form a caching pair. Let x and y be such that $right_{T_c}(y,x)$ and f_x and f_y agree for every input value. Then:

- for every $p \in upreds(P)$: $p \notin \textsc{ct}(x)$ iff $p \notin \textsc{ct}(y)$, or in other words $\textsc{ct}(x) = \textsc{ct}(y)$;
- for every $p \in \textsc{ct}(x)/\textsc{ct}(y)$: $rank(p(x)) = rank(p(y))$. Thus, $isp(r,x) = isp(r,y)$ for every $1 \leqslant r \leqslant n2^{2n} - 2^n + 1$;
- $connpr_{G_{T_c}}(c,x) = connpr_{G_{T_c}}(c,y)$, if $rank(x) = rank(y) = 1$.

It is clear then that (x,y) is a caching pair.

For every tree $T \in F$ consider now the tree T' obtained by removing all cached nodes from T together with all redundant nodes and their successors. Due to the construction of EF we have that at any time there is at most a redundant node in the completion structure. As such, $|T'| \geqslant |T| - |fr(T)| - 1$, where $fr(T)$ is the frontier of T (we remove at most $|fr(T)| + 1$ nodes from T, as cached nodes and successors of redundant nodes are nodes in $fr(T)$). We have that $|fr(T')| \leqslant \mathcal{F}$, otherwise there would be two nodes $x, y \in fr(T')$ such that $f_x = f_y$, and thus x and y form a caching pair - this is in contradiction with the fact that T' does not contain any cached node.

Thus, $|fr(T')|$ is exponential in n, the depth of T' is also at most $n2^{2n} - 2^n$, and the number of nodes in T' is exponential in n. As the maximum degree of T is bounded by $degree(P)$, it follows that $|fr(T)|$ is bounded again by an exponential in the size of P. Hence, $|T|$ is bounded by an exponential in the size of P. As the number of trees in a completion structure is at most $cts(P) + 1$, it follows that there are at most an exponential number of nodes in the size of P in any complete $\mathcal{A}_3$-completion structure. □

The following complexity result is an immediate consequence of Proposition 19:

Corollary 6. $\mathcal{A}_3$ runs in the worst case in non-deterministic exponential time.

5.6 Soundness

In this section we prove that algorithm $\mathcal{A}_3$ is sound. Note that the proof is quite involved and as such it will use a number of lemmas.

Proposition 20 (Soundness of $\mathcal{A}_3$)**.** Let P be a FoLP and let $p \in \mathit{upreds}(P)$. If there exists a complete clash-free $\mathcal{A}_3$-completion structure for p with respect to P, then p is satisfiable with respect to P.

Proof. From a clash-free complete completion structure for p with respect to P, we construct an open interpretation, and show that this interpretation is an open answer set of P that satisfies p. Let $\langle \mathit{EF}, \textsc{ct}, \textsc{st}, G\rangle$ be such a clash-free complete completion structure with $\mathit{EF} = \langle F, \mathit{ES}\rangle$ the interconnected forest and $G = (V, A)$ the corresponding dependency graph, and let bl and ch be the sets of blocking pairs and caching pairs corresponding to the completion. Let $\mathit{blocked} = \{y \mid (x, y) \in \mathit{bl}\}$ and $\mathit{cached} = \{y \mid (x, y) \in \mathit{ch}\}$ be the sets of blocked and cached nodes, respectively.

1. *Construction of the open interpretation.*

   We construct a new graph $G_{\mathit{ext}} = (V_{\mathit{ext}}, A_{\mathit{ext}})$ by extending G in the following way: for every blocking pair and caching pair, the content of the blocking/caching node is copied into the content of the blocked/cached node, and all connections from the blocking/caching node to its successors or within itself are replicated by connections from the blocked/cached node to the successors of the blocking/caching node or within itself (or, in other words, the content of the blocked/cached node is identical to the content of the blocking/caching node and it is motivated in a similar way):

   - $V_{\mathit{ext}} = V \cup \{a_{x|y} \mid a \in V \wedge \mathit{arg}_1(a) = x \wedge (x, y) \in \mathit{bl} \cup \mathit{ch}\}$;
   - $A_{\mathit{ext}} = A \cup \{(a_{x|y}, b_{x|y}) \mid (a, b) \in A \wedge \mathit{arg}_1(a) = x \wedge (x, y) \in \mathit{bl} \cup \mathit{ch}\}$;

   Then, let (U, M) be the following open interpretation:

   - $U = N_{\mathit{EF}}$, i.e., the universe is the set of nodes in the interconnected forest, and
   - $M = V_{\mathit{ext}}$, i.e., the interpretation corresponds to the set of nodes in the extended graph.

   Before showing that (U, M) is an open answer set, i.e. that M is a minimal model of P_U^M, we also extend the interconnected forest EF with arcs from every blocked/cached node to all successors of the corresponding blocking/caching node. This gives rise to an extended forest $\mathit{EF}^{\mathit{ext}} = (F, \mathit{ES}^{\mathit{ext}})$, where:

   $$\mathit{ES}^{\mathit{ext}} = \mathit{ES} \cup \{(y, z) \mid (x, y) \in \mathit{bl} \cup \mathit{ch} \wedge (x, z) \in A_{\mathit{EF}}\}.$$

   For every $c \in \mathit{cts}(P) \cup \{\varepsilon\}$, we denote with T_c^{ext} the extended tree induced by $\mathit{EF}^{\mathit{ext}}$. The extended forest captures in a more accurate way the structure of M: blocked/cached nodes are connected to successors of the corresponding blocking/caching nodes, as their contents are justified similarly to the content of the blocking/caching nodes. Figure 5.7 depicts how an interconnected forest is extended in the case where it contains a caching pair (x, y).

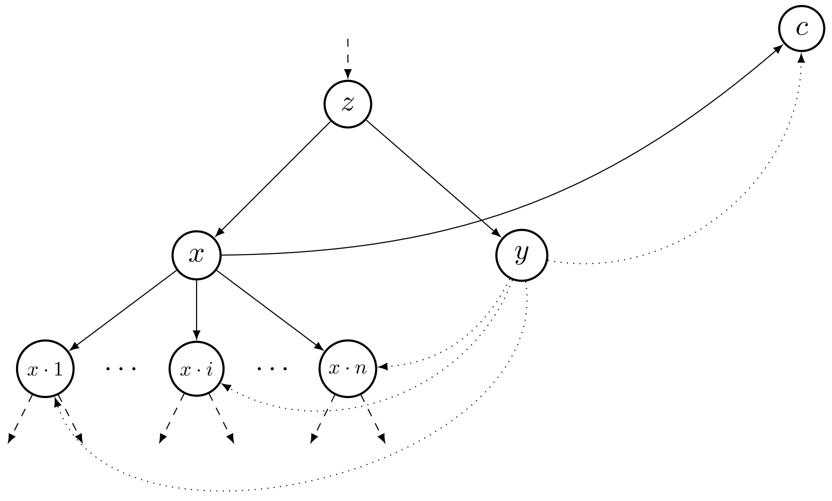


Figure 5.7: Justifying a cached node y by reusing the successors of its corresponding caching node x

2. *M is a model of P_U^M.* We observe that $M \models P_U^M$ if $M \models P_U$. In the following we will show that $M \models P_U$. We start with some helping lemmas.

   Lemma 12. Let $(x, y) \in bl \cup ch$ and $G_{ext} = (V_{ext}, A_{ext})$ be constructed as described above. Then, for any ground rule $r \in P_{N_{EF}}$: $V_{ext} \models r$ iff $V_{ext} \models r_{x||y}$ iff $V_{ext} \models r_{y||x}$.

   Proof. By construction of V_{ext}. □

   Lemma 13. Let $UC = \langle EF, \text{CT}, G\rangle$ be a unit completion structure for a FoLP P with root ε, where $EF = (F, ES)$, and $G = (V, A)$. Then, the following holds:

   $$V \models \bigcup_{r \in P_{N_{EF}}} r_{(arg_1(head(r)))||\varepsilon}.$$

   Proof. By construction of a unit completion structure: the root node of UC, ε, is saturated. Hence, every rule which might deduce something about atoms having ε as the first argument is satisfied by V. □

   Returning to the actual proof of M being a model of P_U, we first observe that the following identity holds as regards P_U:

   $$P_U = \bigcup_{x \in U} \bigcup_{r \in P_U} r_{arg_1(head(r)))||x}.$$

Consequently, in the following we will show that for every node $x \in U$:

$$M \models \bigcup_{r \in P_U} r_{arg_1(head(r))||x} \tag{5.1}$$

That is, all rules in P_U which might deduce atoms having as first argument a certain individual x, are satisfied by M.

We distinguish between:

- (i) $x \notin blocked \cup cached$. Then, at some point during the construction of CS, x has been expanded by replacing it with a unit completion structure $UC = \langle EF', \text{CT}', G' \rangle$, where $G' = (V', A')$. According to Lemma 13: $V' \models \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||\varepsilon}$. Then $(V')_{\varepsilon||x} \models \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||x}$. We have that $(V')_{\varepsilon||x} \subseteq M$, and thus:

  $$M \models \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||x} \tag{5.2}$$

  We next observe that for every rule $r \in P_U$ such that:

  $$r_{arg_1(head(r))||x} \notin \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||x},$$

  it is the case that:

  $$M \not\models r_{arg_1(head(r))||x} \tag{5.3}$$

  Intuitively, the body of $r_{arg_1(head(r))||x}$ must contain a successor ground term t which is not a successor of x in EF and an atom of the form $f(x,t)$; otherwise, $r_{arg_1(head(r))||\varepsilon} \in \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||\varepsilon}$, and consequently $r_{arg_1(head(r))||x} \notin \bigcup_{r \in P_{N_{EF'}}} r_{arg_1(head(r))||x}$. But, by construction of the open interpretation (U, M), for every atom atom of the form $f(x,y) \in M$, where x is neither blocked, nor cached, $y \in succ_{EF}(x)$. Thus $f(x,t)$ is not satisfied by M and $r_{arg_1(head(r))||x}$ is not satisfied by M, either.

  From (5.3) and (5.2), (5.1) follows.

- (ii) suppose $x \in blocked \cup cached$. Then, according to Lemma 12, for every $r \in P_U$: $M \models r_{arg_1(head(r))||x}$ iff $M \models (r_{arg_1(head(r))||x})_{y||x}$ iff $M \models r_{arg_1(head(r))||y}$, where y is the corresponding blocking or caching node. That $M \models r_{arg_1(head(r))||y}$ follows from case (i).

3. *M is a minimal model of P_U^M.* Before proceeding with the actual proof we introduce a notation and a lemma which will prove useful in the following. By 'local' cycles in G we denote cycles in which all unary atoms have identical arguments or there are no unary atoms. All other cycles in G are said to be 'non-local'.

The following lemma associates paths in the dependency graphs G/G_{ext} to paths in the underlying interconnected/extended forest: EF/EF^{ext}. It basically says that by projecting a path in the dependency graph on the arguments of every atom in the path and eliminating all binary arguments and redundant unary arguments, one obtains a path in the extended forest.

Lemma 14. Let $Pt = (a_1, \ldots, a_n) \in paths_G/paths_{G_{ext}}$, with $pred(a_1) \in upreds(P)$, and $pt = (b_1, \ldots, b_p)$ be a tuple obtained by considering the arguments of unary atoms in Pt in the order in which these atoms appear in Pt, and by eliminating successively occurring duplicates. Formally, $b_i = arg_1(a_{k_i})$, for every $1 \leqslant i \leqslant p$, where $(k_1, \ldots, k_p)$ is a sequence of indices, $1 \leqslant k_i \leqslant n$, which satisfies the following conditions:

- $k_i < k_j$, for every $1 \leqslant i < j \leqslant n$;
- for every $1 \leqslant i \leqslant n$, there exists $1 \leqslant j \leqslant p$ such that $k_j = i$ iff all of the following hold:
  - $pred(a_i) \in upreds(P)$;
  - $i = 1$ or $pred(a_{i-1}) \in bpreds(P)$ or $pred(a_{i-1})$ is different from $pred(a_i)$.

Then, $pt \in paths_{EF}/paths_{EF^{ext}}$. We will also call pt, the *argument path* of Pt and denote it as $argpath(Pt)$.

Furthermore, if Pt is a non-local cycle in G/G_{ext}, than pt is a cycle in EF/EF^{ext}.

Example 32. Let

$$Pt_1 = (a(x), f(x,y), d(y), g(y,z), b(z), c(z))$$

be a path in G, and

$$Pt_2 = (a(x), b(x), f(x,y), c(x), d(y), f(y,x), a(x))$$

be a path in G_{ext}.

The projection of Pt_1 on the arguments of unary atoms is (x, y, z, z). By eliminating successive duplicates we obtain that: $argpath(Pt_1) = (x, y, z)$, which is a path in EF. In the case of Pt_2, its projection on the arguments of unary atoms is (x, x, x, y, x). Again, by eliminating successive duplicates we obtain that$argpath(Pt_2) = (x, y, x)$, which is a path in EF^{ext}. Further on, $argpath(Pt_2)$ is a cycle in EF^{ext}; this is due to the fact that Pt_2 is a cycle in G_{ext}.

Proof. [Lemma 14] We construct a sequence of pairs of indexes $((k_1, q_1), \ldots, (k_{p-1}, q_{p-1}))$ such that k_i is the greatest index for which $args(a_{k_i}) = b_i$ and q_i is the smallest index for which $args(a_{q_i}) = b_{i+1}$, for every $1 \leqslant i < p$.

Then, we consider subpaths of Pt of the form $(a_{k_i}, \ldots, a_{q_i})$, for $1 \leqslant i < p$. Every such subpath has the form: $(p(b_i), f_1(b_i, b_{i+1}), \ldots, f_s(b_i, b_{i+1}), q(b_{i+1}))$, with $p, q \in$

$upreds(P)$, $f_1, \ldots, f_s \in bpreds(P)$, and $s \geqslant 0$ (if $s = 0$ there is no binary atom in the subpath). Thus, $(b_i, b_{i+1}) \in A/A'$ for every $1 \leqslant i < p$ and pt is a path in EF/EF^{ext}.

If Pt is a cycle, then $a_1 = a_n$ and $args(a_n) = args(a_{q_{p-1}}) = b_p$. Then, $args(a_1) = b_1 = b_p$, and thus, pt is a cycle as well. □

Now we can proceed to the actual proof of minimality of M. Assume there is a model $M' \subset M$ of $Q = P_U^M$. Then there exists some $l_1 \in M$ such that $l_1 \notin M'$. Take a rule $r_1 \in Q$ of the form $l_1 \leftarrow \beta_1$ with $M \models \beta_1$; note that such a rule always exists by construction of M and expansion rule (i). If $M' \models \beta_1$, then $M' \models l_1$ (as M' is a model), a contradiction. Thus, $M' \not\models \beta_1$, and there must be the case that there exists some $l_2 \in \beta_1$ such that $l_2 \notin M'$. Continuing with the same line of reasoning, one obtains an infinite sequence $\{l_1, l_2, \ldots\}$ with $(l_i \in M)_{1 \leqslant i}$ and $(l_i \notin M')_{1 \leqslant i}$. However, M is finite (the complete clash-free completion structure has been constructed in a finite number of steps, and when constructing M (V_{ext}) we added only a finite number of atoms to the ones already existing in V). Hence, there exist $1 \leqslant i, j$, $i \neq j$, such that $l_i = l_j$. We observe that $(l_i, l_{i+1})_{1 \leqslant i} \in A_{ext}$ by construction of A_{ext} and by the definition of the (i) Expansion-Unary-Positive rule, so our assumption leads to the existence of a cycle in G_{ext}.

Assume G_{ext} has a cycle $C = (a_1, \ldots, a_n = a_1)$. As G does not have any cycle (by construction), every cycle in G_{ext} is a result of introducing new nodes/arcs in G: consequently, every cycle must contain at least one atom having as an argument a blocked or a cached node.

The potential cycles in G_{ext} fall in one of the following categories:

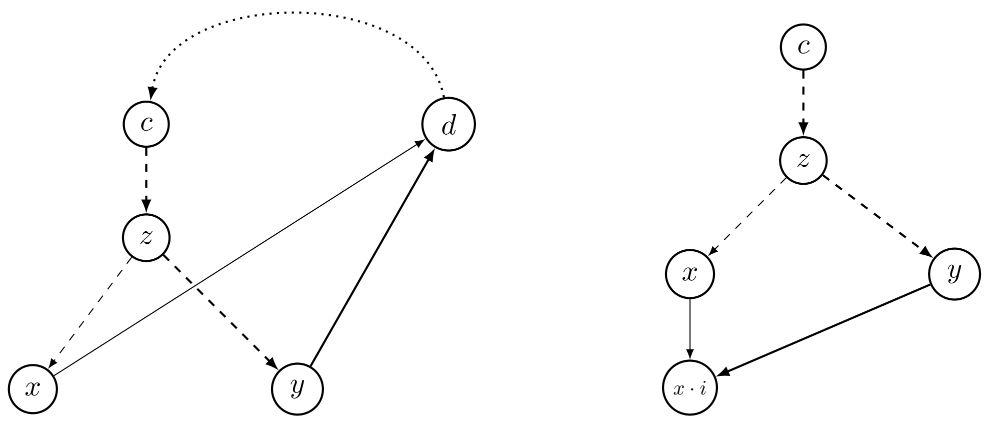
- 'local' cycles: as previously introduced, cycles in which all unary atoms have identical arguments or there are no unary atoms.
- 'caching' cycles: non-local cycles in which some atoms have as arguments cached nodes, but there is no atom which has as one of its arguments a blocked node;
- 'blocking' cycles: non-local cycles in which some atoms have as arguments blocked nodes.

We show by reductio ad absurdum that each of these types of cycles cannot appear in G_{ext}.

Lemma 15. There are no local cycles in G_{ext}.

Proof. Assume $C = (a_1, \ldots, a_n = a_1)$ is a local cycle in G_{ext}. Then C contains only atoms of the form $p(x)$, and $f(x, y)$, for $p \in upreds(P)$, $f \in bpreds(P)$, and $x, y \in N_{EF}$. Assume $x \in blocked/cached$. Then let $z \in N_{EF}$ be such that $(z, x) \in bl/ch$. Then $C_{x|z} = ((a_1)_{x|z}, \ldots, (a_n)_{x|z} = (a_1)_{x|z})$ is a cycle in G. Contradiction with the fact that there are no cycles in G. □

In the following we will denote as:



a) the caching arc goes outside the original tree b) the caching arc stays within the original tree

Figure 5.8: The argument path of a caching cycle

- 'blocking arc': an arc of the form $(y, x \cdot i)$, where (x, y) is a blocking pair in a tree T and $x \cdot i$ is a successor of x in T;
- 'caching arc': an arc of the form $(y, x \cdot i)$, where (x, y) is a caching pair in a tree T and $x \cdot i$ is a successor of x in T;
- 'blocking path': a path in a tree T: $path_T(x, y)$, where (x, y) is a blocking pair.

Lemma 16. There are no caching cycles in G_{ext}.

Proof. Assume C is a caching cycle in G_{ext}: then, as C is non-local, it must contain at least two unary atoms with distinct arguments, one of these arguments being a cached node. In other words, $argpath(C)$ contains at least two nodes, one of them being a cached node. Let T_c be a tree in which such a cached node appears, and y be the right-most cached node in $argpath(C)$ with respect to its position in T_c, i.e. there is no cached node $z \in argpath(C)$ such that $right_{T_c}(z, y)$. Also, let x be its corresponding caching node: $(x, y) \in ch$.

Claim: $path_{T_c}(c, y) \subset argpath(C)$.

Proof: There has to be an outgoing arc from y which is part of $argpath(C)$. As y is a cached node, such an arc can be only a caching arc, and thus, has one of the following forms:

- (y, d), where $d \in cts(P) \cup \{\varepsilon\}$, and $(x, d) \in ES$ (Figure 5.8 a)): then, $argpath(C)$ spans across several trees. As $argpath(C)$ is a cycle, in order to reach y from a tree other than T_c, it is necessary to pass through c: that is, there is a path in T_c^{ext} from

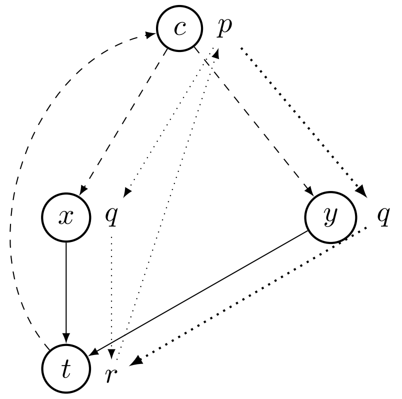


Figure 5.9: A caching cycle

c to y: but the only such path is $path_{T_c}(c, y)$ (once one exits $path_{T_c}(c, y)$ it is not possible to re-enter it as y is the right-most cached node in $argpath(C)$; also it is not possible to return to a node already visited on the path as the cycle contains no blocked nodes).

- $(y, x \cdot i)$, where $x \cdot i \in T_c$ (Figure 5.8 a)): then, there should be a path from $x \cdot i$ to y in EF^{ext}. There is no such path in T_c^{ext} as from $x \cdot i$ one can only reach successors or nodes which are at the left of $x \cdot i$, and implicitly of y. Thus, the only possibility is that such a path includes nodes outside T_c. In this case, the reasoning is similar to the previous case.

Thus, the claim is true. We now show how to reduce a caching cycle with n cached nodes among the arguments of the atoms in the cycle to a caching cycle with $n-1$ cached nodes among the arguments of the atoms in the cycle.

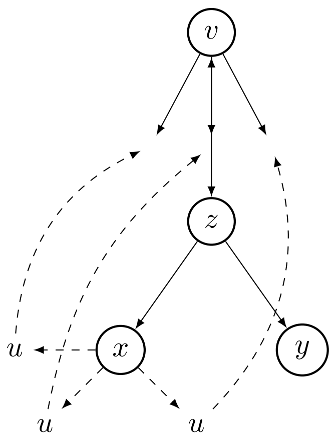
Let (x, y) be a caching pair as above, $z = lca_{T_c}(x, y)$, and let t be a successor of x in EF such that $(y, t) \subset argpath(C)$. From the claim and the construction of $argpath(C)$ we have that:

- $C = Pt_1\hat{\ }Pt_2\hat{\ }Pt_3$, where $Pt_1 \in paths_{G_{T_c}}(p(c), q(y))$, $Pt_2 \in paths_{G_{T_c}}(q(y), r(t))$, and $Pt_3 \in paths_{G_{ext}}(r(t), p(c))$, for some $p, q, r \in upreds(P)$ (not necessarily distinct) (Figure 5.9).
- $argpath(C) = path_{T_c}(c, y)\hat{\ }(y, t)\hat{\ }pt$, where $pt \in paths_{EF^{ext}}(t, c)$ and pt contains $n-1$ cached nodes (the n-th node in $argpath(C)$ is y).

From the caching condition, as $rank(y) = 1$, we know that:

$$connpr_{T_c}(c, y) \subseteq connpr_{T_c}(c, x).$$

Then, there must be a path $Pt_4 \in paths_{G_{T_c}}(p(c), q(x))$. Also, let $Pt_5 = (Pt_2)_{y|x}$. As $(x, y) \in ch$, it follows that $Pt_5 \in paths_{G_{T_c}}(q(x), r(t))$. By replacing Pt_1 with Pt_4 and



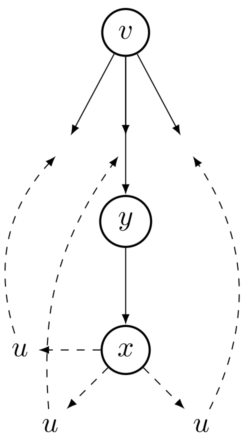


Figure 5.10: Reaching a node y from x where y is either to the right or above x in the tree

Pt_2 with Pt_5 in C one obtains a cycle C' with $argpath(C') = path_{T_c}(c, x)\hat{}(x, t)\hat{}pt$. As x is a cached node, $path_{T_c}(c, x)$ contains no cached node, and thus $argpath(C')$ contains $n - 1$ cached nodes.

Thus, it is always possible to transform a caching cycle to a cycle with less cached nodes, and eventually to a cycle which contains no cached nodes. As we started with cycles with no blocked nodes, and no blocked nodes are introduced in the construction, the end cycle contains no blocked nodes either. As such, it is a cycle in G - this is in contradiction with the fact that there are no cycles in G. Thus, the original assumption was false: there are no caching cycles in G_{ext}.

We will show next that there are no blocking cycles.

Lemma 17. There is no path Pt in G_{ext} such that its argument path contains a blocking path: for every $(x, y) \in bl$ such that $x, y \in T$ and $Pt \in Paths_{G_{ext}}$, $path_T(x, y) \not\subseteq argpath(Pt)$.

Proof. Assume $path_T(x, y) \subseteq argpath(Pt)$. Then, there are two nodes $a_1, a_2 \in G$, with $args(a_1) = x$, and $args(a_2) = y$ and a path $Pt' \in paths_G(a_1, a_2)$ such that $Pt' \subseteq PtC$. But this contradicts with the fact that $connpr_G(x, y) = \emptyset$. Thus, the initial assumption was false. □

Lemma 18. Let $Pt \in paths_{T^{ext}}(x, y)$, for some tree $T \in F$ such that $right_T(y, x)$ or $y <_T x$. Then, there must be a blocking pair (v, u) such that $u \in Pt$ and $v <_T z$ (if $right_T(y, x)$) or $v <_T y$ (if $y <_T x$), where $z = lca_T(x, y)$.

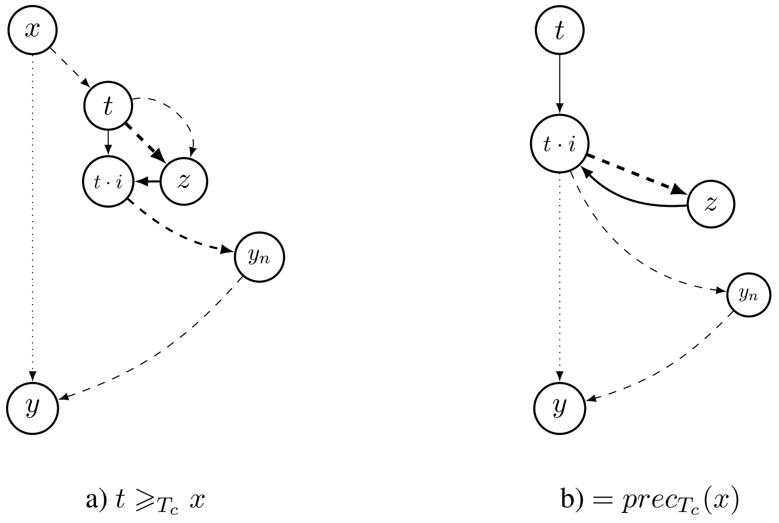


Figure 5.11: If z is a blocked node and $(t, z) \in bl$, then there is a path in G with argument path $Path_T(t, z)$

Proof.

From x one can reach successors of one of its ancestors via blocking arcs, nodes to the left of x in T via caching arcs, or situated below x in T via arcs in the tree. As y is to the right of x in T or above x in T, it cannot be reached via caching arcs unless one first reaches some node which is an ancestor of y or to the right of y. Intuitively, this can only be done when linking from a successor of x via a blocking arc to the successor of a node on the common path between x and y. Figure 3 a) describes the situation where $right_T(y, x)$, while Figure 3 b) describes the situation where $y <_T x$.

Lemma 19. Let Pt be a path in G_{ext} from $p(x)$ to some $q(y)$: $Pt \in paths_{G_{ext}}(p(x), q(y))$ such that x is an ancestor of y in some tree T_c: $x <_{T_c} y$, and for every $z \in argpath(Pt)$, it holds that $z \in T_x$. Then, there is a path in G_{T_c} from some $r(x)$ to $q(y)$.

Proof. Similarly to Lemma 16 we first show how to reduce a path in G_{ext} containing n cached nodes to another path in G_{ext} containing $n - 1$ cached nodes. Assume the first cached node which appears in $argpath(Pt)$ is y_n and its corresponding caching node is x_n. Then there must be an arc $(y_n, t_n) \subseteq argpath(Pt)$ such that $(x_n, t_n) \in A_{T_c}$. Thus, $argpath(Pt) = pt_1\hat{}(y_n, t_n)\hat{}pt_2$, with $pt_1 \in paths_{EF^{ext}}(x, y_n)$, and $pt_2 \in paths_{EF^{ext}}(t_n, y)$. We show how to construct from Pt a path $Pt_1 \in paths_{G_{ext}}(r(x), q(y))$, for some $r \in upreds(P)$ such that $argpath(Pt_1) = path_{T_c}(x, t_n)\hat{}pt_2$. Thus, all potential blocked nodes between x and t_n are eliminated and the cached node y_n is eliminated as well. Obviously, $argpath(Pt_1)$ has $n - 1$ cached nodes.

First, we show that $pt_1 = path_{T_c}(x, y_n)$. Assume the opposite. Then, pt_1 must contain at least a blocked node. Let z be the closest such node to x: as pt_1 contains no cached nodes, that is equivalent to saying that $path_{T_c}(x, z) \subseteq pt_1$. There must also be a node $t <_{T_c} z$ such that: $(t, z) \in bl$ and $(z, t \cdot i) \subseteq pt_1$, for some $i \in \mathbb{N}_{>0}$. As $t \cdot i \in T_x$, it follows that either $t \in T_x$ or $t = prec_{T_c}(x)$.

In the first case ($t \in T_x$) (Figure 5.11 a)) we have that $(t, z) \subseteq path_{T_c}(x, z) \subseteq pt_1$ and thus, there exists a path in G from some $p(t)$ to some $q(z)$, where (t, z) is a blocking pair. This is in contradiction with Lemma 17.

In the second case (Figure 5.11 b)), $t \cdot i = x$, and thus $(z, x) \subseteq pt_1$. Thus, there exist paths $T_1 \in paths_G(p(z), q(x))$ and $T_2 \in paths_G(q(x), r(z))$, for some $p, q, r \in upreds(P)$. Then $(T_1)_{z|t} \in paths_G(p(t), q(x))$ and $(T_1)_{z|t}{}^\frown T_2 \in paths_G(p(t), r(z))$. Again, this is in contradiction with Lemma 17.

Thus, $pt_1 = path_{T_c}(x, y_n)$ and $argpath(Pt) = path_{T_c}(x, y_n){}^\frown(y_n, t_n){}^\frown pt_2$. Then, there must be paths $T_1 \in paths_{G_{T_c}}(p(x), s(y_n))$, $T_2 \in paths_{G_{ext}}(s(y_n), w(t_n))$, and $T_3 \in paths_{G_{ext}}(w(t_n), q(y))$ such that $Pt = T_1{}^\frown T_2{}^\frown T_3$. We distinguish between:

- $t_n = x$. Then $Pt_1 = T_3$. It is simple to verify that it satisfies the properties mentioned above.
- $t_n > x$. Then $x_n \in T_x$ and $z_n = lca_{T_c}(x_n, y_n) \in T_x$. Let $r_n = rank(s(y_n))$. From the existence of T_1 we know that $r_n \leqslant ||x|| \leqslant ||z||$. Obviously, $s \in isp(r_n, y_n)$. Then, from the fact that (x_n, y_n) is a caching pair we have that $isp(r_n, x_n) \supseteq isp(r_n, y_n)$. Thus, $s \in isp(r_n, x_n)$: there must a path in G_{T_c} from a node above x to $s(x_n)$. Let l be the intersection of this path with x (this is guaranteed to exist as $x_n \in T_x$).

  Thus, there exists a path $T_4 \in paths_{G_{T_c}}(l(x), s(x_n))$. As $(x_n, y_n) \in ch$, it follows that there exists also a path $(T_2)_{y_n|x_n} \in paths_{G_{T_c}}(s(x_n), w(t_n))$ (see Figure 5.12). Then: $T_5 = T_4{}^\frown(T_2)_{y_n|x_n} \in paths_{G_{T_c}}(l(x), w(t_n))$. Let $Pt_1 = T_5{}^\frown T_3$. Again, Pt_1 fulfills the conditions required in the construction.

By successive applications of the transformation previously described, we obtain a sequence of paths $Pt_1, \ldots, Pt_n \in paths_{G_{ext}}$, which all have $q(y)$ as their final element, and where $argpath(Pt_i)$ contains $n - i$ cached nodes, for every $1 \leqslant i \leqslant n$. In particular $argpath(Pt_n)$ contains no cached node. Following a similar argument to the one used above to show that $pt_1 = path_{T_c}(x, y_n)$, one can show that $argpath(Pt_n)$ contains no blocked nodes, either. Thus, $argpath(Pt_n)$ is actually a path in T_x: $argpath(Pt_n) = path_{T_c}(x, y)$, and $Pt_n \in paths_{G_{T_c}}$. □

The lemma can be generalized to the case where the argument path contains also nodes above x.

Lemma 20. Let Pt be a path in G_{ext} from $p(x)$ to some $q(y)$: $Pt \in paths_{G_{ext}}(p(x), q(y))$, where $p, q \in upreds(P)$ such that x is an ancestor of y in some tree T_c: $x <_{T_c} y$. Then, there is a path in G_{T_c} from some $r(x)$ to $q(y)$, where r is in $upreds(P)$ as well.

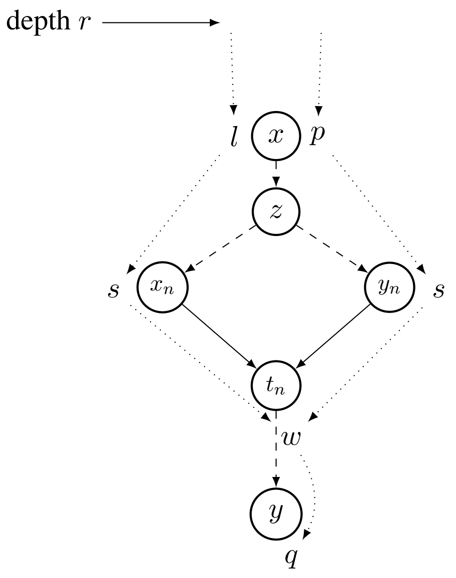


Figure 5.12: Reducing the number of cached nodes in blocking cycles

Proof.

If $c \in argpath(Pt)$, let Pt_1 be a path such that $Pt_1 \in paths_{G_{ext}}(r(c), q(y))$, for some $r \in upreds(P)$, and c occurs only once in $argpath(Pt_1)$. Then, for every $z \in argpath(Pt_1)$ it holds that $z \in T_c$ and we can apply Lemma 19 with respect to Pt_1: there has to be a path $Pt_2 \in paths_{G_{T_c}}(s(c), q(y))$, for some $s \in upreds(P)$. But, $argpath(Pt_2) = path_{T_c}(c, y)$, and then by construction of $argpath$ (Lemma 14), there must a path $Pt_3 \in paths_{G_{T_c}}(v(x), q(y))$, for some $v \in upreds(P)$.

If $c \notin argpath(Pt)$, then for every $z \in argpath(Pt)$, we have $z \in T_c$. Let $z \in path_{T_c}(c, y)$ be such that:

- z is a blocking node in CS,
- there exists a node $t \in T_c$ such that $(z, t) \in bl$, and $t \in argpath(Pt)$,
- there exists $i \in \mathbb{N}_{>0}$ such that $z \cdot i \in argpath(Pt)$ and $(t, z \cdot i) \subseteq argpath(Pt)$.

In other words, z is the first blocking node on the path $path_{T_c}$ which corresponds to some blocked node on the path Pt. In the following we will distinguish between:

- $z \leqslant_{T_c} x$. As $(t, z \cdot i) \subseteq argpath(Pt)$, and $Pt \in paths_{G_{ext}}(p(x), q(y))$, there must be some paths $Pt_1 \in paths_{G_{ext}}(p(x), r(t))$, $Pt_2 \in paths_{G_{ext}}(r(t), s(z \cdot i)$, $Pt_3 \in paths_{G_{ext}}(s(z \cdot i), q(y))$ such that $Pt = Pt_1{}^\wedge Pt_2{}^\wedge Pt_3$.
  As $(t, z) \in bl$, there must be some path $(Pt_2)_{t|z} \in paths_{G_{ext}}(r(z), s(z \cdot i)$.
  Let $Pt_4 = (Pt_2)_{t|z}{}^\wedge Pt_3$. Then $Pt_4 \in paths_{G_{ext}}(r(z), q(y))$.

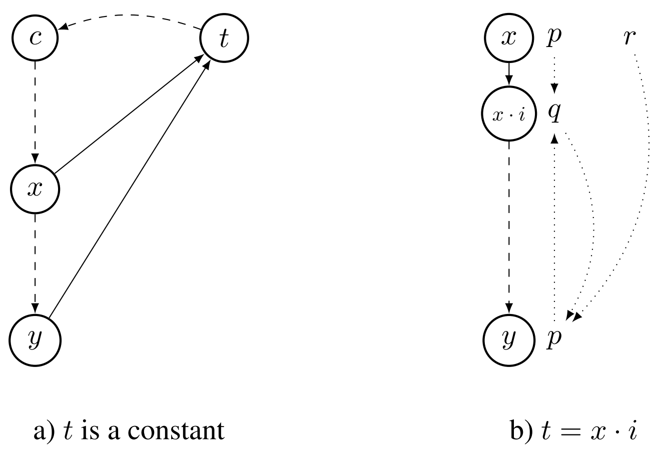


Figure 5.13: Unfolding a blocking cycle

We show next that every node $v \in argpath(Pt_4)$ is within T_z. Assume the opposite. Then there is a node $u \in T_c$ such that $u \in argpath(Pt_4)$ and either $right_{T_c}(u,z)$, $u <_{T_c} z$, or $right_{T_c}(z,u)$. From the fact that $u \in argpath(Pt_4)$ it follows that there exist paths $pt_1 \in paths_{T^{ext}}(z,u)$, and $pt_2 \in paths_{T^{ext}}(u,y)$.

- If $right_{T_c}(u,z)$ or $u <_{T_c} z$, given pt_2, according to Lemma 18 there must be a blocking pair (t,v) such that $t \in pt_2$ and $v <_{T_c} lca_{T_c}(y,u)$ or $v <_{T_c} u$. Clearly, the last condition translates into $v <_{T_c} z$. This is in contradiction with the original assumption that z is the highest node with this property.
- If $right_{T_c}(z,u)$, together with the existence of pt_1, one can apply again Lemma 18 and obtain a contradiction.

In both cases we obtained a contradiction, thus, for every $v \in argpath(Pt_4)$: $v \in T_z$. Together with the existence of Pt_4, one can apply Lemma 19 and obtain that there is a path $Pt_5 \in paths_G(s(z),q(y))$, for some $s \in upreds(P)$. But as $argpath(Pt_5) = path_{T_c}(z,y)$ and $x \in path_{T_c}(z,y)$, by construction of the argument path (Lemma 14), there must be a path $Pt_6 \in paths_{G_{T_c}}(w(x),q(y))$.

- $z >_{T_c} x$. Again, one can show that every node $v \in argpath(Pt)$ is within T_x using a similar argument as we used above to show that every node $v \in argpath(Pt_4)$ is within T_z. Thus, one can apply Lemma 19 with respect to Pt and obtain the desired result.

Lemma 21. There are no blocking cycles in G_{ext}.

Proof. Assume there is a blocking cycle C in G_{ext}. As C is at the same time a non-local cycle and a blocking cycle, $argpath(C)$ contains at least two nodes from N_{EF}, one of which should be a blocked node. Let $y \in T_c$ be such a blocked node and let x be such that: $(x, y) \in bl$. Then there exists a node $t \in N_{EF}$ such that $(x, t) \in A_{EF}$ (t is a successor of x in EF) and $(y, t) \subseteq argpath(C)$. Depending on the position of t in EF we distinguish between:

- $t \in cts(P)$ (Figure 5.13 a)): then, $c \in argpath(C)$ and there must be a path Pt in G_{ext} such that $argpath(Pt) \in paths_{EF^{ext}}(c, y)$. According to Lemma 20, there is a path $Pt' \in G$ with $argpath(Pt') = path_{T_c}(c, y)$. As $c <_{T_c} x <_{T_c} y$, by construction of $argpath(Pt')$ as in Lemma 14, there must be a path $Pt'' \in G$ with $argpath(Pt'') = path_{T_c}(x, y)$. But this is in contradiction with Lemma 17.
- $t = x \cdot i$, for some $i \in \mathbb{N}_{>0}$ (Figure 5.13 b)) and $C = Pt_1 \hat{} Pt_2$ with $Pt_1 \in paths_{G_{ext}}(p(y), q(x \cdot i))$ and $Pt_2 \in paths_{G_{ext}}(q(x \cdot i), p(y))$, for some $p, q \in upreds(P)$. As $(y, x \cdot i)$ is a blocking arc, $(Pt_1)_{y|x} \in paths_G(p(x), q(x \cdot i))$.

  We have that $(Pt_1)_{y|x} \hat{} Pt_2 \in paths_{G_{ext}}(p(x), p(y))$, and according to Lemma 20, there is a path $Pt_3 \in paths_{G_T}(r(x), p(y))$, for some $r \in upreds(P)$, such that: $argpath(Pt_3) = path_{T_c}(x, y)$. Again, this is in contradiction with Lemma 17.

Thus, in both cases we obtained a contradiction and there cannot be any blocking cycle in G_{ext}. $\square$

5.7 Completeness

In this section we show that the algorithm $\mathcal{A}_3$ is *complete*.

Proposition 21 (completeness of $\mathcal{A}_3$)**.** Let P be a FoLP and $p \in upreds(P)$. If p is satisfiable with respect to P, then there exists a complete clash-free $\mathcal{A}_3$ completion structure for p with respect to P.

Proof.

If p is satisfiable with respect to P then p is forest-satisfiable with respect to P (Proposition 5). We construct a clash-free complete $\mathcal{A}_3$-completion structure for p with respect to P, by guiding the application of the match, blocking, caching, and redundancy rules with the help of a forest model of P which satisfies p. The proof is similar to the completeness proof for the algorithm $\mathcal{A}_1$ described in Section 3.5, but requires additional mechanisms to deal with the new redundancy rule and with the new caching rule. Again, we make use of several lemmas.

Lemma 22. Let (U, M) be a forest model for a FoLP P, with:

- $EF = \langle \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\}, ES \rangle$, and
- $\mathcal{L} : \{T_\varepsilon\} \cup \{T_a \mid a \in cts(P)\} \cup A_{EF} \to 2^{preds(P)}$,

as in Definition 11.

Then, for every node $x \in U$, there is a unit completion structure $UC = \langle EF', \text{CT}, G \rangle$ for P, with $EF' = (\{T_{\varepsilon'}\}, ES')$ and $G = (V, A)$, which satisfies the following:

- $y \in N_{EF'}$ iff $y_{\varepsilon'||x} \in N_{EF}$;
- $(\varepsilon', y) \in A_{EF'}$ iff $(x, y_{\varepsilon'||x}) \in A_{EF'}$;
- $\textsc{ct}(\varepsilon') = \mathcal{L}(x) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(x))$;
- $\textsc{ct}(y) \subseteq \mathcal{L}(y_{\varepsilon'||x}) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(y_{\varepsilon'||x}))$, for every $y \in N_{EF'}$;
- $\textsc{ct}(\varepsilon', y) = \mathcal{L}(x, y_{\varepsilon'||x}) \cup \mathit{not}\ (\mathit{upreds}(P) - \mathcal{L}(x, y_{\varepsilon'||x}))$, for every $y \in N_{EF'}$.

Proof. Follows from the completeness of algorithm $\mathcal{A}_2$.

Now we proceed to the actual construction. Let (U, M) be the forest model which guides the expansion with $EF = \langle\{T_\varepsilon\} \cup \{T_a \mid a \in \mathit{cts}(P)\}, ES\rangle$, where $p \in \mathcal{L}(\varepsilon)$ and let $CS = \langle EF', \textsc{ct}, \textsc{st}, G\rangle$ be an initial $\mathcal{A}_3$-completion structure for checking satisfiability of p with respect to P with $EF' = \langle\{T'_{\varepsilon'}\} \cup \{T'_a \mid a \in \mathit{cts}(P)\}, ES'\rangle$, where $p \in \textsc{ct}(\varepsilon')$. We will expand CS in a depth-first fashion (the order of processing trees is not important, just that their contents are expanded depth-first; the expansions of different trees can also be interleaved). Always a node with status unexp is selected for expansion.

Let π be a function which relates nodes/arcs from the interconnected forest in the completion structure in construction to nodes/arcs in the forest model: $\pi : N_{EF'} \to U$. We show that at any point during the construction, the following property (‡) holds:

$$(\ddagger)\begin{cases}\{q \mid q \in \textsc{ct}(z)\} \subseteq \mathcal{L}(\pi(z)), \text{ for all } z \in N_{EF'}\\ \{q \mid \mathit{not}\ q \in \textsc{ct}(z)\} \cap \mathcal{L}(\pi(z)) = \emptyset, \text{ for all } z \in N_{EF'}\\ \textsc{ct}(z) = \mathcal{L}(\pi(z)) \cup \mathit{not}\ (\mathit{bpreds}(P) - \mathcal{L}(\pi(z))), \text{ for all } z \in A_{EF'}.\end{cases}$$

That is, the positive content of a node in the completion structure is contained in the label of the corresponding forest model node, the negative content of a node in the completion structure is disjoint with the label of the corresponding forest model node, and the content of an arc in the completion structure is identical to the label of the corresponding arc in the forest model.

The property will be proved by induction and it is used at every step of the construction (for nodes for which it was already proved to hold): as such the induction step coincides with the construction step.

Base case: We set $\pi(\varepsilon') := \varepsilon$ and $\pi(a) := a$, for every $a \in \mathit{cts}(P)$. That the induction hypothesis is fulfilled, follows from the way the initial completion structure for p with respect to P was defined.

Induction/Construction step: Let x be the node currently selected for expansion in EF': $\textsc{st}(x) := \mathit{unexp}$. Perform the following operations:

(i) Check whether the blocking or caching conditions are met:

- assume there is a node $y \in N_{EF'}$ such that (y, x) is a blocking pair. Then mark x as a blocked node and stop its expansion.
- assume there is a node $y \in N_{EF'}$ such that (y, x) is a caching pair. Then mark x as a cached node and stop its expansion.

Naturally, in both cases (‡) still holds, as we have not modified the content of nodes and we also did not add any new nodes. Note that when applying the blocking or caching rule, we no longer use the guidance of the model (U, M), as it might justify in a different way the atoms which have x and its successors as one of their arguments; we are interested in finding a finite representation of a model which satisfies p, not necessarily of the original model which we used for guidance (actually the soundness proof constructs a non-forest model from a clash-free complete completion structure).

(ii) If x is neither blocked nor cached, according to the induction hypothesis, there is a node $\pi(x) \in N_{EF}$ such that $\textsc{ct}(x) \subseteq \mathcal{L}(\pi(x)) \cup not\ (upreds(P) - \mathcal{L}(\pi(x)))$. Let UC be a unit completion structure with root ε' corresponding to node $\pi(x)$ as in Lemma 22. UC has the property that:

$$\textsc{ct}(\varepsilon') = \mathcal{L}(\pi(x)) \cup not\ (upreds(P) - \mathcal{L}(\pi(x))) \text{ and}$$

$$\textsc{ct}(a) \subseteq \mathcal{L}(a) \cup not\ (upreds(P) - \mathcal{L}(a)), \text{ for every } a \in cts(P).$$

Then, from the induction hypothesis, it follows that $x \in N_{EF'}$ is matchable with UC. Apply the *Match* rule for x and UC.

For every node y added to/updated from EF' by addition of UC: $y \in N_{EF'}$ and $(x, y) \in A_{EF'}$, we have that:

$$\textsc{ct}(y) \subseteq \mathcal{L}(y_{x||\pi(x)}) \cup not\ (upreds(P) - \mathcal{L}(y_{x||\pi(x)})) \text{ and}$$

$$\textsc{ct}(x, y) = \mathcal{L}(\pi(x), y_{x||\pi(x)}) \cup not\ (bpreds(P) - \mathcal{L}(\pi(x), y_{x||\pi(x)})).$$

We set $\pi(y) = y_{x||\pi(x)}$, for every such node, and the induction hypothesis holds.

(iii) Check whether the redundancy rule condition is met: assume there is a node $y \in N_{EF'}$ such that (y, x) form a redundancy pair. Note that unlike the models constructed by our algorithm, arbitrary forest models might contain 'redundant' nodes (or better said they translate to completion structures which contain such nodes). A redundancy pair (y, x) signals a redundant computation in the form of a sub-tree in the interconnected forest from y to x. The way to overcome this is to simply skip the redundancy when constructing a completion structure. As the redundant part of the model is first incorporated in the completion structure, when encountering such a redundancy pair we modify the structure by cutting out the redundant part: y is replaced with x. In order to implement this policy we reuse the $collapse$ operation introduced in the Completeness proof for $\mathcal{A}_1$ in Section 3.5.3 (with the slight adjustments to deal with $\mathcal{A}_3$-completion structures instead of $\mathcal{A}_2$-completion structures). Thus, when (y, x) is a redundancy pair, we simply apply $collapse_{CS}(y, x)$.

As concerns the image of y under π in EF, it is changed to the previous image of x: $\pi(y) := \pi(x)$. The induction hypothesis still holds. □

From Corollary 6, Proposition 21, and Proposition 20 it follows that:

Corollary 7. Satisfiability checking of unary predicates with respect to FoLPs is in NExpTime.

As another corollary of the soundness and completeness results, we obtain a bound on the size of models needed to satisfy a certain unary predicate p with respect to a FoLP P. The bound improves the bound obtained in Section 3.5.5 by one exponential:

Proposition 22. FoLPs have the bounded finite model property: for a FoLP P and a unary predicate $p \in \mathit{upreds}(P)$, if p is satisfiable with respect to P, then there exists an open answer set (U, M) of P, which satisfies P such that the size of U is bounded by an exponential in the size of P.

Again, this result opens the way for standard ASP reasoning as discussed in Section 3.5.5, this time being needed to consider programs with only up to exponential-sized universes. All advantages and disadvantages of the technique, mentioned in Section 3.5.5, hold here as well.

5.8 Simple Reasoning with FoLPs: The Case of Simple Forest Logic Programs

In this section we introduce a fragment of Forest Logic Programs, called simple Forest Logic Programs, which are obtained from FoLPs by restricting the usage of predicate recursion in rules. For this class of programs, $\mathcal{A}_3$ can be simplified such that both the blocking and the caching conditions collapse in a simple subset-based anywhere blocking condition. We will refer to the new simplified algorithm as $\mathcal{A}_3^s$. The fragment is a superset of the language of acyclic FoLPs, and as such can simulate reasoning within the DL $\mathcal{SHOQ}$.

5.8.1 Simple FoLPs: Definitions

We start with some preliminary notations.

Definition 35. For a FoLP P, let $D(P) = (V_d, A_d)$ be the following graph:

- $V_d = \{p \in \mathit{preds}(P) \mid p \text{ is not free}\}$: the set of vertices contains all non-free predicates from P,
- $A_d = \{(p, q) \mid \exists \alpha \leftarrow \beta \in \mathit{rules}(P).\alpha = \{l_1\}, l_2 \in \beta^+, \mathit{pred}(l_1) = p, \mathit{pred}(l_2) = q\}$: the set of arcs is composed of all tuples (p, q) such that there exists either a rule of the form (2.3) or a rule of the form (2.4) with a head literal l_1 and a positive body literal l_2 such that $\mathit{pred}(l_1) = p$, and $\mathit{pred}(l_2) = q$.
- an arc $(p, q) \in A_d$ is said to be *marked*, iff there exists a rule r of the form (2.3) or of the form (2.4) such that $\alpha = \{p\}$ and $q \in \delta_m$, for some $1 \leqslant m \leqslant n$, if r is of the form (2.3), and $q \in \delta$, if r is of the form (2.4).

We call D_P the *marked positive predicate dependency graph* of P.

Definition 36. Let P be a FoLP. Then P is said to be *simple*, iff $D(P)$ does not contain any cycle that has a marked edge.

The restriction on $D(P)$ ensures that there is no path from some atom $p(x)$ to some atom $p(y)$ in the atom dependency graph of P_U which does not contain some atom $q(z)$ such that q is free, where $p \in \mathit{upreds}(P)$, $q \in \mathit{preds}(P)$, U is some arbitrary universe, and $x, y \in U$, $x \neq y$.

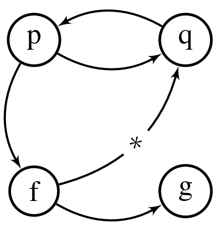


Figure 5.14: Marked Dependency Graph $D(P)$

Indeed, observe that any marked cycle in $D(P)$ contains a unary predicate and thus corresponds to a path from some $p(x)$ to some $p(y)$ in the atom dependency graph of P_U, which does not contain any atom formed with a free predicate. Furthermore, as the cycle is marked, x has to be different from y, as any marked edge marks a change of argument from X or X, Y in the head of a rule to some Y_m or Y in the body, and thus the cycle is 'non-local'.

Example 33. Consider the program P:

$$\begin{array}{rrcl} r_1: & p(X) & \leftarrow & q(X), f(X, Y), not\ p(Y) \\ r_2: & q(X) & \leftarrow & p(X) \\ r_3: & f(X, Y) & \leftarrow & g(X, Y), q(Y) \end{array}$$

The marked positive dependency graph is depicted in Figure 5.14, where the arc (f, q) is marked. While (p, q, p) is an unmarked cycle, (q, p, f, q) is a marked cycle, and thus P is not a simple FoLP. However, if the last rule is dropped, P becomes a simple FoLP.

As the fragment of acyclic FoLPs disallows any form of recursion it follows that:

Proposition 23. Let P be an acyclic FoLP. Then, P is a simple FoLP.

The converse is not valid: there are simple FoLPs which are not acyclic (cf. Example 33). A corollary of Proposition 23 and Proposition 2 is the following lower bound for reasoning with simple FoLPs:

Corollary 8. Satisfiability checking of unary predicates with respect to simple FoLP is EXPTIME-hard.

5.8.2 Reasoning with Simple FoLPs

The restriction on simple FoLPs ensures that anywhere subset blocking can be used as a termination mechanism. As such, the notion of redundant nodes is no longer needed and we do not need to keep track of dependencies in the constructed model via the dependency graph G. We proceed with defining $\mathcal{A}_3^s$-completion structures as usual.

An (initial) $\mathcal{A}_3^s$-*completion structure* is a tuple $\langle EF, \text{CT}, \text{ST}\rangle$, with EF, CT, and ST being defined similarly to their counterpart in an $\mathcal{A}_3$-completion structure.

Note however, that the fragment still allows for local cycles, i.e. cycles induced by rules of type $a(X) \leftarrow a(X)$. As such, in the process of constructing the set of non-redundant UCSs, the dependency graph G is still needed.

An (initial) $\mathcal{A}_3^s$*-completion structure* is evolved using the rule (xii$'$) Match introduced in Section 5.3. As already mentioned, for termination purposes we employ a single applicability rule, which is defined formally as follows:

Rule. ***Anywhere Blocking***. Let P be a simple FoLP, $p \in \mathit{upreds}(P)$, and $\langle EF, \text{CT}, \text{ST}\rangle$ be an $\mathcal{A}_3^s$*-completion structure* for p with respect to P.

A node $x \in N_{EF} - \mathit{cts}(P)$ such that $\text{ST}(x) = \mathit{unexp}$ is *blocked* iff there exists a node $y \in N_{EF} - \mathit{cts}(P)$, such that:

- $\mathit{right}_{T_c}(x, y)$,
- $\text{ST}(y) = \mathit{exp}$, and
- $\text{CT}(x) \subseteq \text{CT}(y)$.

In this case (y, x) is said to be a *blocking pair*. No expansions can be performed on a blocked node.

In the light of what we said so far, the notions of complete and clash-free completion structures become:

Definition 37. A *complete* $\mathcal{A}_3^s$*-completion structure* for a unary predicate p with respect to a simple FoLP P, is an $\mathcal{A}_3^s$-completion structure that results from applying the rule (xii') Match to an initial $\mathcal{A}_3^s$-completion structure for p with respect to P, taking into account the applicability rule Anywhere Blocking, until no further expansions can be applied.

Definition 38. An $\mathcal{A}_3^s$-completion structure for a unary predicate p with respect to a FoLP P, $CS = \langle EF, \text{CT}, \text{ST}\rangle$, is *clash-free* iff there is no node $x \in N_{EF}$ such that $\text{ST}(x) = \mathit{unexp}$.

Algorithm 5.2 provides an overview of the new algorithm $\mathcal{A}_3^s$ for reasoning with simple FoLPs. The algorithm can be seen as a simplified version of $\mathcal{A}_3$.

Termination and worst-case running time complexity of $\mathcal{A}_3^s$ follow immediately from the usage of the anywhere blocking rule and from the fact that for every node $x \in N_{EF}$, there are finitely many values which $\text{CT}(x)$ may take:

Proposition 24. $\mathcal{A}_3^s$ runs in the worst case in non-deterministic exponential time in the size of the simple FoLP under consideration.

Proof. First, we show that the maximum number of nodes in a complete $\mathcal{A}_3^s$-completion structure is exponential in the size of the program. The size of such a completion structure is bounded by the following factors: if we leave all the leaves of the trees in the completion apart, there are at most $2^p + c$ nodes, where $p = |\mathit{upreds}(P)|$, and $c = |\mathit{cts}(P)|$, as there are at most 2^p different possible configurations for the content of a unary node, and all the nodes which are not leaves or constants have to have different contents (otherwise they would form blocking pairs

Algorithm 5.2: Overview of $\mathcal{A}_3^s$, an algorithm for reasoning with simple FoLPs

```
input  : simple FoLP $P$, unary predicate $p$;
output: yes, if $p$ is satisfiable with respect to $P$; false, otherwise;
1) Construct the set of non-redundant Unit Completion Structures (UCSs) for $P$ (if not
available already);
2) Construct an $\mathcal{A}_3^s$-initial completion structure $CS$ for $p$ with respect to $P$;
3) $S = N_{EF}$;
repeat
    Pick up a node $x \in S$ such that $\textsc{st}(x) = unexp$;
    $S = S - \{x\}$;
        a) if there is a node $y \in N_{EF} - cts(P)$ such that $\textsc{st}(x) = unexp$ and
        $\textsc{ct}(x) \subseteq \textsc{ct}(y)$ then
            $x$ is blocked;
            $\textsc{st}(x) = exp$;
        end
        b) if $\textsc{st}(x) = unexp$ then
            non-deterministically choose a unit completion structure $UC$ which
            matches $x$ and perform $expand_{CS}(x, UC)$;
            if $\textsc{st}(x) = unexp$ then
                return false;
            end
        end
until $S = \emptyset$;
return true;
```

and at least one of them would be a leaf). The maximum number of leaves is $r(2^p + c - 1)$, where $r = rank(P)$ is the maximum arity of any of the trees in the extended forest. So, the completion structure has in the worst case a number N of nodes that is exponential in the size of the program: $N \leqslant (2^p + c)(r + 1) - r$.

As computing the set of non-redundant UCSs can be done in the worst case in exponential time, and the algorithm is non-deterministic, it follows that $\mathcal{A}_3^s$ runs in the worst case in non-deterministic exponential time in the size of the program. $\square$

The algorithm is sound and complete:

Proposition 25. A unary predicate p is satisfiable with respect to a simple FoLP P iff there is a complete clash-free $\mathcal{A}_3^s$-completion structure for p with respect to P.

Proof. We only sketch the soundness and completness proofs for $\mathcal{A}_3^s$ here as they can be seen as simplified versions of the respective soundness and completeness proofs for $\mathcal{A}_3$.

Soundness: From a complete clash-free $\mathcal{A}_3^s$-completion structure for p with respect to P, it is possible to construct a model by always reusing the successors of y as successors of x whenever (y, x) is a blocking pair.

As in the case of $\mathcal{A}_3$, the challenge is to show that the model constructed in this way is minimal. This amounts to showing that there are no non-local cycles in the atom dependency graph of the program P grounded with N_{EF} (the local cycles are taken care of in the phase of constructing the set of non-redundant UCSs). Assume there exists such a cycle. But then, one can show that there must be a cycle in D_P which contains a marked edge; this raises a contradiction with the fact that P is simple.

Completeness: If p is satisfiable with respect to P, then p is forest-satisfiable with respect to P. Again, starting with a forest model of P which satisfies p, we construct a clash-free complete $\mathcal{A}_3^s$-completion structure for p with respect to P. Whenever a choice has to be made by the (xii′) Match rule regarding which UCS to use for expanding a certain node, the forest model is used as a guidance and the Anywhere Blocking rule is used to enforce termination. □

5.9 Discussion and Related Work

Due to the new caching rule which allows nodes to reuse computation across branches of a completion structure and the new notion of redundant nodes, the worst-case running time complexity of $\mathcal{A}_3$ is one exponential level lower than that of $\mathcal{A}_1$ and $\mathcal{A}_2$. Furthemore, the new redundancy rule does no longer take in the best case exponential time to be applicable, as was the case for its original counterpart.

The main device that made these improvements possible was the new technique to show the finite bounded model property of FoLPs which is established during the completeness proof described in Section 5.7. The technique works by reducing possibly infinite forest models to ones of bounded size by collapsing nodes of the model with identical contents. However, in order for the collapsed model to still be a minimal model, some complex conditions have to be met concerning paths in the dependency graph associated to the original model.

The termination conditions for $\mathcal{A}_1$ and $\mathcal{A}_2$ stem also from such a technique to reduce models to a finite, bounded size: however, while the technique used by $\mathcal{A}_1$ merely achieved a bound of double exponential size, the current technique improves the bound to single exponential size. The difference between the two reduction techniques lies also in their impact on the redundancy rule:

- in the case of $\mathcal{A}_1$ and $\mathcal{A}_2$, the conditions which two nodes have to fulfil in order to be collapsed at proof-time, i.e. when a model is reduced, cannot be checked at run-time, i.e. during the construction of a completion structure: the required information is simply not available in the absence of an actual model. As such, the conditions do not translate into a direct condition for identifying redundancy at run-time. Instead the established bound on the model size has to be employed for termination.

- in the case of $\mathcal{A}_3$, a different set of conditions is employed to check that it is safe to collapse two nodes. These conditions can be checked also at run-time. Hence, the new notion of redundant nodes we introduced in this chapter.

Both techniques to reduce models and the ensuing termination conditions (in particular, the redundancy rules) are rather unusual: we are not aware of any algorithms which use similar book-keeping to enforce termination.

While in tableau algorithms, it is common to use some form of blocking across branches, this comes usually in the form of anywhere blocking, where the conditions that have to be fulfilled by a pair of nodes to be in a blocking relation are the same, disregarding whether the two nodes are on the same branch or on different branches. However, in our case, blocking across branches (which we call caching) has a different flavour compared to blocking within a branch: its role is actually to preserve blocking within extended branches, i.e. branches which are obtained by copying the subtree which has as root the caching node where the cached node resides.

As we will see in next chapter, in the case of CoLPs, the determinization of $\mathcal{A}_3$, which among others involves the modification of termination conditions such that both satisfiability and unsatisfiability are preserved when reusing computation via one of the termination mechanisms, will lead to very similar conditions for redundancy and caching; however, there will still be a gap between the blocking and the caching conditions. This was the main reason why we decided to refer to our new termination condition as caching, and not as anywhere blocking.

In a quest to find classes of logic programs which always have a stable model, [Fages, 1994] introduced the class of *positive-order-consistent* programs whose stable models coincide with the models of Clark's completion: these are programs in which all recursion between positive literals is restricted (no dependencies are allowed in the positive atom dependency graph). All acyclic FoLPs are positive-order-consistent. While simple FoLPs are not positive-order-consistent, the pre-compiling step, in which all local cycles are eliminated makes them behave de facto as positive-order-consistent programs: no further check on the dependency graph of the model is needed.

As such, in the case of simple FoLPs, we obtain a very simple blocking condition, subset-based anywhere blocking, which is similar to the blocking conditions employed by tableau algorithms for logics from the DL realm which do not contain inverses [Baader and Sattler, 2001].

CHAPTER 6

Worst-Case Optimal Reasoning with Conceptual Logic Programs and Simple Forest Logic Programs

The algorithm $\mathcal{A}_3$ described in Chapter 5 reduced the maximal size of completion structures to an exponential number of nodes in the size of the program. However, being a non-deterministic algorithm, it still ran in non-deterministic exponential time. This was also the case for $\mathcal{A}_3^s$, the algorithm for reasoning with simple FoLPs described in Section 5.8. In this chapter, we show how both $\mathcal{A}_3$ and $\mathcal{A}_3^s$ can be transformed into deterministic procedures that check satisfiability of unary predicates with respect to CoLPs and simple FoLPs, respectively, which run in the worst case in exponential time. Both procedures are worst-case optimal.

The new procedures which we call $\mathcal{A}_{3,c}^{det}$ and $\mathcal{A}_{3,s}^{det}$ consist in constructing an AND/OR tree and an AND/OR forest, respectively, with depth double in the size of the largest depth encountered when running the non-deterministic algorithm. At odd levels, there are OR nodes with unexpanded content (they contain just the constraints imposed by their predecessor), while at even levels, there are AND saturated nodes which are 'realizations' of their predecessor, i.e., they (together with their outgoing arcs and direct successors) describe a possible way to expand the predecessor node. Such a structure will be called an AND/OR completion structure and it is described in Section 6.1.

Each algorithm will evolve such an AND/OR completion structure by matching UCSs against the contents of OR nodes and creating for each match an AND successor for the corresponding OR node. The algorithms use similar mechanisms for termination to their non-deterministic counterparts. While in the case of $\mathcal{A}_{3,s}^{det}$, this consists in a simple anywhere blocking condition similar to the anywhere blocking condition employed by its non-deterministic counterpart $\mathcal{A}_3$, in the case of $\mathcal{A}_{3,c}^{det}$, all previous three termination conditions for $\mathcal{A}_3$: blocking, caching, and redundancy are refined and employed. Special care has to be taken when using anywhere blocking and caching, as in order to obtain a worst-case optimal behaviour, these conditions have to be applicable on pair of nodes which might be situated on branches belonging to different models. In the case of CoLPs, an AND/OR completion structure represents the space of all potential tree models which satisfy a unary predicate p with respect to a CoLP P and as such the algorithm does no longer need to backtrack, i.e. there is no non-determinism. In the case of simple FoLPs, due to the presence of constants, this is no longer the case; each OR node corresponding to some constant can potentially be expanded in several different ways, but for technical reasons we have to constrain such OR nodes to have just one successor in any AND/OR completion structure. This leads to a non-deterministic choice regarding the AND successor for each constant OR node. However, we show that the number of distinct AND/OR completion structure for a given simple FoLP is exponential in the size of the program and as such this does not harm the worst-case optimal behaviour of the algorithm.

Every node in an AND/OR completion structure will be assigned eventually a truth value based on the truth values of its successors or on the fact that there are no such successors. The answer to a satisfiability checking problem is true when all root nodes in an AND/OR structure are evaluated to true. As such, both algorithms have now two phases: an expansion phase, where an AND/OR completion structure is constructed, and an evaluation phase, where truth values are assigned to nodes and propagated through the structure such that eventually every node in the structure has assigned a certain truth value.

Section 6.2 describes $\mathcal{A}_{3,c}^{det}$, the deterministic worst-case optimal algorithm for reasoning with CoLPs, while Section 6.3 describes $\mathcal{A}_{3,s}^{det}$, the deterministic worst-case optimal algorithm for reasoning with simple FoLPs. Finally, Section 6.4 discusses why the deterministic approach does not scale in the case of (full) FoLPs and presents some related work.

6.1 AND/OR Completion Structures

As mentioned in the introduction, an AND/OR completion structure represents more or less an exhaustive description of the search space that has to be explored when constructing a tree model/forest model that satisfies a unary predicate p with respect to a certain CoLP/simple FoLP P. The underlying data structure can be seen as an extension of the data structure underlying an $\mathcal{A}_3$-completion structure. The notion of AND/OR completion structure which we introduce below is generic and as such contains some fields which will prove to be superfluous in one or the other case where it is employed, i.e. when constructing a tree model for a CoLP or when constructing a forest model for a simple FoLP. This will become clear in the definitions of the respective initial completion structures.

As mentioned in the introduction, an AND/OR completion structure has two types of nodes: AND nodes, which are saturated, and OR nodes, which are typically unsaturated, as they are successor nodes in UCSs used to expand the AND nodes. A function TYPE is introduced in the data structure to pinpoint the type of each node in the structure.

For an AND node to be blocked, cached, or redundant, its corresponding blocking, caching or redundancy witness node is an AND node which has been expanded using the same UCS as the original node. Thus, in order to check the blocking, caching, and redundancy conditions, the UCSs used to construct the AND nodes under comparison have to be known. As such, there is an extra function which keeps track for each node of the respective UCS. Note that, in this case, the content function stores redundant information. However, for simplicity, we keep this function as part of the definition of an AND/OR completion structure as well.

As nodes in a blocking, caching, or redundancy pair have been expanded using the same UCS, the predecessor OR node of a blocked, cached or redundant node could reuse as a successor node the corresponding blocking, caching or redundant node, instead of the node itself. We take this approach, since, as we will see later, it will turn out to be useful in the evaluation phase of the algorithms. Whenever a blocking, caching, or redundancy pair is identified, the blocked, cached, or redundant node is removed and a link (new arc) is created from its OR predecessor to the corresponding blocking, caching, or redundancy witness AND node. We want to keep track of these new arcs and the reason for their creation: as such, a new function is introduced which assigns to (some) arcs a special value, which indicates whether they link OR nodes to blocking, caching or redundancy witness nodes, respectively.

Definition 39. An *AND/OR completion structure* for a CoLP/simple FoLP P is a tuple $\langle EF$, CT, ST, G, TYPE, UCS, EVAL, SPEC, CONST $\rangle$ where:

- EF is an extended tree/forest;
- CT : $N_{EF} \cup A_{EF} \to 2^{preds(P) \cup not\ (preds(P))}$ is the 'content' function;
- ST : $N_{EF} \to \{exp, unexp\}$ is the 'status' function (note that AND nodes are always trivially expanded in this setting);
- $G = \langle V, A \rangle$ is the dependency graph with $V \subseteq \mathcal{B}_{P_{N_{EF}}}$;
- TYPE : $N_{EF} \to \{AND, OR\}$ is a function which keeps track of which nodes are AND nodes and which nodes are OR nodes;
- UCS : $N_{EF} \to \mathcal{UCS}_P$ is a (partial) function which keeps for each AND node track of which UCS has been used to expand the node[1], where $\mathcal{UCS}_P$ is the set of all (non-redundant) UCSs constructed with respect to P;
- EVAL : $N_{EF} \to \{true, false, unknown\}$ is a function which assigns one of the truth values $true$, $false$, or $unknown$, to every node in the AND/OR structure;
- SPEC : $A_{EF} \to \{bl, re, ca\}$ is a partial function which singles out arcs in EF which have a special status, like blocking, caching, or redundancy arcs;
- CONST : $N_{EF} \to cts(P)$ is a partial function which assigns to some OR nodes a particular constant from P.

An AND/OR completion structure is expanded as anticipated by 'realizing' OR nodes by way of matching unit completion structures against their content and creating a successor AND node for every successful match. Again, we define a general notion of expansion which applies to the most general notion of AND/OR completion structure as described in Definition 39.

[1]For OR nodes, the function is simply not defined.

The notion of matching a UCS against a node of an AND/OR completion structure is a straightforward adaptation of the similar notion of matching a UCS against an $\mathcal{A}_2$-completion structure.

Definition 40. Let $CS = \langle EF, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec}, \textsc{const} \rangle$ be an AND/OR-completion structure. An OR node $x \in N_{EF}$ is *matchable* with a unit completion structure $UC = \langle EF', \textsc{ct}', G' \rangle$ with root ε, where $EF' = (F', ES')$, iff:

- $\textsc{st}(x) = unexp$,
- $x = \varepsilon$, if $\varepsilon \in cts(P)$,
- UC locally satisfies $\textsc{ct}(x)$, and
- for every arc $(x, c) \in ES'$, and for every $\pm p \in \textsc{ct}'(c)$: $\mp p \notin \textsc{ct}(c)$.

We say that *UC matches x*.

Next we redefine the operation which expands a completion structure by addition of UCSs to deal with the case of AND/OR completion structures. In this case (partially expanded) OR nodes are not replaced with a matchable UCS, but for every possible match, the UCS is added to the completion structure as a successor to the OR node. Typically, an AND node is created for the root of the UCS and a sequence of OR nodes for the successor nodes in the UCS. In the case of newly created successor OR nodes which correspond to constants, the function CONST is defined on such nodes and its value is the respective constant. We will sometimes refer to such nodes as proxy constant nodes. Subsequently, such proxy constant nodes are expanded by linking to an AND successor of the corresponding constant OR node (if their contents match). We defer this operation to Section 6.3 as it is specific to the expansion of AND/OR completion structures for checking satisfiability of simple FoLPs.

Additionally, when keeping track of dependencies between atoms in the prospective models, some new arcs have to be introduced in G between atoms formed using predicates in the content of OR nodes and their realizations in the AND nodes (the atoms formed using the same predicate but having as argument the AND node). Note that in the previous version of the algorithm these connections were implicit.

Suppose that $x \in T$ is an unexpanded OR node in an AND/OR-completion structure $AO = \langle EF, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec}, \textsc{const} \rangle$, with $G = (V, A)$, for which the function CONST is undefined (the node is not a proxy constant node). Suppose also that x is matchable with m unit completion structures UC_i, $1 \leqslant i \leqslant m$ with root ε_i, where for every $1 \leqslant i \leqslant m$: $UC_i = \langle EF_i, \textsc{ct}_i, G_i \rangle$, with $G_i = (V_i, A_i)$. The operation of expanding x in UC by adding an AND successor to x corresponding to each matchable UCS UC_i, $1 \leqslant i \leqslant m$, denoted by $AOExpand_{AO}(x)$, is depicted in Figure 6.1, and modifies AO as follows:

- $\textsc{st}(x) := exp$;
- for every $1 \leqslant i \leqslant m$ do the following:

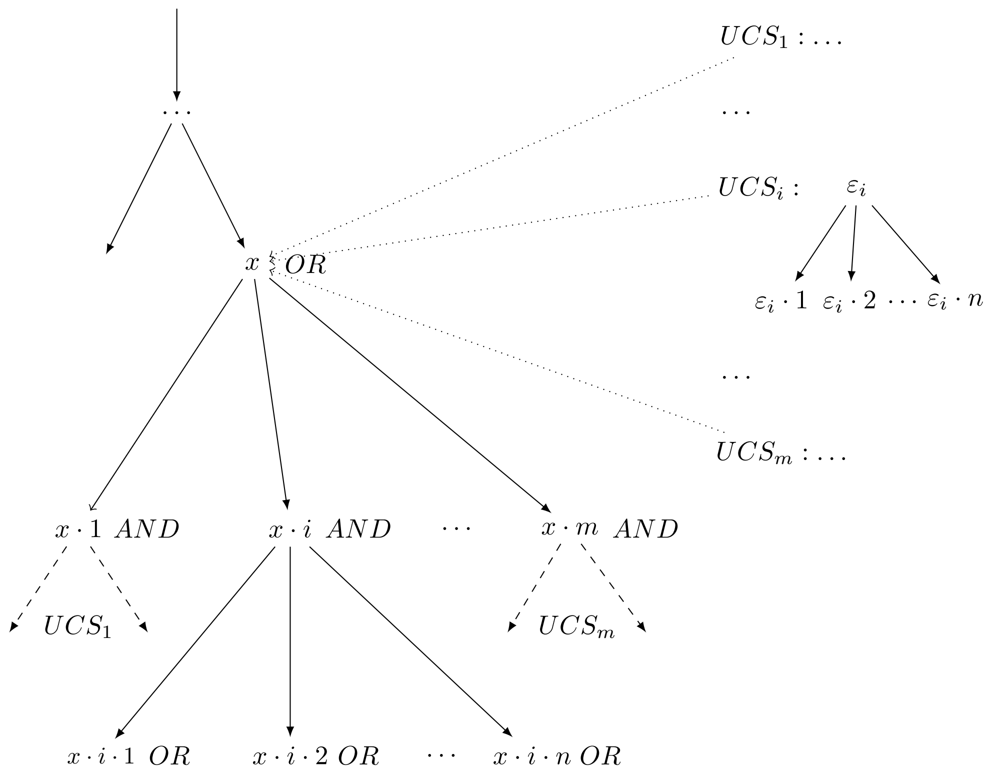


Figure 6.1: The expansion of an OR node in an AND/OR completion structure: the node matches with m UCSs: $UCS_1, UCS_2, \dots, UCS_m$.

- $T := T \cup (T_{\varepsilon_i})_{\varepsilon_i || x \cdot i}$ (adds the tree corresponding to the UCS to the AND/OR completion structure);
- Let $C_i := \{c_1, \dots, c_k\}$ be the set of constant nodes which appear in UC_i, i.e. the set $\{v \in cts(P) \mid (\varepsilon_i, v) \in ES_i\}$. Also let l be the number of non-constant successors of ε_i in UC_i: $l = |T_{\varepsilon_i}| - 1$. Then for every $1 \leqslant j \leqslant k$:
  - create a new successor of $x \cdot i$ in T: $T := T \cup x \cdot i \cdot (l + j)$;
  - set the constant value of the newly created node to c_j: $\textsc{const}(x \cdot i \cdot (l+j)) := c_j$ (the newly created node is a proxy constant node);
- for every $u \in N_{EF_i}$, set $\textsc{ct}(u_{\varepsilon_i || x \cdot i, c_1 || x \cdot i \cdot (l+1), \dots, c_k || x \cdot i \cdot (l+k)}) := \textsc{ct}'(u)$, where c_1, $\dots$, c_k are as above (sets the content of the newly added nodes to be identical to the content of the corresponding nodes in the UCS);
- for every $u \in succ_{EF_i}(\varepsilon_i)$, set $\textsc{ct}(x \cdot i, u_{\varepsilon_i || x \cdot i, c_1 || x \cdot i \cdot (l+1), \dots, c_k || x \cdot i \cdot (l+k)}) := \textsc{ct}'(\varepsilon_i, u)$, where $c_1, \dots, c_k$ are as above (sets the content of the newly added arcs to be identical to the content of the corresponding arcs in the UCS);

- $\textsc{type}(x \cdot i) := AND$ (the newly created node corresponding to the root of the UCS is an AND node);
- $\textsc{type}(y) := OR$, for every $y \in T_{x \cdot i} - \{x \cdot i\}$ (the newly created nodes corresponding to successor nodes in the UCS are OR nodes);
- $\textsc{st}(x \cdot i) := exp$ (the newly created AND node is expanded);
- $\textsc{st}(y) := unexp$, for every $y \in T_{x \cdot i} - \{x \cdot i\}$ (the newly created OR nodes are unexpanded);
- $V := V \cup \{a_{\varepsilon_i || x \cdot i, c_1 || x \cdot i \cdot (l+1), \ldots, c_k || x \cdot i \cdot (l+k)} \mid a \in V_i\}$, where $c_1, \ldots, c_k$ are as above (adds to the graph G the vertices corresponding to vertices in the graph G_i) ;
- $A := A \cup \{(a_{\varepsilon_i || x \cdot i, c_1 || x \cdot i \cdot (l+1), \ldots, c_k || x \cdot i \cdot (l+k)}, b_{\varepsilon_i || x \cdot i, c_1 || x \cdot i \cdot (l+1), \ldots, c_k || x \cdot i \cdot (l+k)}) \mid (a, b) \in A_i\}$, where $c_1, \ldots, c_k$ are as above (adds to the graph G the vertices corresponding to vertices in the graph G_i).
- $A := A \cup \{(p(x), p(x \cdot i)) \mid p \in \textsc{ct}(x)\}$ (some additional arcs are added from unary atoms formed with the original OR node to their realizations in the AND node);
- $\textsc{ucs}(x \cdot i) := UC_i$ (records the UCS used for the expansion of the AND node);
- $\textsc{eval}(y) := unknown$, for every $y \in T_{x \cdot i}$ (in the expansion stage the truth value of every node is 'unknown';

We also overload the $AOExpand$ operator by denoting with $AOExpand_{AO}(x, uc)$ the expansion of the OR node x in AO with a single AND successor node by using the UCS uc which matches x. This will be used later in this chapter, in Section 6.2.5.2 and Section 6.3.

6.2 Worst-Case Optimal Reasoning with Conceptual Logic Programs

In this section we describe $\mathcal{A}_{3,c}^{det}$, the deterministic algorithm for checking satisfiability of unary predicates with respect to CoLPs. As already mentioned, the algorithm works by evolving and evaluating an AND/OR completion structure. In Section 6.2.1 we introduce the notion of initial AND/OR completion structure for checking satisfiability of a unary predicate p with respect to a CoLP P, and describe how such a structure is expanded using the $AOExpand$ operation we introduced in the previous section.

As mentioned in the introduction, $\mathcal{A}_{3,c}^{det}$ uses the same termination conditions as its non-deterministic counterpart $\mathcal{A}_3$ which are blocking, caching, and redundancy. The general intuition for blocking and caching is identifying pairs of (AND) nodes which are structurally similar and in which one of the nodes can reuse the computation used to justify the content of the other node. However, in this case, as the AND/OR completion structure represents the space of all possible computations, both successful, i.e. leading to an open answer set, and unsuccessful, i.e. leading to a clash, the blocked/cached node not only reuses the justification of the corresponding blocking/caching node in an open answer set, but also has to be guaranteed not to be satisfiable when the corresponding blocking/caching node is not satisfiable either.

Also, in the case of redundancy, the intuition is slightly changed: a redundancy pair is formed again from two nodes on the same branch of the structure which have similar content and in between which the set of oldest paths in the dependency graph G which traverse the two nodes increases. However, while the occurrence of a redundant node stops the expansion of a branch, it does not constitute a clash and as such does not trigger the failure of the construction. Similar to the blocking and caching situation, the redundant node will have the same satisfiability status (truth value) as the redundancy witness.

In order to enforce these properties, the conditions which have to be fulfilled by pairs of nodes to form a blocking, caching, or redundancy pair, are stronger than in the non-deterministic version of the algorithm: in particular, always UCSs associated to specific AND nodes are compared, as opposed to the typical content subset condition we had before. Section 6.2.2 describes these termination conditions and when an AND/OR completion structure is fully expanded.

From what we said so far, there seems to be no difference between the de facto usage of blocking and redundancy pairs: they both refer to pairs of nodes on the same branch in the structure, in which one node caches the truth value/satisfiability status of the other. They are distinguished only by the conditions the corresponding computation paths between the two nodes have to fulfil: in the case of blocking, the chains of dependencies in the atom dependency graph have to dwindle, such that there are no dependencies between atoms formed with the blocking node and atoms formed with the blocked node, while in the case of redundancy such dependency chains have to increase.

The difference comes into play when the truth values of the blocking/redundancy witness nodes are set during the evaluation phase. While the satisfiability of a blocking node can rely on the satisfiability of its corresponding blocked node (which in its turn relies on the satisfiability of the blocking node, i.e. truth can be circularly motivated via cycles which involve blocking pairs), this is not the case for a redundancy witness node: if it is satisfiable, this is due to some alternative computation path which does not involve the corresponding redundant node. This is reflected in the evaluation phase which consists in a fix-point procedure. At every iteration, a set of nodes is assumed to be true: these all have to be blocked OR nodes; initially this is the set of all blocked OR nodes in the structure. The truth values of the nodes in the hypothesized set are further propagated through the structure to their parents, ancestors, etc. When the propagation leads to cycles which contain blocking arcs in which every node has truth value true, the corresponding blocked nodes stays in the hypothesized set. For a given blocked node, if no such cycle exists the node is removed from the set. Eventually a fixed point is reached – in this case all nodes assumed to be true are actually set to true. Note that no redundant node is part of such a hypothesized set. The evaluation strategy is described in Section 6.2.3.

Section 6.2.4 shows that the algorithm terminates and that it runs in the worst case in deterministic exponential time, while Section 6.2.5 shows that the procedure is sound and complete.

6.2.1 Evolving an AND/OR Completion Structure for a COLP

We start by introducing the notion of initial AND/OR-initial completion structure for checking satisfiability of a unary predicate p with respect to a CoLP P. Such a structure reuses the data structures which appear in a generic AND/OR completion structure as described in Definition 39 except for the last field $\textsc{const}$. Also, in this case, the extended forest which occurs in a generic AND/OR completion structure is actually an extended tree as CoLPs do not allow for constants.

Definition 41. An *AND/OR-initial completion structure for checking satisfiability of a unary predicate p with respect to a CoLP P* is a completion structure $\langle ET$, $\textsc{ct}$, $\textsc{st}$, G, $\textsc{type}$, $\textsc{ucs}$, $\textsc{eval}$, $\textsc{spec}\rangle$, where:

- $ET := (T, \emptyset)$ is an extended tree,
- $T := \{\varepsilon\}$, with ε being an arbitrary constant,
- $\textsc{ct}(\varepsilon) := \{p\}$,
- $\textsc{st}(\varepsilon) := unexp$,
- $G := \langle\{p(\varepsilon)\}, \emptyset\rangle$,
- $\textsc{type}(\varepsilon) := OR$,
- $\textsc{eval}(\varepsilon) := unknown$.

Such an initial AND/OR completion structure will be evolved as usually by applications of the basic expansion operation, $AOExpand$, in a certain order, which guarantees that no yo-yo caching will take place.

Rule. DetMatch. Let $AO = \langle ET$, $\textsc{ct}$, $\textsc{st}$, G, $\textsc{type}$, $\textsc{ucs}$, $\textsc{eval}$, $\textsc{spec}\rangle$, with $ET = (T, ES)$, be an AND/OR completion structure. If for some node $x \in T$:

- $\textsc{type}(x) = OR$,
- $\textsc{st}(x) = unexp$, and
- for every node y such that $right_T(y, x)$: $\textsc{st}(y) = exp$,

then apply $AOExpand_{AO}(x)$.

6.2.2 Termination Conditions: Blocking, Caching, and Redundancy

This section describes the new blocking, caching, and redundancy rules which are used by $\mathcal{A}^{det}_{3,c}$ (Sections 6.2.2.1, 6.2.2.3, and 6.2.2.2, respectively). It also specifies when an AND/OR completion structure for checking satisfiability of a unary predicate with respect to a CoLP is fully expanded (Section 6.2.2.4).

6.2.2.1 Deterministic Blocking Rule

In this version of blocking, we compare building blocks of the AND/OR completion structure, in the form of UCSs associated to AND nodes. Again, in order for the AND nodes to form a blocking pair, there should be no running paths in G between atoms formed with the respective nodes: to this purpose, it is enough to check for paths in G which have as underlying path in ET the path in T which links the two nodes, where the argument path of a path Pt in G, $argpath(Pt)$, is as defined in Section 5.6.

To this purpose, we introduce a new notation which will also prove useful in subsequent sections: given two nodes $x, y \in T$, by $connpr_{G|T}(x, y)$, we understand the set:

$$\{(p, q) \mid \exists Pt \in paths_G(p(x), q(y)).argpath(Pt) = path_T(x, y)\}.$$

As anticipated, in this version of the blocking rule, the blocked node and its successors are removed from the AND/OR completion structure and an arc is created from its OR predecessor to the corresponding blocking AND node. The arc is said to be a *blocking arc* and its SPEC value is updated to reflect this. The predecessor OR node which is linked to the blocking node is called a blocked OR node[2].

Rule. ***DetBlocking.*** If there exist two AND nodes $x, y \in T$ such that:

- y is an ancestor of x in T: $y <_T x$,
- $\text{UCS}(x) = \text{UCS}(y)$, and
- $connpr_{G|T}(x, y) = \emptyset$,

then x is *blocked* and y is its corresponding *blocking* node. Let:

- $u := prec_T(x)$,
- $ES := ES \cup \{(u, y)\}$,
- $\text{SPEC}(u, y) := bl$,
- $A := A \cup \{(p(u), p(y)) \mid p \in \text{CT}(u)\}$,
- $T := T - T_x$.

Arc (u, y) is a *blocking* arc, and u is a *blocked OR node*.

Note that, due to the construction of an AND/OR completion structure and to the blocking conditions, it also holds that there is no path $Pt \in paths_G$ such that $argpath(Pt) = path_T(y, z)$, where (z, y) is blocking arc.

[2]Note that the OR node might already be a blocked OR node by virtue of one of its other successors.

6.2.2.2 Deterministic Redundancy Rule

As for the non-deterministic case, the intention is to limit the expansion of a branch in case some redundant computation has been performed: however, here, once a redundancy pair has been identified it does not necessarily mean failure: there might be an open answer set which contains the UCS used to expand the two nodes, but the current path is not the right path to arrive to such an open answer set. Due to the exhaustive nature of an AND/OR completion structure, if there is such an open answer set, the completion will contain an alternative path to justify the content of the redundancy witness and its successors. Thus, in a sense, a redundant node can be seen as caching the truth value of its redundancy witness: this is also the case with blocking pairs; however, unlike in the case of blocking, a redundant node does not serve as a justification for the truth value of its counterpart node, the redundancy witness. In order for both nodes to be true, there has to be an alternative way to justify the content of the redundancy node which does not include the redundancy path.

Technically, to capture this relationship between nodes, similar conditions to the ones introduced in the non-deterministic algorithm are checked on pairs of AND nodes with identical corresponding UCSs:

Rule. *DetRedundancy*. If there exist two AND nodes $x, y \in N_{ET}$ such that:

- y is an ancestor of x in T: $y <_T x$,
- $\textsc{ucs}(x) = \textsc{ucs}(y)$,
- $rank(x) = rank(y) = r$, and
- $isp(k, y) = isp(k, x)$, for every $r \leqslant k \leqslant ||y||$,

then x is *redundant* and y is its corresponding *redundancy witness* node. Let:

- $u := prec_T(x)$,
- $ES := ES \cup \{(u, y)\}$,
- $\textsc{spec}(u, y) := re$,
- $A := A \cup \{(p(u), p(y)) \mid p \in \textsc{ct}(u)\}$, and
- $T := T - T_x$.

Arc (u, y) is a *redundancy* arc, and u is a *redundant OR node*.

As was the case with blocking, a redundant node and its successors are removed from the AND/OR completion structure, and an arc is created from its OR predecessor to the corresponding redundancy witness node. The arc is said to be a *redundancy arc* and its $\textsc{spec}$ value is updated to reflect this. The predecessor OR node which is linked to the redundancy witness node is called a redundant OR node[3]

[3]Again, the OR node might already be a redundant OR node by virtue of one of its other successors. Also, the OR node can be at the same time a blocked, redundant, and, as will be seen in the next expansion rule, a cached OR node.

6.2.2.3 Deterministic Caching Rule

As mentioned in the introduction, caching is about reusing (successful or unsuccessful) computation performed during the expansion of a node found on a different branch than the current node. When there exists some blocking node which is situated above the lowest common ancestor of the cached and caching node and whose corresponding blocked node is a descendant of the cached node, by reusing the computation of the caching node a new node will be created which is a copy of the above-mentioned blocked node. We would like for this copy of the blocked node to be in a blocking relation with the original blocking node. As such, some conditions regarding sets of paths in the dependency graph between atoms formed with nodes which are ancestors of both nodes in a caching pair and the nodes in the caching pair themselves were imposed in the non-deterministic version of the algorithm. Those conditions were enough to warrant that when the cached node is expanded successfully, by copying its expansion we obtain a successful expansion for the caching node, too. Here, the conditions are strengthened to enforce that if the expansion of the cached node fails (the node is evaluated to false), the hypothetical expansion of the caching node would lead to failure, as well.

A cached node and its successors are removed from the AND/OR completion structure and an arc is created from its OR predecessor to the corresponding caching node. The arc is said to be a *caching arc* and the predecessor OR node is said to be a cached OR node.

Rule. *DetCaching*. If there exist two AND nodes $x, y \in T$ such that:

- $\textsc{ucs}(x) = \textsc{ucs}(y)$,
- $isp(r, y) = isp(r, x)$, for every $1 \leqslant r \leqslant ||z||$, where z is the lowest common ancestor of x and y: $z = lca_T(x, y)$, and
- $right_T(y, x)$,

then let:

- $u := prec_T(x)$,
- $ES := ES \cup \{(u, y)\}$,
- $\textsc{spec}(u, y) := ch$,
- $A := A \cup \{(p(u), p(y)) \mid p \in \textsc{ct}(u)\}$, and
- $T := T - T_x$.

Arc (u, y) is a *caching* arc, and u is a *cached OR node*.

Note the similarity between the redundancy conditions and the caching conditions in the current version of the algorithm. Redundancy can be seen as a limit case of caching in which the caching node is situated not to the right but above the cached node (it is the lowest common ancestor of itself and of the cached node). Also, the conditions of the two rules, the deterministic caching and the deterministic redundancy rule, can be seen as a combination of the conditions for caching and redundancy in the non-deterministic version of the algorithm. This is not surprising, as, as discussed in Section 6.2.2.2, the redundancy rule has in this case a caching flavour since the redundancy node caches the truth value of the redundancy witness.

While the two rules could be merged into one, they are kept separate; this is both for consistency reasons – with the previous version of the algorithm, but also to stress the similarities between the blocking rule and the redundancy rule which work at the branch level, and the differences between both of these rules and the caching rule which works across branches.

6.2.2.4 Complete AND/OR Completion Structures

Definition 42. An AND/OR-*complete completion structure* for a CoLP P and a unary predicate p, is an AND/OR-completion structure that results from the repeated application of the rule *DetMatch* to an initial AND/OR-completion structure for p and P, taking into account the applicability rules *DetBlocking*, *DetRedundancy*, and *DetCaching* such that no rules can be further applied.

Note that in this case we do not deal explicitly with clashes: if a node cannot be expanded it will be assigned the truth value $false$ in the next section, while redundant nodes as discussed in Section 6.2.2.2 do not necessarily mean failure.

6.2.3 Evaluation of an AND/OR Completion Structure

After a complete AND/OR completion structure has been constructed, it has to be evaluated: every node in the completion is assigned eventually one of the truth values $true$ or $false$. After the expansion stage, the truth value of every node in the completion is $unknown$. The intuition regarding the evaluation of particular nodes, is that a node is assigned the truth value $true$ iff there is some open answer set where the content of that node is satisfied, and the truth value $false$ whenever there is no such open answer set.

At the same time, the assignment of truth values has to respect the AND/OR structure of the completion, that is, an OR node is true iff one of its successors is true, and an AND node is true iff each of its successors is true. In order to impose this last constraint, every time a set of nodes is assigned some truth value, the assignment is propagated upwards to the predecessors of the nodes in the set to as much of an extent as possible. That is, if some OR node has an ancestor which is true, one can already infer that its truth value is true. The same about an AND node which has a false successor: its truth value will be false.

Algorithm 6.1 describes a propagation procedure which given a complete AND/OR completion structure CS and a set S of nodes which have truth value $true$ or $false$, transforms the structure such that as many nodes as possible have assigned one of the truth values $true$ or $false$ (for which the justification of the truth values can be traced back to the initial set S). Note that this procedure might also change the truth value of some node to $true$ from $false$ or vice versa (in case one node had a preassigned value which does not agree with the value computed as a function of the successors which make part from S or whose truth value has been computed based on the truth values of some nodes from S).

Algorithm 6.1: The AND/OR Truth Value Propagation Procedure

```
input  : an AND/OR completion structure $AO = \langle EF$, CT, ST, $G$, TYPE, UCS, EVAL,
         SPEC $\rangle$, with $EF = (F, ES)$, and a set of nodes $S$ from $N_{EF}$ with truth value
         $true$ or $false$;
output: an updated AND/OR completion structure
repeat
    for every $x \in N_{EF}$ do
        $switch := false$;
        if $\exists y \in S$ such that $(x, y) \in A_{EF}$ then
            let v:=$\begin{cases} \bigvee_{z \in N_{EF},(x,z) \in A_{EF}} \text{EVAL}(z), & \text{if } type(x) = OR; \\ \bigwedge_{z \in N_{EF},(x,z) \in A_{EF}} \text{EVAL}(z), & \text{if } type(x) = AND; \end{cases}$
            if $v \neq \text{EVAL}(x)$ then
                $\text{EVAL}(x) := v$;
                $S := S \cup \{x\}$;
                $switch := true$;
            end
        end
    end
until $switch = false$;
```

In the following, we describe informally how every node of a complete AND/OR completion structure is assigned one of the truth values *true* or *false* by making repeated use of the propagation procedure just presented. The main steps of the evaluation procedure are as follows:

- *an initialization step*: in this step, all AND nodes which have no successors are assigned the truth value $true$ and all OR nodes which have no successors, are assigned the truth value *false*. Further on, these values are propagated using the propagation procedure.

- *a fix-point procedure*: at each iteration, the procedure hypothesizes a set of nodes as having truth value $true$, and checks whether the hypothesis was correct or was an overestimation. At every subsequent iteration, the hypothesized set is shrunk until an exact match is found. The original hypothesized set is the set of all blocked OR nodes. Every iteration consists of two steps:

    1. a copy CS' of the original AND/OR structure CS is created in which all blocked OR nodes are assigned the truth value $true$. The newly assigned truth values are propagated through the new AND/OR completion structure using the propagation procedure.

       The intuition for this step is to obtain an overestimation of the set of true nodes in the AND/OR completion structure. Besides nodes which are obviously true – the ones set as such during the initialization step, some nodes can be true (their content is satisfiable) by relying (at least partially) on circular justifications: in other words, they are part of a cycle in ET such that every node in the cycle is justified using the next node in the cycle as a successor.

However, every atom in an open answer set has to be well-supported, thus, there can be no cyclic dependencies between atoms. As such, cycles in ET are 'good' as long as they do not give rise to cycles in G. This brings us to the explanation we provided in the introduction to this chapter regarding the difference between blocking pairs and redundancy pairs. Due to the specific conditions of each termination rule, cycles which involve a blocking arc can be shown not to give rise to any underlying cycle in G, while all other cycles can be shown to give rise to such cycles (in particular, these are cycles which contain redundancy arcs, but no blocking arcs).

By assigning true to every blocked OR node (which is always part of a cycle which contains a blocking arc, and naturally every such cycle contains a blocked OR node) and propagating these assignments throughout the structure, we obtain an over-estimation of truth in the AND/OR completion structure as from such nodes one can reach every node in a good cycle. The resulting truth value assignment might be an overestimation, as some nodes in good cycles might still turn out to be false. This is the case, for example, for an AND node which belongs to such a cycle and has besides its OR successor on the cycle another successor which eventually fails. Then, the AND node cannot be true either, and neither can it serve as a justification for nodes which are its predecessors in ET, including the blocking node and the blocked OR node which are part of the blocking cycle. This leads to the second iteration step.

2. during this step, it is checked whether the hypotheses hold, i.e. every blocked OR node assumed to be true is actually true. This is done by elimination: in order for such a node to stay true at the next iteration, it must be the case that all nodes in one of the cycles induced by one of its outgoing blocking arcs are true (see the explanation at the previous step). All nodes for which this is not the case are eliminated from the set of hypotheses. A new iteration is performed with the new hypotheses set unless all hypotheses were confirmed at this step. This is needed as the false hypotheses might have propagated truth in some cycles which are not good and this process has to be reversed.

- *a final assignment step*: eventually, during the last iteration of the previous step, all hypothesized truth is confirmed, and, as such, it is transferred to the original structure. All nodes which still have truth value $unknown$ are set to $false$.

Thus, eventually, the evaluation procedure assigns to every node of an AND/OR completion structure for p with respect to a program P one of the truth values $true$ or $false$. To simplify the formal exposition of the evaluation procedure, we introduce the following notation:

Algorithm 6.2: The Evaluation Procedure for an AND/OR Completion Structure

```
input  : an AND/OR complete completion structure $AO = \langle ET$, CT, ST, $G$, TYPE, UCS,
         EVAL, SPEC $\rangle$, with $ET = (T, S)$
output: $AO$ is updated such that every node has truth value $true$ or $false$ according to the
         satisfisfiability of its content
1) Initialization: nodes with no successors
$S := \emptyset$;
for every $x \in T$ such that $x$ has no successors in $ET$ do
    if $type(x) = OR$ then
        EVAL$(x) :=$ false;
    end
    else
        EVAL$(x) :=$ true;
    end
    $S := S \cup \{x\}$;
end
$Propagate(AO, S)$;
2) Iterative evaluation procedure
$S := \{x \mid \text{SPEC}(x, y) = bl,$ for some $y \in T\}$;
$switch := false$;
repeat
    $AO' := AO$;
    i) Overestimating truth
    for $x \in S$ (in $AO'$) do
        EVAL$(x) :=$ true;
    end
    $Propagate(AO', S)$;
    ii) Hypothesis check
    for $x \in S$ do
        if there exists no true cycle cy in ET (in $AO'$) such that there exists a node
        $y \in N_{EF}$ with $(x, y) \subseteq cy$ and SPEC$(x, y) = bl$ then
            $S := S - \{x\}$;
            $switch := true$;
        end
    end
until $switch = false$;
3) Transfer of truth to $AO$; final assignment and propagation
$AO := AO'$;
for every $x \in T$ do
    if EVAL$(x) = unknown$ then
        EVAL$(x) :=$ false;
    end
end
```

Definition 43. Let $AO = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$, with $ET = (T, ES)$ be a complete AND/OR completion structure for a CoLP P and $C \in paths_{ET}$ be a cycle in ET. We say that C is a *true cycle* iff for every $x \in C$: $\textsc{eval}(x) = true$.

The evaluation procedure is formally described by Algorithm 6.2.

In the following, we will refer to the result of applying the evaluation procedure to a complete AND/OR completion structure as an *evaluated AND/OR completion structure*. Given such a structure, one can decide whether p is satisfiable by looking at the truth value of its root node:

Definition 44. An evaluated AND/OR completion structure for p with respect to a CoLP P is *successful* iff the truth value of the root node in the structure is true: $\textsc{eval}(\varepsilon) = true$.

Theorem 2. Given a CoLP P and a predicate $p \in upreds(P)$, p is satisfiable with respect to P iff there exists a successful evaluated AND/OR completion structure for checking satisfiability of p with respect to P.

The proofs (in both directions) of the above result are provided in Section 6.2.5.

6.2.4 Termination and Complexity

There are two main factors to consider regarding termination and running time of $\mathcal{A}^{det}_{3,c}$: first, the termination and running time of the expansion procedure, and second, the termination and running time of the evaluation procedure.

6.2.4.1 Computation of Complete AND/OR Completion Structures

Similarly to the non-deterministic case, we show in a first stage that every branch in a complete AND/OR structure[4] has at most an exponential number of nodes in the size of the program, and in a second stage that the number of nodes in the whole completion has at most an exponential number of nodes in the size of the program.

Proposition 26. Every branch in a complete AND/OR-completion structure for a unary predicate p with respect to a CoLP P has at most an exponential number of nodes in the size of P.

Proof. Let $u = |\mathcal{UCS}_P|$ and $n = |upreds(P)|$. We show that any branch in T has at most:

$$2((n2^{n^2+1} - 1)(u - 1) + n2^{n^2+1}) - 1 \text{ nodes.}$$

Assume the opposite: then, there exists a branch with at least:

$$2((n2^{n^2+1} - 1)(u - 1) + n2^{n^2+1}) \text{ nodes,}$$

[4]As the completion has now also backward arcs, whenever we refer to a branch of the completion we mean a branch in the underlying tree of the structure which might have as terminal node some blocked, redundant, or cached OR node.

or in other words, there exists a branch with at least

$$(n2^{n^2+1} - 1)(u - 1) + n2^{n^2+1} \text{ AND nodes.}$$

There is a finite number of AND nodes with different values for the UCS field: at most u, on any branch of the completion structure and in the completion structure itself. As such, there exists a unit completion structure, uc_1, and a sequence of non-terminal AND nodes $x_1, \dots, x_{n2^{n^2+1}}$ belonging to the branch such that: $\text{UCS}(x_i) = uc_1$, for every $1 \leqslant i \leqslant n2^{n^2+1}$. We assume that $x_i <_T x_j$, for every $1 \leqslant i < j \leqslant n2^{n^2+1}$.

For every $1 \leqslant i \leqslant n2^{n^2+1}$, let $r_i^1, \dots, r_i^n$ be the ordered sequence of ranks of unary predicates in $\text{CT}(x_i)$ such that:

- $\{r_i^j \mid 1 \leqslant j \leqslant n\} = \{k \mid p \in \text{CT}(x_1) \wedge rank(p(x_i)) = k\}$;
- $r_i^j \geqslant r_i^{j+1}$, for every $1 \leqslant j < n$;
- if $j = |\{k \mid p \in \text{CT}(x_i) \wedge rank(p(x_i)) = k\}| < n$, then $r_i^m := max\{k \mid p \in \text{CT}(x_i) \wedge rank(p(x_i)) = k\}$, for every $m > j$.

In the following, we will show by induction that:

– for every $1 \leqslant j \leqslant n$: if $x_{n2^{n^2+1}}$ is not redundant, then $rank(x_{j2^{n^2+1}}) > r_1^j$.

Intuitively, this captures the fact that there is no path in G from an atom with rank less or equal to r_1^j to an atom having as argument $x_{j2^{n^2+1}}$.

Base case $j = 1$. If $(x_1, x_{2^{n^2+1}})$ is a blocking pair the claim is obvious. We prove now that $rank(x_{2^{n^2+1}}) > r_1^1$ in the case where $(x_1, x_{2^{n^2+1}})$ is not a blocking pair. We have that:

- $rank(x_i) \geqslant rank(x_1) = r_1^1$, for every $1 \leqslant i \leqslant 2^{n^2+1}$, and
- (x_i, x_k) is neither a blocking nor a redundancy pair, for every $1 \leqslant i < k \leqslant 2^{n^2+1}$.

Assume that $rank(x_{2^{n^2+1}}) = r_1^1$. Then $rank(x_i) = r_1^1$, for every $1 \leqslant i \leqslant 2^{n^2+1}$. Let $l_1 = 2^{n^2+1-n}$. Then, there must be a sequence $x_i^1, \dots, x_{l_1}^1$ of nodes such that:

- for every $1 \leqslant i \leqslant l_1$ there exists some $1 \leqslant k \leqslant 2^{n^2+1}$ such that $x_i^1 = x_k$, and
- there exists a set $S_1 \in 2^{upreds(P)}$ such that $S_1 \neq \emptyset$ and $isp(r_1^1, x_i^1) = S_1$, for every $1 \leqslant i \leqslant l_1$: (given that $|\{isp(r,x), \mid x \in T \text{ and } r \in \mathbb{N}\}| = 2^n$).

Further on, we distinguish between the following situations:

- $r_{l_1}^2 > ||x_1^1||$, or in other words $isp(k, x_{l_1}^1) = \emptyset$, for every $r_1^1 < k \leqslant ||x_1^1||$: this means that there are no paths in the dependency graph running between atoms formed with the first node in the x^1 sequence and the last node in the same sequence, other than those containing atoms with rank r_1^1. As the two nodes have the same rank: r_1^1, and $isp(r_1^1, x_1^1) = isp(r_1^1, x_{l_1}^1)$, it follows that x_1^1 and $x_{l_1}^1$ are redundant – contradiction with the fact that $x_{l_1}^1$ is not the last node on the branch.

- $r^2_{l_1} \leqslant ||x^1_1||$: this means that $r^2_{l_1} = r^2_{x^1_1}$, or there are some paths in the dependency graph running between atoms formed with the first node in the x^1 sequence and the last node in the same sequence with rank greater than r^1_1. Then, similarly to before, we argue that there exists a sequence of nodes $x^2_1, \dots x^2_{l_2}$, with $l_2 = 2^{n^2+1-2n}$ such that:

  - for every $1 \leqslant i \leqslant l_2$ there exists some $1 \leqslant k \leqslant l_1$ such that $x^2_i = x^1_k$, and
  - there exists a set $S_2 \in \mathit{upreds}(P)$ such that $S_2 \neq \emptyset$ and $\mathit{isp}(r^2_1, x^2_i) = S_2$, for every $1 \leqslant i \leqslant l_2$.

  Continuing in the same vein we obtain that either x^2_1 and $x^2_{l_2}$ are redundant or there exists a sequence x^3 with similar properties as before, etc. As the number of predicates in the content of a node is bounded by n, it follows that we can construct at most n such sequences and for the last one, $1 \leqslant k \leqslant n$, the nodes x^k_1 and x^k_l, where $l_k = 2^{n^2-kn+1}$, form a redundancy pair.

Thus, based on the assumption that $\mathit{rank}(x_{2^{n^2+1}}) = r^1_1$, it follows that there must be some redundant node in the sequence $x_1, \dots, x_{2^{n^2+1}}$. This is in contradiction with the fact that the complete AND/OR completion structure contains no redundant AND node (every such node is removed as part of the redundancy rule). Thus, the assumption was false and $\mathit{rank}(x_{2^{n^2+1}}) > r^1_1$.

Induction step: This consists in proving that $\mathit{rank}(x_{(j+1)2^{n^2+1}}) > r^{j+1}_1$, when it is known that $\mathit{rank}(x_{j(2^{n^2+1})}) > r^j_1$, for some $0 < j < n$.

Assume that it is not the case that $\mathit{rank}(x_{(j+1)2^{n^2+1}}) > r^{j+1}_1$. Then, for every $2^{n^2+1} < i \leqslant 2^{n^2+2}$, it is the case that $\mathit{rank}(x_i) = r^2_1$. As previously, it is possible to construct sequences x^1, x^2, …, and show that eventually some redundancy pair is encountered. But this is in contradiction with the fact that there are no redundant AND nodes in the structure. Thus, the original claim is true: for every $1 \leqslant j \leqslant n$, if $x_{n2^{n^2+1}}$ is not redundant, then $\mathit{rank}(x_{j2^{n^2+1}}) > r^j_1$.

From here on, the proof follows the line of the argument used to show the similar claim for the non-deterministic case: as $\mathit{rank}(x_{n2^{n^2+1}}) > r^n_1$, it follows that the set of oldest paths in G traversing $x_{n2^{n^2+1}}$ started at a node below x_1, and thus, there are no paths in G running between x_1 and $x_{n2^{n^2+1}}$. As $\textsc{ucs}(x_1) = \textsc{ucs}(x_{n2^{n^2+1}})$, this implies that $(x_1, x_{n2^{n^2+1}})$ is a blocking pair and thus $x_{n2^{n^2+1}}$ is a blocked OR node. But, again, there are no explicit blocked AND nodes in a complete AND/OR completion structure. Thus, the original assumption was false, and there are at most:

$$2((n2^{n^2+1} - 1)(u - 1) + n2^{n^2+1}) - 1 \text{ nodes}$$

on every branch, where $n = |\mathit{upreds}(P)|$, and $u = |\mathcal{UCS}_P|$. □

Proposition 27. A complete AND/OR-completion structure for a unary predicate p and a CoLP P has at most an exponential number of nodes in the size of P.

Proof.

Similarly to the non-deterministic case, we associate with every AND node in a complete AND/OR completion structure $x \in T$, a function

$$f_x : \mathit{upreds}(P) \to \{r \mid 0 \leqslant r \leqslant 2((n2^{n^2+1} - 1)(u-1) + n2^{n^2+1}) - 1\},$$

where $n = |\mathit{upreds}(P)|$ and $u = |\mathcal{UCS}_P|$. The function is defined as follows:

$$f_x(p) = \begin{cases} 0, & \text{if } p \notin \textsc{ct}(x) \\ \mathit{rank}(p(x)), & \text{if } p \in \textsc{ct}(x). \end{cases}$$

The number of such functions is:

$$\begin{aligned} \mathcal{F} &= |2^{\mathit{upreds}(P)} \times \{r \mid 0 \leqslant r \leqslant 2((n2^{n^2+1} - 1)(u-1) + n2^{n^2+1}) - 1\}|^{|\mathit{upredsP}|} \\ &= (2^n(2((n2^{n^2+1} - 1)(u-1) + n2^{n^2+1}) - 1)^n, \end{aligned}$$

which is exponential in the size of P. As previously, one can show that every two AND nodes on different branches in T, that have associated identical functions and which *have been expanded using the same unit completion structure* form a caching pair. The reasoning from this point on follows the same line as the reasoning for the non-deterministic case. □

As computing the set of unit completion structures and finding a match between a unit completion structure against a node in T can be performed in exponential time (this in turn, has to be performed at most an exponential number of times - the size of the AND/OR complete structure), and checking the blocking, redundancy, and caching conditions can be done in time linear/polynomial in the size of the longest branch of the structure it follows that:

Proposition 28. Given an initial AND/OR-completion structure for a unary predicate p and a CoLP P AO, a complete AND/OR completion structure can always be evolved from AO in the worst case in time exponential in the size of P.

6.2.4.2 Complexity Analysis for the Evaluation Procedure

We show that the evaluation procedure for a complete AND/OR completion structure $AO = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ terminates and runs in the worst-case in exponential time by looking at the different stages of the procedure:

- *Termination and complexity of the initialization step*: We show that every node in AO changes its truth value at most once during this step. Assume the opposite: then there exists a node $x_1 \in T$ which changes its truth value at least twice as a result of the call to the propagation procedure. This can only by virtue of one of its successors changing values twice as well. Every terminal node changes values at most once from $\mathit{unknown}$ to true or false. As at this stage there cannot be circular motivations of truth values, such there should be an infinite chain of distinct nodes in which every node changes its value twice by virtue of its successor in the chain. This is in contradiction with the fact that T has a finite number of nodes. Thus, the initial assumption was false: each node changes its value at most once. From Proposition 19 it follows that this step takes in the worst case exponential time.

- *Termination and complexity of an iteration step of the alternating iterative procedure*: during this step nodes can change their truth values only from $unknown$ to $true$. Nodes which have truth value $false$ will not change their value as it has been calculated based on the truth value $false$ of some terminal OR nodes during the initialization step. The value of these latter nodes is not changed during this step as only some nodes with truth value $unknown$ are switched to $true$. Also, once a node becomes $true$ its truth value will not change: all nodes in S (either added initially or during the propagation procedure) have truth value $true$ and the only nodes which change their value are doing so based on the fact that some of their successors belong to S. Thus, again, each node changes its value at most once.

  From Proposition 19 it follows that this step takes in the worst case exponential time.

- *Convergence of the alternating iterative procedure*: At every subsequent iteration, the number of nodes in the hypothesized set decreases. Thus, the maximum number of iterations coincides with the maximum number of blocked OR nodes in a complete AND/OR structure, which is exponential in the size of the program.

- *Final assignment*: trivial.

As a result of the analysis above one can conclude that:

Proposition 29. The evaluation procedure for a given complete AND/OR completion structure runs in the worst case in exponential time.

From Proposition 28 and Proposition 29 it follows that:

Proposition 30. $\mathcal{A}_{3,c}^{det}$ runs in the worst case in deterministic exponential time.

Thus, $\mathcal{A}_{3,c}^{det}$ provides an effective worst-case optimal procedure for reasoning with CoLPs.

6.2.5 Soundness and Completeness

This section shows that $\mathcal{A}_{3,c}^{det}$ is sound and complete. We start with some notations and lemmas which will be useful for showing both results.

First, we introduce the notion of traversal of an AND/OR completion structure which consists in a projection of the structure to another AND/OR structure such that the trees corresponding to the two structures have identical roots, for every OR node in the traversal, exactly one successor is carried over from the original structure (in case such a successor exists), and for every AND node in the traversal, all successors are carried over from the original structure. While an AND/OR completion structure is an exhaustive representation of all possible models for a CoLP P which satisfy a certain unary predicate p, a traversal is a representation of a tentative such model (where at every choice point, exactly one choice is considered).

Definition 45. A *traversal* of an AND/OR completion structure for a CoLP P, $AO = \langle ET$, $\textsc{ct}$, $\textsc{st}$, G, $\textsc{type}$, $\textsc{ucs}$, $\textsc{eval}$, $\textsc{spec}\ \rangle$, with $ET = (T, ES)$ is an AND/OR completion structure $Tr = \langle ET'$, $\textsc{ct}'$, $\textsc{st}'$, G', $\textsc{type}'$, $\textsc{ucs}'$, $\textsc{eval}'$, $\textsc{spec}'\ \rangle$ defined as follows:

- $ET' := (T', ES')$, where T' has the same root ε as T,
- for every $x \in T'$ it holds that:
  - $x \in T$,
  - $\textsc{ct}'(x) := \textsc{ct}(x)$,
  - $\textsc{st}'(x) := \textsc{st}(x)$,
  - $\textsc{type}'(x) := \textsc{type}(x)$,
  - $\textsc{ucs}'(x) := \textsc{ucs}(x)$,
  - $\textsc{eval}'(x) := \textsc{eval}(x)$,
  - if $\textsc{type}'(x) = OR$ and some $y \in T$ exists such that $(x, y) \in A_{ET}$, then there exists a unique $z \in T'$ such that $(x, z) \in A_{ET}$ and $(x, z) \in A_{ET'}$;
  - if $\textsc{type}'(x) = AND$, then for every $y \in T$ such that $(x, y) \in A_{ET}$, it is the case that $y \in T'$ and $(x, y) \in A_{ET'}$;
- for every $(x, y) \in A_{ET'}$ it holds that: $\textsc{spec}'(x, y) := \textsc{spec}(x, y)$;
- $G' := \langle V', A' \rangle$ with:
  - $V' := V \cap \mathcal{B}_{P_T}$;
  - $A' := A \cap (V' \times V')$;

The following propositions follow directly from the definitions of an evaluated AND/OR completion structure and of a traversal.

Proposition 31. An evaluated AND/OR completion structure is successful iff it admits a successful traversal.

Proposition 32. A traversal of an evaluated AND/OR completion structure is successful iff all nodes in the traversal have truth value $true$.

While the previous proposition characterizes a successful traversal syntactically, we move next towards a more semantic characterization of such traversals, which as already anticipated, are meant to be representations of actual models. We first provide a formal definition for the notion of good cycle, which was mentioned in the informal description of the evaluation procedure.

Definition 46. Let $AO = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ be an AND/OR completion structure and let $cy \in paths_{ET}$ be a cycle in ET. We say that cy is a *good cycle* iff there exists no cycle $Cy \in paths_G$ with $argpath(Cy) = cy$.

Next we introduce the notion of 'good' traversal which, as we will see later, is in a one-to-one correspondence with the notion of a successful traversal.

Definition 47. A traversal $Tr = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ is good iff every cycle in ET is a good cycle and there exists no node $x \in T$ such that $\textsc{type}(x) = OR$ and x has no successors.

As anticipated in Section 6.2.3, the notion of good cycles is tightly related to the presence of a blocking arc in such cycles. Depending on the presence of such an arc we distinguish between blocking and non-blocking cycles:

Definition 48. Let $AO = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$, with $ET = (T, ES)$ be an AND/OR completion structure and $cy \in paths_{ET}$ be a cycle in ET.

We say that cy is a *blocking cycle* iff there exists a blocking arc $(x, y) \in ES$, $\textsc{spec}(x, y) = bl$, which is part of the cycle: $(x, y) \subset cy$.

Otherwise, we say that cy is a *non-blocking cycle*.

In order to prove the connection between good cycles and blocking cycles, we start with some helping lemmas:

Lemma 23. Let $AO = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ be a complete AND/OR-completion structure, with $ET = (T, ES)$, and let x and y be some nodes in T such that (x, y) is a redundancy or a caching arc in AO.

Then, for every node $t \in T$ such that $||t|| \leqslant_T lca_T(x, y)$ and unary predicate $p \in upreds(P)$, it holds that: there exists a path Pt_1 in G from some $q(t)$ to $p(y)$ with $argpath(Pt_1) = path_T(t, y)$ iff there exists a path Pt_2 in G from some $r(t)$ to $p(y)$ with $argpath(Pt_2) = path_T(t, x)^\wedge(x, y)$.

Proof. From the construction of an AND/OR completion structure. See Section 6.2.2.2 and Section 6.2.2.3. □

Lemma 24. Let $CS = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ be an AND/OR-completion structure, let $x <_T y$ be some nodes in ET, and let pt be a path in $paths_{ET}(x, y)$ such that:

- for every $z \in pt$: $z \in T_x$, and
- there exists no blocking arc (z, t) such that $(z, t) \subseteq pt$.

Then, there exists some path $Pt \in paths_G(q(x), r(y))$ such that $argpath(Pt) = pt$ iff there exists some path $Pt' \in paths_G(s(x), r(y))$ with $argpath(Pt') = path_T(x, y)$, where $q, r, s \in upreds(P)$.

Proof. For the purpose of this lemma let $\sharp(pt)$ be the number of special arcs, i.e. redundancy and caching arcs, in some path $pt \in paths_{EF}$ which does not contain any blocking arc. We will show that the claim in the lemma holds by induction on $\sharp(pt)$.

Induction base: $\sharp(pt) = 0$, that is $pt = path_T(x, y)$. The claim trivially holds.

Induction step: Assume that for all $pt \in paths_{ET}$ which contain no blocking arc and for which $\sharp(pt) = n$, for some $n \in \mathbb{N}_{>0}$, the claim holds. Let $pt \in paths_{ET}$ be a path which contains no blocking arcs with $\sharp(pt) = n + 1$. Then there must be some nodes z, t, with (z, t) being a redundancy or caching arc, and some paths $pt_1 \in paths_T(x, z)$, and $pt_2 \in paths_{ET}(t, y)$ (the paths are depicted in Figure 6.2 using dashed arcs), such that:

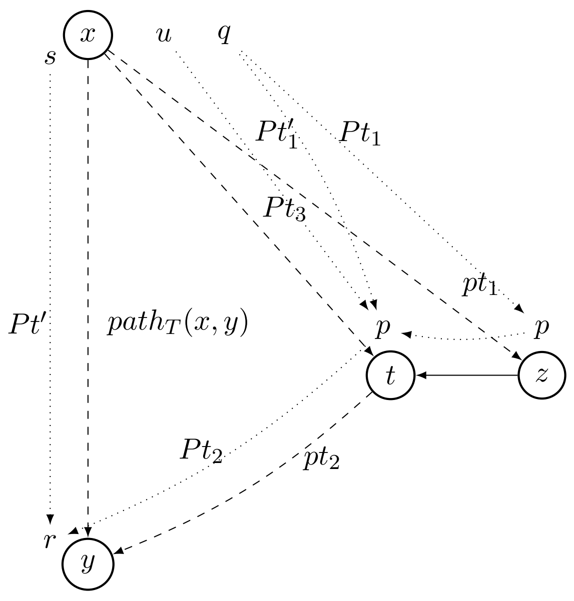


Figure 6.2: A path in ET from x to y containing a redundancy or a caching arc (z,t) and corresponding paths in the dependency graph G

- $pt = pt_1\hat{}(z,t)\hat{}pt_2$, and
- $\sharp(pt_2) = n$.

We next show that the claim in the lemma holds for pt, that is, that there is there is some path $Pt \in paths_G(q(x), r(y))$ such that $argpath(Pt) = pt$ iff there exists some path $Pt' \in paths_G(s(x), r(y))$ with $argpath(Pt') = path_T(x,y)$. Figure 6.2 can again be used as a reference for the proof of the claim in both directions: it depicts paths in the dependency graph G using dotted arcs.

"⇒": Assume that there exists a path $Pt \in paths_G(q(x), r(y))$ with $argpath(Pt) = pt$. Then there must be some paths $Pt_1 \in paths_G(q(x), p(z))$, and $Pt_2 \in paths_G(p(t), r(y))$, with $argpath(Pt_1) = pt_1$, $argpath(Pt_2) = pt_2$, and some arc $(p(z), p(t)) \in A_G$ (the only type of arcs in G between atoms having as arguments z, and t, where (z,t) is a special arc in EF), such that: $Pt = Pt_1 \;\hat{}\; (p(z), p(t)) \;\hat{}Pt_2$.

We notice that $argpath(Pt_1\hat{}(p(z), p(t))) = path_T(x,z)\hat{}(z,t)$. Then, from Lemma 23 it follows that there must be a path $Pt_3 \in paths_G(u(x), p(t))$, with $rank(u(x)) = r$ and $argpath(Pt_3) = path_T(x,t)$. By concatenation of Pt_3 with Pt_2, one obtains a path $Pt_4 \in paths_G(u(x), r(y))$ with $argpath(Pt_4) = path_T(x,t)\hat{}pt_2$. As $\sharp(argpath(Pt_4)) = n$, from the induction hypothesis it follows that there exists a path $Pt' \in paths_G(s(x), r(y))$ such that $argpath(Pt') = path_T(x,y)$, which is exactly the original claim.

"⇐": Assume that there exists a path $Pt' \in paths_G(s(x), r(y))$ with $argpath(Pt') = path_T(x,y)$. Let $pt_4 = path_T(x,t)\hat{}pt_2$. As $\sharp(pt_2) = n$, it follows that also $\sharp(pt_4) = n$. Then, from the induction hypothesis for pt_4 and from the existence of Pt', it follows that

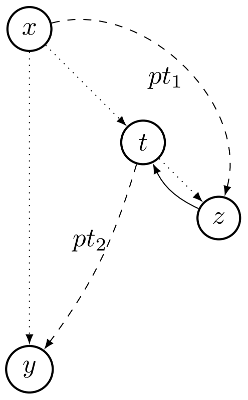


Figure 6.3: A path in ET from x to y which contains a blocking arc (z, t)

there must be some path $Pt_4 \in paths_G(u(x), r(y))$ such that $argpath(Pt_4) = pt_4$. Subsequently, there must be some paths $Pt_3 \in paths_G(u(x), p(t))$, $Pt_2 \in paths_G(p(t), r(y))$ such that $argpath(Pt_3) = path_T(x, t)$ and $argpath(Pt_2) = pt_2$. Due to the existence of $Pt3$ from Lemma 23 it follows that there must be some path $Pt'_1 \in paths_G(q(x), p(y))$ with $argpath(Pt'_1) = path_T(x, z)\hat{}(z, t)$. Then, $Pt'_1\hat{}Pt_2 \in paths_G(u(x), r(t))$ and $\sharp(Pt'_1\hat{}Pt_2) = n + 1$.

Lemma 25. Let $AO = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be an AND/OR-completion structure, and let (x, y) be a blocking arc in AO. Then, $connpr_{G|T}(y, x) = \emptyset$.

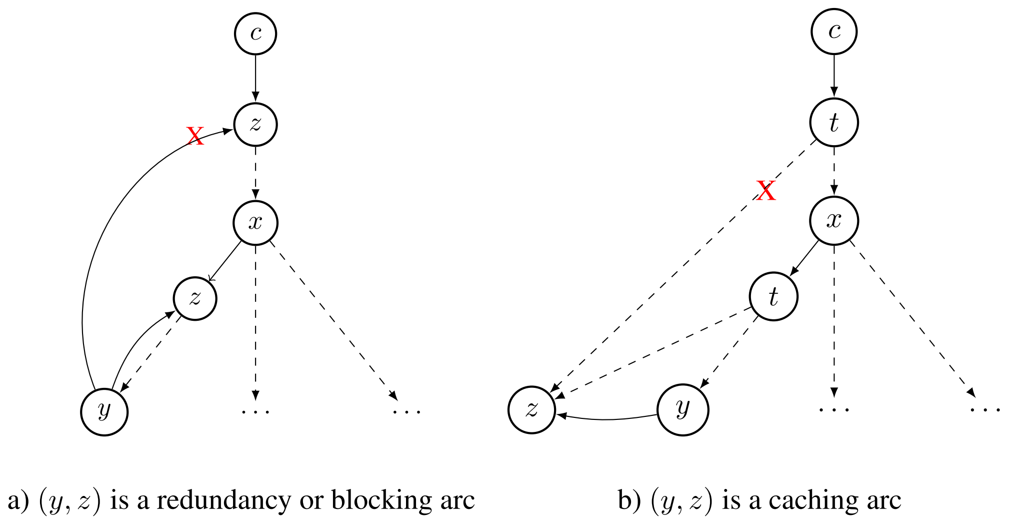
Proof. From the construction of an AND/OR completion structure. See Section 6.2.2.1. □

Lemma 26. Let $CS = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be an AND/OR-completion structure, let $x <_T y$ be some nodes in ET, and let pt be a path in $paths_{ET}(x, y)$ such that:

- for every $z \in pt$: $z \in T_x$, and
- there exists $t \in T$ such that SPEC$(z, t) = bl$ ((z, t) is a blocking arc) and $(z, t) \subseteq pt$.

Then, there exists no path $Pt \in paths_G$ with $argpath(Pt) = pt$.

Proof. Let $pt = pt_1\hat{}(z, t)\hat{}pt_2$, where (z, t) is the first blocking arc in pt: for every $(u, v) \subseteq pt_1$: SPEC$(u, v) \neq bl$. Then, $t <_T z$, and $x <_T y$, thus $t \in path_T(x, z)$ (Figure 6.3). As z is a blocked OR node, from Lemma 25 it follows that $connpr_{G|T}(t, z) = \emptyset$, thus also $connpr_{G|T}(x, z) = \emptyset$. As pt_1 contains no blocking arcs, one can apply Lemma 24: there exists some path $Pt_1 \in paths_G(q(x), r(z))$, with $q, r \in upreds(P)$ such that $argpath(Pt_1) = pt_1$ iff there exists some path $Pt_2 \in paths_G(s(x), r(z))$ with $argpath(Pt_2) = path_T(x, z)$, with $rank(s(x)) = rank(q(x))$. But $connpr_{G|T}(x, z) = \emptyset$, and thus, there exists no such paths Pt_1 or Pt_2: thus, there exists also no path $Pt \in paths_G$ with $argpath(Pt) = pt$. □



c
z
X
x
z
y
...
...
a) (y, z) is a redundancy or blocking arc
c
t
X
x
t
z
y
...
...
b) (y, z) is a caching arc

Figure 6.4: Every cycle has a 'top' node x such that every node in the cycle is part of T_x

Lemma 27. Let $CS = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be a complete AND/OR-completion structure, and let cy be a cycle in ET. Then, there exists a node x such that for every $y \in cy$: $y \in T_x$. We call x the *top node* of cy and denote it with $top(cy)$.

Proof. Let $x \in cy$ be such that there is no node $y \in cy$ with $y <_T x$ or $right_T(y, x)$ (x is the top node on the 'left'-most branch from which there are some nodes which are part of the cycle). We show inductively that only nodes in T_x can be reached. The induction partial order is given by the distance $dist(x, y)$ from x to a node y in cy, defined as $dist(x, y) = length(pt)$, where pt is the path from x to y in cy: $pt \in paths_{EF}(x, y)$ and $pt \subseteq cy$.

Induction base: Let $y \in cy$ be such that $dist(x, y) = 1$. Then, $(x, y) \in A_{ET}$. As there exists no node $z \in cy$ such that $z <_T x$ or $right_T(z, x)$, any arc (x, y) with $y \in cy$ is a regular arc: $(x, y) \in A_T$, thus $y \in T_x$.

Induction step: Assume all nodes y for which $dist(x, y) = n$, are in T_x (IH). Let z be such that $dist(x, z) = n + 1$ and y be such that $(y, z) \subset cy$. Then, $dist(x, y) = n$, and thus $y \in T_x$. Depending on the type of (y, z) we distinguish between:

- SPEC(y, z) = *bl* or SPEC(y, z) = *re*: then, $z \in paths_T(c, y)$. But $y \in T$, thus $x \in paths_T(c, y)$, and there is no $t <_T x$. Then, $z \geqslant_T x$: $z \in T_x$ (Figure 6.4 a)).

- SPEC(y, z) = *ch*: then, $right_T(y, z)$. Let $t = lca_T(y, z)$. If $t <_T x$, it follows that $right_T(x, z)$ – in contradiction with the original assumptions on x. Thus: $t \geqslant_T x$, and $z >_T x$, or in other words $z \in T_x$ (Figure 6.4 b)).

- SPEC(y, z) is undefined, or in other words (y, z) is a regular arc in T. Then, from the fact that $y \in T_x$, it follows straightforwardly that $z \in T_x$.

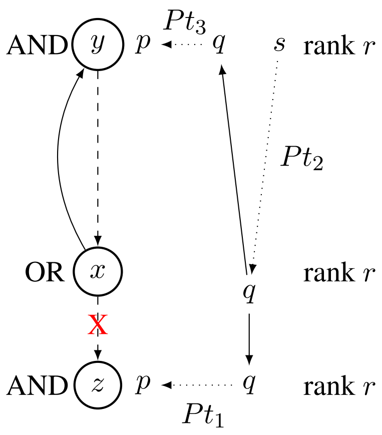


Figure 6.5: If (x, y) is a redundancy arc and $p \in \text{CT}(y)$ there is a path in G from $q(x)$ to $p(y)$

Thus, for every node in $y \in cy$, it holds that $y \in T_x$, and x, as chosen above is $top(cy)$. □

Lemma 28. Let $AO = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be an AND/OR-completion structure, and let (x, y) be a redundancy arc in AO. Then, for every unary predicate $p \in \text{CT}(y)$, there is some unary predicate $q \in \text{CT}(x)$ such that $(q(x), p(y)) \in conn_G$ and $rank(q(x)) = rank(p(y))$.

Proof. As (x, y) is a redundancy arc in AO, there must have been some AND node z which was identified as being redundant while constructing AO having y is a redundancy witness (Figure 6.5). According to the redundancy rule (Section 6.2.2.2) the node was subsequently deleted and the redundancy arc (x, y) has been created. In the following we consider AO as if it still contains the redundant node z. From the redundancy rule, it follows that for every $s \in \text{CT}(y)$, it is the case that $s \in \text{CT}(z)$ and $rank(s(y)) = rank(s(x))$.

Let $r = rank(p(y))$. Then, there must be some path in G of rank r which contains $p(z)$. Let s and q be the intersection of this path with y and x, respectively. Also let Pt be the subpath of this path which ranges from $s(y)$ to $p(z)$. From the definition of the $AOExpand$ operation and from the definition of the redundancy rule we know that there must be some arcs $(q(x), q(z))$ and $(q(x), q(y))$ in G, respectively. Furthermore, there is no arc of type $(q(x), s(z))$ where s is different from q. Thus, there must be some paths $Pt_1 \in paths_G(s(y), q(x))$, and $Pt_2 \in paths_G(q(z), p(z))$ such that $Pt = Pt_1 \hat{} (q(x), q(z)) \hat{} Pt_2$.

From the fact that z and y were expanded using the same UCS and the existence of Pt_2, it follows that there must be a path $Pt_3 \in paths_G(q(y), p(y))$. Then, by concatenation of arc $(q(x), q(y))$ and of path Pt_3 we obtain a path $Pt_4 \in paths_G(q(x), p(y))$. □

We next show that every non-blocking cycle in ET has an overlying cycle in G and every blocking cycle in ET has no overlying cycle in G.

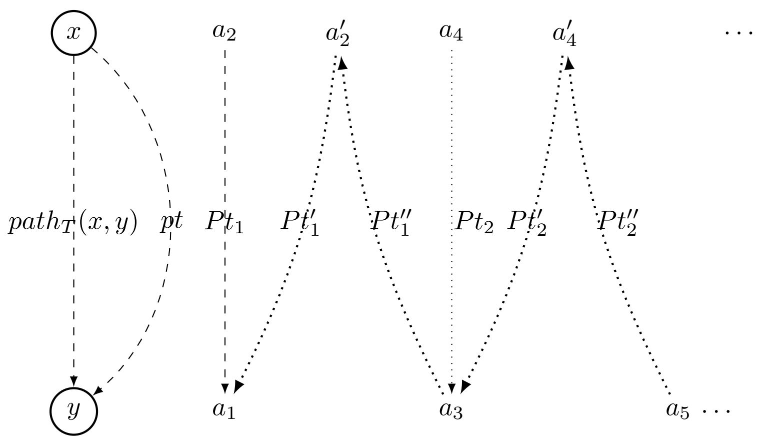


Figure 6.6: Paths in G in a non-blocking cycle

Lemma 29. Let $CS = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be a complete AND/OR-completion structure, and let cy be a non-blocking cycle in ET. Then, there exists a cycle $Cy \in paths_G$ such that $argpath(Cy) = cy$.

Proof. As cy is a non-blocking cycle, according to Lemma 27, there exists a node $x \in cy$ such that for every $y \in cy$, $y \in T_x$. Let y be the direct predecessor of x in the cycle: $(y, x) \in cy$ and $pt \in paths_{ET}(x, y)$ such that $cy = pt\hat{}(y, x)$. Then, as $y \in T_x$ and cy contains no blocking arc, (y, x) is a redundant arc.

Let $r_1 = rank(x)$. Then, there must be some path $Pt_1 \in paths_G(a_2(x), a_1(y))$, with $a_1, a_2 \in upreds(P)$, $argpath(Pt_1) = path_T(x, y)$, and $rank(a_2(x)) = rank(a_1(y)) = r_1$. One can apply Lemma 24 for $Pt_1/path_T(x, y)$ and pt: there must be some path $Pt_1' \in paths_G(a_2'(x), a_1(y))$ with $argpath(Pt_1') = pt$ and $a_2' \in upreds(P)$. Let $r_2 = rank(a_2'(x))$.

Then, according to Lemma 28 there exists a predicate $a_3 \in isp(r_2, y)$ and a path $Pt_1'' \in paths_G(a_3(y), a_2'(x))$. As $a_3 \in isp(r_2, y)$ and $r_2 \leqslant ||x||$, it follows that there must be a path $Pt_2 \in paths_G(a_4(x), a_3(y))$, with $a_4 \in upreds(P)$, and $argpath(Pt_2) = path_T(x, y)$. Again, by applying Lemma 24 one obtains, that there must be a path $Pt_2' \in paths_G(a_4'(x), a_3(y))$ with $argpath(Pt_2') = pt$, and $a_4' \in upreds(P)$.

Following the same line of reasoning one obtains that there must be a sequence of paths:

$$Pt_i' \in paths_G(a_{2i}'(x), a_{2i-1}(y)) \text{ with } argpath(Pt_i') = pt, \text{ for every } i \geqslant 1 \tag{6.1}$$

and a sequence of paths:

$$Pt_i'' \in paths_G(a_{2i+1}(y), a_{2i}'(x)) \text{ with } argpath(Pt_i'') = (y, x), \text{ for every } i \geqslant 1 \tag{6.2}$$

By combining (6.1) and (6.2) (Figure 6.6), we obtain that for every $k > 1$ and for every $1 \leqslant j < k$, there exists a path $Pt_{j,k} \in paths_G(a_{2k+1}(x), a_{2j-1}(y))$ such that:

$$Pt_{j,k} = Pt_k''\hat{}Pt_k'\hat{}\ldots\hat{}Pt_j''\hat{}Pt_j'.$$

But, as $|upreds(P)|$ is finite, there must be two indices $1 \leqslant j \leqslant k$ such that $a_{2j-1} = a_{2k+1}$: then, there exists an arc $(a_{2j-1}(y), a_{2k+1}(x)) \in A_G$ and a path $Cy = Pt_{j,k}\hat{}(a_{2j-1}(y), a_{2k+1}(x))$ is a cycle in G with $argpath(Cy) = pt\hat{}(y, x) = cy$.

Lemma 30. Let $CS = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be a complete AND/OR-completion structure, and let cy be a blocking cycle in ET. Then, there exists no cycle $Cy \in paths_G$ such that $argpath(Cy) = cy$.

Proof.
Let (z, t) be a blocking arc in cy such that $(z, t) \subseteq cy$, $\text{SPEC}(z, t) = bl$ and $x = top(cy)$.

Then $cy = pt_1\hat{}pt_2$, with $pt_1 \in paths_{ET}(x, t)$ and one can apply Lemma 26 for the path pt_1: there is no path $Pt_1 \in paths_G$ such that $argpath(Pt_1) = pt_1$, thus there is no path $Pt \in paths_G$ such that $argpath(Pt) = cy$ (and subsequently no cycle). □

From Lemma 30, Lemma 29, and Definition 46, it is possible to conclude that:

Proposition 33. Let $CS = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be an evaluated AND/OR-completion structure. Then, a cycle cy in ET is good iff cy is a blocking cycle.

Finally, we state and prove the following result which underpins the evaluation procedure:

Proposition 34. An evaluated AND/OR completion structure AO for a CoLP P and unary predicate p is successful iff it admits a good traversal.

Proof. Let $AO = \langle ET$, CT, ST, G, TYPE, UCS, EVAL, SPEC $\rangle$ be an evaluated AND/OR completion structure.

"$\Rightarrow$": Assume AO is successful. Before proceeding to the construction of a good traversal, we observe that every traversal $Tr = \langle ET'$, CT$'$, ST$'$, G', TYPE$'$, UCS$'$, EVAL$'$, SPEC$'$ $\rangle$ of AO is completely characterized by ET' and AO, i.e. it can always be reconstructed from these two data structures. In the following, we will construct explicitly only the extended tree part, ET', of a good traversal, Tr, of AO. We will also use the function ST$'$ to keep track of nodes which have been already expanded in the traversal in construction.

The procedure initializes Tr with the root of the tree T of the AND/OR successful completion structure. Progressively, it considers unexpanded OR nodes in T' and, using AO as a guidance, for every such node picks up an AND successor which is true in ET.

Assume that an unexpanded OR node $x \in T'$ is going to be expanded by reusing an AND successor y of x in ET. Then, the operation of updating ET' by addition of the new AND node y as a successor to x is denoted with $trExpand(ET', x, y)$ and is formally defined as follows:

- $\text{ST}'(x) = exp$;
- $A_{ET'} := A_{ET'} \cup \{(x, y)\}$;
- if $y \notin T'$ then:
  - $T' := T' \cup \{y\}$;
  - for every z such that $(y, z) \in A_{ET}$:

- $*$ $T' := T' \cup \{z\}$;
- $*$ $A_{ET'} := A_{ET'} \cup \{(y, z)\}$;
- $*$ $\textsc{st}'(z) = unexp$;

A strategy is needed for the selection of the one and only successor for each OR node in the traversal which ensures that no bad cycles are introduced in the construction. To this end, every true node in AO is annotated with a natural number, signifying the distance from the node to terminal 'good' nodes, i.e. AND nodes with no successors or blocked/proxy constant OR nodes in true good cycles.

The distance is defined as follows:

$$d(x) := \begin{cases} 0, & \text{if } x \text{ is a terminal AND node or} \\ & \text{a blocked OR node in a true cycle} \\ min\{d(y)+1 \mid (x,y) \in A_{ET} \text{ and } d(y) \text{ is defined }\}, & \text{if } x \text{ is an OR node, other than above} \\ max\{d(y)+1 \mid (x,y) \in A_{ET} \text{ and } d(y) \text{ is defined }\}, & \text{if } x \text{ is an AND node, other than above.} \end{cases}$$

To see that the distance is well-defined, consider the propagation procedure described by Algorithm 6.1 and the evaluation procedure described by Algorithm 6.2. A node is true either by virtue of being a terminal AND node, a blocked OR node in a true cycle, or as a result of one of the calls to $Propagate$ in the initial step of the evaluation procedure and in the last iteration of the same procedure. Assume that there is a counter c which initially is set to 0 and which runs through the two propagation calls. Also, let $e : T \to \mathbb{N}$ be a function such that every time a new node is assigned the truth value $true$ as a result of one of the two calls, the counter c is incremented and $e(x) := c$. Additionally, $e(x) = 0$, for every node x which is a terminal AND node or a blocked OR node in a true cycle.

As the two calls to the propagation procedure always terminate, as discussed in Section 6.2.4, it is clear that $e(x)$ is defined and finite, for every node $x \in T$ for which $\textsc{eval}(x) = true$.

We show by induction on $e(x)$ that e is an over-estimation of d: for every node $x \in T$ for which $\textsc{eval}(x) = true$, $d(x) \leqslant e(x)$.

Induction base: $e(x) = 0$. Then, $d(x) = 0$.

Induction hypothesis: Assume $d(x) \leqslant e(x)$, for every x such that $e(x) \leqslant k$, for a fixed k. We show that: $d(x) \leqslant e(x)$, for every x such that $e(x) = k + 1$.

Let x be such that $e(x) = k + 1$. We distinguish between:

- x is an OR node: then, there must be a successor y of x in ET such that $e(y) \leqslant k$. From the definition of d: $d(x) \leqslant e(y) + 1 \leqslant k + 1 = e(x)$.

- x is an AND node: then, for every successor $y_1, \ldots, y_n$ of x in ET: $e(y_i) \leqslant k$. From the definition of d: $d(x) \leqslant max_{1 \leqslant i \leqslant n}(e(y_i)) + 1 \leqslant k + 1 = e(x)$.

Thus, d, being an under-estimation of e, is also defined and finite, for every node $x \in T$ such that $\textsc{eval}(x) = true$. In the following we will use d in the construction of the traversal Tr. For every OR node $x \in T'$ such that $\textsc{st}'(x) = unexp$ do one of the following:

- if x is an OR blocked node in a true good cycle cy in ET ($d(x) = 0$), for which there exists a node $y \in T$ such that $(x, y) \in cy$ and $\textsc{spec}(x, y) = bl$, make y the successor of x in ET': $trExpand(ET', x, y)$.

- if x is an ordinary node ($d(x) > 0$) pick up a successor node y of x in ET for which $d(y) = d(x) - 1$ (by construction of d such a successor must exist) and use it as a successor of x in ET': $trExpand(ET', x, y)$.

It is clear that the procedure described above terminates and constructs a traversal: for every OR node for which there exists a successor in ET, there will be exactly one successor in ET'. Also, if a node is re-encountered it will not be expanded again. It remains to be shown that the traversal is a good one, i.e. all (potential) cycles in ET' are good ones. Assume the opposite: that, there exists a cycle cy which is not good. Then, by construction of d, for every two nodes $x, y \in T'$, if $(x, y) \in A_{ET'}$ and $d(x) \neq 0$, $d(x) > d(y)$. It follows that every cycle should contain a node x such that $d(x) = 0$. But then, x is a blocked OR node, and its only successor in ET is the node $y \in T$ such that $\textsc{spec}(x, y) = bl$, and thus cy is a blocked cycle. But then, cy is good.

" $\Leftarrow''$: Assume that there exists a good traversal $GT = \langle ET'$, $\textsc{ct}'$, $\textsc{st}'$, G', $\textsc{type}'$, $\textsc{ucs}'$, $\textsc{eval}'$, $\textsc{spec}' \rangle$ of AO. Then, all the cycles in GT are good and there are no terminal OR nodes, i.e. OR nodes with no successors.

We show that GT is a successful traversal by induction: in particular, we show that the truth value of every node x in the traversal is $true$ by induction on $d(x)$, which is the distance from x to some terminal AND node or blocked OR node in T' (in the traversal!), formally defined as follows:

$$d(x) = \begin{cases} 0, & \text{if } x \text{ is a blocked OR node in a good cycle in } ET' \\ & \text{or a terminal AND node in } T' \\ max\{1 + d(y) \mid (x, y) \in A_{ET'}\} & \text{otherwise.} \end{cases}$$

As for the distance used for the counter-proof, we show first that $d(x)$ is well-defined: in this case this means showing that $d(x)$ is finite, for every $x \in T'$. Assume the opposite. Then there must be an infinite length path in ET' which contains x such that for every node y on the path: $d(x) > 0$. Such a path can only be a cycle. All cycles in ET' are good, thus, according to Proposition 33 they are also blocking cycles. As such, any such cycle contains a blocked OR node y such that $d(y) = 0$ – contradiction.

We next show by induction on $d(x)$, that the truth value of every node $x \in T'$ is true.

Base case: if $d(x) = 0$, then we distinguish between:

- x is a terminal AND node in T/T' (a node with no successors): then, it is set to $true$ in the initialization step of the evaluation procedure.

- x is a blocked OR node in a good cycle in ET'. Then x is set to $true$ in the first iteration of the evaluation procedure.

Induction step: Assume $\textsc{eval}(x) = true$, for all nodes $x \in T'$ such that $d(x) \leqslant k$, for some $k \in \mathbb{N}$, and let $y \in T'$ be such that $d(y) = k + 1$. For every successor x of y it is the case that $d(x) \leqslant k$ (from the definition of d), thus every successor of y has truth value $true$ after the first iteration of the evaluation procedure. Then, $\textsc{eval}(y) = true$, as well, after the first iteration of the evaluation procedure.

Given that $\textsc{eval}(x) = true$, for every $x \in T'$ after the first iteration of the evaluation procedure, it means that every blocked node in a good cycle is actually part of a true cycle. Thus, the said nodes will stay in the hypothesized set also in the second iteration. One can repeat the reasoning and show that after the second iteration the truth values of all nodes in T' is true, and so on, for every iteration.

Thus, for every $x \in T'$: $\textsc{eval}(x) = true$ and GT' is a successful traversal. According to Lemma 31, it follows that AO is also successful.

6.2.5.1 Soundness

Proposition 35. The algorithm $\mathcal{A}_{3,s}^{det}$ is sound: given a CoLP P and a predicate $p \in upreds(P)$, if there exists a successful evaluated AND/OR completion structure for checking satisfiability of p with respect to P, then p is satisfiable with respect to P.

Proof Sketch. Let AO be a successful evaluated AND/OR completion structure for checking satisfiability of p with respect to P. Then, according to Proposition 34, there exists a good traversal $GT = \langle ET, \textsc{ct}, \textsc{st}, G, \textsc{type}, \textsc{ucs}, \textsc{eval}, \textsc{spec} \rangle$ of AO.

We construct an open answer set (U, M) from GT by simply considering GT as a model representation: for every unary/binary predicate in the label of an AND node/outgoing arc from an AND node, there must be a unary/binary atom in the model formed with the respective predicate and node/arc:

$$\begin{aligned} U =& T \\ M =& \{p(x) \mid p \in \textsc{ct}(x), x \in T, type(x) = AND\} \cup \\ & \{f(x, z) \mid f \in \textsc{ct}(x, y), (x, y), (y, z) \in A_{ET}, type(x) = AND\} \end{aligned}$$

That (U, M) is a model follows directly from the construction of an AND/OR completion structure (See the soundness proof for the non-deterministic case). To show that (U, M) is a minimal model one can employ a similar argument as in the Soundness proof for the non-deterministic case: non-minimality would imply that there must be a cycle in G. But GT is good, thus every cycle in ET is good as well, and as such G cannot contain any cycle. Thus, (U, M) is minimal.

6.2.5.2 Completeness

Proposition 36. The algorithm $\mathcal{A}_{3,s}^{det}$ is complete: given a CoLP P and a predicate $p \in upreds(P)$, if p is satisfiable with respect to P, then there exists a successful evaluated AND/OR completion structure for checking satisfiability of p with respect to P.

Proof. Let CS' be a complete clash-free $\mathcal{A}_3$-completion structure for p with respect to P, which has been evolved from an initial $\mathcal{A}_3$-completion structure CS_0 for p with respect to P.

As p is satisfiable, from Proposition 21, such a completion structure must exist. Also, following the Completeness proof for $\mathcal{A}_3$ in Section 5.7, it is easy to see how CS' can be transformed into a structure $CS = \langle ET, \text{CT}, \text{ST}, G \rangle$, with $ET = (T, ES)$, which is evolved from CS_0 using the same rules used to evolve an $\mathcal{A}_3$-completion structure, with the exception of the caching rule (by simply appending the justification for caching nodes to the corresponding cached nodes). Then, CS is clash-free as well, and in particular, G is acyclic.

Let $AO_1 = \langle ET_1, \text{CT}_1, \text{ST}_1, G_1, \text{TYPE}_1, \text{UCS}_1, \text{EVAL}_1, \text{SPEC}_1 \rangle$ be an evaluated AND/OR completion structure for p with respect to P, with $ET_1 = (T_1, ES_1)$. Using CS as a guidance, we mark a 'walk' in AO which mimics a depth-first exploration of CS, where sometimes we loop around blocking paths, i.e. when encountering a blocked node we jump to its corresponding blocking node and continue the exploration. To this purpose, we introduce a partial function $mark : T_1 \cup A_{ET_1} \rightarrow \{yes\}$, which marks arcs and nodes from ET_1 which are part of the constructed walk. We also introduce a function $\bar{\cdot} : T \rightarrow T_1$ which relates nodes in CS to OR nodes in AO and we make use of the function ST to keep track of nodes in CS which were already processed.

We start the traversal of CS and the marking of AO by setting $\overline{\varepsilon} = \varepsilon$, $mark(\varepsilon) = yes$, and $\text{ST}(\varepsilon) = unexp$. Then, inductively, we consider nodes $x \in T$ such that $\text{ST}(x) = unexp$ and do as follows:

1. if $\overline{x}$ is a blocked OR node in AO with $(\overline{x}, y)$ being a blocking arc, and there exists a cycle $cy \in paths_{ET_1}$ such that:
   - $(\overline{x}, y) \subseteq cy$, and
   - for every $z \in cy$: $mark(z) = yes$,

   then let $\text{ST}(x) := exp$.
2. if $\overline{x}$ is not blocked in AO or there exists no cycle $cy \in paths_{ET_1}$ with the properties enumerated above, we distinguish between:
   - x is a blocked OR node in CS: in this case we loop along a blocking path in CS. The intention is to loop until a blocking cycle is reached in AO (case 1. above). Let y be the corresponding blocking node for x in CS, and let uc be the UCS which has been used to evolve y while constructing CS. Then let:
     - $\text{ST}(x) := exp$,
     - $\text{ST}(y) = unexp$, and
     - $\overline{y} = \overline{x}$.
   - if x is an ordinary node in CS, i.e. a non-blocked node. Let uc be the UCS which has been used to evolve x while constructing CS. Then there must be some node $y_1 \in T_1$ such that:
     - $\text{UCS}'(y') = uc$, and
     - $(x', y') \in A_{ET'}$ (possibly a blocking, redundancy or caching arc).

     Apply the following:

- $\text{ST}(x) := exp$.
- $mark(x, y_1) := yes$,
- $mark(y_1) = yes$
- for every i such that $y_1 \cdot i \in T_1$:
  - $mark(y_1, y_1 \cdot i) = yes$,
  - $mark(y_1 \cdot i) = yes$,
  - $\overline{y \cdot i} = y_1 \cdot i$, and
  - $\text{ST}(y \cdot i) = unexp$.

Note that as we loop along blocking paths in CS, there is the potential for non-termination. To see that this is not the case, consider the opposite. Then, intuitively, we must take an infinite walk in AO in order to mimic such an infinite loop. As the number of nodes in AO is finite, such an infinite walk can take place only along some cycle(s). If a blocking cycle is encountered according to case (1) above, the construction stops. Thus, the walk must take place along a non-blocking cycle. But this means that there exists some circular dependency between atoms formed with nodes from the non-blocking cycle. As the walk mimics the walk along the branch in CS, this translates in the presence of circular dependencies between atoms formed with nodes from the blocking path – contradiction with the blocking condition. Thus, the process always terminates when using CS as a guidance.

In the following, we show how to construct a good traversal $GT = \langle ET''$, CT'', ST'', G'', TYPE'', UCS'', EVAL'', $\text{SPEC}'' \rangle$ of AO, with $ET'' = (T'', ET'')$, based on the set of nodes and arcs in ET' which are marked. First, from the finiteness of the marking procedure, we observe that every node $x \in T'$ which is marked is within finite distance to a marked terminal AND node or a to a marked blocked OR node which belongs to a marked blocking cycle, i.e. a cycle in which every node is marked. We introduce a distance $d : T' \to \mathbb{N}$ which reflects this:

$$d(x) := \begin{cases} 0, & \text{if } x \text{ is a marked terminal AND node or} \\ & \text{a blocked OR node in a marked cycle} \\ min\{d(y) + 1 \mid (x, y) \in A_{ET} \text{ and } y \text{ is marked }\}, & \text{if } x \text{ is an OR node, other than above} \\ max\{d(y) + 1 \mid (x, y) \in A_{ET} \text{ and } y \text{ is marked }\}, & \text{if } x \text{ is an AND node, other than above.} \end{cases}$$

It is clear that $d(x)$ is finite, for every $x \in T'$ such that $mark(x) = yes$ (†).

Then, GT is evolved inductively, such that it contains only marked nodes:

- initially, $T'' = \{\varepsilon\}$.

- for every non-blocked OR node $x \in T''$ which has more than one marked successor in AO, pick up the successor y with minimal d. Let uc be the UCS which has been used to expand y. Then, $AOExpand_{GT}(x, uc)$.

- for every blocked OR node $x \in T''$, let y be such that $\text{SPEC}'(x, y) = bl$ and $(x, y) \subseteq cy$ with cy a marked cycle in AO. Then, connect x and y in GT: $A_{ET''} = A_{ET''} \cup \{(x, y)\}$.

That GT is a traversal, follows from the fact that each OR node has one and only one successor and each AND node preserves all successors from AO_2. To see that it is good, consider the opposite. Then, there must be some non-blocking cycle cy in GT. As each OR node has just one successor (the one with minimum d value) and every node has $d > 0$, it follows that the value of d for every node in the cycle is infinite – contradiction with (†).

Thus, GT is a good traversal of AO. From Proposition 34, it follows that GT is successful, while from Proposition 31, it follows that AO is successful as well.

Thus, if p is satisfiable with respect to a CoLP P, there exists a successful AND/OR completion structure for p with respect to P. $\Box$

As satisfiability checking of unary predicates with respect to CoLPs is EXPTIME-hard (Proposition 1) and $\mathcal{A}_{3,c}^{det}$ runs in exponential time (Proposition 30), is sound (Proposition 35) and complete (Proposition 36), it follows that:

Proposition 37. Algorithm $\mathcal{A}_{3,c}^{det}$ is worst-case optimal.

6.3 Worst-Case Optimal Reasoning with Simple Forest Logic Programs

In this section we describe $\mathcal{A}_{3,s}^{det}$, a deterministic worst-case optimal algorithm for reasoning with simple Forest Logic Programs, which builds on the non-deterministic algorithm $\mathcal{A}_3^s$ for reasoning with the same fragment introduced in Section 5.8.2.

$\mathcal{A}_{3,s}^{det}$ is similar in spirit to $\mathcal{A}_{3,c}^{det}$, the deterministic algorithm for reasoning with CoLPs described in the previous section. In some respects, it can be seen as a simplified variant of $\mathcal{A}_{3,c}^{det}$. This is due to the fact that predicate recursion in simple FoLPs is restricted such that no infinite chain of atoms can occur in the atom dependency graph of any model. An exception to this are the so-called local cycles induced by rules of the form $a(X) \leftarrow a(X)$, but these are dealt with in the process of constructing the set of non-redundant UCSs. As such, all cycles which are created in EF are 'good cycles'. Also, in line with $\mathcal{A}_3^s$, a form of anywhere blocking is employed where the blocking condition does not make any reference to the atom dependency graph.

However, as the algorithm deals in this case with forests instead of trees, there is an additional complication as regards the expansion of an AND/OR completion structure. This is explained and dealt with in Section 6.3.1. Section 6.3.2 describes the new blocking condition and when a completion structure is fully expanded. Section 6.3.3 describes the evaluation procedure for AND/OR completion structures for simple FoLPs, while Section 6.3.4 discusses termination and complexity issues.

6.3.1 Evolving AND/OR Completion Structure for Simple FoLPs

As usual, we start by defining the notion of initial AND/OR completion structure for a simple FoLP P. In this case, the underlying data structure is an extended forest. Also, as explained in the introduction to this chapter, we make use of a function CONST which assigns to some OR nodes in the structure, called proxy OR nodes, a constant from P.

Definition 49. An *AND/OR initial completion structure for checking satisfiability of a unary predicate p with respect to a simple FoLP P* is an AND/OR completion structure $\langle EF$, CT, ST, TYPE, UCS, EVAL, SPEC, CONST $\rangle$ for P, where:

- $EF = (F, \emptyset)$ is an extended forest with:
  - $F = \{T_c \mid c \in cts(P) \cup \{\varepsilon\}\}$, with ε being an anonymous individual or one of $cts(P)$,
  - $T_x = \{x\}$, for every $x \in cts(P) \cup \{\varepsilon\}$,
- $\text{CT}(x) = \{p\}$, for some $x \in cts(P) \cup \{roo\}$,
- $\text{TYPE}(x) = AND$, for every $x \in cts(P) \cup \{roo\}$, and
- $\text{EVAL}(x) = unknown$, for every $x \in cts(P) \cup \{roo\}$.

After defining the notion of initial completion structure, the reader would expect to employ the $AOExpand$ operation introduced in Section 6.1 to expand the structure. However, we cannot expand the structure straight away. By doing so, every root OR node in the structure could potentially have more than one AND successor. While an AND/OR completion structure was an exhaustive representation of the search space in the case of CoLPs, in the case of simple FoLPs potential interactions between AND nodes corresponding to constants in the structure preclude this.

As Section 6.3.3 will show, the set of nodes assumed to be true in the first iteration of the evaluation procedure will include the set of proxy nodes which are part of cycles which contain constants in the extended forest: this is due to the fact that there are no cycles in the atom dependency graph of the model(s) in construction which have as argument path such a cycle in the extended forest. However, care must be taken when propagating the truth values of such nodes. Distinct proxy nodes corresponding to the same constant could potentially be matched against different AND successors of the root node corresponding to that constant. Thus, we could potentially obtain a true traversal which contains more than one UCS having as root some constant in the program. Obviously, this should be disallowed as it could lead to inconsistencies: for every constant in the program, every model should contain just one UCS which has as root the respective constant.

To ensure that a true AND/OR completion structure always admits a true traversal, we introduce an additional notion of AND/OR pre-completion structure which is the result of expanding an initial AND/OR completion structure such that every constant OR node has exactly one successor:

Definition 50. An *AND/OR pre-completion structure for checking satisfiability of a unary predicate p with respect to a simple FoLP P* is an AND/OR completion structure derived from an initial AND/OR completion structure for checking satisfiability of p with respect to P: $A0 = \langle EF$, CT, ST, G, TYPE, UCS, EVAL, SPEC, CONST $\rangle$, by applying the operation $AOExpand_{AO}(c, uc)$ exactly once for every $c \in cts(P)$, where uc is a non-redundant UCS for P with root c.

Regular OR nodes in a pre-completion structure can be expanded as usual, by introducing an AND successor for every matching UCS using the $AOExpand$ operation. What is still missing, is a rule for expanding proxy OR nodes: these are nodes whose successors are restricted to be constant OR nodes, that is successors of root OR nodes standing for constants. Due to the constraint imposed on pre-completion structures, that each such structure contains at most such an AND node for every constant, it is obvious that every proxy OR node could have at most one match, the AND node in question. Thus, the expansion operation consists in this case only in checking that the content of the proxy OR node x is not in contradiction with the content of its corresponding constant AND node (if that exists); if that condition is fulfilled, the two nodes are linked via an arc and the proxy node is marked is expanded. We denote this operation again with $AOExpand_{AO}(x)$, where x is a proxy OR node and formally define it as follows:

- let $c := \text{CONST}(x)$ and let i be such that $c \cdot i \in N_{EF}$;
- if $\text{CT}(x) \subseteq \text{CT}(c \cdot i)$, then:
  - create an arc from x to $c \cdot i$ and add it to ES: $ES := ES \cup \{(x, c \cdot i)\}$,
  - $A := A \cup \{(p(x), p(c \cdot i)) \mid p \in \text{CT}(x)\}$,
  - $\text{ST}(x) := exp$.

We next define the expansion rule ***DetSMatch*** as follows:

Rule. DetSMatch. Let $AO = \langle EF, \text{CT}, \text{ST}, \text{TYPE}, \text{UCS}, \text{EVAL}, \text{SPEC}, \text{CONST} \rangle$, with $ET = (F, ES)$, be an AND/OR completion structure . If for some node $x \in N_{EF} - cts(P)$:

- $\text{TYPE}(x) = OR$,
- $\text{ST}(x) = unexp$, and
- for every node y such that $right_T(y, x)$: $\text{ST}(y) = exp$,

then apply $AOExpand_{AO}(x)$.

Note that the notion of pre-completion structure we introduced in this section is defined non-deterministically. However, it is easy to see how the set of all pre-completion structures can be computed by iterating through all UCSs which match a certain constant root OR node.

6.3.2 Anywhere Blocking for Deterministic Reasoning with Simple FoLPs

As a termination mechanism, similarly to the non-deterministic case, we employ a form of anywhere blocking:

Rule. *DetABlocking*. If there exist two AND nodes $x, y \in N_{EF}$ such that:

- $x, y \notin cts(P)$,
- $y <_{T_c} x$ or $right_{T_c}(x, y)$, for some $c \in cts(P) \cup \{\varepsilon\}$,

- $\text{UCS}(x) = \text{UCS}(y)$,

then x is a *blocked* node and y is its corresponding *blocking* node. Let:

- $u = prec_T(x)$,
- $ES = ES \cup \{(u, y)\}$,
- $\text{SPEC}(u, y) = bl$,
- $A = A \cup \{(p(u), p(y)) \mid p \in \text{CT}(u)\}$,
- $T_c = T_c - T_x$.

Node u is said to be a *blocked OR node*, arc (u, x) is a *blocking arc*.

Finally:

Definition 51. An AND/OR-*complete completion structure* for a simple FoLP P and $p \in upreds(P)$, is an AND/OR-completion structure that results from the repeated application of the rule *DetSMatch* to an AND/OR pre-completion structure for p and P, taking into account the applicability rule *DetABlocking* such that no rules can be further applied.

6.3.3 Evaluating AND/OR Completion Structures for Simple FoLPs

As was the case for AND/OR completion structures for CoLPs, complete AND/OR completion structures for simple FoLPs have to be evaluated: every node in such a completion is assigned eventually one of the truth values $true$ or $false$. The evaluation procedure consists again of an initialization step and a fix-point procedure. The initialization step is identical to the initialization step used by $\mathcal{A}^{det}_{3,c}$.

As concerns the fix-point procedure, again a set of nodes is assumed to be true at every iteration: this time the set consists in the union of the set of blocked OR nodes and the set of proxy OR nodes which are part of some cycle in EF. The intuition for the presence of the blocked OR nodes is the same as in the case of CoLPs (with the difference that here anywhere blocking is applicable). As regards proxy OR nodes, as we explained in Section 6.3.1, the syntactical restriction on simple FoLPs makes it possible to have arbitrary circular justifications between nodes of EF as there are no overlying cycles in the atom dependency graph. The modified evaluation procedure which still uses the $Propagation$ operation and the notion of 'true cycle' introduced in Section 6.2.3 is described by Algorithm 6.3.

The result of applying Algorithm 6.3 to a complete AND/OR completion structure for p with respect to P is said to be *an evaluated AND/OR completion structure for* p *with respect to a simple FoLP* P.

Definition 52. An evaluated AND/OR completion structure is said to be *successful* iff the truth value of every root node in the structure is true: $\text{EVAL}(x) = true$, for every $x \in cts(P) \cup \{roo\}$.

Algorithm 6.3: Evaluation of an AND/OR Completion Structure for a simple FoLP

```
input : an AND/OR complete completion structure $AO = \langle EF, \textsc{ct}, \textsc{st}, \textsc{type}, \textsc{ucs},$
        $\textsc{eval}, \textsc{spec}, \textsc{const} \rangle$, with $EF = (F, S)$, $F = \{T_x \mid x \in ctsP \cup \{\varepsilon\}\}$ for
        checking satisfiability of $p$ with respect to a simple FoLP $P$
output: $AO$ is updated such that every node has truth value $true$ or $false$ according to its
        satisfiability status

1) Initialization: nodes with no successors $S := \emptyset$;
for every $x \in N_{EF}$ such that $x$ has no successors in $EF$ do
    if $type(x) = OR$ then
        $\textsc{eval}(x) := false$;
    end
    else
        $\textsc{eval}(x) := true$;
    end
    $S := S \cup \{x\}$;
end
$Propagate(AO, S)$;

2) Iterative evaluation procedure
$S := \{x \mid \textsc{spec}(x, y) = bl,$ for some $y \in N_{EF}\} \cup \{x \mid \textsc{const}(x)$ is defined, $x \in$
$cy, cy$ is a cycle in $EF\}$;
$switch := false$;
repeat
    $AO' := AO$;
    i) Overestimating truth for $x \in S$ (in $AO'$) do
        $\textsc{eval}(x) := true$;
    end
    $Propagate(AO', S)$;
    ii) Hypothesis check for $x \in S$ do
        if there exists no true cycle $cy$ in $EF$ (in $AO'$) such that there is some node
        $y \in N_{EF}$ with $(x, y) \subseteq cy$ and $\textsc{spec}(x, y) = bl$ or $\textsc{const}(x) = y$ then
            $S := S - \{x\}$;
            $switch := true$;
        end
    end
until $switch = false$;

3) Transfer of truth to $AO$; final assignment and propagation
$AO := AO'$;
for every $x \in N_{EF}$ do
    if $\textsc{eval}(x) = unknown$ then
        $\textsc{eval}(x) := false$;
    end
end
```

Proposition 38. Given a simple FoLP P and a predicate $p \in \mathit{upreds}(P)$, p is satisfiable with respect to P iff there exists a successful evaluated AND/OR completion structure for checking satisfiability of p with respect to P.

Proof. As we will see the proof for this proposition is strictly related to the soundness and completeness proofs for $\mathcal{A}_3^s$, the non-deterministic algorithm for reasoning with simple FoLPs.

We first introduce the notion of traversal of an AND/OR completion structure for p with respect to a simple FoLP P.

Definition 53. A *traversal of an AND/OR completion structure* $AO = \langle EF$, $\textsc{ct}$, $\textsc{st}$,, $\textsc{type}$, $\textsc{ucs}$, $\textsc{eval}$, $\textsc{spec}$, $\textsc{const}\ \rangle$ for p with respect to a simple FoLP P, with $EF = (F, ES)$, is an AND/OR completion structure $Tr = \langle EF'$, $\textsc{ct}'$, $\textsc{st}'$, G', $\textsc{type}'$, $\textsc{ucs}'$, $\textsc{eval}'$, $\textsc{spec}'$, $\textsc{const}\ \rangle$ defined as follows:

- $EF' = (F', ES')$, with $F' = \cup_{x \in cts(P) \cup \{\varepsilon\}} T'_x$, where ε is the same as the one used to define EF,
- for every $x \in N_{EF'}$ it holds that:
  - $x \in N_{EF}$,
  - $\textsc{ct}'(x) = \textsc{ct}(x)$,
  - $\textsc{st}'(x) = \textsc{st}(x)$,
  - $\textsc{type}'(x) = \textsc{type}(x)$,
  - $\textsc{ucs}'(x) = \textsc{ucs}(x)$,
  - $\textsc{eval}'(x) = \textsc{eval}(x)$,
  - $\textsc{const}'(x) = \textsc{const}(x)$ (if defined),
  - if $\textsc{type}'(x) = OR$: if there exists $y \in T$ such that $(x, y) \in A_{ET}$, there exists a unique $z \in T'$ such that $(x, z) \in A_{ET}$ and $(x, z) \in A_{ET'}$;
  - if $\textsc{type}'(x) = AND$: for every $y \in T$ such that $(x, y) \in A_{ET}$, $y \in T'$ and $(x, y) \in A_{ET'}$;
- for every $(x, y) \in A_{ET'}$ it holds that: $\textsc{spec}'(x, y) = \textsc{spec}(x, y)$;

As before, there is a bidirectional correspondence between successful evaluated AND/OR completion structures and successful traversals.

Lemma 31. An evaluated AND/OR completion structure for a simple FoLP P and a unary predicate p is successful iff it admits a successful traversal.

Lemma 32. A traversal of an evaluated AND/OR completion structure for a simple FoLP P and a unary predicate p is successful iff all nodes in the traversal have truth value true.

Further on, one can relate successful traversals to complete clash-free $\mathcal{A}_3^s$-completion structures for p with respect to P:

Lemma 33. Let P be a simple FoLP and $p \in \mathit{upreds}(P)$. Then, there exists a complete clash-free $\mathcal{A}_3^s$-completion structures for p with respect to P iff there exists a successful traversal of an AND/OR completion structure for p with respect to P.

Proof. It follows from the definitions of the two entities: it is easy to see that it is possible to construct a complete clash-free $\mathcal{A}_3^s$-completion structure from a successful traversal and vice versa. □

Then, from soundness and completeness of $\mathcal{A}_3^s$ (Proposition 25), Lemma 31, Lemma 32, and Lemma 33, it follows that $\mathcal{A}_{3,s}^{det}$ is sound and complete as well. □

6.3.4 Termination and Complexity

Again, there are two main issues to consider regarding termination and running time of $\mathcal{A}_{3,s}^{det}$: first, the termination and running time of the expansion procedure, and second, the termination and running time of the evaluation procedure.

- *Termination and running time of the expansion procedure*:
  - there are an exponential number of AND/OR pre-completion structures as the number of (non-redundant) UCSs is exponential in the size of the program and for every constant c in the program, we have to make a choice regarding the UCS which is used to expand it;
  - the number of nodes in a complete AND/OR completion structure for a simple FoLP P is bounded by the number of (non-redundant) UCSs for P, which is exponential in the size of P;
  - the expansion of an OR node in the structure can be performed in linear time in the number of UCSs, thus exponential in the size of the program;
  - checking the anywhere blocking condition and performing the necessary steps when the condition is fulfilled can be done in polynomial time in the size of the program.

  From the analysis above, it follows that the set of all complete AND/OR completion structure for a simple FoLP P can be constructed in the worst case in exponential time.
- *Termination and running time of the evaluation procedure*: following an argument similar to the one used to show the termination of the evaluation procedure in the case of CoLPs, we obtain that this step can again be performed in the worst case in exponential time in the size of the program.

We thus obtain:

Proposition 39. Algorithm $\mathcal{A}_{3,s}^{det}$ runs in the worst case in deterministic exponential time.

As satisfiability checking of unary predicates with respect to simple FoLPs is EXPTIME-hard (Corollary 8), from Proposition 38 and Proposition 39, we obtain the following main result:

Theorem 3. Satisfiability checking of unary predicates with respect to simple FoLPs is EXPTIME-complete. Furthermore, $\mathcal{A}_{3,s}^{det}$ is a worst-case optimal algorithm for solving this task.

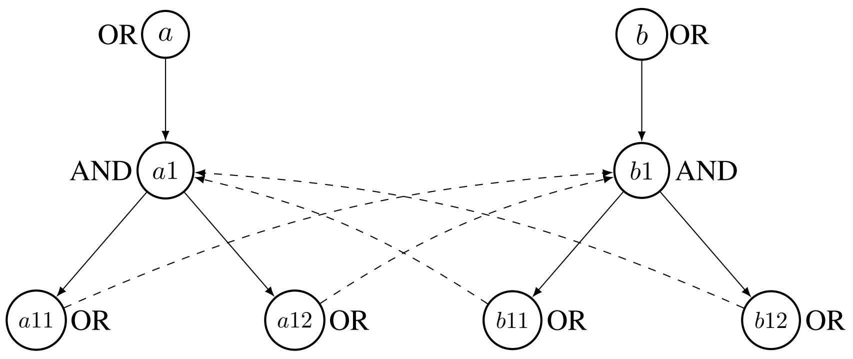


Figure 6.7: AND/OR completion structure with proxy OR nodes which belong to both good and bad cycles

6.4 Discussion and Related Work

We presented two deterministic worst-case optimal algorithms for reasoning with CoLPs and simple FoLPs, both of which are fragments of FoLPs. The natural question is why not having a single such algorithm which deals with the whole fragment of FoLPs? Unfortunately, the technique used in this chapter to obtain worst case optimal algorithms, i.e. constructing AND/OR completion structures and applying termination conditions across branches of such structures, does not work in the presence of both constants and unrestricted recursion, as is the case for FoLPs.

Consider the AND/OR completion structure shown in Figure 6.7 which depicts a pattern that could easily occur when constructing such a structure for a FoLP. There are two OR nodes corresponding to constants a and b each with just one successor AND node: $a1$ and $b1$, respectively. Each of the two AND nodes has in turn two OR successors: $a11$ and $a12$ for $a1$, and $b11$ and $b12$ for $b1$, respectively. There are four elementary cycles in the underlying extended forest corresponding to this structure:

$$C_1 = (a1, a11, b1, b11, a1)$$
$$C_2 = (a1, a12, b1, b11, a1)$$
$$C_3 = (a1, a11, b1, b12, a1)$$
$$C_4 = (a1, a12, b1, b12, a1)$$

Now assume that C_1 and C_4 are good cycles, while C_2 and C_3 are bad, i.e. non-good, cycles. Then, each of the proxy OR nodes: $a11$, $a12$, $b11$, and $b12$, belongs both to a good and to a bad cycle. The question in this situation is which truth value to assign to these proxy nodes? Should they be part of the set of nodes initially assumed to be true? If we decide to include also such nodes in the set, then we obtain that all nodes of the structure in Figure 6.7 have truth value $true$. However if one looks at the structure, it can be seen that it is actually a traversal, which contains bad cycles. Thus, it is not an actual model. If we decide not to include such nodes, we might find other situations in which the structure admits a good traversal although some proxy OR nodes are in bad cycles, but in which all nodes will be evaluated as false due to our initial decision regarding proxy OR nodes.

We did not succeed to identify any deterministic strategy to assign truth values to such nodes. We conjecture that there might not be any such strategy and that, in fact, Forest Logic Programs are NEXPTIME-complete. It is subject to future work to prove this conjecture.

Related Work

As we mentioned in the introduction to this thesis, while tableau algorithms behave well in practice, they are usually sub-optimal. In the case of EXPTIME-complete logics, they generate many times a double exponential number of nodes. The explosion of the universe size might be alleviated by the use of anywhere blocking, but that does not always help [Motik et al., 2009b]. Another reason why such algorithms are sub-optimal is that they are typically non-deterministic and as such they run in the worst-case in non-deterministic exponential/double exponential time.

[De Giacomo et al., 1996] sketches one of the first attempts to obtain a worst-case optimal algorithm for an EXPTIME logic, $\mathcal{ALC}$. The work is extended in [Donini and Massacci, 2000]: the resulted algorithm constructs a tree-shaped tableau in which every branch stands for a potential model. Thus, the tableau represents an exhaustive representation of the search space, similar in this respect to an AND/OR completion structure in the case of $\mathcal{A}_{3,c}^{det}$. However the underlying tree is an OR tree: every branching point is caused by a disjunction. This is unlike the case of the AND/OR completion structures which we introduced in this chapter and the completion structures employed by $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ which are AND structures: every branching point stands for a conjunction. The algorithm in [De Giacomo et al., 1996] reuses proven inconsistencies by means of an inconsistency propagation calculus which allows the information about such inconsistencies to be propagated also across different branches of the tableau (thus, across different potential models).

More recently, an EXPTIME tableau for $\mathcal{ALC}$ has been described in [Goré and Nguyen, 2013]. The algorithm constructs an AND/OR graph which is similar to the AND/OR completion structures employed by $\mathcal{A}_{3,c}^{det}$ (which are extended trees). The graph is initialized with a node which contains in its label the concept checked to be satisfiable together with the TBox axioms. However, as the underlying logic is $\mathcal{ALC}$, the construction of such an AND/OR tree uses a simple anywhere blocking technique: whenever the same label is encountered twice, the node where the label occurs first is reused.

Like our deterministic algorithms $\mathcal{A}_{3,c}^{det}$ and $\mathcal{A}_{3,s}^{det}$, the algorithm described in [Goré and Nguyen, 2013] is a two-staged procedure: after the construction of the AND/OR graph, nodes are evaluated to establish the satisfiability status of their label. Two distinct evaluation procedures are provided, the second one being an optimization of the first. Both procedures assign to every node which contains an inconsistency the status `unsat`. The first procedure works by propagating unsatisfiability in the AND/OR graph in a similar manner we propagate falsehood in an AND/OR completion structure. However, here everything which is not explicitly unsatisfiable after the propagation procedure is satisfiable – this is due to the fact that atoms in an $\mathcal{ALC}$ model do not have to be supported. If there is any atom which has no successors, its status is always `sat`. Also, any cycles in the AND/OR graph, are good cycles: tautological supports are not disallowed.

The second procedure interleaves the construction of the AND/OR graph with the process of evaluating the satisfiability status of nodes in the graph. It also uses a so-called “sound global caching” strategy by which the authors understand caching across models of both satisfiability and unsatisfiability. In this last respect, it is more similar to our algorithm as we also propagate in a first stage both truth and falsehood. While our algorithms and also the first procedure described in [Goré and Nguyen, 2013] have the drawback that their best-case performance is identical to their worst-case performance – this is due to the fact that they construct in a first stage an exhaustive representation of the search space for all possible models –, the interleaving approach has the potential to terminate before actually constructing the whole search space for models. It is the subject of future work to formalize such an interleaving approach also for the case of $\mathcal{A}_{3,c}^{det}$ and $\mathcal{A}_{3,s}^{det}$.

The approach described in [Goré and Nguyen, 2013] is generalized in [Goré and Nguyen, 2008] to the case of sound global caching for abstract modal tableaux. It has been also extended to deal with $\mathcal{ALCI}$ [Goré and Widmann, 2009], fixpoint logics [Goré, 2009], and $\mathcal{SHIQ}$ [Goré and Nguyen, 2007].

CHAPTER 7

Summary and Future Work

7.1 Further Related Work

In this section we discuss some further approaches for combining rules and ontologies which are related to FoLPs by virtue of their hybrid modeling capacities, but which were not touched upon as part of one of the related work discussions at the end of each chapter. As explained in Section 1.1, FoLPs are an interesting formalism as they allow to combine features from both the classical First Order Logic world and the non-monotonic Logic Programming world. In particular, having the ability to simulate reasoning within the DL $\mathcal{SHOQ}$, they allow the integration of $\mathcal{SHOQ}$ KBs and themselves in the form of f-hybrid KBs. As such here we will discuss formalisms which are related to f-hybrid KBs as well.

A possible way to look at hybrid formalisms [de Bruijn et al., 2006] is based on the way the LP and the DL component interact. At one end of the spectrum, the LP component sees the classical component as a black box which can be accessed via a query entailment interface; approaches which adopt such a communication mechanism are called *loosely-coupled approaches*. At the other end, are the so-called *integrating* approaches; these are approaches in which one does not speak about a combination per se, but about a unified formalism which has features from both knowledge representation paradigms. FoLPs fall into this category. Finally, a third category of combinations of rules and ontologies is the class of *tightly-coupled approaches*: in this case the two components are maintained distinct, but the semantics of the combination is defined via common interpretations. This is the case for f-hybrid knowledge bases.

In the following, we have a look at different approaches which fall into one or the other of the above-mentioned categories and their relation to FoLPs/f-hybrid KBs. As we will see, in most of the cases it is either not possible to reason with unknown individuals within the rule component or the rule component is syntactically restricted in ways which are orthogonal to the syntactic restrictions imposed on FoLPs.

7.1.1 Loosely-coupled Approaches

The most well-known loosely-coupled approach is *dl-programs* [Eiter et al., 2008]. The formalism integrates logic programs under the stable model semantics with DL knowledge bases by allowing the logic program to query the DL knowledge base. At the same time, it is possible to send (controlled) input from the logic program to the DL knowledge base. Reasoning is done via a stable model computation of the logic program, interwoven with queries that are oracles to the DL part. The approach has also been adapted for the case of logic programs under the well-founded semantics [Eiter et al., 2011].

As both semantics for dl-programs – the stable model semantics and the well-founded semantics –, assume that the dl-program is grounded in a first step using the set of names occurring in the two components, no reasoning with unnamed individuals takes place at the level of the rule component. This is unlike the case of FoLPs whose semantics is defined with respect to open domains, i.e. domains which extend the set of names occurring in the program with unknown individuals.

7.1.2 Tightly-coupled Approaches

An early attempt to combine rules and ontologies was *Semantic Web Rule Language (SWRL)* [Horrocks and Patel-Schneider, 2004]: there, OWL DL KBs, having as underlying logic the DL $\mathcal{SHOIN}$, were combined with function-free Horn rules. In the general case the formalism is undecidable. This is due to the interaction between the open domain semantics of OWL DL and the rule component. Some decidable restrictions were defined in the form of *DL-safe rules* [Motik et al., 2005] and *Description Logic Rules (DL rules)* [Krötzsch et al., 2008a].

DL-safe rules impose a safety-condition on the variables occurring in the rule component which can be seen as an extrapolation of the classical Datalog safety condition. A variable occurring in a rule is said to be *DL-safe* iff it occurs in a non-DL-atom in the body of the rule. A combined KB is DL-safe iff every variable occurring in the rule component is DL-safe. This syntactic restriction ensures that all facts which can be inferred using the rule component are about named individuals which occur in rules.

Description Logic Rules (DL rules) is another family of decidable fragments of SWRL. All rules are restricted such that they allow only for unary and binary predicates corresponding to concept expressions and role names in a specific DL. Similar to FoLPs, they are tree-shaped rules. However, while variables in FoLP rules can be seen as nodes in a tree of depth 1, here the chaining of variables can have arbitrary length, i.e. they allow for constructions like $f(X,Y)$, $g(Y,Z)$, $\ldots$ in the body of a rule. Such constructions can also be simulated in FoLPs by introduction of new unary atoms and new rules. For example, a rule of the form $a(X) \leftarrow f(X,Y), g(Y,Z), b(Z)$ can be transformed into two FoLP rules $a(X) \leftarrow f(X,Y), c(Y)$ and $c(X) \leftarrow g(X,Y), b(Y)$, where x is a freshly introduced unary predicate. Although Description Logic Rules have tree-shaped bodies and are from this perspective similar to FoLPs, their semantics is not a minimal model semantics. Like Description Logics, their semantics is first-order based.

Depending on the underlying DL, [Krötzsch et al., 2008a] distinguishes between $\mathcal{SROIQ}$ rules, $\mathcal{EL}^{++}$ rules, and Description Logic Program rules:

- $\mathcal{SROIQ}$ rules are the most expressive fragment by allowing a $\mathcal{SROIQ}$ ontology to serve as the DL component. They do not actually extend the DL $\mathcal{SROIQ}$, as the rules can be mapped to $\mathcal{SROIQ}$. In order to ensure that such a translation is possible some more restrictions are imposed on the rule component.

- $\mathcal{EL}^{++}$ rules are combinations of $\mathcal{EL}^{++}$ KBs with DL rules. In this case, the DL rules are the core expressive mechanism to which the $\mathcal{EL}^{++}$ components are reduced.

- Description Logic Program rules have as an underlying formalism the language Description Logic Programs (DLP) [Grosof et al., 2003]. DLP represents the common subset of OWL-DL ontologies and Horn logic programs. So-called DL2 KBs are defined as combinations of DLP rules KBs with DLP KBs, which additionally might contain role disjunction axioms and/or role asymmetry axioms. Such a KB can be transformed into a set of function-free first-order Horn rules.

[Krötzsch et al., 2008b] extends DL rules to a new type of rules, called *extended DL rules*. Unlike regular DL rules, the extended DL rules allow for 'role conjunctions' in rule bodies. A new decidable fragment has been defined, called *ELP rules*, which builds on extended $\mathcal{EL}^{++}$ rules and employs at the same time a notion of DL-safety.

As a general observation concerning DL rules, the focus in all of the approaches is on extending DLs with rule bases which are as expressive as possible while at the same time preserving the computational properties of the initial DL. This leads sometimes to rather intricate syntactic characterizations of different fragments. Syntactically, some of these fragments allow for more complex rule shapes than FoLP rules, but FoLPs distinguish themselves through the fact that they have a *negation as failure* operator and adopt a minimal model semantics, thus adding a different type of expressivity to such combinations of rules and ontologies, which is not specific to the DL world.

r-hybrid knowledge bases [Rosati, 2008] are another tightly-coupled approach for combining rules and ontologies. They extend $\mathcal{DL}$+*log* [Rosati, 2006] with inequalities and negated DL atoms. Similar to DL-safe rules, they employ a notion of safety for variables in the rule component. However, in this case the safety condition is relaxed from the classical one to the so-called *weakly DL-safety*: a rule is weakly DL-safe iff every variable in the rule appears in a positive atom in the body of the rule (*Datalog safeness*), and every variable either occurs in a positive non-DL atom in the body of the rule, or it only occurs in positive DL atoms in the body of the rule.

As in the case of DL-safe rules, reasoning with r-hybrid knowledge bases can be performed by grounding the rule component with the set of constants appearing explicitly in the knowledge base. However, here, due to the weaker safety condition, some variables from rule bodies which occur in DL atoms might remain uninstantiated. The conjunction of all atoms in a rule body containing such variables is seen as a conjunctive query which is delegated to the DL part of the knowledge base. Decidability for satisfiability checking of r-hybrid knowledge bases is guaranteed if decidability of the conjunctive query containment/union of conjunctive queries containment problems is guaranteed for the DL at hand. Note that f-hybrid KBs do not impose any restriction whatsoever on the interaction between the $\mathcal{SHOQ}$ component and the FoLP one. However, in the case of f-hybrid KBs, the DL is fixed to be $\mathcal{SHOQ}$ or one of its fragments. This is a consequence of the fact that reasoning with f-hybrid KBs relies on a translation of $\mathcal{SHOQ}$ KBs to FoLPs.

MKNF$^+$ knowledge bases [Motik and Rosati, 2010], consist of a DL component and a component of so-called MKNF$^+$ rules. The latter allow for modal operators $\mathbf{K}$ and $\mathbf{not}$ in front of atoms, but also for non-modal atoms, unlike their predecessor, hybrid MKNF knowledge bases [Motik and Rosati, 2006, Motik et al., 2006]; non-modal atoms can be eliminated by a transformation leading to MKNF knowledge bases. Also, unlike the rules in hybrid MKNF knowledge bases, atoms in MKNF$^+$ rules are 'generalized', in the sense that they can be arbitrary first-order formulae. This allows the approach to capture languages like *EQL-Lite*$(\mathcal{Q})$ [Calvanese et al., 2007a], dl-programs by [Eiter et al., 2008] and disjunctive dl-programs by [Lukasiewicz, 2004]. Other approaches to integrating ontologies and rules which are generalized by MKNF$^+$ knowledge bases are: CARIN-style rules [Levy and Rousset, 1996], $\mathcal{AL}$-log [Donini et al., 1998], DL-safe rules [Motik et al., 2005], the Semantic Web Rule Language (SWRL) [Horrocks and Patel-Schneider, 2004], and r-hybrid knowledge bases [Rosati, 2008].

MKNF knowledge bases are in the general case undecidable. Again, in order to regain decidability a DL-safety condition is imposed, together with a notion of admissibility which concerns decidability for the DL inference. The same considerations hold regarding the relation with f-hybrid knowledge bases as in the case of r-hybrid knowledge bases.

7.1.3 Integrating Approaches

Datalog$^\pm$ [Gottlob and Lukasiewicz, 2009, Calì et al., 2009] is an extension of Datalog which allows for a special type of rules with existentially quantified variables in the head, called tuple generating dependencies (TGDs). The formalism is undecidable in the general case. Like in the case of OASP, several syntactic restrictions have been imposed on the shape of TGDs in order to regain decidability. Two such restrictions are: (1) every rule should have a guard, an atom which contains all variables in the rule body, giving rise to *guarded Datalog*$^\pm$, and (2) every rule should have a singleton body atom, giving rise to *linear Datalog*$^\pm$. It is possible to simulate some DLs from the DL-Lite family [Calvanese et al., 2007b] within these fragments. The guardedness condition has been relaxed to *weakly-guardedness*, where the weak guard has to contain only the variables in the body that appear in so-called affected positions, positions where newly invented values can appear during reasoning [Calì et al., 2008].

Some further generalizations to the guarded fragment of Datalog$^\pm$ are so-called *sticky sets* of TGDs [Calì et al., 2010a], *weakly-sticky* sets of TGDS, and *sticky-join* sets of TGDs [Calì et al., 2010b] which generalize both sticky sets and linear TGDs. All these fragments are defined by imposing restrictions on multiple occurrences of variables in rule bodies. The syntactic restrictions on rules bodies are orthogonal to the ones imposed on FoLPs: neither Datalog$^\pm$ rules are enforced to have a tree-shape like FoLPs, nor do variables in FoLP rules have to fulfill the conditions required for the different sets of TGDs to belong to one of the previously mentioned decidable fragments of Datalog$^{\pm}$. Usually, TGDs do not negation. However, so-called stratified normal TGDs have been introduced [Lukasiewicz et al., 2012], which are TGDs whose body atoms can appear in a negated form together with a semantics in terms of canonical models. FoLPs support full negation as failure (under the stable models semantics).

7.2 Summary

The main objective of this thesis was to design algorithms for reasoning with (fragments of) Forest Logic Programs (FoLPs). FoLPs is a fragment of Open Answer Set Programming that has the forest model property. We presented the following results:

- Chapter 3 described $\mathcal{A}_1$, a tableau-based algorithm for reasoning with FoLPs. The algorithm runs in non-deterministic double exponential time. It was the first algorithm to deal with the whole language of FoLPs and as such it also established the decidability of the language. $\mathcal{A}_1$ uses a set of expansion rules for evolving a potential forest model for a FoLP. Due to the non-monotonicity of the language, in order to distinguish between minimal and non-minimal models it employs a complex blocking condition which considers among others dependencies between atoms in the constructed model. In a minimal model, no atom should depend on itself or on an infinite chain of atoms.

- Using $\mathcal{A}_1$ we devised a knowledge compilation technique which pre-computes all potential building blocks of forest models in the form of trees of depth one. Such a building block is called unit completion structure (UCS). We also introduced a notion of redundant UCSs – these are UCSs which can be discarded in the process of constructing a model without losing completeness. The technique and the new algorithm $\mathcal{A}_2$ which builds on this technique are described in Chapter 4.

- An optimized algorithm for reasoning with FoLPs, $\mathcal{A}_3$, was presented in Chapter 5. The worst-case running time performance of $\mathcal{A}_3$ improves the one of of $\mathcal{A}_1$ and $\mathcal{A}_2$ by one exponential. The improvement is due to different termination conditions used by the new algorithm, in particular due to the a rule which allows one to reuse computation across branches of a model. The new termination conditions are a reflection of a new strategy to establish the finite bounded model property of the language, which is part of the completeness proof of $\mathcal{A}_3$.

  The same chapter introduced a restricted fragment of FoLPs, called simple FoLPs, for which the termination conditions of $\mathcal{A}_3$ collapse into a much simpler subset-based anywhere blocking condition. Simple FoLPs generalize previous fragments of FoLPs known to be decidable, like local and acyclic FoLPs.

- Chapter 6 described worst-case optimal procedures for dealing with the fragments of CoLPs – the restriction of FoLPs to a language which does not allow for the occurrence of constants, and simple FoLPs, the fragment introduced in Chapter 5. The procedures build exhaustive representations of the search space for tree/forest models in the form of AND/OR trees/forests. They employ similar termination conditions as $\mathcal{A}_3$, but this time it is possible to reuse computation across branches belonging to different models, which in turn leads to worst-case optimal behaviour.

7.3 Future Work

The main open question is the one raised at the end of Chapter 6: is satisfiability checking for FoLPs feasible in exponential time or is the problem harder? We have the feeling that the latter is the case, and that the problem might be NEXPTIME-hard.

In Section 3.7 of Chapter 3 we pinpointed the relation between the expansion rules of our initial algorithm for reasoning with FoLPs, $\mathcal{A}_1$, and different strategies employed by classical ASP solvers. We think that it would be interesting to pursue this affinity further for both theoretical and practical reasons. Theoretically, it would give us insight into how to devise smarter less non-deterministic expansion rules. This might lead to an efficient procedure to compute the set of UCSs for a Forest Logic Program which could serve as the basis for a practical implementation.

An interesting point which the work on this thesis touched upon is the connection between automata-based procedures and tableau-based procedures: in Section 4.5.1 of Chapter 4 we discussed at length the work in [Baader et al., 2003b] concerning abstract tableau systems for Description Logics and their translation into looping tree automata. On the other hand, [Hladik and Sattler, 2003] and [Hladik, 2007] explored this link in the opposite direction: it provided a translation from alternating looping tree automata into the Description Logic $\mathcal{FLEUI}_f$. As we explained in Section 5.9, our tableau algorithms do not fit into the framework of [Baader et al., 2003b]. At the same time, we know that for CoLPs it is possible to reduce satisfiability checking of unary predicates to emptiness testing for two-way alternating tree automata with a parity condition. This raises some questions:

- would it be possible to reduce the task of checking emptiness of two-way alternating tree automata with a parity condition to satisfiability checking with respect to CoLPs?
- could satisfiability checking with respect to FoLPs be polynomially encoded as an emptiness check of two-way alternating tree automata? Of course, this question is very much related to the one concerning the theoretical complexity of FoLPs. If the answer is positive, then FoLPs are EXPTIME-complete. Even though there exists such an encoding for CoLPs, the answer to this question is not trivial. This is for similar reasons for which it is also not trivial to lift the deterministic algorithm $\mathcal{A}_{3,c}^{det}$ introduced in Section 6.2 to the full FoLP case. For more details, see the discussion in Section 6.4.

Finally, we mentioned briefly at the end of Chapter 6 several possible optimizations that could be brought to the deterministic algorithms $\mathcal{A}_{3,c}^{det}$ and $\mathcal{A}_{3,s}^{det}$ for reasoning with CoLPs and simple FoLPs, respectively, in the direction of interleaving the construction of AND/OR completion structures with their evaluation. This would enable the identification of models before the whole search space is explored. Such an interleaving algorithm would be at the same time worst-case optimal and exhibit pay-as-you-go behaviour. We plan to also investigate further that dimension.

Bibliography

[Anger et al., 2005] Anger, C., Gebser, M., Linke, T., Neumann, A., and Schaub, T. (2005). The nomore++ approach to answer set solving. In Sutcliffe, G. and Voronkov, A., editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lecture Notes in Computer Science*, pages 95–109.

[Baader et al., 2003a] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and (eds), P. F. P. (2003a). The description logic handbook: Theory, implementation, and applications. In *Description Logic Handbook*. Cambridge University Press.

[Baader et al., 2003b] Baader, F., Hladik, J., Lutz, C., and Wolter, F. (2003b). From tableaux to automata for description logics. *Fundamenta Informaticae*, 57(2-4):247–279.

[Baader and Hollunder, 1995] Baader, F. and Hollunder, B. (1995). Embedding defaults into terminological representation systems. *Journal of Automated Reasoning*, 14(2):149–180.

[Baader and Sattler, 2001] Baader, F. and Sattler, U. (2001). An overview of tableau algorithms for description logics. *Studia Logica*, 69(1):5–40.

[Baral, 2002] Baral, C. (2002). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

[Baral and Gelfond, 1994] Baral, C. and Gelfond, M. (1994). Logic programming and knowledge representation. *J. Logic Program.*, 19 & 20(Suppl. 1):73–148.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Sci. Am.*, 284(5):34–43.

[Berrueta et al., 2011] Berrueta, D., Korf, R., Kiss, E. M., Hoppenbrouwers, J., Nijssen, S., Nazarenko, A., Ghali, A. E., Citeau, H., and de Sainte Marie, C. (2011). D6.1 - Specification of the ONTORULE platform. Technical report, ONTORULE IST-2009-231875 Project.

[Bonatti et al., 2006] Bonatti, P., Lutz, C., and Wolter, F. (2006). Expressive non-monotonic description logics based on circumscription. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 400–410.

[Bonatti et al., 2008] Bonatti, P., Pontelli, E., and Son, T. C. (2008). Credulous resolution for answer set programming. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)*, pages 418–423. AAAI.

[Bonatti, 2011] Bonatti, P. A. (2011). On the decidability of $\mathbb{FDNC}$ programs. *Intelligenza Artificiale*, 5(1):89–93.

[Calì et al., 2008] Calì, A., Gottlob, G., and Kifer, M. (2008). Taming the infinite chase: Query answering under expressive relational constraints. In Baader, F., Lutz, C., and Motik, B., editors, *Description Logics'08*, volume 353. CEUR-WS.org.

[Calì et al., 2009] Calì, A., Gottlob, G., and Lukasiewicz, T. (2009). Datalog$\pm$: A unified approach to ontologies and integrity constraints. In *Proceedings of the International Conference on Database Theory ICDT*, volume 9, pages 14–30.

[Calì et al., 2010a] Calì, A., Gottlob, G., and Pieris, A. (2010a). Advanced processing for ontological queries. *Proceedings of the VLDB Endowment*, 3(1):554–565.

[Calì et al., 2010b] Calì, A., Gottlob, G., and Pieris, A. (2010b). Query answering under non-guarded rules in Datalog$\pm$. In *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems (RR 2010)*, pages 1–17.

[Calvanese et al., 2007a] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007a). Eql-Lite: Effective first-order query processing in description logics. In *Proceedings of IJCAI'2007*, pages 274–279.

[Calvanese et al., 2007b] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007b). Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *JAR*, 39(3):385–429.

[Clark, 1978] Clark, K. L. (1978). Negation as failure. In Minker, J., editor, *Logic and Data Bases*, volume 1, pages 293–322. Plenum Press, New York, London.

[de Bruijn, 2009] de Bruijn, J. (2009). D3.1 - State-of-the-art survey of issues. Technical report, ONTORULE IST-2009-231875 Project.

[de Bruijn et al., 2006] de Bruijn, J., Eiter, T., Polleres, A., and Tompits, H. (2006). On representational issues about combinations of classical theories with nonmonotonic rules. In *Knowledge Science, Engineering and Management, First International Conference, KSEM 2006, Guilin, China, August 5-8, 2006, Proceedings*, volume 4092 of *Lecture Notes in Computer Science*, pages 1–22. Springer.

[De Giacomo et al., 1996] De Giacomo, G., Donini, F. M., and Massacci, F. (1996). EXPTIME tableaux for $\mathcal{ALC}$. In Padgham, L., Franconi, E., Gehrke, M., McGuinness, D. L., and Patel-Schneider, P. F., editors, *Description Logics*, volume WS-96-05 of *AAAI Technical Report*, pages 107–110. AAAI Press.

[Dean and Schreiber, 2004] Dean, M. and Schreiber, G. (2004). OWL web ontology language reference. W3C recommendation, W3C.

[Dimopoulos et al., 1997] Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. In Steel, S. and Alami, R., editors, *Recent Advances in AI Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 169–181.

[Donini et al., 1998] Donini, F., Lenzerini, M., Nardi, D., and Schaerf, A. (1998). AL-log: Integrating Datalog and description logics. *Journal of Intelligent and Cooperative Information Systems*, 10:227–252.

[Donini and Massacci, 2000] Donini, F. and Massacci, F. (2000). EXPTIME tableaux for $\mathcal{ALC}$. *Journal of Artificial Intelligence*, 124(1):87–138.

[Donini et al., 2002] Donini, F. M., Nardia, D., and Rosati, R. (2002). Description logics of minimal knowledge and negation as failure. *ACM Trans. on Comput. Logic*, 3(2):177–225.

[Eiter et al., 2007] Eiter, T., Faber, W., Fink, M., and Woltran, S. (2007). Complexity Results for Answer Set Programming with Bounded Predicate Arities. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165.

[Eiter et al., 2004] Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. (2004). A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Logic*, 5(2):206–263.

[Eiter et al., 2011] Eiter, T., Ianni, G., Lukasiewicz, T., and Schindlauer, R. (2011). Well-founded semantics for description logic programs in the semantic web. *ACM Transactions in Computational Logic*, 12(2).

[Eiter et al., 2008] Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., and Tompits, H. (2008). Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539.

[Eiter et al., 2012] Eiter, T., Ortiz, M., and Šimkus, M. (2012). Conjunctive query answering in the description logic $\mathcal{SH}$ using knots. *Journal of Computer and System Sciences*, 78(1):47–85.

[Eiter and Šimkus, 2009] Eiter, T. and Šimkus, M. (2009). Bidirectional answer set programs with function symbols. In Boutilier, C., editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 765–771.

[Faber et al., 2004] Faber, W., Leone, N., and Pfeifer, G. (2004). Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings JELIA-2004*, volume 3229 of *LNCS/LNAI*, pages 200–212. Springer.

[Fages, 1991] Fages, F. (1991). A new fix point semantics for generalized logic programs compared with the well-founded and the stable model semantics. *New Generation Computing*, 9(4).

[Fages, 1994] Fages, F. (1994). Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1(1):51–60.

[Feier, 2012] Feier, C. (2012). Worst-case optimal reasoning with forest logic programs. In Brewka, G., Eiter, T., and McIlraith, S. A., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14*.

[Feier and Heymans, 2008] Feier, C. and Heymans, S. (2008). A sound and complete algorithm for simple conceptual logic programs. In *Proceedings of the 3rd International Workshop on Applications of Logic Programming to the (Semantic) Web and Web Services (ALPSWS2008), co-located with the 24th International Conference on Logic Programming (ICLP) Udine, Italy, December 12, 2008*, volume 434 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Feier and Heymans, 2009] Feier, C. and Heymans, S. (2009). Hybrid reasoning with forest logic programs. In *Proceedings of 6th Annual European Semantic Web Conference (ESWC 2009)*, volume 5554, pages 338–352. Springer.

[Feier and Heymans, 2010] Feier, C. and Heymans, S. (2010). An optimization for reasoning with forest logic programs. In *Proceedings of 3rd International Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP), July 20th, 2010, Edinburgh.*

[Feier and Heymans, 2013] Feier, C. and Heymans, S. (2013). Reasoning with forest logic programs and f-hybrid knowledge bases. *TPLP*, 3(13):395–463.

[Furbach et al., 2009] Furbach, U., Günther, H., and Obermaier, C. (2009). A Knowledge Compilation Technique for $\mathcal{ALC}$ TBoxes. In *Proc. of the Twenty-Second International Florida Artificial Intelligence Research Society Conference, May 19-21, 2009, Sanibel Island, Florida, USA*.

[Gebser and Schaub, 2006] Gebser, M. and Schaub, T. (2006). Tableau calculi for answer set programming. In *Proc. of 22nd Int. Conf. on Logic Programming (ICLP)*, volume 4079 of *LNCS*, pages 11–25. Springer.

[Gelder et al., 1991] Gelder, A. V., Ross, K., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *Proc. of ICLP'88*, pages 1070–1080.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385.

[Giunchiglia et al., 2006] Giunchiglia, E., Lierler, Y., and Maratea, M. (2006). Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377.

[Goré, 2009] Goré, R. (2009). Global caching, inverse roles and fixpoint logics. In Grau, B. C., Horrocks, I., Motik, B., and Sattler, U., editors, *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*, volume 477 of *CEUR Workshop Proceedings*.

[Goré and Nguyen, 2007] Goré, R. and Nguyen, L. (2007). EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 4548 of *Lecture Notes in Computer Science*, pages 133–148. Springer.

[Goré and Nguyen, 2013] Goré, R. and Nguyen, L. (2013). EXPTIME tableaux for $\mathcal{ALC}$ using sound global caching. *Journal of Automated Reasoning*, 50(4):355–381.

[Goré and Widmann, 2009] Goré, R. and Widmann, F. (2009). Sound global state caching for $\mathcal{ALC}$ with inverse roles. In Giese, M. and Waaler, A., editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 5607 of *Lecture Notes in Computer Science*, pages 205–219. Springer.

[Goré and Nguyen, 2008] Goré, R. and Nguyen, L. A. (2008). Sound global caching for abstract modal tableaux. In et al., G. L., editor, *Proceedings of Concurrency, Specification and Programming (CS&P'2008)*, pages 157–167.

[Gottlob and Lukasiewicz, 2009] Gottlob, A. C. G. and Lukasiewicz, T. (2009). A general Datalog-based framework for tractable query answering over ontologies. In *In Proc. PODS-2009*, pages 77–86. ACM Press.

[Grädel, 2003] Grädel, E. (2003). Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In *Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged*.

[Grädel and Walukiewicz, 1999] Grädel, E. and Walukiewicz, I. (1999). Guarded Fixed Point Logic. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99, Trento*, pages 45–54.

[Grimm and Hitzler, 2007] Grimm, S. and Hitzler, P. (2007). Reasoning in Circumscriptive $\mathcal{ALCO}$. Technical report, FZI at University of Karlsruhe, Germany.

[Grimm and Hitzler, 2008] Grimm, S. and Hitzler, P. (2008). Defeasible Inference with Circumscriptive OWL Ontologies. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*.

[Grimm and Hitzler, 2009] Grimm, S. and Hitzler, P. (2009). A preferential tableaux calculus for circumscriptive $\mathcal{ALCO}$. In Polleres, A. and Swift, T., editors, *Proceedings of the 3rd*

International Conference on Web Reasoning and Rule Systems (RR 2009), volume 5837, pages 40–54. Springer.

[Grosof et al., 2003] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. WWW 2003*, pages 48–57. ACM.

[Heymans, 2006] Heymans, S. (2006). *Decidable Open Answer Set Programming*. PhD thesis, Theoretical Computer Science Lab (TINF), Department of Computer Science, Vrije Universiteit Brussel.

[Heymans et al., 2009] Heymans, S., Feier, C., and Eiter, T. (2009). A reasoner for simple conceptual logic programs. In Polleres, A. and Swift, T., editors, *Proc. of the 3rd Int. Conf. on Web Reasoning and Rule Systems (RR 2009)*, volume 5837, pages 55–70. Springer.

[Heymans et al., 2006] Heymans, S., Van Nieuwenborgh, D., and Vermeir, D. (2006). Conceptual logic programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)*, 47(1–2):103–137.

[Heymans et al., 2007] Heymans, S., Van Nieuwenborgh, D., and Vermeir, D. (2007). Open answer set programming for the semantic web. *Journal of Applied Logic*, 5(1):144–169.

[Heymans et al., 2008] Heymans, S., Van Nieuwenborgh, D., and Vermeir, D. (2008). Open answer set programming with guarded programs. *Transactions on Computational Logic*, 9(4):1–53.

[Hladik, 2007] Hladik, J. (2007). *To and Fro between Tableaus and Automata for Description Logics*. PhD thesis, Dresden University of Technology.

[Hladik and Sattler, 2003] Hladik, J. and Sattler, U. (2003). A translation of looping alternating automata into description logics. In Baader, F., editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19), Miami Beach, FL, USA*, volume 2741 of *Lecture Notes in Computer Science*, pages 90–105. Springer.

[Horrocks, 2003] Horrocks, I. (2003). Implementation and optimisation techniques. In Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press.

[Horrocks and Patel-Schneider, 2004] Horrocks, I. and Patel-Schneider, P. F. (2004). A Proposal for an OWL Rules Language. In *Proc. of the World Wide Web Conference (WWW)*, pages 723–731. ACM.

[Horrocks and Sattler, 2001] Horrocks, I. and Sattler, U. (2001). Ontology reasoning in the SHOQ(D) description logic. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence*.

[Horrocks et al., 1999] Horrocks, I., Sattler, U., and Tobies, S. (1999). Practical reasoning for expressive description logics. In *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in LNCS, pages 161–180. Springer.

[Horrocks et al., 2000] Horrocks, I., Sattler, U., and Tobies, S. (2000). Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 482–496, London, UK. Springer-Verlag.

[Kazakov, 2008] Kazakov, Y. (2008). $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 274–284. AAAI Press.

[Kifer, 2008] Kifer, M. (2008). Rule interchange format: The framework. In Calvanese, D. and Lausen, G., editors, *Web Reasoning and Rule Systems*, volume 5341 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg.

[Kifer and Boley, 2010] Kifer, M. and Boley, H. (2010). RIF overview. Working Group Note, 22nd of july.

[Krötzsch et al., 2008a] Krötzsch, M., Rudolph, S., and Hitzler, P. (2008a). Description logic rules. In *Proc. ECAI*, pages 80–84. IOS Press.

[Krötzsch et al., 2008b] Krötzsch, M., Rudolph, S., and Hitzler, P. (2008b). ELP: Tractable rules for OWL 2. In *Proc. 7th Int. Semantic Web Conf. (ISWC-08)*, pages 649–664.

[Lee and Lifschitz, 2003] Lee, J. and Lifschitz, V. (2003). Loop formulas for disjunctive logic programs. In Palamidessi, C., editor, *Proc. of International Conference of Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 451–465.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562.

[Levy and Rousset, 1996] Levy, A. Y. and Rousset, M. (1996). CARIN: A representation language combining Horn rules and description logics. In *Proc. of ECAI'96*, pages 323–327.

[Lierler, 2008] Lierler, Y. (2008). Abstract answer set solvers. In *Proc. of the 26th Int. Conf. on Logic Programming (ICLP)*, pages 377–391.

[Lifschitz, 2008] Lifschitz, V. (2008). What is Answer Set Programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17*, pages 1594–1597. AAAI Press.

[Lifschitz and Turner, 1994] Lifschitz, V. and Turner, H. (1994). Splitting a logic program. In *Proceedings ICLP-1994*, pages 23–38. MIT Press.

[Lifschitz and Woo, 1992] Lifschitz, V. and Woo, T. Y. C. (1992). Answer sets in general nonmonotonic reasoning (preliminary report). In Nebel, B., Rich, C., and Swartout, W. R., editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, October 25-29*, pages 603–614. Morgan Kaufmann.

[Lin and You, 2002] Lin, F. and You, J. (2002). Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artificial Intelligence*, 140:175–205.

[Lin and Zhao, 2002] Lin, F. and Zhao, Y. (2002). ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. of 18th National Conf. on Artificial Intelligence*, pages 112–117. AAAI.

[Lin and Zhao, 2004] Lin, F. and Zhao, Y. (2004). ASSAT: computing answer sets of a logic program by SAT solvers. *Journal of Artificial Intelligence*, 157(1–2):115 – 137.

[Lloyd, 1987] Lloyd, J. W. (1987). *Foundations of Logic Programming; (2nd Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA.

[Lukasiewicz, 2004] Lukasiewicz, T. (2004). A novel combination of answer set programming with description logics for the semantic web. In *In Proceedings of KR-2004*, pages 141–151. AAAI Press.

[Lukasiewicz et al., 2012] Lukasiewicz, T., Calì, A., and Gottlob, G. (2012). A general Datalog-based framework for tractable query answering over ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14(0).

[Magka et al., 2013] Magka, D., Krötzsch, M., and Horrocks, I. (2013). Computing stable models for nonmonotonic existential rules. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13), Beijing, China, August 3-9*. AAAI Press/IJCAI.

[Magka et al., 2012] Magka, D., Motik, B., and Horrocks, I. (2012). Modelling structured domains using Description Graphs and logic programming. In *Proceedings of the 9th Extended Semantic Web Conference*, pages 330–344, Berlin, Heidelberg. Springer-Verlag.

[Marek, 1999] Marek, V. W. (1999). Stable models and an alternative logic programming paradigm. In *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag.

[Marek and Truszczynski, 1989] Marek, V. W. and Truszczynski, M. (1989). Stable semantics for Logic Programs and default theories. In Lusk, E. L. and Overbeek, R. A., editors, *NACLP*, pages 243–256. MIT Press.

[Minsky, 1985] Minsky, M. (1985). A Framework for Representing Knowledge. In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 245–262. Kaufmann, Los Altos, CA.

[Motik et al., 2009a] Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., and Lutz, C., editors (27 October 2009a). *OWL 2 Web Ontology Language: Profiles*. W3C Recommendation. Available at `http://www.w3.org/TR/owl2-profiles/`.

[Motik et al., 2006] Motik, B., Horrocks, I., Rosati, R., and Sattler, U. (2006). Can OWL and logic programming live together happily ever after? In *Proc. of the Int. Semantic Web Conf. (ISWC)*, volume 4273 of *LNCS*, pages 501–514. Springer.

[Motik and Rosati, 2006] Motik, B. and Rosati, R. (2006). Closing semantic web ontologies. Technical report, University of Manchester, UK.

[Motik and Rosati, 2010] Motik, B. and Rosati, R. (2010). Reconciling description logics and rules. *Journal of the ACM*, 57(5):30:1–30:62.

[Motik et al., 2005] Motik, B., Sattler, U., and Studer, R. (2005). Query answering for OWL-DL with rules. *Journal of Web Semantics*, 3(1):41–60.

[Motik et al., 2009b] Motik, B., Shearer, R., and Horrocks, I. (2009b). Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, 36(1):165–228.

[Niemelä, 1999] Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273.

[Nieuwenhuis et al., 2006] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2006). Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977.

[Nogueira et al., 2001] Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. (2001). An A-Prolog decision support system for the Space Shuttle. In *Proceedings of Practical Aspects of Declarative Languages*, pages 169–183. Springer.

[OWL 2, 2009] OWL 2 (2009). OWL 2 web ontology language structural specification and functional-style syntax. Recommendation 27 October 2009, W3C.

[Przymusinski, 1991] Przymusinski, T. C. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424.

[Reiter, 1978] Reiter, R. (1978). On Closed World Data Bases. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York.

[Rosati, 2006] Rosati, R. (2006). DL+log: Tight integration of description logics and disjunctive Datalog. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 68–78.

[Rosati, 2008] Rosati, R. (2008). On combining description logic ontologies and nonrecursive datalog rules. In *Proc. of the 2nd Int. Conf. on Web Reasoning and Rule Systems (RR 2008)*.

[Schild, 1991] Schild, K. (1991). A correspondence theory for terminological logics: Preliminary report. In Mylopoulos, J. and Reiter, R., editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI 1991)*, pages 466–471. Morgan Kaufmann.

[Simons et al., 2002] Simons, P., Niemelá, I., and Soininen, T. (2002). Extending and implementing the stable model semantics. *Journal of Artificial Intelligence*, 138(1-2):181–234.

[Soininen and Niemelä, 1999] Soininen, T. and Niemelä, I. (1999). Developing a declarative rule language for applications in product configuration. In *Proceedings of Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer.

[Subrahmanian and Zaniolo, 1995] Subrahmanian, V. S. and Zaniolo, C. (1995). Relating stable models and AI planning domains. In *Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16*, pages 233–247. MIT Press.

[Tiihonen et al., 2003] Tiihonen, J., Soininen, T., Niemelä, I., and Sulonen, R. (2003). A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED'03)*, pages 1290–1299.

[Tobies, 2001] Tobies, S. (2001). *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD thesis, RWTH-Aachen.

[Vardi, 1998] Vardi, M. Y. (1998). Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloquium on Automata, Languages and Programming*, pages 628–641. Springer.

[Šimkus and Eiter, 2007] Šimkus, M. and Eiter, T. (2007). $\mathbb{FDNC}$: Decidable non-monotonic disjunctive logic programs with function symbols. In *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007).*

[Wang, 1961] Wang, H. (1961). Proving theorems by pattern recognition II. *Bell System Technical Journal*, 40:1–42.