Practical Reasoning with Transaction Logic Programming for Knowledge Base Dynamics

A DISSERTATION PRESENTED

BY

PAUL FODOR

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

May 2011

Stony Brook University
The Graduate School

Paul Fodor

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

Professor Michael Kifer, Advisor
Department of Computer Science

Professor Yanhong Annie Liu, Chairman of Defense
Department of Computer Science

Professor David S. Warren
Department of Computer Science

Dr. Christopher A. Welty, External
IBM Watson Research Center, Hawthorne, NY, USA

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Practical Reasoning with Transaction Logic Programming for Knowledge Base Dynamics

by

Paul Fodor

Doctor of Philosophy

in

Computer Science

Stony Brook University

2011

Transaction Logic is an extension of classical predicate calculus for representing declarative and procedural knowledge in logic programming, databases, and artificial intelligence. Since it provides a logical foundation for the phenomenon of state changes, it has been successful in areas as diverse as workflows, planning, reasoning about actions, Web services, security policies, active databases and more. Although a number of implementations of Transaction Logic exist, none is logically complete due to the time and space complexity of such implementations.

In the first part of this thesis, we describe an approach for performing actions in the logic, which has better complexity and termination properties via a logically complete tabling evaluation strategy. Then we describe a series of optimizations, which make this algorithm practical and analyze their performance on a set of benchmarks. Our performance evaluation study shows that the tabling algorithm can scale well both in time and space.

In the second part of the thesis, we extend Transaction Logic in the direction of defeasible reasoning, which has a number of interesting applications, including specification of defaults in action theories and heuristics for directed search in planning. In this setting we show that heuristics expressed as defeasible actions can significantly reduce the search space and thus the execution time and space requirements.

To my family and teachers

Contents

List of Figures vi

List of Tables viii

Acknowledgements ix

1 Introduction 1

2 Preliminaries 4

2.1 Transaction Logic 4

2.1.1 Serial Transaction Logic Syntax 5

2.1.2 Transaction Logic Semantics 8

2.1.3 A Proof Theory for the serial-Horn Transaction Logic 12

2.2 Tabled Logic Programming 14

2.3 Defeasible Reasoning 15

3 Tabling for Transaction Logic 17

3.1 Tabling for Definite Serial Horn-Transaction Logic Programs 19

3.1.1 Tabled-$\mathcal{TR}$ derivation trees 23

3.1.1.1 A Step-by-step Tabling Example for Definite Serial Horn Transaction Logic 25

3.2 Problems and Solutions in Implementing Tabled Transaction Logic . 28

3.2.1 Main Difficulties with Implementing Tabled Transaction Logic 28

3.2.2 The Space of Possible Solutions 30

3.2.2.1 Space issues in tabled $\mathcal{TR}$ 30

3.2.2.2 Time issues in tabled $\mathcal{TR}$ 33

3.3 Implementations of Tabled Transaction Logic 36
3.4 Applications and performance evaluation 37
3.4.1 Hamiltonian cycles 37
3.4.2 Artificial Intelligence planning in the blocks world 38
3.4.3 Evaluation for tabled $\mathcal{TR}$ implementations 40
3.5 Tabling for Concurrent Transaction Logic Programs 44

4 A Well-founded Semantics for Transaction Logic with Defaults and Argumentation Theories 59
4.1 Defeasibility in Transaction Logic 60
4.1.1 $\mathcal{TR}^{DA}$ Syntax 60
4.1.2 $\mathcal{TR}^{DA}$ Well - founded Semantics 61
4.2 Argumentation theory representatives 68
4.2.1 The $GCLP^{\mathcal{TR}}$ courteous argumentation theory 68
4.2.2 An argumentation theory for defeasible logic 71
4.3 $\mathcal{TR}^{DA}$ discussion and related work 73
4.4 Applications, implementation and evaluation 76
4.4.1 $\mathcal{TR}^{DA}$ Aplications in action priorities, planning and workflows 76
4.4.2 $\mathcal{TR}^{DA}$ Evaluation 82

5 Conclusion and future work 85

A Application of Transaction Logic in CEP 98

B Appendix: Tabled $\mathcal{TR}$ Soundness and Completeness 104

C Tabled $\mathcal{TR}$ Termination 110

D Unique Least Model for `not`-free $\mathcal{TR}$ Programs 113

E $\mathcal{TR}^{DA}$ Fixpoint and Well-founded Model 116

F $\mathcal{TR}^{DA}$ Reduction to Transaction Logic 124

List of Figures

1 An initial graph for the consuming paths reachability example 18

2 SLD-style tree for the query $reach(a, X)$ in the consuming paths example with an infinite derivation branch 18

3 The tabled resolution tree at step 2 for the query $reach(a, X)$ in the consuming paths example 25

4 The tabled resolution tree at step 3 for the query $reach(a, X)$ in the consuming paths example 26

5 The tabled resolution tree at step 7 for the query $reach(a, X)$ in the consuming paths example 26

6 The tabled resolution tree at step 11 for the query $reach(a, X)$ in the consuming paths example 27

7 The tabled resolution tree at step 12 for the query $reach(a, X)$ in the consuming paths example 27

8 The tabled resolution tree at step 14 for the query $reach(a, X)$ in the consuming paths example 28

9 Rule trie example 32

10 State trie example 32

11 Part of the resolution tree for the Concurrent $\mathcal{TR}$ Example 3.3 49

12 A successful branch in the resolution tree for the Concurrent $\mathcal{TR}$ Example 3.3 51

13 Tabling of only hot components for the Concurrent $\mathcal{TR}$ Example 3.3 51

14 The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.3 53

15 The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.4 53

16 A transaction workflow example for defeasible reasoning in $\mathcal{TR}$ . . . 81

List of Tables

1 A set of states saved during the tabling algorithm 31
2 Times for finding consuming paths in graphs 55
3 Numbers of tabled states and state comparisons for finding consuming paths in graphs 55
4 Time and space for building 10 consuming paths in 10 graphs 55
5 Numbers of tabled states and state comparisons for building 10 consuming paths in 10 graphs 56
6 Times for finding Hamiltonian cycles in graphs 56
7 Numbers of tabled states and state comparisons for finding Hamiltonian cycles 56
8 Times for finding 10 Hamiltonian cycles in 10 graphs 57
9 Time and space requirements for building pyramids of N blocks in blocks worlds 57
10 Numbers of tabled states and state comparisons for building pyramids in blocks worlds 57
11 Time and space requirements for building pyramids of N blocks in 10 parallel blocks worlds 58
12 $\mathcal{CTR}$ formulas and their hot components 58
13 Time, space, tabled states and state comparisons for planning in the blocks world with and without preferential heuristics 84

Acknowledgements

I would like to thank my thesis adviser, Professor Michael Kifer, for his generous and continuous support, guidance and lessons throughout my graduate studies. His enthusiasm, technical approach into research problems and dedication for teaching were important examples for me on how to do research. He gave me freedom to work alone and explore different areas which helped me grow in confidence, but also he kept a close guidance on my work channeling it into rule-based declarative languages. Without his lessons on research and commitment to intellectual development it would not have been possible to write this thesis. I would also like to thank the other members of my committee: Professor David Warren for everything I learned about Prolog and WAM from him during these years, Professor Annie Yauhong Liu for invaluable lessons about research in programming languages and to Dr. Christopher Welty for agreeing to be my external committee member and his patience with my progression on this thesis.

I would like to sincerely thank all those who gave me support and encouragement to complete this thesis. My sincere thanks to all my teachers at Stony Brook, including Professor C.R. Ramakrishnan for great lessons into computation with logic and programming languages, Professor Steven Skiena for infusing his in-depth knowledge and interest in algorithms into me during his great course on Computational Biology, Professor Radu Grosu for teaching a great and interesting course on compilers, Professor Radu Sion for introducing me to computer security, Professor Tziker Chiueh for getting me ready for a Ph.D. with a marathon in Computer Networks and an interesting project in the first semester when I arrived at Stony Brook, and to Prof. I.V. Ramakrishnan for his coordination of the lab and useful tips to me about almost everything. I would also like to extend a thank you to the systems and administrative staff at the department including Brian Tria, Cynthia Scalzo, Betty Knitweiss, and

Kathy Germana.

I was fortunate enough to work with some very hardworking and imaginative fellow graduate students during these last few years: Hui Wan, Senlin Liang, Tuncay Tekle, Diptikalyan Saha, Chang Zhao and Reza Basseda. This material is based on work supported in part by Dr. Benjamin Grosof from Vulcan Inc. I would like to thank him and to Mike Dean for great discussions on logic programming and great brain storming face to face meetings. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of any company.

It was my pleasure to interact and collaborate with some fine people outside Stony Brook: Dr. Terrance Swift, Darko Anicic, Dr. Sebastian Rudolph, Dr. Jurgen Angele, Dave Gunning, Peter Clark, Dr. Michael Gelfond, Roland Stuhmer, Jurgen Baier, Imad Abdallah, Daniela Inclezan, Dumitru Roman and Martin Rezk. I worked with lots of excellent researchers during my summers in IBM, including: Dr. David Ferrucci, Dr. David Lubensky, Dr. Juan Huerta, Dr. Eric Brown, Dr. Jennifer Chu-Carroll, Adam Lally, Dr. John Prager, Dr. Branimir Boguraev, Dr. Liwei Dai, Dr. James William Murdock, Dr. Pablo Duboue, and many others.

It was a great pleasure to make wonderful friends during my studies making this journey enjoyable and eventful: Joy Dutta, Anu Singh, Rahul Agarwal, Beata Sarna-Starosta, Amit Sasturkar, Song Feng, Corina Weidinger, Cristina Turdean, Steve Giovino, Marcy Newton, Dave Weidinger, Oana Tudor, Dan Cojocaru, Abhishek Rai, Gang Wu, Alok Tongaonkar, Andrei Todor, Ionel Ovidiu Patu, Bo Lin, Eli Packer, Ibtisam Ul Haque, Simona Boboila, Xianjin Zhu, Xin Li, Svetlana Stenchikova, Jae-hyuk Her, Jalal Mahmud, Yevgen Borodin, Yue Wang, Yury Puzis, Zan Sun, Katia Hristova, Anand Kashyap, Saikat Mukherjee, Ambrish Tyagi, Steven Rennie, Ravi Vijaya Satya, Ghinwa Choueiter, Sameer Maskey, and many more.

I would not have gone this far without the continuous love, support and encouragement from my great family, especially my parents, my wife Andreea, Andreea's parents, my brother Petru Fodor, my sister in law Alina Lazar and the Young family. It is futile for me to even begin describing what I owe to each of them.

Chapter 1

Introduction

Transaction logic (abbr., $\mathcal{TR}$) [BK93, BK94b, BK98c] is a general logic for representing knowledge base dynamics. Its model and proof theories cleanly integrate declarative and procedural knowledge and, as a result, the logic has been employed in domains ranging from reasoning about actions ([BK98a, Bon97]), to AI planning ([BK95, Fod09]), knowledge representation ([BK94a]), event processing ([AFSS09b]), workflow management and Semantic Web services ([DKRR98, DKR04, RK07, RK08]), security policy frameworks [BN07], and general knowledge base programming ([BK98b]). In logic programming, $\mathcal{TR}$ provides a clean, logical alternative to the *assert* and *retract* operators of Prolog, while, in databases, $\mathcal{TR}$ is a declarative language for programming transactions, for updating database views, and for specifying active rules. Moreover, in AI, $\mathcal{TR}$ can be used for representing procedural knowledge, planning, hypothetical reasoning, subjunctive queries and counterfactuals.

A couple of implementations of $\mathcal{TR}$ exist [Hun96a, Hun96b, Sle00, F.S00, YKZ03, Kif] but, unfortunately, all are logically incomplete. The major barrier to completeness for these implementations is similar to the reasons for Prolog incompleteness: the computation is based on an SLD-like resolution procedure with a depth - first goal selection strategy. This problem has been studied extensively in the logic programming literature [TS86, CW96], and this led to the development of *tabling* (or *memoing*)— an efficient algorithm for logically complete implementation of logic programs based on SLD resolution [War92, SW94]. The best known implementation of

tabling is XSB,[1] but there are others, such as Yap,[2] B-Prolog,[3] and Mercury.[4]

The success of this tabled technique in Prolog makes it a natural candidate for solving the analogous problems in Transaction Logic. The major difference in $\mathcal{TR}$ is that the latter deals with the phenomenon of *changing states*, which is not an issue in XSB and similar systems, where state changes are viewed as a non - logical feature that is best left outside of the scope of the tabling mechanism. In contrast, state updates have both *model - theoretic* and *procedural* semantics in Transaction Logic, and their correct treatment is essential.

The first part of the thesis will be about extending Transaction Logic with the tabling algorithm (published in [FK10b]). The issue is that tabling for $\mathcal{TR}$ requires memoing of the underlying database state and not just memoing of the previously called subgoals. Clearly, this is a major problem both in terms of space and time. Of course, a powerful formalism such as Transaction Logic does not come without a price, but our contribution is in showing that there is ample room for optimization. After describing the extended tabling algorithm, we discuss the major trade - offs in its implementation and propose several time and space optimizations. We implemented a dozen of algorithms, which combine our optimizations in various ways. Here we discuss six of those that illustrate the most salient points. We discuss the rationale behind each of them, and then present our experimental results. These results show that a proper integration of our techniques results in a system with the best overall performance and scalability characteristics.

We are not aware of any work that directly deals with problems similar to ours. However, we are building on a host of results, which became ingredients in our optimization techniques or could be used for further optimization. These include the already mentioned works on tabling, the various indexing data structures, such as B^+ trees and other balanced trees (like AVL, Red-Black, and 23-trees), tries, sets, and others [Com79, GS78, SRV01, Pon92, DPR96, Liu98].

Defeasible reasoning is another important paradigm, which has been extensively studied as a knowledge representation paradigm, including in fields such as policies,

[1] `http://xsb.sourceforge.net`
[2] `http://www.dcc.fc.up.pt/~vsc/Yap`
[3] `http://www.probp.com`
[4] `http://www.cs.mu.oz.au/mercury`

regulations, law, learning, and others [BH95, BE99, BE00, DST03, DS01, EFLP03, GS98, Gro99, Nut94, Pra93, SI00, WZL00, ZWB01]. In the second part of the thesis, we will combine $\mathcal{TR}$ with defeasible reasoning (published in [FK11]) and show that the resulting logic language has many important applications. This logic is called $\mathcal{TR}^{DA}$ (*Transaction Logic with Defaults and Argumentation Theories*) and it extends $\mathcal{TR}$ in the direction of the *Logic Programming with Defaults and Argumentation theories* (LPDA) [WGK+09b], a unifying framework for defeasible reasoning that we proposed recently. Along the way we define the *well - founded semantics* [VRS91] for $\mathcal{TR}$, which allows the computation of three valued models for Transaction Logic programs that use the negation - as - failure operator over actions. The combined logic enables a number of interesting applications, such as specification of defaults in action theories and heuristics for pruning search in such search - intensive applications and planning. We also demonstrate the usefulness of the approach by experimenting with a prototype of $\mathcal{TR}^{DA}$ and showing how heuristics expressed as defeasible actions can significantly reduce the search space as well as execution time and space requirements.

Apart from the contributions in the main part of the thesis we also worked on applications of $\mathcal{TR}$ in the complex event processing domain. Results for these applications are presented in the Appendix A.

Chapter 2

Preliminaries

In this chapter we present the basic notions and definitions used in the rest of this thesis. In Section 2.1 we present the general logic of state change for deductive databases and logic programs named Transaction Logic, its model theory, its Horn subset, a specialized proof theory and procedural interpretation. In Section 2.2, we shortly describe the classical technique of tabling applied in our new implementation of $\mathcal{TR}$. In Section 2.3, we present defeasible reasoning for logic programming and our former work on Logic Programming with Defaults and Argumentation theories (LPDA), work that sits at the base of the new extension to Horn - $\mathcal{TR}$, called Transaction Logic with Defaults and Argumentation theories ($\mathcal{TR}^{DA}$).

2.1 Transaction Logic

Transaction logic ($\mathcal{TR}$) ([BK93, BK94b, BK98c]) is a general logic of state change, which extends classical first - order logic with new connectives that make it suitable for representing both procedural and declarative knowledge. The alphabet of the language $\mathcal{L}_{\mathcal{TR}}$ of general $\mathcal{TR}$ is similar to that of first-order logic: a countably-infinite set of variables $\mathcal{V}$, a countably-infinite set of function symbols $\mathcal{F}$ (where each functor f has an arity and constants are 0-ary function symbols), a countably-infinite set of predicate symbols $\mathcal{P}$, logical connectives (disjunction $\vee$, classical conjunction $\wedge$, serial conjunction $\otimes$, classical negation $\mathtt{neg}$, default negation $\mathtt{not}$, concurrent conjunction $|$, isolation $\odot$), and the quantifiers $\forall$ and $\exists$. We will describe the operands and we will

define formulas in $\mathcal{TR}$ later in this thesis for various subsets of the language. One the most important contribution of $\mathcal{TR}$ is that $\mathcal{TR}$ comes with a pair of oracles, one called the *data oracle* $\mathcal{O}^d$ which specifies the *static* semantics of states and one called *transition oracle* which specifies the *dynamic* semantics of states (or *updates*).

2.1.1 Serial Transaction Logic Syntax

In this thesis we use a subset of Transaction Logic called serial Horn-$\mathcal{TR}$. This subset is interesting because it is sufficiently expressive for many applications, including planning, workflow management, and action languages [BK95, BK98a, Bon97, BK94a, DKRR98, DKR04, RK07, RK08].

The syntax of Horn-$\mathcal{TR}$ is derived from that of standard Horn logic programming. As described above, the alphabet of the language $\mathcal{L}_{\mathcal{TR}}$ of $\mathcal{TR}$ contains an infinite number of constants, function symbols, predicate symbols, and variables. The *atomic formulas* have the form $p(t_1, ..., t_n)$, where p is a predicate symbol, and t_i are terms (variables, constants, function terms). However, unlike standard logic programming, predicate symbols are partitioned into *fluents* and *actions*. Fluents are predicates whose execution does not change the state of the database, while actions are predicates that can change the state of the database. Fluents are further partitioned into *base fluents* and *derived fluents*. Base fluents correspond to the classical base predicates in relational databases; they represent stored data and are inserted and deleted in the database. Derived fluents correspond to derived predicates, which represent database views. An atomic formula $p(t_1, ..., t_n)$ will be also called a *fluent* or an *action atomic formula* if p is a fluent or an action symbol, respectively. Furthermore, if p is a derived (respectively, base) fluent symbol then $p(t_1, ..., t_n)$ is a derived (respectively, base) fluent atomic formula. An expression is called *ground* if it does not contain any variables.

The symbol `neg` will be used to represent the explicit negation (also called "strong" negation) and `not` will be used for default negation, that is, negation as failure. A *fluent literal* is either an atomic fluent or has one of the following negated forms:

- `neg` atm, `not` atm, `not neg` atm,

where *atm* is a fluent atomic formula, An *action literal* is an action atomic formula or has the form $\texttt{not}\,\alpha$, where α is a action atomic formula. Literals of the form $\texttt{neg}\,\alpha$ are not allowed.

A *database state* is a set of ground base fluents. All database states are assumed to be *consistent*, meaning that it is not possible for both f and $\texttt{neg}\,f$ belong to the same database state, for any base fluent f.

Transaction Logic distinguishes a special sort of actions, called *elementary transitions* or *elementary updates*. Intuitively, an elementary transition is a "builtin" action that transforms a database from one state into another. All other actions are defined via rules using the elementary transitions and fluents. In this thesis, elementary transitions are deletions and insertions of base fluents. Formally, an *elementary state transition* is an action atomic formula of the form $insert(f)$ or $delete(f)$, where f is a ground base fluent or has the form $\texttt{neg}\,g$, where g is a ground base fluent. For any given database $\mathbf{D}$,

- $insert(f)$ causes a transition from $\mathbf{D}$ to the state $\mathbf{D} \cup \{f\} \setminus \{\texttt{neg}\,f\}$; and
- $delete(f)$ causes a transition from $\mathbf{D}$ to $\mathbf{D} \setminus \{f\} \cup \{\texttt{neg}\,f\}$.

In addition to the classical connectives $\wedge$, $\vee$, and quantifiers, $\mathcal{TR}$ has new logical connectives (we will add them per need basis in this thesis). Two of the new connectives are: the sequential conjunction $\otimes$ and the modal operator of hypothetical execution $\diamond$. The formula $\phi \otimes \psi$ represents an action composed of an execution of ϕ followed by an execution of ψ, while the formula $\diamond\phi$ is an action of *hypothetically* testing whether ϕ can be executed at the current state, but no actually state changes take place. For example, executing $delete(on(blk1, table)) \otimes insert(on(blk1, blk2))$ means, in procedural terms, "first delete $on(blk1, table)$ from the database, and then insert $on(blk1, blk2)$ into the database." The current database state changes as a result. In contrast, $\diamond move(blk1)$ is only a "hypothetical" execution: it checks whether $move(blk1)$ can be executed in the current state, but whether it can or not the current state does not change.

The semantics of Transaction Logic is such that when f_1 and f_2 are fluents, $f_1 \otimes f_2$ is equivalent to $f_1 \wedge f_2$ and $\diamond f$ to f. Therefore, when no actions are present, $\mathcal{TR}$ reduces to classical logic. This will explain later our use of $\wedge$ in Example 3.4.2 where

it can be replaced with $\otimes$ without changing the meaning (but, the uses of $\otimes$ in the Examples 4.6 and 3.4.2 cannot be replaced with $\wedge$ without changing the meaning).

Definition 2.1 (Serial goal) *Serial goals are defined recursively as follows:*

- *If P is a fluent or an action literal then P is a serial goal. Note that fluent literals can contain both* `not` *and* `neg`*, and action literals can contain* `not`*.*
- *If P is a serial goal, then so are* `not` *P and $\diamond P$.*
- *If P_1 and P_2 are serial goals then so are $P_1 \otimes P_2$ and $P_1 \wedge P_2$.*

Definite serial goals are defined similarly to serial goals, the only difference being that they can contain only atomic fluents and atomic actions instead of fluent and action literals (that is, definite serial goals do not contain the `not` *and* `neg` *operators).*
□

Definition 2.2 (Serial rules) *A **serial rule** is an expression of the form:*

$$H \; :- \; B. \tag{1}$$

where H is a `not`*-free literal and B is a serial goal. We will be dealing with two classes of serial rules:*

- ***Fluent rules:*** *In this case, H is a derived fluent or the explicit negation of a derived fluent and $B = f_1 \otimes ... \otimes f_n$, where each f_i is a fluent literal (and thus $\otimes$ could be replaced with $\wedge$).*
- ***Action rules:*** *In this case, H must be an atomic action formula, while the body of the rule, B, is a* serial goal.

*A **transaction base** is a finite set of serial rules.*
*A **definite serial rule** is a serial rule where all the fluents f_i in the bodies $B = f_1 \otimes ... \otimes f_n$ of fluent rules are atomic fluents, while the serial goals in the bodies of action rules are definite serial goals.*

□

In the above definition, the literal H is called the *head* of the rule, while the serial goal B is called the *body* of the rule. The rule can be viewed as a procedure declaration, and the rule body can be viewed as a procedure call. This is also the operational interpretation similar to the logic programming SLD-style resolution (Linear resolution with Selection function for Definite programs) that we will formalize later in this section.

Definition 2.3 (Serial transaction formula) *A* ***serial transaction formula*** *in the language $\mathcal{TR}$ is a literal, a serial goal or a serial rule.* $\Box$

2.1.2 Transaction Logic Semantics

As described in Section 2.1, general $\mathcal{TR}$ uses plug - ins for the data oracle $\mathcal{O}^d$ and the transaction oracle $\mathcal{O}^t$. These oracles come with a set of database state identifiers (or states). The key concept underlying the semantics of $\mathcal{TR}$ is the concept of *execution paths*, which are sequences of database states.

Definition 2.4 (Paths and Splits) *A* ***path*** *of length k, or a* k-path*, is a finite sequence of states, $\pi = \langle \mathbf{D}_1 \ \ldots \ \mathbf{D}_k \rangle$, where $k \geq 1$.*

A ***split*** *of π is any pair of sub paths, π_1 and π_2, such that $\pi_1 = \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle$ and $\pi_2 = \langle \mathbf{D}_i \ ... \ \mathbf{D}_k \rangle$ for some i $(1 \leq i \leq k)$. If π has a split into π_1 and π_2 then we write $\pi = \pi_1 \circ \pi_2$.* $\Box$

The $\mathcal{TR}$ model theory is defined using *path structures*, which are mappings from paths to classical interpretations conforming to the data and the transaction oracles as plug - ins. A *path structure* $\mathbf{I}$ over $\mathcal{L}$ is a quadruple $\langle U, I_{\mathcal{F}}, I_{path} \rangle$, where U is the *domain* of $\mathbf{I}$, $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$ assigning a function $U^n \longmapsto U$ to every n-ary function symbol in $\mathcal{F}$ and I_{path} is a total mapping that assigns to every path a first - order semantic structure in $Struct(U, I_{\mathcal{F}})$, compliant with the data and transaction oracles: $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$ for every formula $\phi \in \mathcal{O}^d(\mathbf{D})$, and $I_{path}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle) \models^c u$ for every atom $u \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ (where the symbol $\models^c$ denotes satisfaction in these structures).

Here, we depart from the general Transaction Logic framework and the plug - in type of definitions and models that consider oracles and we give direct semantics

for specialized transaction logics variants used in this thesis. We address tabling for definite Horn-$\mathcal{TR}$ programs in Section 2.2, extending it with concurrency in Section 3.5, and with a 3-valued well-founded version in Section 4. Like in classical Horn rules, we only consider Herbrand interpretations and models. We start with the definite Horn-$\mathcal{TR}$ programs and their semantics.

Definition 2.5 (Herbrand universe and base of $\mathcal{TR}$) *The **Herbrand universe** of $\mathcal{TR}$, denoted $\mathcal{U}$, is the set of all ground terms built using the constants and function symbols of the language of $\mathcal{TR}$.*

*The **Herbrand base**, denoted $\mathcal{B}$, is the set of all ground* `not`*-free literals that can be constructed using the language of $\mathcal{TR}$. Within this set we distinguish the following subsets:*

- *$\mathcal{B}_F$, the **Herbrand Base of fluents** is a subset of $\mathcal{B}$ that consists of the fluent-literals.*
- *$\mathcal{B}_{EU}$, the **Herbrand Base of elementary updates** is a subset of ground insert- and delete-literals that are used for elementary transitions.*
- *$\mathcal{B}_A$, the **Herbrand Base of actions** is the subset of $\mathcal{B}$ that consists of action-literals.* □

As in classical logic programming, a *variable assignment* is a mapping $\nu : \mathcal{V} \rightarrow \mathcal{U}$, which takes a variable and returns a Herbrand term as output. The mapping is extended to terms as follows: *i.e.*, $\nu(f(t_1, \ldots, t_n)) = f(\nu(t_1), \ldots, \nu(t_n))$. We can omit variable assignment for formulas with no free variables (called *sentences*) and, from now on, we will deal only with sentences, unless explicitly stated otherwise.

The definite Horn-$\mathcal{TR}$ model theory uses the usual two truth values **t** and **f**, which stand for the usual *true* and *false*, respectively. In Section 4, we will add a third truth value, **u**, that stays for *undefined*.

Definition 2.6 (2-valued Herbrand interpretation for definite programs) *A **2-valued Herbrand interpretation for definite programs** is a mapping $\mathcal{H}$ that assigns* **f** *or* **t** *to every formula L in $\mathcal{B}$.* □

Definition 2.7 (2-valued Herbrand Path Structure for Horn-$\mathcal{TR}$)
A ***2-valued Herbrand Path Structure*** *is a mapping* $\boldsymbol{I}$ *that assigns a 2-valued Herbrand interpretation to every* ***path*** *subject to the following restrictions:*

1. $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{t}$, *if* $d \in \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{f}$, *if* $d \notin \mathbf{D}$;
   for every ground base fluent literal d *and every database state* $\mathbf{D}$.

2. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p\}$ *and* $\mathbf{P}$ *is a ground fluent literal;*
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{f}$, *otherwise.*

3. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \setminus \{p\}$ *and* $\mathbf{P}$ *is a ground fluent literal;*
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{f}$, *otherwise.*

$\square$

Definition 2.8 (Truth valuation in 2-valued path structures) *Let* $\boldsymbol{I}$ *be a path structure for Horn-*$\mathcal{TR}$*,* π *a path,* L *a ground* `not`*-free literal, and let* F*,* G *ground Horn-serial goals We define* ***truth valuations*** *with respect to the 2-valued path structure* $\boldsymbol{I}$ *as follows:*

- *If* ϕ *and* ψ *are serial goals and* $\pi = \pi_1 \circ \pi_2$ *then*
  $\boldsymbol{I}(\pi)(\phi \otimes \psi) = \mathbf{f}\ if\ (\ \boldsymbol{I}(\pi_1)(p) = \mathbf{f}$ *or* $\boldsymbol{I}(\pi_2)(q)) = \mathbf{f}\ )$
  $\boldsymbol{I}(\pi)(\phi \otimes \psi) = \mathbf{t}$, *otherwise.*

- *If* ϕ *and* ψ *are serial goals then*
  $\boldsymbol{I}(\pi)(\phi \wedge \psi) = \mathbf{f}\ if\ (\ \boldsymbol{I}(\pi)(p) = \mathbf{f} or \boldsymbol{I}(\pi)(q)) = \mathbf{f}\ )$
  $\boldsymbol{I}(\pi)(\phi \wedge \psi) = \mathbf{t}$, *otherwise.*

- *If* ϕ *is a serial goal and* $\pi = \langle \mathbf{D} \rangle$*, where* $\mathbf{D}$ *is a database state, then*
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \mathbf{t}\ if\ \boldsymbol{I}(\pi')(\phi)\ |\ \ \pi'$ is there is a path that starts at $\mathbf{D}$
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \mathbf{f}$, *otherwise.*

- *For a definite serial rule* $F :\!- G$*,*
  $\boldsymbol{I}(\pi)(F :\!- G) = \mathbf{t}$ *iff* $\boldsymbol{I}(\pi)(F) = \boldsymbol{I}(\pi)(G)$ *or* $(\boldsymbol{I}(\pi)(F) = \mathbf{t}$ *and* $\boldsymbol{I}(\pi)(G) = \mathbf{f})$
  $\boldsymbol{I}(\pi)(F :\!- G) = \mathbf{f}$, *otherwise.*

We will say that ϕ is **satisfied** *on path π in the path structure $\boldsymbol{I}$ and write $\boldsymbol{I}, \pi \models \phi$ if $\boldsymbol{I}(\pi)(\phi)=\mathbf{t}$.*

□

As we said before, in most of this thesis we deal only with sentences and we will omit the variable assignments ν from these definitions. However, for completeness, if ν is variable assignment, then we write that under ν, ϕ is ***satisfied*** on path π in the path structure $\boldsymbol{I}$, as $\boldsymbol{I}, \pi \models_\nu \phi$.

Definition 2.9 *A 2-valued path structure, $\boldsymbol{I}$, is a* 2-valued model *of a transaction formula ϕ if $\boldsymbol{I}, \pi \models \phi$ for every path π. In this case, we write $\boldsymbol{I} \models \phi$ and say that say that $\boldsymbol{I}$ is a* ***model*** *of ϕ or that ϕ is* ***satisfied*** *in $\boldsymbol{I}$. A path structure $\boldsymbol{I}$ is a model of a set of formulas if it is a model of every formula in the set.*

A path structure $\boldsymbol{I}$ is a 2-valued model *of a definite serial Horn-$\mathcal{TR}$ transaction base* $\mathbf{P}$ *if all the rules in* $\mathbf{P}$ *are satisfied in $\boldsymbol{I}$ (that is, $\boldsymbol{I} \models R$ for every $R \in \mathbf{P}$).* □

We now define two order relations between path structures. In classical logic programming, a Herbrand interpretation σ_1 *precedes* another interpretation σ_2, written $\sigma_1 \preceq \sigma_2$ if all `not`-free literals that are true in σ_1 are also true in σ_2 and all `not`-literals that are true in σ_2 are also true in σ_1. We also say that a Herbrand interpretation σ_1 is *smaller* (that is, it contains *less* information) than another interpretation σ_2, written $\sigma_1 \leq \sigma_2$ if all `not`-free literals that are true in σ_1 are also true in σ_2 and all `not`-literals that are true in σ_1 are also true in σ_2.

If $\mathbf{M}_1$ and $\mathbf{M}_2$ are two Herbrand path structures, then $\mathbf{M}_1 \preceq \mathbf{M}_2$ if $\mathbf{M}_1(\pi) \preceq \mathbf{M}_2(\pi)$ for every path, π. We also have $\mathbf{M}_1 \leq \mathbf{M}_2$ if $\mathbf{M}_1(\pi) \leq \mathbf{M}_2(\pi)$ for every path, π.

A model $\mathbf{M}$ of $\mathbf{P}$ is **minimal** with respect to $\preceq$ iff for any other model, $\mathbf{N}$, of $\mathbf{P}$ $\mathbf{N} \preceq \mathbf{M}$ implies $\mathbf{N} = \mathbf{M}$. The **least** model of $\mathbf{P}$ is a minimal model that is unique. In [BK95], it was shown that every definite Horn $\mathcal{TR}$ program has a unique least total model.

An existential serial goal is a statement of the form $\exists \bar{X} \psi$ where ψ is a serial goal and $\bar{X}$ is a list of all free variables in ψ. For instance, $\exists X move(X, blk2)$ is an existential serial goal. Informally, the truth value of an existential goal in $\mathcal{TR}$ is determined over sequences of states, called *execution paths*, which makes it possible

to view truth assignments in $\mathcal{TR}$'s models as executions. If an existential serial goal, ψ, defined by a program $\mathbf{P}$, evaluates to true over a sequence of states $\mathbf{D_0}, \ldots \mathbf{D_n}$, we say that it can *execute* at state $\mathbf{D_0}$ by passing through the states $\mathbf{D_1}$, ..., $\mathbf{D_{n-1}}$, and ending in the final state $\mathbf{D_n}$. Formally, this is captured by the notion of *executional entailment*, which is written as follows:

$$\mathbf{P}, \mathbf{D_0}, \ldots \mathbf{D_n} \models \psi$$

2.1.3 A Proof Theory for the serial-Horn Transaction Logic

The $\Im^I$ proof theory for serial-Horn $\mathcal{TR}$, described in [BK95, BK98c], resembles the well-known SLD resolution proof strategy for Horn clauses, but it has additional inference rules and axioms. The theory aims to prove statements of the form $\mathbf{P}, \mathbf{D}_0 \text{---} \vdash \psi$, which are called *sequents*. Here $\mathbf{P}$ is a set of serial-Horn rules and ϕ is a *serial-Horn goal*, i.e., a formula that has the form of a body of a serial-Horn rule. An inference succeeds if and only if it finds an execution for the transaction ψ—a sequence of database states $\mathbf{D}_1, \ldots, \mathbf{D}_n$—such that $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$. Informally, this statement says that transaction ψ can successfully execute starting from state $\mathbf{D}_0$.

The axiom of the $\Im^I$ proof theory for serial-Horn $\mathcal{TR}$ uses a special propositional constant in the $\mathcal{TR}$ language, namely *state* (also abbreviated as ( )), which is true only on all paths of length 1 (those are all database states). In the model-based declarative semantics, that is, for any path structure $\mathbf{M}$ and path π, it is the case that $\mathbf{M}, \pi \models \texttt{state}$ if and only if π is a path of length 1. `state` is false (that is, $\mathbf{M}(\pi)(\texttt{state}) = \mathbf{f}$) on every path having more than one state.

Axioms: $\mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$

Inference Rules: In Rules 1–3 below, σ is a substitution, a and b are atomic formulas, and ϕ and *rest* are serial goals.

1. *Applying transaction definitions*:
Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed apart so that the rule shares no variables with $b \otimes rest$. If a and b unify with a most general unifier σ, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(b \otimes rest)}$$

2. *Querying the database*:

If b is a fluent literal, $b\sigma$ and $rest\,\sigma$ share no variables, and $b\sigma$ is true in the database state $\mathbf{D}$ then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(b \otimes rest)}$$

3. *Performing elementary updates*:

If $b\sigma$ and $rest\,\sigma$ share no variables, and $b\,\sigma$ is an elementary action that changes state $\mathbf{D}_1$ to state $\mathbf{D}_2$ then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

Given an inference system, an *executional deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, seq_1, seq_2, $\ldots$, seq_{n-1}, seq_n, where each seq_i is either an axiom-sequent or is derived from earlier sequents by one of the above inference rules. If $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_{n-1}$, $\mathbf{D}_n$ are the database states of these sequents, respectively, then $\mathbf{D}_n, \mathbf{D}_{n-1}, \ldots, \mathbf{D}_1, \mathbf{D}_0$ is called the *execution path* of the deduction.

Theorem 2.1 (Soundness and Completeness [BK95]) *If ϕ is a serial-Horn goal, the executional entailment*

$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi$

holds if and only if there is an executional deduction of $(\exists)\,\phi$ whose execution path is $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.

It is important to keep in mind that this completeness result does not prescribe any particular way of applying the inference rules. If these rules are applied in the forward direction, then all execution paths will be enumerated and completeness will be realized. However, such proofs are undirected, exhaustive,and impractical. In contrast, if we apply the rules backwards, then we obtain a strategy that generalizes the usual SLD resolution with left-to-right literal selection—exactly the strategy used in Prolog. This strategy provides an efficient, goal-directed search strategy for proofs, but it is, unfortunately, incomplete. In many cases, recursive (especially left-recursive)

rules cause SLD resolution with left-to-right literal selection to get stuck in an infinite depth-first search of the proof tree. Just as in ordinary logic programming, to make the above proof theory complete for an SLD-style strategy, it is necessary for the first rule (the one that most resembles SLD resolution) to be applied in a breadth-first manner, but this is hard to implement efficiently.

2.2 Tabled Logic Programming

The paradigm of Tabled Logic Programming (TLP) was invented to circumvent Prolog's incompleteness: the computation based on an SLD-like resolution procedure with a depth - first goal selection strategy. This incompleteness problem has been studied extensively in the logic programming literature [TS86, CW96], leading to the development of *tabling* (or *memoing*) as an efficient algorithm for logically complete implementation of logic programs based on SLD resolution [War92, SW94]. The best known implementation of tabling is XSB,[1] but there are others, such as Yap,[2] B-Prolog,[3] and Mercury.[4]

The idea behind tabling is to maintain in a table all subgoals encountered in a query evaluation and answers to these subgoals. If a subgoal encountered more than once, the evaluation reuses information from the table rather than re - performing resolution against program clauses.

The technique is simple, but it has very important consequences. The tabling technique ensures termination of programs with the *bounded term - size property*, those are the programs where the size of subgoals and answers produced during an evaluation is less than some fixed number. This leads to an easier technique to reason about termination than in basic Prolog and better termination properties. For instance, using tabling, a query to a Prolog predicate for transitive closure over a graph would terminate computing all the reachability pairs of nodes avoiding infinite branches and redundant computation due to repeated subgoals in the search space of SLD resolution. The technique avoids redundant evaluation of subgoals.

[1] `http://xsb.sourceforge.net`
[2] `http://www.dcc.fc.up.pt/~vsc/Yap`
[3] `http://www.probp.com`
[4] `http://www.cs.mu.oz.au/mercury`

Tabling can also be used to evaluate programs with negation according to the Well-Founded Semantics (WFS) [VRS91] (including programs that have recursion through negation). Tabling can achieve the optimal complexity for query evaluation for queries to Datalog programs with negation (with or without function symbols) and other large classes of programs, since it does not recompute the answers for any goals that it already encountered. Finally, tabling integrates closely with Prolog and implicitly with Transaction Logic because of the top - down evaluation strategy.

2.3 Defeasible Reasoning

We conclude this preliminary chapter by introducing preliminaries to the second part of this thesis, namely defaults and defeasible reasoning over Transaction Logic. Defeasible reasoning in logic programming (LP) has been successfully used to model a broad range of application domains and tasks, including security policies, regulations, laws, Web services, aspects of inductive/scientific learning and natural language understanding. There has been a multitude of formal approaches to defeasibility based on a large variety of intuitions about the desired behavior [BH95, BE99, BE00, DST03, DS01, EFLP03, GS98, Gro99, Nut94, Pra93, SI00, WZL00, ZWB01]. Most of these are based on Reiter's Default Logic [Rei80], stable models [GL88], and only a few [Gro99, MN06, WGK$^+$09a] use the well - founded semantics [VRS91].

Our approach builds upon our previous work on unifying research on defeasible reasoning in classical logic programming [WGK$^+$09a]. The $\mathcal{LPDA}$ novel approach has the advantages that generalizes Courteous Logic [Gro99] and other previous defeasible LP approaches to include HiLog-style higher - order [CKW93] and F-logic style object - oriented features [KLW95], and has the ability to combine multiple defeasible LP approaches within a single system.

$\mathcal{LPDA}$ deals with **tagged rules**, expressions of the form

$$@r\ L :\!-\ Body \tag{2}$$

where r, called the **tag** of the rule, is a term, L, called the **head** of the rule, is a `not`-free literal in L, and $Body$, called the **body** of the rule, is a conjunction of literals in L. A **logic program with defaults and argumentation theories** is a set

of tagged rules. The $\mathcal{LPDA}$ framework abstracts the intuitions about defeasibility into *argumentation theories*, a separate set of rules that contain a special predicate $\texttt{\$defeated}_{AT}$ that does not appear in the rule heads of the main program. The semantics of $\mathcal{LPDA}$ are based on well - founded models [VRS91] and stable models [GL88], and for further details the reader can examine its details in [WGK$^+$09a].

Chapter 3

Tabling for Transaction Logic

A number of implementations of $\mathcal{TR}$ exist [Hun96a, Hun96b, Sle00, F.S00, YKZ03, Kif] but, unfortunately, all are logically incomplete. The major barrier to completeness for these implementations is similar to the reasons for Prolog incompleteness: the computation is based on an SLD- like resolution procedure with a depth - first goal selection strategy. We extend Transaction Logic with tabling, keeping into consideration that $\mathcal{TR}$ deals with the phenomenon of *changing states*, which is not an issue in classical logic programming, where state changes are viewed as a non - logical feature that is best left outside of the scope of the tabling mechanism.

The following example shows the effects of the original proof theory presented in Section 2.1 of this thesis.

Example 3.1 (Consuming paths)

Suppose *edge* is a binary fluent and $delete(edge(N, M))$ denotes the action of deleting the edge that goes from node N to node M. The following rules compute reachability in the graph by traversing edges and then swallowing them:

$$\begin{aligned} & reach(X, Y) \ :- \\ & \quad reach(X, Z) \otimes edge(Z, Y) \otimes delete(edge(Z, Y)). \\ & reach(X, X). \end{aligned} \tag{3}$$

Note that the first rule defines the action *reach* recursively.

Lets consider the initial graph in Figure 1 and a query $reach(a, X)$ to find all nodes X reachable from the node a and return the states obtained after the deletion of each

path. Notice that the example has a recursive definition. Just as in Prolog, it is not hard to see that the SLD strategy for the above proof theory will get stuck in infinite derivation paths. As seen in Figure 2, the proof tree is infinite and the $reach(a, X)$ query will run into an infinite loop by applying the first rule for $reach(X, Y)$ over and over again before it would return any single solution. □

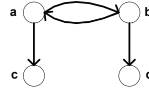

Figure 1: An initial graph for the consuming paths reachability example

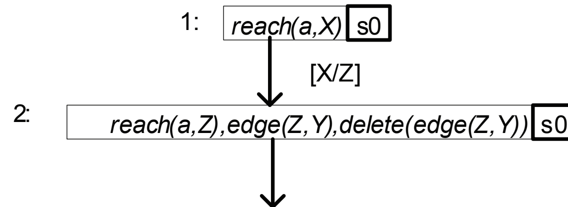

Figure 2: SLD-style tree for the query $reach(a, X)$ in the consuming paths example with an infinite derivation branch

Just as in ordinary logic programming, to make the above proof theory complete for an SLD-style strategy, it is necessary for the first rule (the one that most resembles SLD resolution) to be applied in a breadth - first manner, but this is hard to define and implement efficiently.

$\mathcal{TR}$ tabling requires memoing of the underlying database state and not just memoing of the previously called subgoals. Clearly, this is a major problem both in terms of space and time. Of course, a powerful formalism such as Transaction Logic does not come without a price, but our contribution is in showing that there is ample room for optimization. After describing the extended tabling algorithm, we discuss the major trade - offs in its implementation and show several time/space optimizations. We implemented a dozen of algorithms, which combine our optimizations in various ways. In the end of this chapter we discuss six of those that illustrate the most salient points, the rationale behind each of them, and then present our experimental results. These results show that a proper integration of our techniques results in a system with the best overall performance and scalability characteristics.

3.1 Tabling for Definite Serial Horn-Transaction Logic Programs

Tabling for definite serial-Horn $\mathcal{TR}$ is analogous to tabling for Datalog, but with one major difference: not only the goals that are yet to be proved need to be memoized, but also the database states in which the calls to those goals were made. Likewise, not only the answers to these goals must be memoized, but also the states that get created by execution of those goals. We first describe the main principles of the algorithm and then incorporate it into the proof theory of Section 2.1 by modifying the first inference rule (we call the new inference system $\mathcal{F}^T$).

The main idea in tabled logic programming is to re - use answers that were computed for previous calls to the same goal. First, predicates of the program are partitioned into the *tabled* ones and those that are *not* tabled. In principle, all predicates could be tabled and query execution would still be correct. However, in some cases, knowing that some predicates do not have to be tabled (while still preserving completeness) can lead to significant savings (Sections 3.2, 3.4). One tabled goal is said to *dominate* another in tabled resolution if the two goals are *variants* of each other (variant tabling), i.e., are identical up to variable renaming, or if the first goal *subsumes* the second (subsumption - based tabling). When a subgoal to a tabled predicate starting in a particular state is encountered, a check is made to see whether this is the first occurrence of this subgoal in that state (i.e., no dominating goal call was made before in the same state).

- If the call is new, the pair $(goal, state)$ is saved in a global data structure called the *table space*, and evaluation uses normal clause resolution to compute answers and generate new database states for the subgoal. The computed $(answer\text{ - }unification, new\text{ - }state)$ pairs are recorded in the *answer table* created for the aforesaid $(goal, state)$ pair each time they are computed.

- If the call is *not* new, i.e., a pair $(goal, state)$ exists in the table space for a dominating goal, the answers to the call are returned directly from the answer table for $(goal, state)$ and no clause resolution is used.

The evaluation goes on by returning new answers to subgoals until all answers for all

goals generated during this process are computed.

The Inference System $\mathcal{F}^T$):

As in the previous section, $\mathbf{P}$ is a transaction base, $\mathbf{D}$ and $\mathbf{D}_i$ are any databases, σ denotes substitutions, a and b atomic formulas, and ϕ, $rest$ are definite serial-Horn goals.

Axioms: $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash state$

Rule 1a. *Applying transaction definitions for tabled predicates:*

Suppose b's predicate is tabled and there is no dominating pair $(c, \mathbf{D}_1)$ in the table space. Let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ whose variables have been renamed apart from $b \otimes rest$ (i.e., the rule shares no variables with the goal) and suppose that a and b unify with the most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\begin{array}{c}\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest) \\ (b, \mathbf{D}_1) \in \texttt{table space} \\ \forall\ \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\gamma,\ (b\gamma, \mathbf{D}_i) \in \texttt{answer table}(b, \mathbf{D}_1)\end{array}}$$

That is, given a sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma$, the rule allows us to derive $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)$. In addition, $(b, \mathbf{D}_1)$ is added to the table space, and for all γ such that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\gamma$ is derivable, the answer $(b\gamma, \mathbf{D}_i)$ is added to the answer table for $(b, \mathbf{D}_1)$.

Rule 1b. *Returning answers from answer tables:*

Suppose: (1) b's predicate symbol is declared as tabled, (2) there is a dominating pair $(c, \mathbf{D}_1)$ in the table space, (3) the answer table for $(c, \mathbf{D}_1)$ has an entry $(a, \mathbf{D}_i)$, and (4) a and b unify with most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_i \text{---} \vdash (\exists)\,(rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

Rule 1c. *Applying transaction definitions for non - tabled predicates:*

This rule is identical to Rule 1 in the proof theory of Section 2.1: let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ and a's predicate symbol is *not* tabled. Assume that this rule's variables have been renamed apart from $b \otimes rest$ and that a and b unify with most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

Rule 2. *Querying the database*:

If b is a fluent literal, $b\sigma$ and $rest\,\sigma$ share no variables, and $b\sigma$ is true in the database state $\mathbf{D}_1$, then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

Rule 3. *Performing elementary updates*:

If $b\sigma$ and $rest\,\sigma$ share no variables, and $b\,\sigma$ is an elementary action that changes state $\mathbf{D}_1$ to state $\mathbf{D}_2$, then:

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\,rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

The rest of the tabling proof theory for Transaction Logic (Rules 2 and 3) is identical to the original theory of Section 2.1. The inference system $\mathcal{F}^T$ also contains the same axiom of the $\Im^I$ proof theory for serial-Horn $\mathcal{TR}$ that says that the propositional constant *state* is true only on all paths of length 1: $\mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$.

The rules 1a–1c modify the original proof theory for $\mathcal{TR}$ by capturing the effects of tabling. Rule 1a creates new entries in the table space and their associated answer tables. When a call to a subgoal is complete, the corresponding answer (both the substitution and the resulting database state) are added to an appropriate answer table. Rule 1b deals with calls for which dominating table entries already exist. In those cases, no clause resolution is used and answers are returned directly from the appropriate answer tables. Rule 1c is identical to Rule 1 of the original proof theory for Transaction Logic, but here it is applied only to non - tabled predicates. It simply does clause resolution SLD-style. Notice that Rule 1b might change the current database state after returning an answer for b, since the returned answer might have been obtained as a result of execution of state - changing actions.

We show first a simple example where the main properties of tabling are easy to observe: completeness and termination.

Example 3.2 (Simple infinite derivations) *Suppose $flag$ is a 0-ary fluent and $insert(flag)$ denotes the action of inserting the flag in the current database, while $delete(flag)$ denotes the action of deleting the flag from the current database.*

$$\begin{aligned} a(X) &:- insert(flag) \otimes b(X). \\ b(X) &:- delete(flag) \otimes a(X). \\ b(X) &:- test(X). \end{aligned} \tag{4}$$

$$test(1). \quad test(2). \quad test(3). \tag{5}$$

Suppose that the initial database is the empty database. Given the query $?- a(X).$, the original system performs consecutive insertions and deletions of $flag$ ad infinitum. This type of depth - first execution is neither complete nor terminates. However, it is easy to observe that the tabled execution performs one insertion and one deletion of flag and is suspended, because it's useless to continue these consecutive updates. The system will then query for $test(X)$ and terminate. Although in the original system, there were an infinite number of possible executions of $a(X)$, corresponding to the path - answers of the form $\langle \{\}, \{flag\}, \{\}, \ldots, \{flag\} \rangle$, it is pointless to compute all of them if all that the user wants is the initial and the final database states. In this case, the transaction succeeds for $a(1)$, $a(2)$ and $a(3)$, all the solution paths starting in the database state $\{\}$ and ending in the database state $\{flag\}$. □

We modify the definition of deduction in Section 2.1 to accommodate tabling: A *tabled* deduction for $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$, is a series of sequents, where each sequent is an axiom or is derived from earlier sequents by an inference rule of the above tabling inference system.

Theorem 3.2 (Soundness and Completeness) *Suppose ϕ is a definite serial-Horn goal.*

Soundness: *If there is a tabled deduction of the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \phi$ with the execution path $\langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle$ then the executional entailment $\mathbf{P}, \mathbf{D_1} \ldots \mathbf{D_n} \models (\exists)\, \phi$ holds.*

Completeness: *If the executional entailment $\mathbf{P}, \mathbf{D_1 D_2} \ldots \mathbf{D_{n-1} D_n} \models (\exists)\, \phi$ holds then there exists a tabled deduction of the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \phi$ with an execution*

path $\langle \mathbf{D}_1, \mathbf{D}'_2 \ldots \mathbf{D}'_m, \mathbf{D}_n \rangle$ *that starts in the database state* $\mathbf{D}_1$ *and ends in* $\mathbf{D}_n$*.*

Proof: See Appendix B. □

This theorem is different from the completeness of the proof theory in Section 2.1 (Theorem 2.1) in that it does not guarantee that all execution paths will be found: it only guarantees that all final states will be found. This is a very essential difference because the number of all execution paths can be infinite (even in simple cases where function symbols are not involved), while the number of final states is often finite. The user typically is interested in finding out whether a particular transaction can execute starting at a particular state and finish in a particular final state (or a group of states). The fact that there are additional executions where the same sub - sequence sequence of states repeats itself (which is the main cause of infiniteness) is usually of no interest to the user. This leads to the following important termination results.

Theorem 3.3 (Termination)
Let $\mathbf{P}$ *be a program with no function symbols with arity greater than 0, that is, it allows only constants (i.e., 0-ary function symbols). Let us further assume that all recursive predicates in* $\mathbf{P}$ *are marked as tabled. Then, for any definite serial-Horn goal* ϕ*, the* tabled *proof theory finds one or more proofs of* $\mathbf{P}, \mathbf{D} \dashrightarrow\!\vdash (\exists)\, \phi$ *and* terminates.

Proof: See Appendix C. □

Note that the above theorem does not guarantee that all executions found by the original proof theory of [BK95] will also be found by the tabling proof theory, and this is a good thing! In this way, the new proof theory will find all the executions *that matter*, and will be able to terminate. In the Appendix C we compute an upper bound for the number of sequents in the proof of any transaction using the tabled derivation trees defined in the Section 3.1.1).

3.1.1 Tabled-$\mathcal{TR}$ derivation trees

In this section we introduce the *tabled derivation trees*, a formalism used in the proof of termination of the tabled inference system $\mathcal{F}^T$ and also to exemplify the inference in a user friendly way similar to that of *Extended SLG* (SLG_X) in [Swi99].

Given the tabled proof theory, a program $\mathbf{P}$, an initial database $\mathbf{D}$ and a definite serial-Horn goal ϕ, we can build a *Tabled-$\mathcal{TR}$ derivation tree* with a root corresponding to the goal ϕ and the database $\mathbf{D}$, whose nodes correspond to sequents in the proof theory. Each arc is labeled with an inference rule and a set of substitutions σ, while each node is a pair $\langle answer - substitution, database\ state \rangle$ that corresponds to a new sequent obtained by applying the inference rule on the parent sequent. In fact, each node in the tree is associated with one and only one database, namely the current state of the database that starts the paths on which the transaction has to be proved true. The construction of this tree proceeds as follows. In the initial step, the goal ϕ becomes the root of the tree and is associated with the initial database, bD, constituting parts of the sequent: $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$. A node is *empty* if the current goal in the sequent is empty (these are the success leaves of the derivation tree). These empty goals correspond to application of axioms in the proof theory and they still contain a current database state. A node is *completed* (completely evaluated) if it is empty, or if it is a node that is not a suspended sequent (that is, the inference rule 1b is applied for it returning answers from answer table of a dominating goal) and no inference rule can be applied to the node, or it is a suspended sequent where there is a completed node for the dominating pair $(c, \mathbf{D})$ in the table space whose all entries $(a, \mathbf{D}')$ were applied with most general unifiers to the current node (all of its possible answers were fed to the node) and all possible operations have been done on its nodes, and the nodes of subtree upon which the node depends. A ground subgoal is completely evaluated when an answer is derived for it and all the returning databases have been determined. A node is *active* if it not completed. At each step in the tree construction that follows the initial step, an active node is chosen in the tree and we proceed by applying a resolution which has not been performed yet. If the resolution succeeds and we reach an empty goal, we add a child node to the current node for the sequent resulting from the resolution. A tree is *completed* if all it's nodes are completed.

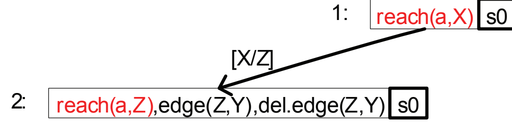

Figure 3: The tabled resolution tree at step 2 for the query $reach(a, X)$ in the consuming paths example

3.1.1.1 A Step-by-step Tabling Example for Definite Serial Horn Transaction Logic

Lets consider the consuming paths example 3 to provide a detailed example for tabled Horn-$\mathcal{TR}$. We reproduce this short example here:

$$\begin{aligned} reach(X,Y)\ &:- \\ &\quad reach(X,Z) \otimes edge(Z,Y) \otimes delete(edge(Z,Y)). \\ reach(X,X)&. \end{aligned} \tag{6}$$

The tabled resolution (implemented with delaying) uses two tables: a *solution table* to save the tabled queries and a *lookup table* to mark the answers tried in each node in the evaluation tree for calls that were not *dominant* calls. In the Figure 3, node 1 is a dominating call because it was the first time the evaluation encountered the goal $reach(a, Variable)$, while node 2 is a dominated node for $reach(a, Variable)$. The call $reach(a, X)$ and the initial state of the graph is saved in the $call-initial\ state$ column of the table, while the node 2 is added to the lookup table with no solutions currently tried. The computation in node 2 is stalled until we have additional solutions for the query.

The Figure 4 shows the resolution for the consuming paths top - down tabled example where the second rule is applied and an answer to the query is computed and added to the solution table: a can be reached from a (i.e., $reach(a, a)$), leaving the database in the initial database state, s_0.

The Figure 5 shows that the solution $reach(a, a)$ in state s_0 for the query $reach(a, X)$ was applied in the node 2 and two new solutions are found by the algorithm: $reach(a, b)$ bringing the database into the new state $\{edge(a, c), edge(b, a), edge(b, d)\}$ and $reach(a, c)$ bringing the database into the new

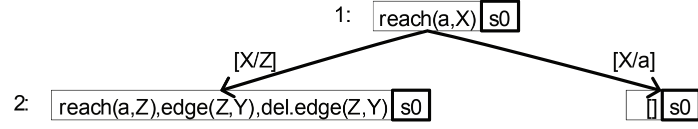

Figure 4: The tabled resolution tree at step 3 for the query $reach(a, X)$ in the consuming paths example

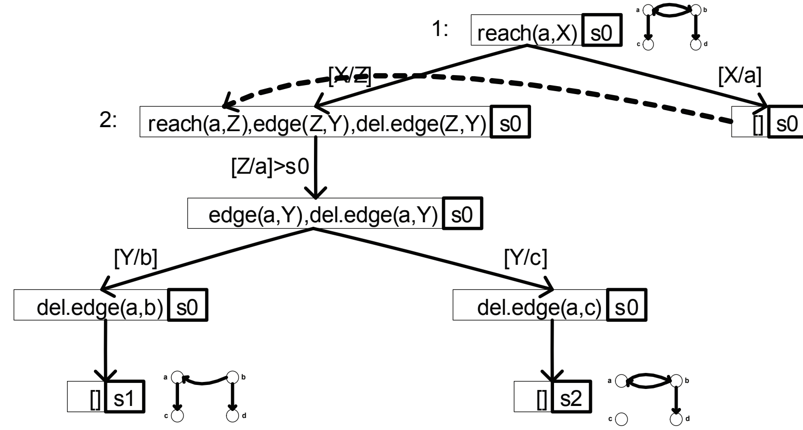

Figure 5: The tabled resolution tree at step 7 for the query $reach(a, X)$ in the consuming paths example

database state $\{edge(a, b), edge(b, a), edge(b, d)\}$. The answer $reach(a, a)$ with resulting state s_0 is marked in the lookup table as tested for the node 2.

The Figure 5 shows that the solution $reach(a, a)ins_0$ for the query $reach(a, X)$ was applied in the node 2 and two new solutions are found by the algorithm: $reach(a, b)$ bringing the database into the new state $\{edge(a, c), edge(b, a), edge(b, d)\}$ and $reach(a, c)$ bringing the database into the new database state $\{edge(a, b), edge(b, a), edge(b, d)\}$. The answer $reach(a, a)$ with resulting state s_0 is marked in the lookup table as tested for the node 2.

Following the application of the two solutions $reach(a, b)$ and $reach(a, c)$ to the dominated call in the node 2, additional solutions are added to the dominant goal $reach(a, X)$, namely $reach(a, a)$, bringing the database into the new state $\{edge(a, c), edge(b, d)\}$ and $reach(a, d)$ bringing the database into the new database state $\{edge(a, c), edge(b, a)\}$ (see Figure 6. The dominated goal 2 in the lookup table marks that the answers $reach(a, b)$ and $reach(a, c)$ were applied to 2.

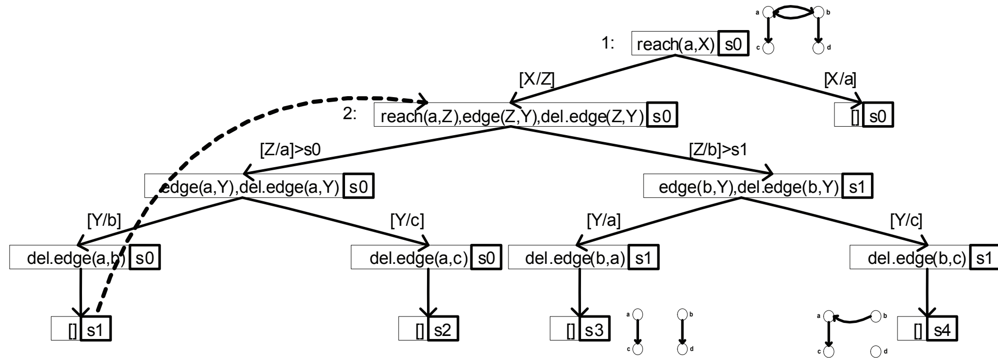

Figure 6: The tabled resolution tree at step 11 for the query $reach(a, X)$ in the consuming paths example

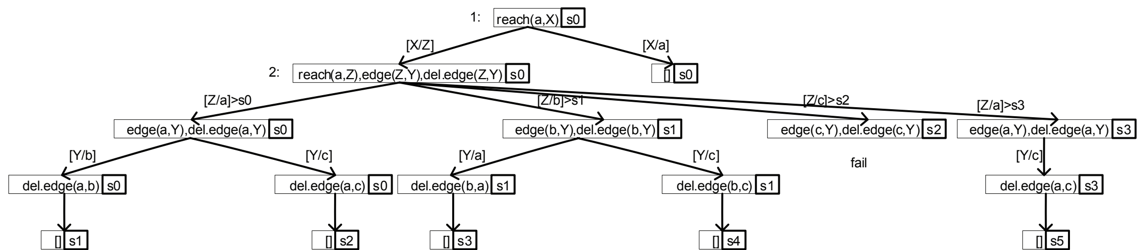

Figure 7: The tabled resolution tree at step 12 for the query $reach(a, X)$ in the consuming paths example

The algorithm continues by feeding all answers to the dominated node 2 and computing all answers to the query $reach(a, X)$. The algorithm finds all the solutions (i.e., answer substitutions and return states) for the query, terminates and does not repeat inferences, being an optimal computation for the query.

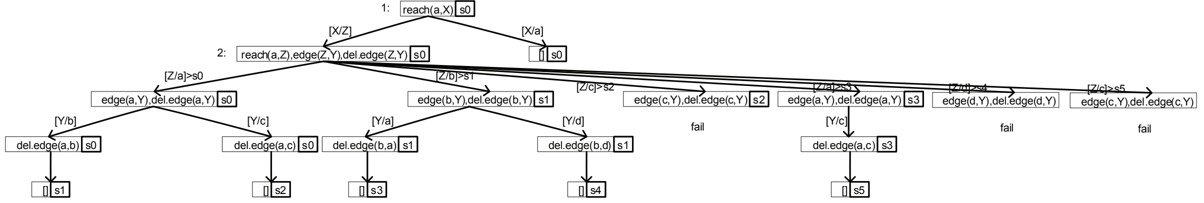

Figure 8: The tabled resolution tree at step 14 for the query $reach(a, X)$ in the consuming paths example

3.2 Problems and Solutions in Implementing Tabled Transaction Logic

In this section we discuss the major hurdles on the way to implementing the algorithm of Section 3.1 and propose a number of solutions. Then we describe six different implementations that progressively adopt these solutions. A performance evaluation of these implementations is described in Section 3.4.

3.2.1 Main Difficulties with Implementing Tabled Transaction Logic

The first obvious problem in implementing Transaction Logic is the transactional semantics of its actions, which requires atomicity. As it turned out, this is the easiest problem to address, and all existing implementations support atomicity.

The hard implementational issues have to do with tabling. These issues stem from the same major difficulty, which is easy to spot after a quick review of the tabling algorithm: unlike normal logic programming, tabling in $\mathcal{TR}$ implies saving the underlying database states as part of the answer tables. This raises the following problems:

1. *Space*. Saving database states in answer tables potentially leads to huge duplication of information. This is particularly troublesome for large database states (e.g., tens of thousands or even millions of facts).

2. *Time*. Tabling database states implies the following time - consuming operations:

(a) *Copying of states.* Since states are modified in the course of transaction execution, tabled states must be copied, since once tabled the contents of that state must stay immutable.

(b) *Comparison of states.* When a transactional subgoal is invoked at a state, we must check whether that particular goal/state pair is already tabled. This involves comparison of states as sets. In the worst case, comparing two states might take $O(n \cdot log(n))$ time, where n is the size of the states. Worse, newly created states might have to be compared with other tabled states to determine if the newly created set of facts is a genuinely new state or has been seen before.

(c) *Querying of states.* The states created during the execution of transactions must be efficiently queryable. We will soon see, however, that there is a tension between the efficiency of querying and the various solutions to the aforementioned problems with time and space.

(d) *Reinstating states.* During backtracking and restarting of suspended goals with new solutions, the data structures for current state need to be resumed for querying and updating. This process might take a constant time when just pointers to the state have to be changed or a variable time when new data structures necessary for querying need to be created. A similar operation in SLG-WAM is called *forward trail* of reinstalling variable bindings. Both operations have the goal to reinstate the environment in the suspended computation.

(e) *Backtracking of updates.* Although, it's a problem inherited from $\mathcal{TR}$ and not a new issue in tabling of $\mathcal{TR}$, backtracking of updates is done differently in various data structures.

Each of the above problems has a number of solutions, but the different solutions involve various trade - offs, so it is not obvious how the different solutions fare when combined. The next section discusses ideas that lead to substantial savings in various situations.

3.2.2 The Space of Possible Solutions

We will now map the space of possible approaches to the problems listed in the previous section and discuss the various trade - offs in adopting the different space - and time - saving solutions. In Section 3.3, we discuss the most interesting combinations of these solutions, and their performance is evaluated in Section 3.4.

3.2.2.1 Space issues in tabled $\mathcal{TR}$

Our first observation is that although the initial state of a transaction might be huge, a typical transaction changes only a few dozens of facts. Transactions that originate in AI or graph algorithms, as in our Examples 3.1, 3.4.1, and 3.4.2, might modify hundreds or even thousands of facts, but this is still far cry from millions or even billions of facts that an initial state might contain. This suggests an obvious idea: *differential logs*. That is, instead of tabling an entire state, we can represent a state as a pair of the form $(initial_state, changelog)$. This representation not only saves space, but also reduces the amount of time required for copying states. A differential log is normally represented as a pair of logs $(InsertLog, DeleteLog)$. The former contains the records of inserted facts and the latter of deleted ones. Differential logs introduce a trade - off between the decreasing cost of storing and copying states and the increasing time for querying database states. Depending on the data structures used for change logs, this overhead could be a constant factor of 2 or more.

The next possibility is to employ the various forms of *compression*, such as:

- *Sharing*. Logs can be stored using data structures, like *tries*, which enable high degree of sharing, so the total space requirement would be less than the sum of the sizes of all the logs.

- *Factoring*. Database facts can be stored on a heap and shared among states. The states themselves can refer to these facts using pointers or a numbering scheme. Thus, duplication of facts that are common to many different states is much less costly (only one word per duplicate fact).

- *Table skipping*. It might be possible to reduce the number of states that need to be tabled by carefully analyzing the rules and determining that only the

$$
\begin{aligned}
s0 = \{edge(a,b), edge(a,c), edge(b,a), edge(b,d)\} \\
s1 = \{edge(a,c), edge(b,a), edge(b,d)\} \\
s2 = \{edge(a,b), edge(b,a), edge(b,d)\} \\
s3 = \{edge(a,c), edge(b,d)\} \\
s4 = \{edge(a,c), edge(b,a)\} \\
s5 = \{edge(b,d)\}
\end{aligned}
$$

Table 1: A set of states saved during the tabling algorithm

states associated with certain subgoals have to be tabled. Other states can be modified directly without the need for storing or copying them. The theoretical basis for skipping is Theorem 3.3. All that is needed is to ensure that enough predicates are tabled to affect termination. The theorem states that it suffices to table just the recursive predicates, but in some cases even that much might be unnecessary.

- *Double-differential logs.* When table - skipping is used, the changes made by the transaction *with respect to the previous tabled state* can be kept in a log and not merged to that state until the next tabled state is reached. In this case, the current state is represented as a pair $(tabled_state, changelog_relative_to_tabled_state)$. In turn, the tabled state is represented as a pair $(initial_state, main_changelog)$, so the entire state is represented using the initial state and two relative change logs. The first of these logs is called the *main* change log and the second is the *residual* change log. We call this state representation strategy *double - differential logging.*

The techniques of sharing and factoring are exemplified in the Figures 9 and 10 where we have an example with 4 fluents $edge(a,b), edge(a,c), edge(b,a)$ and $edge(b,d)$ and 5 possible states (see Table 1). The tries are very compact and reusing common facts in multiple states occupies a relatively compact space.

Subsumptive tabling Finally, in general, a transaction depends on only a small portion of the database state. In such a case, if a transaction repeats itself on different databases, but the database portion that it depends on remains the same, this

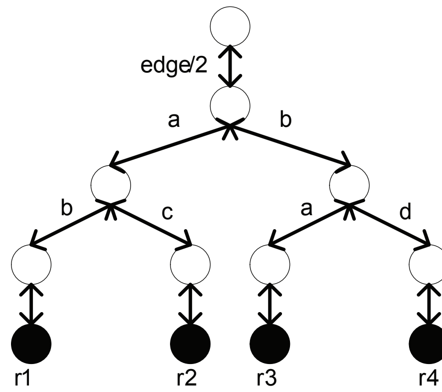

Figure 9: Rule trie example

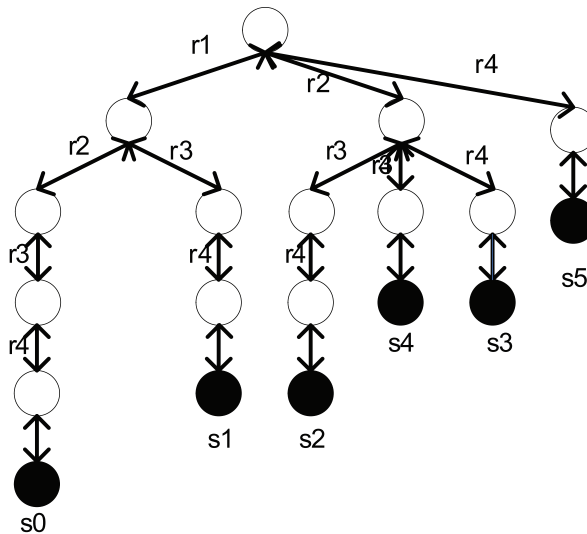

Figure 10: State trie example

derivation branch can be suspended and the proof theory slightly modified to account for this optimization. Such detections of dependencies between transaction calls and database fluents can be done using the dependency graph (like the one that we address later in this thesis in Section 3.5). One example that could take advantage of this feature is the Hamiltonian cycle use case mentioned in the evaluation Section 3.4.1.

3.2.2.2 Time issues in tabled $\mathcal{TR}$

Two of the main time - related issues are copying and comparing of states. The third issue, which stems from the suggestion to use differential logs, has to do with the increased cost of querying the underlying database states.

- *State comparison.* Our first observation is that, in most cases, the newly created states are different from most of the already seen tabled stated. So, we need a fast method to rule out most of the equalities. One such method is based on *incremental hash functions*. Simple incremental hash functions are cardinality, the total number of arities of the facts in the states, or, for example, any function of the form $\bar{h}(State) = \Sigma_{e \in State}\, h(e) \,\mathbf{mod}\, N$. For better results, one can employ several such functions. (These hash functions must be incremental so they could be computed quickly over large sets.)

  If the hash functions fail to differentiate among the new state and some of the tabled states, the sets must be compared directly. This can be made faster if *replicas* of tabled states are kept sorted, since comparing two sorted lists is linear in the size of the lists.

  Still, this is not completely satisfactory if, for example, a newly created state has to be directly compared with multiple tabled states, which the hash functions failed to tell apart. It turns out, however, that the problem can be avoided by the use of data structures, such as *tries*. For instance, we can store the already seen tabled states as sorted lists in a trie. Then, comparison of any new state with *all* the stored states can be done in time linear *in the size of the new state* and will not depend on the number of the already seen states. We can further combine hashing with trie comparison by representing each tabled state as a

list whose prefix (say, the first three elements, if we choose to use three hash functions) are the hash values of our hash functions and the rest is a sorted list of the actual facts that belong to the state. Thus, in searching the trie the first few comparisons will be made based on the hash values and then states will be discriminated based on the actual facts they contain.

- *Separate state repository.* Tabled states and goal calls are typically kept in tries, because this data structure enables fast (linear time) checks to find out whether a pair $(call, state)$ had been seen before. The question is then whether these pairs are stored in one trie (say, as a term $pair(call, state)$) or the calls are stored in one trie and states in another (the latter is called a *state repository*). Storing states and calls in the same trie typically requires more space (because calls and states tend to share less structure in such tries) and time (since call - state comparisons tend to fail later than in the case of separate state repositories).

- *Querying of states.* The data structures used for differential logs make a big difference for the querying time of the states, since in order to find out whether a particular fact is in a state one has to query both the initial state and the log. For double-differential logs, the overhead is even higher, since two logs must be queried in addition to the initial state. Since the initial state is static, it can be designed in the most advantageous way as far as querying is concerned. For the logs, we have to balance the insertion time against the query time.

  Unordered lists are best as far as the update time goes, but they are some of the worst for querying. Also, for state comparison we need *sorted* list, which makes unsorted lists less attractive. Tries are good for querying and updating, but they are poor at maintaining the sorted order among the facts. For state comparison, tries must be converted to lists at the cost of $n \cdot log(n) \cdot |S|$, where $|S|$ is the number of facts in the trie.

  Nevertheless, in our experiments, we stored some logs as tries, since they are the most optimized data structure in our underlying platform, XSB. To compensate for the tries' inefficiency in keeping the logs sorted, we sometimes maintained sorted lists as auxiliary data structures. A much better choice would have been B^+ trees, as they can be made shallow (thus improving the search) and they

naturally keep data sorted.

- *Copying of states.* First, note that the table skipping and factoring methods that were introduced as space - saving techniques are also important time - saving techniques, because the fewer states are tabled — the fewer state - comparisons and copying are needed. Double - differential logs can also reduce the number of times states have to be copied. This happens because in double - differential logging, new tabled states are created by merging the previous tabled state with the residual log. This is done just before entering the *next* tabled state. In contrast, in single - differential logging, states that might get tabled at the next opportunity are initially created by copying the state that was tabled just now. The copy is then modified directly and it gets saved in the state repository when the next tabled call is made. If no new tabled call is reached, the copy has been made in vain. Double-differential logging delays copying of states and thus is less prone to wasteful copying.

  Beyond that, the fastest data structure to copy would be a list. In fact, if logs are represented as *unsorted* lists then no copying would be needed whatsoever. Logs can simply be passed as arguments to the predicates that represent actions. For instance, the log at state k could be
  $(InsertLog_k, DeleteLog_k)$ and the next state (say, after inserting p) it would be $([p|InsertLog_k], DeleteLog_k)$, which shares the lists $InsertLog_k$ and $DeleteLog_k$ with the previous state.

  Unfortunately, as discussed earlier, lists are not efficient for querying, and we need them sorted. In our performance evaluation, we compared list - based implementations with others to validate the trade - off between copying and querying of states. With an eye on querying, balanced trees are reasonably efficient to copy, since their space overhead is a constant factor (compared to lists). In our comparisons, however, red - black trees and AVL trees did worse than tries because tries are highly optimized in XSB. However, an optimized implementation of B$^+$ trees would be far superior than tries. The space overhead factor for B$^+$ trees is only $1 + 1/(k - 1)$, where k is the degree of the tree, and they can be copied very efficiently, if implemented in a low - level language like C.

Thus, a trade - off exists between the costs of querying and copying, which we evaluate in our performance study.

3.3 Implementations of Tabled Transaction Logic

Overall, we implemented more than a dozen of different algorithms, which realize various combination of the above ideas. In this section, we discuss six of the most interesting such implementations[1] .

Common features. All implementations discussed here share the following common features, which were introduced earlier:

- Data compression via factoring.
- Differential logs.
- State comparison:
  - via incremental hash functions — to quickly rule out most false matches
  - state repositories that use tries to store replicas of the main differential logs — to ensure at most linear - time match of newly created states against all previously seen states

Implementation 1. This implementation uses the above common features in which differential logs are single, since table - skipping is *not* used. The logs are maintained as *ordered* lists stored in the state repository. New states are constructed via insertion - sort operations. As noted earlier, lists are a poor choice for querying states, but they are near - optimal for copying. Recall that a differential log has the form $(InsertLog, DeleteLog)$. Moving to the next state is accomplished by inserting a record in the insertion or deletion logs. In the worst case, this is linear in the log size, but the average is under 3/4 of the log size. Since successive states often share their list tails, this can also result in space savings.

[1] http://flora.sourceforge.net/tr-interpreter-suite.tar.gz

Implementation 2. This implementation is similar to #1, but logs are stored both as ordered lists and tries. The ordered lists reside in the state repository, as before, and tries are used to speed up querying. When moving from state to state, the tries are modified directly, without copying, so the only significant overhead here is the need to maintain a query trie. To improve performance, creation of the query trie can be delayed until the first query or update.

Implementations 3a and 3b. These implementations use table skipping to reduce the number of tabled states. Table-skipping avoids state comparison and copying when executing non - tabled (usually non - recursive) actions. State copying is still required at tabled states. Furthermore, since states produced by non - tabled transactions are not saved in tables, there is no need to check if we have seen such states before. Both implementations use sorted lists to represent logs. However, 3a uses single differential log and 3b uses double logs.

Implementations 4a and 4b. Like 3a and 3b, these implementations use table skipping, where 4a uses single differential logs and 4b uses double logs. The difference is that 4a represents its single log as a trie and 4b does the same for its main differential log. The residual differential log in 4b is still maintained as a sorted list. (In our tests, the residual differential logs were generally short, which did not justify the overhead of using tries for them.) Similar to the implementation 2, the creation of the main differential log (e.g., for the implementations 4b) is delayed until such data structure is needed.

3.4 Applications and performance evaluation

The following examples provide a test - bed for performance evaluation study.

3.4.1 Hamiltonian cycles

A Hamiltonian cycle is a cycle in a directed graph that visits each vertex exactly once. Similarly to consuming paths, Hamiltonian cycles are detected here by swallowing the

already traversed vertexes.

$$
\begin{aligned}
& hCycle(Start, Start) \;:-\; \texttt{not}\ vertex(X). \\
& hCycle(Start, X) \;:- \\
& \qquad edge(X,Y) \otimes vertex(Y) \\
& \qquad \otimes delete(vertex(Y)) \otimes insert(mark(X,Y)) \\
& \qquad \otimes hCycle(Start, Y) \otimes insert(vertex(Y)).
\end{aligned} \tag{7}
$$

This solution to Hamiltonian paths relies on the transactional semantics of $\mathcal{TR}$. The second rule does the search and there are many possible ways for it to fail. Due to the transactional semantics of the logic, changes to the database state made while expanding these failing derivation paths are "forgotten" and new derivations are then tried. □

Note that so far we have been describing the consuming paths and Hamiltonian cycle examples procedurally, in terms of search. The actual model - theoretic semantics has none of that. It simply says that (in Example 3.4.1) the transaction $hCycle(Start, Start)$ can execute, i.e., that there is an executional entailment of the form

$$\mathbf{P}, \mathbf{D}_0, \ldots \mathbf{D}_n \models hCycle(Start, Start)$$

where $\mathbf{D_0}$ is the original graph, if and only if there is a Hamiltonian cycle in the graph. The actual cycle can be extracted from the above sequence of states. While the model theory is completely declarative, the aforesaid search does take place: it is performed by the sound and complete proof theory of $\mathcal{TR}$, which appears later.

3.4.2 Artificial Intelligence planning in the blocks world

The following rules define a STRIPS-like planner for building pyramids of blocks. We represent the blocks world using the fluents $on(x, y)$, which say that block x is on top of block y and $isclear(x)$, which says that nothing is on top of block x. The action $pickup(X)$ picks up block X and the action $putdown(X, Y)$ puts it down on top of block Y. The action $move(X, From, To)$ moves block X from its current position on top of block $From$ to a new position on top of block To. This action is defined by combining the afore mentioned actions *pickup* and *putdown* if certain pre - conditions

are satisfied. In addition, it defines the recursive action *stack*, which represents the pyramid building transaction.

$$
\begin{aligned}
& stack(0, Block). \\
& stack(N, X) \ :- \ N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \\
& \qquad \otimes on(Y, X). \\
& stack(N, X) \ :- \ N > 0 \otimes on(Y, X) \otimes unstack(Y) \\
& \qquad \otimes stack(N, X). \\
& unstack(X) \ :- \ on(Y, X) \otimes unstack(Y) \otimes unstack(X). \\
& unstack(X) \ :- \ isclear(X) \wedge on(X, table). \\
& unstack(X) \ :- \ (isclear(X) \wedge on(X, Y) \wedge Y \neq table) \\
& \qquad \otimes move(X, table). \\
& unstack(X) \ :- \ on(Y, X) \otimes unstack(Y) \otimes unstack(X). \\
& move(X, Y) \ :- \ X \neq Y \otimes pickup(X) \otimes putdown(X, Y). \\
& pickup(X) \ :- \ isclear(X) \otimes on(X, Y) \\
& \qquad \otimes delete(on(X, Y)) \otimes insert(isclear(Y)). \\
& putdown(X, Y) \ :- \ isclear(Y) \otimes \mathtt{not}\, on(X, Z1) \\
& \qquad \otimes \mathtt{not}\, on(Z2, X) \otimes delete(isclear(Y)) \\
& \qquad \otimes insert(on(X, Y)).
\end{aligned}
\tag{8}
$$

The above rules represent a straightforward algorithm for building a pyramid. The first rule says that stacking zero blocks on top of X is a no - op. The second rule says that, for bigger pyramids, stacking N blocks on top of X involves moving some other block, Y, on X and then stacking $N-1$ blocks on Y. To make sure that the planner did not remove Y from X while building the pyramid on Y, we are verifying that $on(Y, X)$ continues to hold at the end. Looking down at the definition of *move* we may notice that this action will not be performed if X is not clear. The third rule for *stack* says that in that case the robot should *unstack* whatever is no X and make X clear. The *unstack* action is also recursively defined and is, in a sense, the opposite of stacking. Definition of the other actions is straightforward. □

3.4.3 Evaluation for tabled $\mathcal{TR}$ implementations

The tabled $\mathcal{TR}$ implementations were tested on a workstation with Pentium dual - core 2.4GHz CPU and 3GB memory running on Ubuntu Linux and XSB Prolog version 3.2.

In describing the results of our tests, we use tables that show time (in seconds) and space (in Kb) costs for the different implementations using the problems in this section of gradually increasing size. To increase accuracy, we make the tests run for considerable amounts of time and avoid the possibility where different algorithms might pick up solutions that incur different costs. To this end, our tests compute *all* possible solutions for every problem in our suite and the numbers of solutions for each case are listed in the tables.

One of the important goals of this performance study is to demonstrate the benefits of table - skipping and double - differential logging. To show this, we include tables that display the numbers of tabled (saved) states, the numbers of times states were copied, and the numbers of times new states were compared with the contents of the state repositories (table - skipping implementations should do fewer of these operations). These tables also help us explain the reported times and assess the various trade - offs.

The overall conclusion from the study is that table - skipping and double - differential logging incur relatively small overheads for small problems, but bring substantial savings for larger problems and make them scale better. Likewise, maintaining data structures, like tries, that speed up querying of states brings significant speedups. The main overhead of those of our implementations that rely on tries (implementations 2, 4a, and 4b) is that copying tries is slow (7 times slower than copying lists in XSB). Since XSB's tries do not preserve the order on their contents, we had to also keep states as sorted lists — both time and space overhead. The use of B^+ trees in lieu of tries would have solved both of these problems, if an efficiently integrated version existed for XSB.

The suite of the different implementations of $\mathcal{TR}$ and of the test cases used in this comparison is provided at `http://flora.sourceforge.net/tr-interpreter-suite.tar.gz`.

Consuming paths

Table 2 shows execution times and memory consumption for the consuming paths problem for graphs with 100, 250, and 350 vertices. The row *# of Solutions* also shows the total number of solutions found.

It might seem surprising that Implementation 1, which incorporates only the basic optimizations, is one of the two best performers. Implementation 3, which adds table skipping, does only infinitesimally better. The explanation for this behavior is provided by Table 3: The nature of the consuming paths problem is such that all states must be tabled, so there is no advantage to table - skipping. Indeed, Table 3 shows that the number of tabled states and state comparisons is exactly the same for all implementations and depends only on the problem size. Using efficient data structures for logs, such as tries, does not help either. Only a relatively small number of queries is issued, and the benefits of faster querying using tries are negated by the overhead of copying tries compared to lists (earlier we mentioned that copying a trie takes 7 times longer). Tries also take more space than lists and, since the number of tabled states is the same for all implementations, the ones that maintain the logs using tries require significantly more space.

Nevertheless, it is easy to demonstrate that even for the consuming paths problem the use of table - skipping, tries, and double - differential logging is greatly beneficial. To see this, we can use the consuming paths method to find ten paths simultaneously in ten disjoint graphs. Our solution to this problem was obtained from the original consuming paths problem by simply repeating the “consuming” part of the rules in

(3) ten times on different *edge* predicates.

$$
\begin{aligned}
& reach(X,Y)\ :- \\
& \quad reach(X,Z) \\
& \qquad \otimes\, edge1(Z,Y) \otimes delete(edge1(Z,Y)) \\
& \qquad \otimes\, edge2(Z,Y) \otimes delete(edge2(Z,Y)) \\
& \qquad ... \\
& \qquad \otimes\, edge10(Z,Y) \otimes delete(edge10(Z,Y)). \\
& reach(X,X).
\end{aligned} \tag{9}
$$

Table 4 clearly shows that table - skipping, tries, and differential logging bring substantial time benefits, as Implementation 4a, which incorporates all of these optimizations is by far the best.

Similarly to the ordinary consuming paths example, the explanation is provided by Table 5: the number of tabled states and state comparisons performed by the table - skipping implementations 3 – 4b is ten times less than the corresponding numbers for implementations 1 and 2. We also see that table - skipping is better memory - wise, since implementations 3a and 3b consume half of the memory used by Implementation 1. Implementations 4a and 4b are much more memory hungry compared to implementations 3a and 3b because of the use of query tries, which consume much more memory than lists.

Hamiltonian cycles

Our next experiment computes all Hamiltonian cycles in graphs of sizes 50, 150, and 200 nodes. The results are shown in Table 6. This table provides several interesting observations:

- In constructing Hamiltonian cycles, many more queries are issued than in the case of consuming paths, so efficient data structures for querying are important. Thus, Implementation 2 is much faster than Implementation 1.

- Table 7 shows that using table - skipping reduces the number of tabled states and state comparisons by about 1/3. This is not high enough to offset the benefits of fast querying, so Implementation 1 is still slightly better that Implementations 3a and 3b.

- The querying overhead of double - differential logging is quite noticeable in this case, so the times for implementations 3b and 4b are higher than for implementations 2 and 3a. Nevertheless, Implementation 4b beats 3b (both use double differential logs) because it uses query tries rather than lists.

- The number of state comparisons performed by versions 3a and 4a is higher than in case of 3b and 4b. This validates our earlier observation that, since double - differential logging defers state copying and comparison (unlike single - differential logs), this might lead to fewer of such comparisons and copies being done overall. This problem is partially responsible for the higher runtime of Implementation 4a, which makes a larger number of expensive trie copies and comparisons. The other reason is that the query tries need to be transformed into sorted lists at state comparison.

As with consuming paths, it is easy to demonstrate that, for larger examples, the combination of table - skipping, query tries, and double - differential logging, i.e., Implementation 4b, scales better and is the overall winning combination. To this end, we can use the problem of constructing ten Hamiltonian cycles in ten disjoint graphs analogously to the way the ten simultaneous consuming paths problem was constructed in (9). Table 8 shows the results, which do not require further elaboration.

Blocks World

We conclude our performance study with the blocks world planning example for pyramids of 5, 6, and 7 blocks. Since the number of plans grows exponentially, we

could not evaluate larger problems on our test machine. Our results are shown in Table 9.

Since our largest problem uses only seven active blocks, the main differential logs and, especially, the residual logs tend to be quite small. As a result, there is no significant benefit in using query tries. Similarly, although Table 10 indicates that table skipping reduces the number of comparisons by the factor of 3, the overhead of creating and comparing all those extra states in implementations 1 and 2 is not high. On the other hand, implementations 3b and 4b suffer slightly due to the higher querying overhead associated with double - differential logging.

Interestingly, Table 10 again shows the higher number of state comparisons (and therefore state copies) performed by single - differential logging implementations 3a and 4a. In case of 4a, this leads to a significant overhead because copying and comparing tries takes 7 times more time than in case of lists. Since Implementation 3a uses lists (and these lists are short) this implementation is not seriously affected by all the extra copying.

Once again, transforming our planning problem into one in which ten separate pyramids are being built in ten parallel worlds clearly shows the advantage of our optimizations. The performance figures in Table 11 point to Implementation 4b as a clear winner.

In this section we presented several examples using the tabled $\mathcal{TR}$ interpreter. In Appendix A, we present a new language for complex event processing using transaction logic.

3.5 Tabling for Concurrent Transaction Logic Programs

We end this chapter by attacking the problem of lifting the tabling technique from the Sequential Transaction Logic to its concurrent version. Concurrent Transaction

Logic ($\mathcal{CTR}$) [BK96] extends the sequential version of the $\mathcal{TR}$ with the operator for concurrent or parallel execution "$|$" and the isolation operator "$\odot$". The formula $\phi|\psi$ means that the subtransactions ϕ and ψ execute concurrently (interleaved). The formula $\odot\phi$ means that ϕ must execute "atomically" and its execution should not be interleaved with any other transactions.

In the following paragraphs, we describe $\mathcal{CTR}$ and we show that the same tabling technique used for the sequential version is incapable of functioning for all the programs using the concurrent version of $\mathcal{TR}$ due to the fact that multiple parts of the program can be executed interleaved. We show that memoizing the set of so - called "hot" components and execution candidates at each step does not solve the infinite recursion problem by means of counter - examples.

Formally, Concurrent Transaction Logic extends the concept of *paths* (sequences of databases) to sequences of paths, called *multi - paths*. Formally, a *multi - path* is a finite sequence of paths, where each constituent path represents a period of continuous execution, separated by periods of suspended execution. For example, if $D_0, D_1, D_2, \ldots, D_6$ are database states, then $\langle D_0D_1D_2, D_3D_4, D_5D_6\rangle$ is a multi - path.

If the $\langle D_0D_1D_2, D_3D_4, D_5D_6\rangle$ multi - path was the execution history of an action ϕ then the action had three periods of continuous execution: $D_0D_1D_2$, D_3D_4 and D_5D_6. In the first period, ϕ changed the database from D_0 to D_2 going through the intermediate state D_1 and is suspended, re - awakening at state D_3. Similarly, in ϕ's second period of continuous execution, it changed the database from D_3 to D_4 and is suspended, while in its third period of continuous execution, it changed the database from D_5 to D_6 and finishes.

In the following definitions, we introduce the interleaving and reduction operations on multi - paths. The inverse operation of the *split* operation from Section 2.1 is called *concatenation*.

Definition 3.10 (Concatenation) *Suppose that* $\kappa = \langle D_1 \ldots D_k\rangle$ *and* $\kappa' = \langle D_k \ldots D_{k+l}\rangle$ *are two paths, where* D_k *is the last state of the path* κ *and the first state of the path* κ'*. Then, their concatenation is the path* $\kappa \circ \kappa' = \langle D_1 \ldots D_{k+l}\rangle$.

Definition 3.11 (Interleaving) *Suppose that $\pi_1, \ldots, \pi_n$ are multi - paths, then a multi - path π is an interleaving if it can be partitioned into order - preserving subsequences $\pi_1, \ldots, \pi_n$. The set of all interleavings of two multi - paths π_1 and π_2 is denoted $\pi_1 || \pi_2$ $\kappa' = \langle D_k \ \ldots \ D_{k+l} \rangle$.*

Definition 3.12 (Reduction) *Suppose that $\tau = \langle \kappa_1, \ \ldots, \ \kappa_n \rangle$ is a multi - path. If the paths κ_i and κ_{i+1} can be concatenated, for some i, then τ reduces to the multi - path $\tau' = \langle \kappa_1, \ \ldots, \kappa_{i-1}, \kappa_i \circ \kappa_{i+1}, \kappa_{i+2}, \ \ldots, \ \kappa_n \rangle$.*

$\mathcal{CTR}$ formulas are interpreted by multi - path structures which are used to tell which ground atoms (fluents and actions) are true on what multi - paths. They are similar to Herbrand path structures for Sequential $\mathcal{TR}$, the difference being that the mapping $\boldsymbol{I}$ conforms to the *reduction operation*: if a multi - path π_1 reduces to a multi - path π_2, then $\boldsymbol{I}(\pi_1)(a) = \mathbf{t}$ implies $\boldsymbol{I}(\pi_2)(a) = \mathbf{t}$ for every atom a (i.e., if a can execute along π_1 then it can also execute along π_2). It was also shown in Lemma 3.6 in [BK96] that this property is true for any formula ϕ if a multi - path π_1 reduces to a multi - path π_2, then $\boldsymbol{I}(\pi_1)(\phi) = \mathbf{t}$ implies $\boldsymbol{I}(\pi_2)(\phi) = \mathbf{t}$ (i.e., if ϕ can execute along π_1 then it can also execute along π_2).

Concurrent goals are defined recursively as follows:

- If P is a fluent or an action literal then P is a concurrent goal.
- If P is a concurrent goal, then so are $\odot P$ and $\diamond P$.
- If P_1 and P_2 are concurrent goals then so are $P_1 \mid P_2$, $P_1 \ \otimes \ P_2$ and $P_1 \ \wedge \ P_2$. $\square$

A **concurrent rule** is an expression of the form $H \ :- \ B.$, where H is a `not` - free literal and B is a concurrent goal.

A proof theory for concurrent Horn transaction logic An inference system for $\mathcal{CTR}$, $\mathcal{F}^C$ [BK96], verifies that $P, D_0 \text{---} \vdash (\exists)\, \phi$, saying, informally, that transaction $(\exists)\, \phi$ can successfully execute starting from state D_0 if and only if an execution path is found for the transaction (i.e., a sequence of databases $D_0, D_1, \ldots, D_n$ such that

$P, D_0, D_1, \ldots, D_n \vdash (\exists)\, \phi)$. The inference system $\mathcal{F}^C$ tries to execute transactions left - to - right, that is, left subtransactions first. These "left" subtransactions are called "hot" components and are defined as follows:

Definition 3.13 (Hot Components) *Consider ϕ a concurrent goal. Its set of hot components $hot\phi$ is:*

- $hot(\ ()\ ) = \{\}$, *where ( ) is the empty goal.*
- $hot(\ b\ ) = \{b\}$, *where b is an atomic formula.*
- $hot(\ \phi_1 \otimes \phi_2 \otimes \ \ldots\ \otimes \phi_n\ ) = hot(\phi_1)$.
- $hot(\ \phi_1 \mid \phi_2 \mid \ \ldots\ \mid \phi_n\ ) = hot(\phi_1) \cup \ \ldots\ \cup hot(\phi_n)$.
- $hot(\ \odot\ \phi_1) = \{\odot\ \phi_1\}$. □

Informally, *hot components* are those subprocesses that are ready to execute. In Figure 12, we illustrate the hot components of a few transaction formulas.

Definition 3.14 (Inference in $\mathcal{F}^C$) *Consider a concurrent Horn transaction base* **P** *and* **D** *is any database state.*

Axioms: $\mathbf{P}, \mathbf{D}\ \text{---} \vdash (\,)$, *for any* **D**.

Inference Rules: *In Rules 1–4 below, σ is a substitution, ϕ and ϕ' are concurrent serial conjunctions, and a is an atomic fluent or action in $hot(\phi)$.*

1. Applying transaction definitions*:*

Suppose $b \leftarrow \beta$ is a rule in **P** *whose variables have been renamed apart so that the rule shares no variables with ϕ. If a and b unify with a most general unifier σ, then*

$$\frac{\mathbf{P}, \mathbf{D}\ \text{---} \vdash (\exists)\, \phi'\sigma}{\mathbf{P}, \mathbf{D}\ \text{---} \vdash (\exists)\, \phi}$$ *where ϕ' is obtained from ϕ by replacing a hot occurrence of a with β.*

2. Querying the database*:*

If a is a fluent literal, $a\sigma$ and $\phi'\sigma$ share no variables, and $a\sigma$ is true in the database state $\mathbf{D}$ *then*

$$\frac{\mathbf{P},\mathbf{D}\,\text{---}\vdash(\exists)\,\phi'\,\sigma}{\mathbf{P},\mathbf{D}\,\text{---}\vdash(\exists)\,\phi}$$
where ϕ' is obtained from ϕ by deleting a hot occurrence of a.

3. Performing elementary updates*:*

If $a\sigma$ and $\phi'\sigma$ share no variables, and $a\,\sigma$ is an elementary action that changes state $\mathbf{D}_1$ *to state* $\mathbf{D}_2$ *then*

$$\frac{\mathbf{P},\mathbf{D}_2\,\text{---}\vdash(\exists)\,\phi'\,\sigma}{\mathbf{P},\mathbf{D}_1\,\text{---}\vdash(\exists)\,\phi}$$
where ϕ' is obtained from ϕ by deleting a hot occurrence of a.

4. Executing atomic transactions*:*

If $\odot\alpha$ is a hot component in ϕ then

$$\frac{\mathbf{P},\mathbf{D}_2\,\text{---}\vdash(\exists)\,(\alpha\,\otimes\,\phi')}{\mathbf{P},\mathbf{D}_1\,\text{---}\vdash(\exists)\,\phi}$$
where ϕ' is obtained from ϕ by deleting a hot occurrence of $\odot\alpha$.

Example 3.3 (CTR workflow example 1) *Lets consider the program where we have two recursive actions: a recursive producer a and a recursive consumer b. Note that here $ins/1$ and $del/1$ are considered* ***strict*** *updates, i.e., $ins(i)$ (i is a fluent) fails if t is already in the current database state, respectively, $del(i)$ fails if t is not in the current database state. In fact, this is just syntactical sugar and not a real restriction since strict updates can be always be represented with non - strict updates: $ins(i)$ as* $\textbf{not}\,t\,\otimes\,not-strict-insert(i)$ *and $del(i)$ as $t\,\otimes\,not-strict-delete(i)$.*

Suppose i is a fluent, $ins.i$ denotes the action of strict inserting i in the database and $del.i$ denotes the action of strict deleting i from the database. The following rules define a workflow:

$$\begin{aligned}
&c:-a|b.\\
&a:-a\,\otimes\,ins.i\,\otimes\,ins.i.\\
&a:-ins.i\,\otimes\,ins.i.\\
&b:-b\,\otimes\,del.i\,\otimes\,del.i\,\otimes\,del.i.\\
&b:-del.i\,\otimes\,del.i\,\otimes\,del.i.
\end{aligned}\tag{10}$$

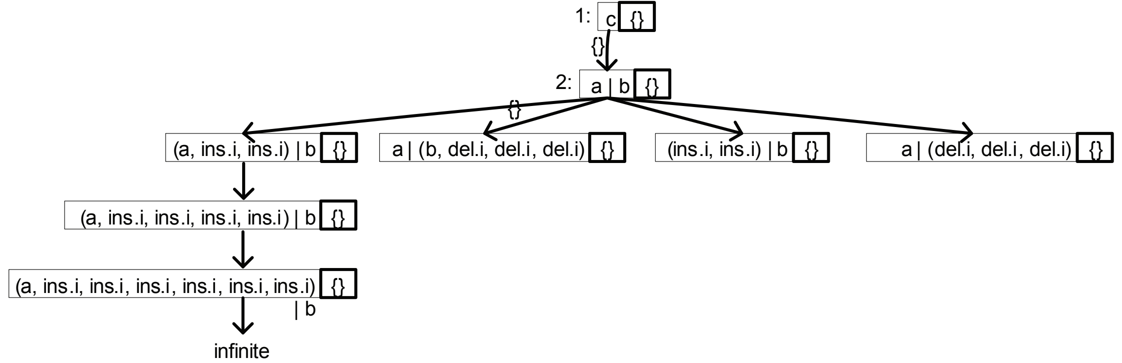

Figure 11: Part of the resolution tree for the Concurrent $\mathcal{TR}$ Example 3.3

And the query: $:-c.$ *is a query to execute the action c, which tries to communicate between the "producer" action a and the "consumer" action b.*

□

Like in serial Horn-$\mathcal{TR}$, the SLD style proof theory for several queries to recursive programs goes into infinite loops. Such an infinite branch in the SLD proof can be seen in Figure 11.

The $\mathcal{CTR}$ proof theory 3.14 can be modified in a similar way the proof theory for the serial Horn-$\mathcal{TR}$ was modified. However, we will show that the tabling algorithm cannot always be applied to $\mathcal{CTR}$ because no suitable tabling component can be found. Tabling the entire goal does not solve the issue showed in Figure 11 because the goal is extended ad infinitum. Tabling the set of hot components is not correct either due to the interleaving of entire concurrent conjunctions and not only of the hot components. Consider a concurrent Horn transaction base $\mathbf{P}$, any database state $\mathbf{D}$, a substitution σ, any concurrent conjunctions ϕ and ϕ', the sets of hot components for the concurrent conjunctions ϕ and ϕ', namely $hot(\phi)$ and $hot(\phi')$, and an atomic fluent or action in $hot(\phi)$, a. For simplicity, lets also consider that all the predicates in the program are tabled. One set of hot components is said to *dominate* another set of hot components in tabled resolution if the for each goal in any of the sets there is a *dominant* goal (see Section 3.1) in the other set. For instance, if the dominance relation is the *variant* relation, i.e., identity up to variable renaming, then this type of $\mathcal{CTR}$ tabling is a form of variant tabling, while if for each goal in the first set of hot components there is a *subsuming* goal in the second set, then the technique is

a subsumption - based tabling. When a goal ϕ is called, the set of hot components $hot(\phi)$ is encountered and one hot component is selected as a candidate to be executed in a particular state, a check is made to see whether this is the first occurrence of this set of hot components and candidate component was tabled before in that state (i.e., no dominating set of hot components and dominant candidate were encountered before in the same state). If pair formed by the set of hot components and the execution candidate is new, the tuple $(hot(\phi), candidate, state)$ is saved in the global data structure called the *table space*, and the evaluation uses normal clause resolution to compute answers and generate new return database states.

Lets consider that we modified the first inference rule in the $\mathcal{CTR}$ proof theory 3.14.

Suppose ϕ is a goal for the program $\mathbf{P}$. If the pair $(hot(\phi), candidate)$ is encountered for the first time at state $\mathbf{D}$ (i.e., no dominating entry $(Set, Candidate, \mathbf{D})$ is in the table space), then the transaction definitions are applied as before. Let $b \leftarrow \beta$ be a rule in $\mathbf{P}$ whose variables have been renamed apart so that the rule shares no variables with ϕ. If the candidate a and the head b unify with a most general unifier σ, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi'\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi}$$

where ϕ' is obtained from ϕ by replacing the hot occurrence of the candidate a with β.

The computed (*candidate answer - unification*, *new - state*) answer pairs cannot be recorded for the aforesaid $(hot(\phi), candidate, state)$ entry because this result is due to the interleaving of all parts of concurrent conjunctions, and not of the execution of only the hot candidate. As a consequence, in the general case, there is nothing to be saved as answers for any tuple $(Set, candidate, state)$ in the table space. We also show by means of a counter - example that such a tabling algorithm for $\mathcal{CTR}$ does not help with termination. Figure 13 shows the application of this tabling algorithm on the Example 3.3 and the query to c fails on all paths. However, Figure 12 shows a successful derivation. In consequence the method in incomplete.

However, the proof procedure $\mathcal{F}^T$ works for programs with no interactions between concurrent branches in the transaction base. If ϕ was a goal for the program $\mathbf{P}$, with a candidate a in the set of hot components $hot(\phi)$, and $b \leftarrow \beta$ the rule in $\mathbf{P}$ whose

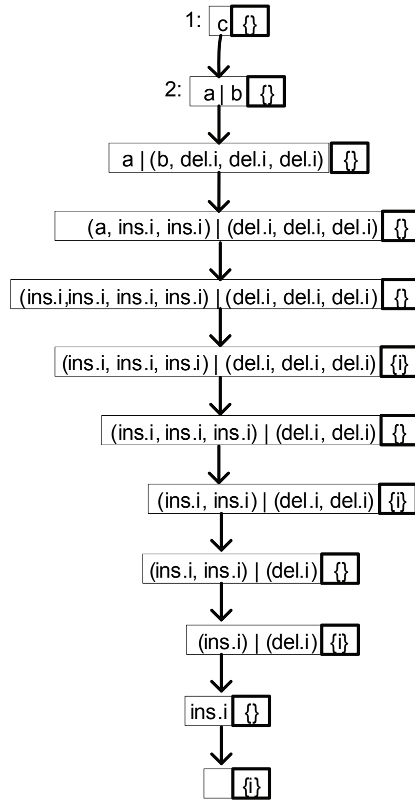

Figure 12: A successful branch in the resolution tree for the Concurrent $\mathcal{TR}$ Example 3.3

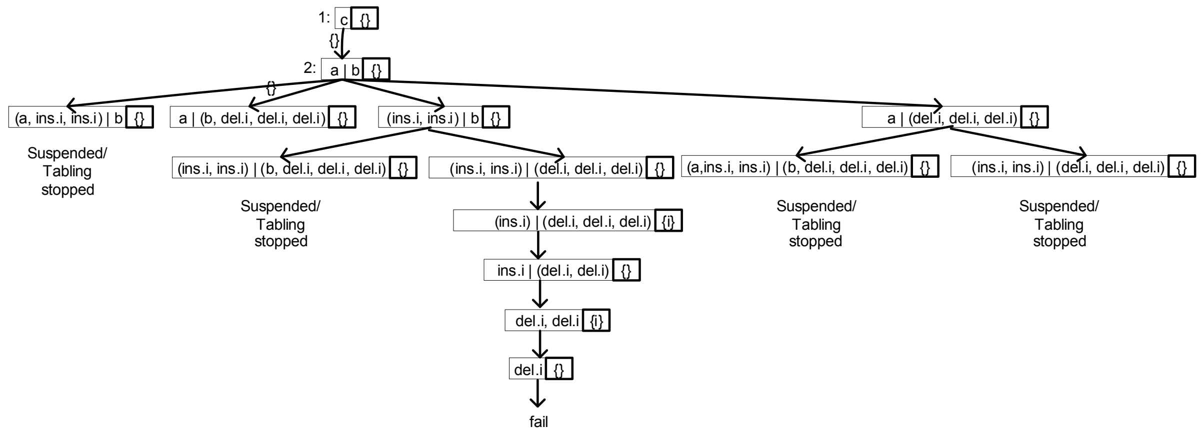

Figure 13: Tabling of only hot components for the Concurrent $\mathcal{TR}$ Example 3.3

variables have been renamed apart so that the rule shares no variables with ϕ, then, for non - interfering computations, if the sequent $\mathbf{P}, \mathbf{D} \text{---} \mathbf{D}' \vdash (\exists)\beta\sigma$ is derived in isolation with the rest of ϕ and the answer $(hot(\phi), \beta\sigma, \mathbf{D}')$ can be added to the answer table associated with the table entry $(hot(\phi), a, \mathbf{D})$.

In non - interfering computations, there are either only queries, only inserts, or only deletes, in which case the outcome of a goal for different interleavings does not change because the different branches do not interact. Such interactions can be found by examining the rules in the transaction base using an algorithm based on the dependency graph to detect the dependencies between predicates. Given a $\mathcal{CTR}$ program, the nodes in the dependency graph are: all predicates that are heads of rules and all queries and elementary operations invoked in the rules. The graph has an edge from X to Y, if for some rules X is the predicate in the head of the rule and Y is a predicate or an elementary operation in the rule body.

Example 3.4 (Financial Transactions) *Lets consider the program where we have the balance of a bank account represented by relation balance(Acct, Amt) and the following transactions: $change_balance(Acct, Bah, Bal2)$, to change the balance of an account from Ball to Bal2, $withdraw(Amt, Acct)$ to withdraw an amount from an account, $sell(Product, Amt, Acct)$ to sell a product to a customer with the account Acct.*

$$
\begin{array}{l}
withdraw(Amt, Acct) :- \ balance(Acct, Bal) \ \otimes \ Bal \geq Amt \\
\qquad\qquad \otimes \ change_balance(Acct, Bal, Bal - Amt). \\
change_balance(Acct, Bal, Bal2) :- \ del.balance(Acct, Ball) \ \otimes \ ins.balance(Acct, Bal2). \\
sell(Amt, Product, Acct) :- \ withdraw(Amt, Acct) \mid ins.delivered(Product, Acct).
\end{array}
\tag{11}
$$

To withdraw Amt from an account, Acct, the balance of the account is retrieved by the query $balance(Acct, Bal)$ and, then the test $Bal \geq Amt$ compares the balance with the amount to ensure that the account will not be overdrawn. The change_balance rule uses two elementary updates, del.balance and ins.balance, to change the balance of an account from Ball to Bal2. The third rule defines the action sell as withdrawing the money form the account and, in parallel, delivering the product. If one of the transfers succeeds and the other fails, then $\mathcal{CTR}$ query for $sell(Amt, Product, Acct)$

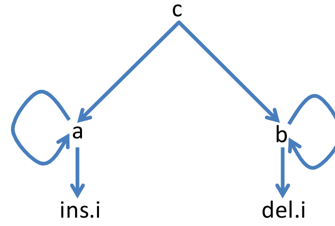

Figure 14: The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.3

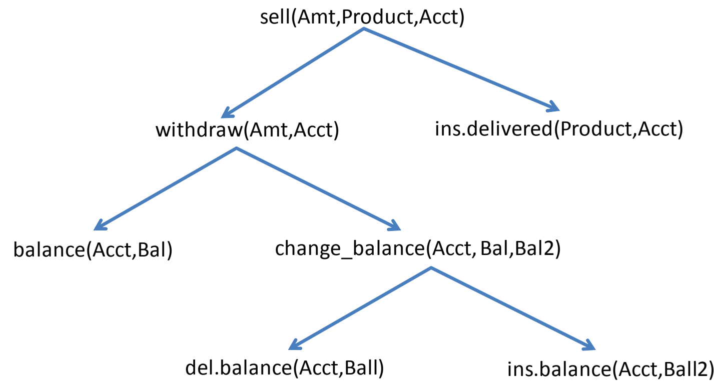

Figure 15: The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.4

behaves correctly, rolling back the entire transaction.

□

The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.3 is depicted in Figure 14, while The dependency graph for the Concurrent $\mathcal{TR}$ Example 3.4 is depicted in Figure 15.

Given an action predicate a, the *insert set* of a, written $Ins(a)$, is the set of all $insert(f)$ predicates reachable from node a in the dependency graph, where f is a fluent predicate. For an action a, the *query set* of a, written $Query(a)$, is the set of all calls to f predicates reachable from node a in the dependency graph, where f is a fluent predicate. Similarly, given an action predicate a, the *delete set* of a, written $Dels(a)$, is the set of all $delete(f)$ predicates reachable from node a in the dependency graph, where f is a fluent predicate. Two action predicates, $a1$ and $a2$, do not interact if the intersection of the set $Ins(a1)$ with $Dels(a2)$, the intersection of the set $Dels(a1)$ with $Ins(a2)$, the intersection of the set $Query(a1)$ with $Ins(a2)$, the intersection of the set $Ins(a1)$ with $Query(a2)$, the intersection of the set $Query(a1)$ with $Dels(a2)$

and the intersection of the set $Dels(a1)$ with $Query(a2)$ are empty. For concurrent goals where the concurrent actions do not interact the tabling technique memoizing the set of "hot" components and the candidate at each step solves the infinite recursion problem because the concurrent process can be reduced to sequential cases. In can be easily seen from the dependency graph 14 for the Concurrent $\mathcal{TR}$ Example 3.3 that the intersection of the set $Ins(a)$ with $Dels(b)$ is non empty. The dependency graph 15 for the Concurrent $\mathcal{TR}$ Example 3.4 also shows that the above condition for the concurrent formula $withdraw(Amt, Acct) \mid ins.delivered(Product, Acct)$ is satisfied, in which case the two actions are non - interacting and the tabling algorithm is possible. The concurrent formula $withdraw(Amt, Acct) \mid ins.delivered(Product, Acct)$ can be reduced to a sequential case by writing the last rule as two definitions of the action *sell*:

$$\begin{aligned} &sell(Amt, Product, Acct) :- ins.delivered(Product, Acct) \otimes withdraw(Amt, Acct). \\ &sell(Amt, Product, Acct) :- withdraw(Amt, Acct) \otimes ins.delivered(Product, Acct). \end{aligned} \tag{12}$$

in which case the tabling algorithm reduces to the Sequential $\mathcal{TR}$ tabling inference system $\mathcal{F}^T$.

Graph size	100		250		350	
# of Solutions	5050		31375		61425	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.128	806	1.544	4843	3.940	9473
2	0.212	5538	2.292	66413	5.996	173389
3a	0.136	807	1.540	4843	3.924	9473
3b	0.152	806	1.672	4843	4.608	9473
4a	0.224	10325	2.796	128434	7.880	337070
4b	0.204	5538	2.128	66413	5.680	172976

Table 2: Times for finding consuming paths in graphs

Graph size	100		250		350	
	States	Comp.	States	Comp.	States	Comp.
1	5051	5050	31376	31375	61426	61425
2	5051	5050	31376	31375	61426	61425
3a	5051	5050	31376	31375	61426	61425
3b	5051	5050	31376	31375	61426	61425
4a	5051	5050	31376	31375	61426	61425
4b	5051	5050	31376	31375	61426	61425

Table 3: Numbers of tabled states and state comparisons for finding consuming paths in graphs

Graph size	100		200		250	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	6.236	4580	47.642	18219	92.425	28881
2	8.568	371762	M.Err.	M.Err.	M.Err.	M.Err.
3a	4.796	2533	37.182	10066	71.840	15620
3b	4.024	2533	30.073	10065	58.083	15620
4a	1.780	77873	13.536	596734	25.929	1155434
4b	1.292	39744	8.564	301429	16.325	582398

Table 4: Time and space for building 10 consuming paths in 10 graphs

Graph size	100		200		250	
	States	Comp.	States	Comp.	States	Comp.
1	50501	50500	201001	201000	313751	313750
2	50501	50500	201001	201000	313751	313750
3a	5051	5050	20101	20100	31376	31375
3b	5051	5050	20101	20100	31376	31375
4a	5051	5050	20101	20100	31376	31375
4b	5051	5050	20101	20100	31376	31375

Table 5: Numbers of tabled states and state comparisons for building 10 consuming paths in 10 graphs

Graph size	50		150		200	
# of Solutions	50		150		200	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.252	2412	8.392	51543	23.405	118248
2	0.244	6111	4.144	132082	9.148	303932
3a	0.164	2362	3.956	51091	10.100	118566
3b	0.236	7337	5.644	187927	13.968	442537
4a	0.300	15284	6.852	330211	16.105	755352
4b	0.300	15446	5.696	379042	12.584	885453

Table 6: Times for finding Hamiltonian cycles in graphs

Graph size	50		150	
	States	Comp.	States	Comp.
1	7403	7500	67203	67500
2	7403	7500	67203	67500
3a	4903	5051	44703	45151
3b	4903	5000	44703	45000
4a	4903	5051	44703	45151
4b	4903	5000	44703	45000

Table 7: Numbers of tabled states and state comparisons for finding Hamiltonian cycles

Graph size	50		150	
	CPU	Mem.	CPU	Mem.
1	4.912	164777	M.Err.	M.Err.
2	6.052	424113	M.Err.	M.Err.
3a	3.076	9878	86.505	255174
3b	4.340	14854	105.814	391963
4a	1.656	58959	M.Err.	M.Err.
4b	1.356	46072	27.4210	1228925

Table 8: Times for finding 10 Hamiltonian cycles in 10 graphs

Blocks	5		6		7	
# of Pyramids	120		720		5050	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.212	576	2.392	5586	29.265	63207
2	0.196	656	2.100	6197	26.265	68636
3a	0.196	546	2.192	5286	27.905	60105
3b	0.228	544	2.528	5284	31.661	60102
4a	0.288	3296	3.268	46269	41.958	1005012
4b	0.204	608	2.268	5793	28.117	64915

Table 9: Time and space requirements for building pyramids of N blocks in blocks worlds

Blocks	5		6		7	
	States	Comp.	States	Comp.	States	Comp.
1	1546	4210	13327	42792	130922	480326
2	1546	4210	13327	42792	130922	480326
3a	501	9767	4051	107882	37633	1364911
3b	501	1300	4051	13020	37633	144354
4a	501	9767	4051	107882	37633	1364911
4b	501	1300	4051	13020	37633	144354

Table 10: Numbers of tabled states and state comparisons for building pyramids in blocks worlds

Blocks	5		6		7	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	1.800	9696	21.457	72741	286.413	128150
2	1.780	29289	19.441	233285	M.Err.	M.Err.
3a	1.140	889	13.208	7346	172.838	55984
3b	1.808	892	21.433	7349	287.413	75930
4a	1.312	30988	15.588	409155	M.Err.	M.Err.
4b	1.096	1614	11.984	12854	148.109	128150

Table 11: Time and space requirements for building pyramids of N blocks in 10 parallel blocks worlds

Formula	Hot components
$a \mid b$	$\{a, b\}$
$a \mid delete(i) \otimes b$	$\{a, delete(i)\}$
$\odot(a \otimes b) \mid delete(i) \otimes b$	$\{\odot(a \otimes b), delete(i)\}$
$(a \mid b) \otimes (c \mid d)$	$\{a, b\}$
$(a \otimes b) \mid (c \otimes d)$	$\{a, c\}$

Table 12: $\mathcal{CTR}$ formulas and their hot components

Chapter 4

A Well-founded Semantics for Transaction Logic with Defaults and Argumentation Theories

Defeasible reasoning is an important paradigm, which has been extensively studied as a knowledge representation paradigm, including in fields such as policies, regulations, law, learning, and others [BH95, BE99, BE00, DST03, DS01, EFLP03, GS98, Gro99, Nut94, Pra93, SI00, WZL00, ZWB01]. We combine $\mathcal{TR}$ with defeasible reasoning and show that the resulting logic language has many important applications. This logic is called $\mathcal{TR}^{DA}$ (*Transaction Logic with Defaults and Argumentation Theories*) because it extends $\mathcal{TR}$ in the direction of the recently proposed *logic programming with defaults and argumentation theories* (LPDA) [WGK$^+$09b], a recently proposed unifying framework for defeasible reasoning. In order to accomplish the above tasks we define the well-founded semantics [VRS91] for $\mathcal{TR}$.

$\mathcal{TR}^{DA}$ extends traditional logic programming, Transaction Logic, and LPDA and their application domains. Moreover, we show that the combined logic enables a number of interesting applications, such as specification of defaults in action theories and heuristics for pruning search in such search - intensive applications as planning. We also demonstrate the usefulness of the approach by experimenting with a prototype of $\mathcal{TR}^{DA}$ and showing how heuristics expressed as defeasible actions can significantly reduce the search space as well as execution time and space requirements.

4.1 Defeasibility in Transaction Logic

In this section we define a form of defeasible Transaction Logic, which we call *Transaction logic with defaults and argumentation theories* ($\mathcal{TR}^{DA}$). The development was inspired by our earlier work on logic programming with argumentation theories, which did not support actions [WGK+09b]. Language-wise, the only difference between $\mathcal{TR}^{DA}$ and serial $\mathcal{TR}$ is that the rules in $\mathcal{TR}^{DA}$ are tagged.

4.1.1 $\mathcal{TR}^{DA}$ Syntax

Definition 4.15 (Tagged rules) *A **tagged rule** in the language $\mathcal{TR}^{DA}$ is an expression of the form*

$$@r\ H\ :-\ B. \tag{13}$$

*where the **tag** r of a rule is a term. The head literal, H, and the body of the rule, B, have the same restrictions as in Definition 2.2.*

*A serial $\mathcal{TR}^{DA}$ **transaction base** $\mathbf{P}$ is a set of rules, which can be **strict** or **defeasible**.* □

Definition 4.16 ($\mathcal{TR}^{DA}$ Transaction formula) *A $\mathcal{TR}^{DA}$ **transaction formula** in the language $\mathcal{TR}^{DA}$ is a literal, a serial goal, a tagged or an untagged serial rule.* □

We note that the rule tag in the above definition is not a rule identifier: several rules can have the same tag, which can be useful for specifying priorities among sets of rules.

Strict rules are used as *definite* statements about the world. In contrast, defeasible rules represent *defeasible defaults* whose instances can be "defeated" by other rules. Inferences produced by the defeated rules are "overridden." We assume that the distinction between strict and defeasible rules is specified in some way: either syntactically or by means of a predicate (note that in this thesis, we consider strict rules to be unlabeled rules as in Definition 2.2).

Definition 4.17 (Rule handle) *Given a rule of the form (13), the term*

$$\mathtt{handle}(r, H) \tag{14}$$

*is called the **handle** of that rule.* □

$\mathcal{TR}^{DA}$ transaction bases are used in conjunction with *argumentation theories*, which are sets of rules that define conditions under which some rule instances in the transaction base may be defeated by other rules. The argumentation theory and the transaction base share the same set of fluent and action symbols.

Definition 4.18 (Argumentation theory) *An **argumentation theory**, AT, is a set of strict serial rules. We also assume that the language of $\mathcal{TR}^{DA}$ includes a unary predicate, $\texttt{\$defeated}_{AT}$, which may appear in the heads of some rules in AT but not in the transaction base. A $\mathcal{TR}^{DA}$ $\mathbf{P}$ is said to be **compatible** with AT if $\texttt{\$defeated}_{AT}$ does not appear in any of the rule heads in $\mathbf{P}$,* □

The rules AT are used to specify how the rules in $\mathbf{P}$ get defeated. This can be accomplished using special predicates defined in $\mathcal{TR}^{DA}$, such as, the `!opposes` and `!overrides` predicates in the courteous argumentation theories. For the purpose of defining the semantics, we assume that the argumentation theories AT are grounded. This grounding can be done by appropriately instantiating the variables and meta - predicates in AT.

Although Definition 4.18 imposes almost no restrictions on the predicate $\texttt{\$defeated}_{AT}$, practical argumentation theories are likely to require that it is executed hypothetically, i.e., that its execution does not change the current state. This is certainly true of the argumentation theories used in this thesis.

4.1.2 $\mathcal{TR}^{DA}$ Well - founded Semantics

We extend the well - founded semantics for logic programing [VRS91] to $\mathcal{TR}^{DA}$ using the Przymusinski-style definition [Prz94]. In the following definition, we use the usual three truth values $\mathbf{t}$, $\mathbf{f}$, and $\mathbf{u}$, which stand for *true*, *false*, and *undefined*, respectively. We also assume the existence of the following total order on these values: $\mathbf{f} < \mathbf{u} < \mathbf{t}$.

Definition 4.19 (3-valued Partial Herbrand interpretation) *A **partial Herbrand interpretation** is a mapping $\mathcal{H}$ that assigns $\mathbf{f}$, $\mathbf{u}$ or $\mathbf{t}$ to every formula L in $\mathcal{B}$.*

A partial Herbrand interpretation $\mathcal{H}$ is ***consistent relative to an atomic formula*** L if it is not the case that $\mathcal{H}(L) = \mathcal{H}(\texttt{neg}\, L) = \mathbf{t}$. $\mathcal{H}$ is ***consistent*** if it is consistent relative to every formula. $\mathcal{H}$ is ***total*** if, for every ground ***not***-free formula L (other than $\mathbf{u}$), either $\mathcal{H}(L) = \mathbf{t}$ and $\mathcal{H}(\texttt{neg}\, L) = \mathbf{f}$ or $\mathcal{H}(L) = \mathbf{f}$ and $\mathcal{H}(\texttt{neg}\, L) = \mathbf{t}$.

Partial Herbrand interpretations are used to define *path structures*, which are used to tell which ground atoms (fluents and actions) are true on what paths. Path structures play the same role in $\mathcal{TR}$ and $\mathcal{TR}^{DA}$ as the role played by the classical semantic structures in classical logic. The semantic structures of $\mathcal{TR}^{DA}$ are *mappings* from paths to partial Herbrand interpretations.

Definition 4.20 (3-valued Herbrand Path Structure) *A* ***partial Herbrand Path Structure*** *is a mapping* $\boldsymbol{I}$ *that assigns a partial Herbrand interpretation to every* ***path*** *subject to the following restrictions:*

1. $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{t}$, *if* $d \in \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{f}$, *if* $\texttt{neg}\, d \in \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{u}$, *otherwise, for every ground base fluent literal* d *and every database state* $\mathbf{D}$.

2. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p\} \setminus \{\texttt{neg}\, p\}$ *and* p *is a ground fluent literal;*
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{f}$, *otherwise.*

3. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \setminus \{p\} \cup \{\texttt{neg}\, p\}$ *and* p *is a ground fluent literal;*
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{f}$, *otherwise.*

Without loss of generality, in defining the semantics of $\mathcal{TR}^{DA}$ we will consider ground rules only. This is possible because all variables in a rule are considered to be universally quantified, so such rules can be replaced with a set of all of their ground instantiations.

We assume that the language includes the following special propositional constants: $\mathbf{u}^{\pi}$ and $\mathbf{t}^{\pi}$, for each path π. Informally, $\mathbf{t}^{\pi}$ is a propositional transaction that is true precisely over the path π and false on all other paths; $\mathbf{u}^{\pi}$ is a propositional transaction that has the value $\mathbf{u}$ over π and is false on all other paths.

Definition 4.21 ($\mathcal{TR}^{DA}$ 3-valued Truth valuation in path structures) *Let* $\boldsymbol{I}$ *be a path structure,* π *a path,* L *a ground* `not` *-free literal, and let* F*,* G *ground serial goals We define* ***truth valuations*** *with respect to the path structure* $\boldsymbol{I}$ *as follows:*

- *If* $\mathbf{P}$ *is a* `not` *-free literal then* $\boldsymbol{I}(\pi)(p)$ *is already defined because* $\boldsymbol{I}(\pi)$ *is a Herbrand interpretation, by definition of* $\boldsymbol{I}$*.*

- *If* ϕ *and* ψ *are serial goals and* $\pi = \pi_1 \circ \pi_2$ *then* $\boldsymbol{I}(\pi)(\phi \otimes \psi) = \boldsymbol{min}(\boldsymbol{I}(\pi_1)(p), \boldsymbol{I}(\pi_2)(q))$*.*

- *If* ϕ *and* ψ *are serial goals then* $\boldsymbol{I}(\pi)(\phi \wedge \psi) = \boldsymbol{min}(\boldsymbol{I}(\pi)(p), \boldsymbol{I}(\pi)(q))$*.*

- *If* ϕ *is a serial goal then* $\boldsymbol{I}(\pi)(\boldsymbol{not}\,\phi) = \sim \boldsymbol{I}(\pi)(\phi)$*, where* $\sim \mathbf{t} = \mathbf{f}$*,* $\sim \mathbf{f} = \mathbf{t}$*, and* $\sim \mathbf{u} = \mathbf{u}$*.*

- *If* ϕ *is a serial goal and* $\pi = \langle \mathbf{D} \rangle$*, where* $\mathbf{D}$ *is a database state, then*
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \boldsymbol{max}\{\boldsymbol{I}(\pi')(\phi) \mid \pi' \text{ is a path that starts at } \mathbf{D}\}$
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \mathbf{f}$*, otherwise.*

- *For a strict serial rule* F `:-` G*,*
  $\boldsymbol{I}(\pi)(F \text{ :- } G) = \mathbf{t}$ *iff* $\boldsymbol{I}(\pi)(F) \geq \boldsymbol{I}(\pi)(G)$*.*

- *For a defeasible rule* $@r\ F$ `:-` G*,*
  $\boldsymbol{I}(\pi)(@\texttt{r F :- G}) = \mathbf{t}$ *iff*
  $\boldsymbol{I}(\pi)(F) \geq \boldsymbol{min}(\,\boldsymbol{I}(\pi)(G),\ \ \boldsymbol{I}(\langle D_0 \rangle)(\boldsymbol{not} \Diamond \texttt{\$defeated}(\texttt{handle}(r, F))))$*,*
  where D_0 *is the first database in the path* π*.*

- *For any path* π*:*
  $\boldsymbol{I}(\pi)(\mathbf{t}^{\pi}) = \mathbf{t}$ *and* $\boldsymbol{I}(\pi')(\mathbf{t}^{\pi}) = \mathbf{f}$*, if* $\pi' \neq \pi$*;*
  $\boldsymbol{I}(\pi)(\mathbf{u}^{\pi}) = \mathbf{u}$ *and* $\boldsymbol{I}(\pi')(\mathbf{u}^{\pi}) = \mathbf{f}$*, if* $\pi' \neq \pi$*.*

We will write $\boldsymbol{I}, \pi \models \phi$ *and say that* ϕ *is* **satisfied** *on path* π *in the path structure* $\boldsymbol{I}$ *if* $\boldsymbol{I}(\pi)(\phi)$=$\mathbf{t}$*.*
We will say that a path structure $\boldsymbol{I}$ *is* ***total*** *if, for every path* π *and every serial goal* ϕ*,* $\boldsymbol{I}(\pi)(L)$ *is either* $\mathbf{t}$ *or* $\mathbf{f}$*.* □

Definition 4.22 ($\mathcal{TR}^{DA}$ 3-valued Model of a transactional formula) *A path structure,* $\boldsymbol{I}$*, is a* model *of a transaction formula* ϕ *if* $\boldsymbol{I}, \pi \models \phi$ *for every path* π*.*

In this case, we write $\boldsymbol{I} \models \phi$ *and say that say that* $\boldsymbol{I}$ *is a* ***model*** *of* ϕ *or that* ϕ *is* ***satisfied*** *in* $\boldsymbol{I}$*. A path structure* $\boldsymbol{I}$ *is a model of a set of formulas if it is a model of every formula in the set.*

Definition 4.23 (Model of $\mathcal{TR}^{DA}$) *A path structure* $\boldsymbol{I}$ *is a model of a serial* $\mathcal{TR}^{DA}$ *transaction base* $\mathbf{P}$ *if all the rules in* $\mathbf{P}$ *are satisfied in* $\boldsymbol{I}$ *(that is, if* $\boldsymbol{I} \models R$ *for every* $R \in \mathbf{P}$*). Given a* $\mathcal{TR}^{DA}$ *transaction base* $\mathbf{P}$*, an argumentation theory AT, and a path structure* $\boldsymbol{M}$*, we say that* $\boldsymbol{M}$ *is a model of* $\mathbf{P}$ *with respect to the argumentation theory AT, written as* $\boldsymbol{M} \models (\mathbf{P}, AT)$*, if* $\boldsymbol{M} \models \mathbf{P}$ *and* $\boldsymbol{M} \models AT$*.* □

Like classical logic programs, the Herbrand semantics of serial $\mathcal{TR}$ can be formulated as a fixpoint theory [BK98a]. In classical logic programming, given two Herbrand partial interpretations σ_1 and σ_2, $\sigma_1 \preceq \sigma_2$ if all `not`-free literals that are true in σ_1 are also true in σ_2 and all `not`- literals that are true in σ_2 are also true in σ_1. Similarly, given two Herbrand partial interpretations σ_1 and σ_2, $\sigma_1 \leq \sigma_2$ if all `not`- free literals that are true in σ_1 are also true in σ_2 and all `not`- literals that are true in σ_1 are also true in σ_2.

Definition 4.24 (Order on Path Structures) *If* $\mathbf{M}_1$ *and* $\mathbf{M}_2$ *are two Herbrand partial path structures, then* $\mathbf{M}_1 \preceq \mathbf{M}_2$ *if* $\mathbf{M}_1(\pi) \preceq \mathbf{M}_2(\pi)$ *for every path,* π *(truth ordering). Similarly, we have* $\mathbf{M}_1 \leq \mathbf{M}_2$ *if* $\mathbf{M}_1(\pi) \leq \mathbf{M}_2(\pi)$ *for every path,* π *(information ordering).*

A model $\mathbf{M}$ of $\mathbf{P}$ is **minimal** with respect to $\preceq$ iff for any other model, $\mathbf{N}$, of $\mathbf{P}$ $\mathbf{N} \preceq \mathbf{M}$ implies $\mathbf{N} = \mathbf{M}$. The **least** model of $\mathbf{P}$ is a minimal model that is unique.

It is well - known that in ordinary logic programming any set of Horn rules always has a least model. In [BK95], it is shown that every definite Horn $\mathcal{TR}$ program has a unique least total model. Theorem 4.4, below, shows that this property is preserved by serial `not`-free $\mathcal{TR}$ programs, but in this case the model might be a partial path structure. Serial `not`-free programs are more general than the positive $\mathcal{TR}$ programs because the undefined propositional symbol $\mathbf{u}^{\pi}$ for some path π may occur in the bodies of the program clauses.

Theorem 4.4 (Unique Least Partial Model for serial `not`-free $\mathcal{TR}$ programs) *If* $\mathbf{P}$ *is a* **`not`**-*free* $\mathcal{TR}$ *program, then* $\mathbf{P}$ *has a least Herbrand model, denoted* $LPM(\mathbf{P})$.

Proof: See Appendix D. □

Example 4.5 *Let the* $\mathcal{TR}$ *program* $\mathbf{P}$ *be:*

$$
\begin{aligned}
a &\ :-\ \texttt{state}.\\
b &\ :-\ a \otimes \mathbf{u}^{\langle \mathbf{D}_\emptyset \rangle}.\\
c &\ :-\ c \otimes \mathbf{u}^{\langle \mathbf{D}_\emptyset \rangle}.
\end{aligned}
$$

where a*,* b*, and* c *are action symbols and* $\mathbf{D}_\emptyset$ *is the empty database state. The least partial model of* $\mathbf{P}$ *is a path structure that maps any 1-path to a classical Herbrand partial model where* a *is true,* c *is false, and* b *is undefined. All other paths are mapped to the classical Herbrand partial model where all formulas are mapped to* $\mathbf{u}$*. Note that* b *is* not *false in* $LPM(\mathbf{P})$ *because the truth value of the sequential conjunction of premises in the second rule is* $\mathbf{u}$*, so the truth value of* b *must be at least* $\mathbf{u}$*.* □

For `not`-free $\mathcal{TR}$ programs, the least partial model $LPM(P)$ can be obtained as the least fixed point of the immediate consequence operator $\hat{T}$, which is applied to all paths. However, we will not pursue this line here.

Next we define well - founded models for $\mathcal{TR}^{DA}$ by adapting the definition from [Prz94]. First, we define the quotient operator, which takes a $\mathcal{TR}^{DA}$ program $\mathbf{P}$ and a path structure $\boldsymbol{I}$ and yields a serial-Horn $\mathcal{TR}$ program $\dfrac{\mathbf{P}}{\boldsymbol{I}}$.

Definition 4.25 ($\mathcal{TR}^{DA}$ Quotient) *Let* $\mathbf{P}$ *be a set of* $\mathcal{TR}^{DA}$ *rules and* $\boldsymbol{I}$ *a path structure for* $\mathbf{P}$*. The* $\mathcal{TR}^{DA}$ ***quotient of*** $\mathbf{P}$ ***by*** $\boldsymbol{I}$*, written as* $\dfrac{\mathbf{P}}{\boldsymbol{I}}$*, is defined through the following sequence of steps:*

1. *First, each occurrence of every* **`not`** *- literal of the form* **`not`** L *in* $\mathbf{P}$ *is replaced by* $\mathbf{t}^\pi$ *for every path* π *such that* $\boldsymbol{I}(\pi)(\textbf{not}\, L) = \mathbf{t}$ *and with* $\mathbf{u}^\pi$ *for every path* π *such that* $\boldsymbol{I}(\pi)(\textbf{not}\, L) = \mathbf{u}$*.*

2. *For each labeled rule of the form @r L :- Body obtained in the previous step, replace it with the rules of the form:*

$$L :\text{-}\ \mathbf{t}^{\langle \mathbf{D}_t \rangle} \otimes Body$$
$$L :\text{-}\ \mathbf{u}^{\langle \mathbf{D}_u \rangle} \otimes Body$$

for each database state $\mathbf{D}_t$ *such that*

$$\boldsymbol{I}(\langle \mathbf{D}_t \rangle)(\boldsymbol{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L)))) = \mathbf{t}$$

and each database state $\mathbf{D}_u$ *such that*

$$\boldsymbol{I}(\langle \mathbf{D}_u \rangle)(\boldsymbol{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L)))) = \mathbf{u}$$

3. *Remove the labels from the remaining rules.*

The resulting set of rules is the quotient $\dfrac{\mathbf{P}}{\boldsymbol{I}}$. □

Note that in Step 1 of the above definition of the quotient each occurrence of $\texttt{not}\, L$ is replaced with different $\mathbf{t}^{\pi}$ and $\mathbf{u}^{\pi}$ for different π's, so every rule in $\mathbf{P}$ may be replaced with several (possibly infinite number of) not-free rules. All combinations of replacements for the not- literals in the body of the rules have to be used. Only the π's where $\boldsymbol{I}(\pi)(\texttt{not}\, L) = \mathbf{f}$ are not used, which effectively means that the rule instances that correspond to those cases are removed from consideration. Also note that, the $\mathcal{TR}^{DA}$ quotient of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to an argumentation theory AT (the program union $\mathbf{P} \cup AT$) for any path structure $\mathbf{I}$, $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{I}}$, is a negation-free $\mathcal{TR}$ program, so, by Theorem 4.4, it has a unique least Herbrand model, $LPM(\dfrac{\mathbf{P} \cup AT}{\boldsymbol{I}})$.

We will now give the definition for the immediate consequence operator Γ. For compatibility with the classical notations in logic programming, we will use the set representation of Herbrand models: $\boldsymbol{I}^{+} = \{L \mid L \in \boldsymbol{I}$ *is a* $\boldsymbol{not}$*-free literal*$\}$, $\boldsymbol{I}^{-} = \{L \mid L \in \boldsymbol{I}$ *is a* $\boldsymbol{not}$*-literal*$\}$ and $\boldsymbol{I} = \boldsymbol{I}^{+} \cup \boldsymbol{I}^{-}$.

Definition 4.26 ($\mathcal{TR}^{DA}$ immediate consequence operator) *The incremental consequence operator,* Γ*, for a* $\mathcal{TR}^{DA}$ *transaction base* $\mathbf{P}$ *with respect*

to the argumentation theory AT takes as input a path structure $\boldsymbol{I}$ *and generates a new path structure as follows:*

$$\Gamma(\boldsymbol{I}) =^{def} LPM\left(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}\right)$$

Suppose $I_\emptyset$ *is the path structure that maps each path* π *to the* empty *Herbrand interpretation in which all propositions are undefined (i.e., for every path* π *and every literal* L*, we have* $I_\emptyset(\pi)(L) = \mathbf{u}$.

The ordinal powers of the immediate consequence operator Γ *are defined inductively as follows:*

- $\Gamma^{\uparrow 0}(I_\emptyset) \; = \; I_\emptyset$*;*
- $\Gamma^{\uparrow \alpha}(I_\emptyset) \; = \; \Gamma(\Gamma^{\uparrow \alpha - 1}(I_\emptyset))$*, for* α *a successor ordinal;*
- $\Gamma^{\uparrow \alpha}(I_\emptyset)(\pi) \; = \; \cup_{\beta<\alpha}\Gamma^{\uparrow \beta}(I_\emptyset)(\pi)$*, for every path* π *and* α *a limit ordinal.*

□

The operator Γ is monotonic with respect to the $\leq$ order relation when $\mathbf{P}$ and AT are fixed (see Appendix E). Because Γ is monotonic, the sequence $\{\Gamma^{\uparrow n}(I_\emptyset)\}$ $(\Gamma^{\uparrow 0}(I_\emptyset),\ \Gamma^{\uparrow 1}(I_\emptyset),\ \Gamma^{\uparrow 2}(I_\emptyset),\ \ldots)$ has a least fixed point and is computable via transfinite induction (see Appendix E).

Definition 4.27 (Well-founded model) *The* ***well - founded model*** *of a* $\mathcal{TR}^{DA}$ *transaction base* $\mathbf{P}$ *with respect to the argumentation theory AT, written as* $WFM(\mathbf{P}, AT)$*, is defined as the limit of the sequence* $\{\Gamma^{\uparrow n}(I_\emptyset)\}$*.* □

The next theorem states that our constructive computation of the least model of the program $(\mathbf{P}, AT)$ is correct.

Theorem 4.5 (Correctness of the Constructive $\mathcal{TR}^{DA}$ Least Model)

$WFM(\mathbf{P}, AT)$ *is the least model of* $(\mathbf{P}, AT)$*.*

Proof: See Appendix E. □

The next theorem shows that $\mathcal{TR}^{DA}$ programs under the well - founded semantics reduce to ordinary $\mathcal{TR}$ programs under the same well - founded semantics. In conclusion, $\mathcal{TR}^{DA}$ can be implemented using ordinary transaction logic programming systems that support the well - founded semantics.

Theorem 4.6 ($\mathcal{TR}^{DA}$ Reduction) *$WFM(\mathbf{P}, AT)$ coincides with the well - founded model of the $\mathcal{TR}$ program $\mathbf{P}' \cup AT$, where $\mathbf{P}'$ is obtained from $\mathbf{P}$ by changing every defeasible rule* (@r L :- Body) $\in$ $\mathbf{P}$ *to the plain rule* L :- not ($\diamond$ \$defeated(handle(r, L))) $\otimes$ Body *and removing all the remaining tags.*

Proof: See Appendix F. □

4.2 Argumentation theory representatives

Various argumentation theories can be defined to abstract the multiple intuitions about defeasibility. These argumentation theories are a set of definitions for concepts that a reasoner might use to argue why certain conclusions are to be defeated or to win over other conclusions. In the following sections we define two such argumentation theories: one for the *generalized courteous logic programs* (GCLP) ([Gro99] being the only commercially available defeasible reasoning formalism, i.e., IBM's Common-Rules[1]), and one for defeasible logic, a popular formalism that attracts a lot of attention in the field.

4.2.1 The $GCLP^{\mathcal{TR}}$ courteous argumentation theory

As our first example of an argumentation theory, we present here a particularly interesting argumentation theory which extends *generalized courteous logic programs* (GCLP) [Gro99] to $\mathcal{TR}$ under the $\mathcal{TR}^{DA}$ framework. This argumentation theory was used in the trade 4.6 and planning 3.4.2 examples in Section 4.4.1. We will call this argumentation theory $GCLP^{\mathcal{TR}}$. As any argumentation theory in this framework, $GCLP^{\mathcal{TR}}$ defines a version of the \$defeated predicate using various auxiliary concepts. We define these concepts first. The user - defined predicates !opposes and !overrides are relations specified over rule handles telling the system what rules are in opposition, respectively, what rules are preferred over the application of other rules. For instance, in the example 4.6, the predicate instance !opposes(handle(_, *sell*(*Stock*)), handle(_, *buy*(*Stock*))) is used to specify that any rule whose head is an instance of the *sell*/1 relation is incompatible with any rule whose head is an

[1] http://www.alphaworks.ibm.com/tech/commonrules

instance of the $buy/1$ with the same argument $Stock$ (that is, selling and buying the same stock in the same state is contradictory). In a parallel manner, the predicate `!overrides` specifies that some actions have higher priority than other actions. For instance, in the same trade example 4.6, the predicate instance $\texttt{!overrides}(\texttt{handle}(sell_action, _), \texttt{handle}(buy_action, _))$ is used to specify that the rule $sell_action$ has higher priority than the rule buy_action, regardless to their rule heads if an opposition situation arises.

The predicate `$defeated` is defined indirectly in terms of the predicates `!opposes` and `!overrides`. In the following definitions the variables R and S are expected to range over rule handles, while the implicit current state identifier D is expected to range over the possible database states. A rule is `$defeated` if it is *refuted* or *rebutted* by some other rule, where the former rule itself is defeasible (in our case, tagged) and the winning rule is not *compromised*, or the rule is *disqualified*. We will define these relations shortly, for the moment we just mention the most common meanings of these predicates: `$refutes` means that a higher - priority rule implies a conclusion that is incompatible with the conclusion implied by the another rule, `$rebuts` means that a pair of rules assert conflicting conclusions without being able to select a conclusion "more important" than the other conclusion, `$compromised` means that it's argument rule handle is defeated by some other rule handle, while `$disqualified` is a special situation when a rule defeats itself (for instance, such a situation is actually possible in the block world when the action of moving an unique block requires this action to beat all other move actions, but not itself).

$$\begin{aligned}
\texttt{\$defeated}(R) &\ :- \ \texttt{\$refutes}(S,R) \wedge \ \texttt{not}\,\texttt{\$compromised}(S). \\
\texttt{\$defeated}(R) &\ :- \ \texttt{\$rebuts}(S,R) \wedge \ \texttt{not}\,\texttt{\$compromised}(S). \\
\texttt{\$defeated}(R) &\ :- \ \texttt{\$disqualified}(R).
\end{aligned} \tag{15}$$

In this thesis we define a single GCLP-style argumentation theory, so we will use the most common interpretation of the aforementioned predicates. However, the reader should keep in mind that the argumentation theory is an input in our theory and can be changed as needed.

A rule R `$refutes` another rule S if R has higher - priority than S and R's conclusion is incompatible with the conclusion of S. Two rule handles are in conflict

if they are both candidates and (their handles) are in opposition to each other. These are defined as follows:

$$\begin{aligned}&\texttt{\$refutes}(R,S)\texttt{ :- }\texttt{\$conflict}(R,S)\ \wedge\ \texttt{!overrides}(R,S).\\&\texttt{\$conflict}(R,S)\texttt{ :-}\\&\quad\texttt{\$candidate}(R),\texttt{\$candidate}(S),\texttt{!opposes}(R,S).\end{aligned} \tag{16}$$

A rule R `$rebuts` another rule S if the two rules assert conflicting conclusions, but neither rule is "more important" than the other, that is, there is preference relation can be inferred between the two rules. This intuition can also be expressed in several different ways, but we have selected the following definition 18 as the most intuitive definition matching the GCLP theory. We define a *candidate* rule handler as a rule instance whose body is hypothetically true in the current database state (that is, it *can be executed hypothetically* in the current state) in the rule 19, and the symmetric `!opposes` relation in the rule 20 with the addition that every literal must oppose its explicit negation (`neg`) in the rule 21. We use two meta-predicates, **body** and **call**, where the **body** meta - predicate in `$candidate` binds B to the body of a rule with handle R, and the **call** meta - predicate takes a serial goal and executes it. We emphasize that the key aspect of the candidacy predicate is the fact that the bodies are executed hypothetically, so they do not modify the current state of the database.

$$\begin{aligned}\texttt{\$rebuts}(R,S)\quad\texttt{:-}\quad&\texttt{\$candidate}(R)\wedge\texttt{\$candidate}(S)\wedge &(17)\\&\texttt{!opposes}(R,S)\wedge\texttt{not \$compromised}(R)\wedge &(18)\\&\texttt{not \$refutes}(_,R)\wedge\texttt{not \$refutes}(_,S).\end{aligned}$$

$$\texttt{\$candidate}(R)\quad\texttt{:-}\quad\mathbf{body}(R,B)\otimes\Diamond\mathbf{call}(B). \tag{19}$$

$$\texttt{!opposes}(X,Y)\ \texttt{:-}\ \texttt{!opposes}(Y,X). \tag{20}$$

$$\texttt{!opposes}(\texttt{handle}(_,H),\ \texttt{handle}(_,\texttt{neg}\ H)). \tag{21}$$

A rule is compromised if it is defeated, and it is disqualified if it transitively defeats itself. The predicate $\texttt{\$defeats}_{tc}$ denotes the transitive closure of the predicate `$defeats`.

$$
\begin{array}{l}
\texttt{\$compromised}(R) \ \texttt{:-} \ \texttt{\$refutes}(_, R) \wedge \texttt{\$defeated}(R). \\
\texttt{\$disqualified}(X) \ \texttt{:-} \ \texttt{\$defeats}_{tc}(X, X). \\
\texttt{\$defeats}_{tc}(X, Y) \ \texttt{:-} \ \texttt{\$defeats}(X, Y). \\
\texttt{\$defeats}_{tc}(X, Y) \ \texttt{:-} \ \texttt{\$defeats}_{tc}(X, Z) \wedge \texttt{\$defeats}(Z, Y).
\end{array}
\tag{22}
$$

As in [WGK+09b], one can define other versions of the above argumentation theory, which differ from the above in various edge cases. However, defining such variations is tangential to our main focus here.

4.2.2 An argumentation theory for defeasible logic

We develop here an argumentation theory that captures the reasoning in the Defeasible Logic family of logics [Nut94, ABGM01, AM02, MN06]. This form of defeasible reasoning is particularly interesting because Governatori, Rotolo and Sadiq used it to execute workflows in [GRS04]. Formally, Defeasible Logic is a triple (R, > , K), where K is a finite set of literals, R is a set of rules, such that if q is any ground literal then the rules whose head is q, $R[q]$ is finite, and ">" is a superiority relation on R. Defeasible Logic partitions the rules R into *strict*, *defeasible*, and *defeater* rules, where :

(a) the *strict* rules are rules which cannot be defeated and need to be satisfied even if the database is inconsistent,

(b) the *defeasible* rules are rules that can be defeated either by facts inferred by the strict rules or by the defeaters, and

(c) the *defeater* rules are used only to defeat other rules, but they do not produce any inferences. The purpose of the defeater rules is to block inferences produced by other rules.

The *opposition* among literals is limited to p and `neg` p, for each fluent p, while the use of the default negation is not allowed, so all literals are `not`-free, and rule tags are unique identifiers of the rules. The theory of Defeasible Logic easily translates into the computation of the fixed point of four sets: ground literals that are strictly

true, ground literals that are strictly false, ground literals that are defeasible true, and ground literals that are defeasible false.

We need a few special predicates provided by the interpreter: a meta - predicate **head/2** that binds the first argument to the head of a rule with the identifier the second argument, a meta - predicate **body/2** that binds the first argument to the body of a rule with the identifier the second argument, a meta - predicate **call/1** that takes a serial goal and executes it on an execution path and and a predicate **break_** $\otimes$ /3 that takes a serial conjunction $B_1 \otimes B_2 \otimes \ldots \otimes B_n$ and returns the first element of the conjunction as the second argument and the rest of the conjunction as the third argument or *state* if the conjunction was a single element B_1.

The program the following extra predicates: `!strict`/1 for rules that cannot be defeated, `$defeater`/1 for rules used only to defeat other rules, but do not produce any inferences, and $>$ /2 as a superiority relation between rules.

A rule is *defeated* if any of the following conditions hold: another conclusion in *conflict* with the current conclusion is *definitely* proved, the conclusion is detected by a defeater (because defeaters make no inferences), or the conclusion is *refuted*.

$$
\begin{array}{l}
\texttt{\$defeated}(\texttt{handle}(T,H)) \ \texttt{:-}\ \texttt{\$conflict}(\texttt{handle}(T,H), \texttt{handle}(S,H)) \\
\qquad\qquad \wedge\ \mathbf{head}(S,H)\ \wedge\ \texttt{\$definitely}(H). \\
\texttt{\$defeated}(\texttt{handle}(T,H)) \ \texttt{:-}\ \texttt{\$defeater}(\texttt{handle}(T,H)). \\
\texttt{\$defeated}(\texttt{handle}(T,H)) \ \texttt{:-}\ \texttt{\$refutes}(_, \texttt{handle}(T,H)).
\end{array}
$$

Two rules are in *conflict* if they are both candidates and their literals are incompatible (i.e., a literal L and its explicit negation $\texttt{neg}\, L$).

$$
\begin{array}{l}
\texttt{\$conflict}(\texttt{handle}(T1,H1), \texttt{handle}(T2,H2)) \ \texttt{:-} \\
\qquad\qquad \texttt{\$candidate}(\texttt{handle}(T1,H1)) \wedge\ \texttt{\$candidate}(\texttt{handle}(T2,H2)) \\
\qquad\qquad \wedge\ \texttt{!opposes}(H1,H2). \\
\texttt{\$candidate}(R)\ \texttt{:-}\ \mathbf{body}(R,B) \otimes \Diamond\mathbf{call}(B). \\
\texttt{!opposes}(L1, \texttt{neg}\, L1). \\
\texttt{!opposes}(\texttt{neg}\, L1, L1).
\end{array}
$$

A literal is *definitely* proved if it is the head of a strict rule whose body is proved only by *strict* clauses. We prove the body by proving all the literals in the body using the meta predicate **break_**$\otimes$ and recursion. The effects of the `$definitely` call are not visible to the rest of the computation.

$$
\begin{aligned}
&\texttt{\$definitely}(L)\ \texttt{:-}\ \Diamond strictly_proved(L).\\
&strictly_proved(state).\\
&strictly_proved(L)\ \texttt{:-}\ \texttt{!strict}(R) \wedge \mathbf{head}(R, L)\\
&\qquad\qquad \wedge\ \mathbf{body}(R, B) \otimes strictly_proved_conj(B).\\
&strictly_proved_conj(state).\\
&strictly_proved_conj(B)\ \texttt{:-}\ \mathbf{break_} \otimes (B, B_1, B_2)\\
&\qquad\qquad \otimes strictly_proved(B_1) \otimes strictly_proved_conj(B_2).
\end{aligned}
$$

An additional rule is added to the instances of the predicate $>/2$ to state that any strict rule has priority to any non - strict rule:

$$> (R1, R2)\ \texttt{:-}\ \texttt{!strict}(R1) \wedge \texttt{not}\ \texttt{!strict}(R1).$$

Finally, the `$refutes`/2 relation is defined using the notions of `$candidate` and `$conflict`.

$$
\begin{aligned}
&\texttt{\$refutes}(S, T)\ \texttt{:-}\ \texttt{\$conflict}(S, T)\ \wedge\ \texttt{\$candidate}(S)\ \wedge\ \texttt{\$candidate}(T)\\
&\qquad \wedge\ \texttt{not}\ \texttt{\$refutes}(_, S).\\
&\texttt{\$refutes}(_, S)\ \texttt{:-}\ \texttt{\$conflict}(S, T)\ \wedge\ \texttt{\$candidate}(T)\\
&\qquad \wedge\ > (T, S)\ \wedge\ \texttt{not}\ \texttt{\$defeater}(T).
\end{aligned}
$$

4.3 $\mathcal{TR}^{DA}$ discussion and related work

Although a great number of works deal with defeasibility in logic programming, few have goals similar to ours: to lift defeasible reasoning from static logic programming to a logic for expressing knowledge base dynamics, such as $\mathcal{TR}$. Such lifting opens up new applications for Transaction Logic by allowing it to take advantage of preferences among rules and defeasibility. As far as the actual chosen approach to defeasible reasoning is concerned, this work is based on [WGK+09b], and extensive in-depth

comparison with other works on defeasible reasoning can be found there. There, we compare LPDA based approaches to the frameworks presented by Gelfond and Son in [GS98] (i.e., the logic of prioritized defaults), by Delgrande, Schaub, and Tompits in [DST03] (i.e., the ordered logic programs), and by Eiter et al. in [EFLP03] (i.e., the meta - interpretation approach to handling preferences) because they allow adaptive behaviours in using the preference information similar to our argumentation various theories. The main difference from [GS98] is that our approach distills all the differences between the different default theories to the notion of an argumentation theory with a simple interface to the user - provided domain description, the predicate `$defeated`. In the case of [DST03], the framework does not come with a unifying model - theoretic semantics, but comes as a transformation of normal logic programs under the stable model semantics. The variable part is the transformation, which encodes a fairly low - level mechanism: the order of rule applications required to generate the preferred answer set. In the following paragraphs we will focus on comparing our work with prior research on defeasibility of actions.

The main contribution here relies not in the use of different argumentation theories, but in the lifting of LPDA to a dynamic logic, such as $\mathcal{TR}$. To our knowledge, none of the works surveyed by [DSTW04] has a similar goal as ours, but some defeasible logic formalisms match various applications of $\mathcal{TR}$, and such, we will compare our work with these works aimed to apply defeasible reasoning to various dynamic domains. In particular, we compare our work with the most representative works aiming to apply defeasible reasoning in planning represented in ASP, namely, the approach by Son and Pontelli in [SP02, SP03, SP04, SP06] and the approach by Delgrande, Schaub and Tompits in [DST04, DST07b, DST07a]. On another hand, the approaches adopted in [GRS04, GMS06, GR10] aim to apply defeasible reasoning for another application of the results presented in this thesis, namely, in modeling, execution and verification of workflows.

The work [SP06] develops a high-level language for the specification of preferences over trajectories and provides a logic programming encoding of the language based on answer set planning. They combine the action language $\mathcal{B}$ [GL98] with the prioritized default theory developed in [GS98]. $\mathcal{TR}^{DA}$ is quite different from [SP06] in that it is a full-fledged logic that combines both declarative and procedural elements,

while [SP06] specifically is geared towards specifying preferences over trajectories in planning. Whereas $\mathcal{TR}^{DA}$ deals with infinite domains and allows function symbols and non-deterministic actions, the approach in [SP06] considers only planning with complete information on finite domains and deterministic actions. Thus, although the two approaches have common applications in the area of planning, they target different knowledge representation scenarios.

The approach in [DST04, DST07a] uses two types of preferences over plans for achieving goals in a plan. The *choice order* specifies when a plan satisfying a goal, ϕ_1, is preferred over another plan satisfying another goal ϕ_2. The *temporal order* specifies when the planning heuristic has a preference concerning the order in which subgoals are to be achieved. That is, when subgoals must become true in a specific order. The set of solution histories is ordered according to these partial order relations, $\leq_c$ (choice) and $\leq_t$ (temporal), and the maximal elements are chosen as the most preferred solutions. Both of these types of preference can be expressed in the $\mathcal{TR}^{DA}$ framework, although due to the difference in the semantics the exact relationship needs further study. In the *choice ordering*, the application of rule definitions for actions whose effect is to update fluents in the fluent serial goal ϕ_2 are defeated when actions that update fluents in the fluent serial goal ϕ_1 can be executed. In the *temporal ordering*, application of rule definitions for actions whose effect is to update fluents in the fluent serial goal ϕ_2 are defeated if some fluents in the fluent serial goal ϕ_1 were not satisfied. This encoding mirrors the dualism between fluents and corresponding actions that update these fluents signaled in [DST07a]. Moreover, while the original work by Delgrande, Schaub, and Tompits in [DST03] was a framework of ordered logic programming that could use a variety of preference handling strategies, its application to planning resumes to a single behaviour of dealing with preferences.

Other systems have also adopted various kinds of preferences in planning, for instance, quality of planning in [Bal04], solving multiple prioritized goals [Bal09], but these works do not study a unified context for an active deductive database such as ours, but special cases of using defeasible logic programming formalisms to implement certain problems or translations of certain dynamic languages (for instance, action languages) in LP formalisms supporting certain kinds of defeasibility. In [EFL+03], a framework for planning with cost preferences is introduced. Each action is assigned a

numeric cost, and plans with the minimal cost are considered to be optimal. Clearly, this work uses a completely different type of preferences and tackles a different and very specific problem in planning, which we do not address. Similar to our work, [EFL+03]'s work is the only other work that deals with planning in the presence of non - deterministic actions.

Finally, regarding dealing with preferences in modeling, execution and verification of workflows, we mention the work of Governatori et al. on modelling notions like delegation of tasks in the execution of a workflow. Another work by the same group, [GMS06], deals compliance of workflows to a given regulation formulated in a variant of deontic logic, allowing expressions similar to what we have in transaction logic (with sequences of task/actions and or branching of actions), but not dealing with defeasible reasoning. Recently, [GR10] extended the work of [GMS06] to model control flow patterns in workflows. However, in the last two papers defeasible reasoning is not studied at all, while in the case of the first paper is just tangential to our goals, being applied in the special case of delegation from one agent to another (more important) agent.

4.4 Applications, implementation and evaluation

We implemented an interpreter for $\mathcal{TR}^{DA}$ in XSB [2] and tested it on a number of examples, including Example 3.4.2. The goal of these tests was to demonstrate how preferential heuristics can be expressed in $\mathcal{TR}^{DA}$ and to evaluate their effects on the efficiency of planning (see Example 4.7).

4.4.1 $\mathcal{TR}^{DA}$ Aplications in action priorities, planning and workflows

In this section, we employ several applications in order to illustrate the advantages of extending Transaction Logic with the well founded semantics and defeasible reasoning. Using the $GCLP^{\mathcal{TR}}$ courteous argumentation theory, the rules in these examples are more powerful than simple $\mathcal{TR}$ rules since they use a relative independence in writing

[2] `http://xsb.sourceforge.net/`

the rule bodies, but retain the concept of defeasible reasoning. The meaning of the `!opposes` and `!overrides` predicates is the same as in the Section 4.2.1.

Example 4.6 (Stock market actions) *Consider a broker who trades stock on the market. He uses a computerized system, which makes various decisions about buying and selling stocks. The system weighs recommendations, which sometimes might conflict with each other, and performs appropriate actions. For simplicity, we ignore issues such as the amount of funds available for purchase and so on.*

$$\begin{aligned}
&@buy_action buy(Stock, Amount)\ :-\\
&\qquad recommendation(buy, Stock) \otimes owns(Stock, Qty) \otimes\\
&\qquad delete(owns(Stock, Qty)) \otimes insert(owns(Stock, Qty + Amount)).\\
&@sell_action sell(Stock, Amount)\ :-\\
&\qquad recommendation(sell, Stock) \otimes owns(Stock, Qty) \otimes\\
&\qquad delete(owns(Stock, Qty)) \otimes insert(owns(Stock, Qty - Amount)).\\
&\texttt{!opposes}(sell(Stock), buy(Stock)).\\
&\texttt{!overrides}(sell_action, buy_action).\\
&recommendation(buy, C)\ :-\ services(X).\\
&recommendation(sell, C)\ :-\ media(X).\\
&services(acme).\\
&media(acme).\\
&owns(acme, 100).\\
&trade(Stock, Amount)\ :-\ buy(Stock, Amount).\\
&trade(Stock, Amount)\ :-\ sell(Stock, Amount).
\end{aligned} \tag{23}$$

The above rules specify that selling and buying the same stock as part of the same decision is contradictory, so these rules are declared to be in conflict. To be on the safe side, the second rule (sell) is said to override the first (buy). Lets consider an existential goal $(\exists)trade(acme, 100)$. Without the `!opposes` *and* `!overrides` *information this goal would have two non - deterministic possible executions: one in which the trader buys an additional 100 stocks in the company acme, and another one in which the trader sells his 100 stocks because he got recommendations both to buy stocks for services companies and to sell the stocks for media companies. However, the second execution is preferred because, in such a contradictory state it's advisable to sell the stocks.* □

Example 4.7 (Blocks world planning) *This example illustrates the use of defeasible reasoning for heuristic optimization of planning in the blocks world. The example is similar to the one used in Section 3.4.2, but here the rules are labeled and additional information about the opposition and the priority between actions is used in the defeasible reasoning. The* $\mathcal{TR}^{DA}$ *program below is designed to build pyramids of blocks that are stacked on top of each other so that smaller blocks are piled up on top of the bigger ones. The construction process is non - deterministic and several different blocks can be chosen as candidates to be stacked on top of the current partial pyramid. The heuristic uses defeasibility to give priority to larger blocks so that higher pyramids can be constructed.*[3]

In this example, we represent the blocks world using the familiar fluents $on(x, y)$ *and* $isclear(x)$ *(see Section 3.4.2), but also the new fluent* $larger(x, y)$*, which says that the size of* x *is larger than the size of* y*. The action* $pickup(X)$ *picks up block* X *and the action* $putdown(X, Y)$ *puts it down on top of block* Y*. These actions are specified by the second and third rules, respectively. The action* $move(Block, From, To)$*, specified by the first rule, moves* $Block$ *from its current position on top of block* $From$ *to a new position on top of block* To*, where the block* $Block$ *is smaller then the block* To*. This action is defined by combining the afore mentioned actions pickup and putdown if certain pre - conditions are satisfied. The stacking action (included later in this section) then uses the* move *action to construct pyramids. The key observation here is that at any given point several different instances of the rule tagged with* mv_rule *might be applicable and several different moves might be performed. The predicate* `!opposes` *stipulates that two different move - actions for different block are considered to be in conflict (because only one action at a time is allowed).*

[3] *For more information about planning with* $\mathcal{TR}$ *the reader is referred to [BK95].*

$$
\begin{aligned}
&@mv_rule(Block, To) \quad move(Block, From, To) \;:- \\
&\qquad (on(Block, From) \wedge larger(To, Block)) \otimes \\
&\qquad pickup(Block, From) \otimes putdown(Block, To). \\
&pickup(X, Y) \;:- \; (isclear(X) \wedge on(X, Y)) \otimes \\
&\qquad delete(on(X, Y)) \otimes \; insert(isclear(Y)). \\
&putdown(X, table) \;:- \; (isclear(X) \wedge \textbf{not}\, on(X, Z)) \\
&\qquad \otimes insert(on(X, table)). \\
&putdown(X, Y) \;:- \; (isclear(X) \wedge isclear(Y) \wedge \textbf{not}\, on(X, Z)) \\
&\qquad \otimes delete(isclear(Y)) \otimes insert(on(X, Y)). \\
&!\mathtt{opposes}(move(B1, F1, T1), move(B2, F2, T2)) \;:- \; B1 \neq B2.
\end{aligned}
\tag{24}
$$

Note that the first rule is tagged with a term, $mv_rule(Block, To)$ *and, according to our conventions, such a rule is defeasible. Various heuristics can be used to improve construction of plans for building pyramid of blocks. In particular, we can use preferences among the rules to cut down on the number of plans that need to be looked at. For instance, the following rule says that move - actions that move bigger blocks are preferred to move - action that move smaller blocks (unless the blocks are moved down on the table surface).*

$$
\begin{aligned}
&!\mathtt{overrides}(mv_rule(B2, To), mv_rule(B1, To)) \;:- \\
&\qquad larger(B2, B1) \wedge To \neq table.
\end{aligned}
\tag{25}
$$

Consider the configuration of blocks in (26).

$$
\begin{aligned}
&on(blk1, blk4).\; on(blk2, blk5). \\
&on(blk3, table).\; on(blk4, table).\; on(blk5, table). \\
&isclear(blk1).\; isclear(blk2).\; isclear(blk3). \\
&larger(blk2, blk1).\; larger(blk3, blk1).\; larger(blk3, blk2). \\
&larger(blk4, blk1).\; larger(blk5, blk2).\; larger(blk2, blk4).
\end{aligned}
\tag{26}
$$

Although, both $blk1$ *and* $blk2$ *can be moved on top of* $blk3$*, moving* $blk2$ *has higher priority because it is larger.*

For moving blocks to the table surface, we use the opposite heuristic, one which

prefers unstacking smaller blocks:

$$\texttt{!overrides}(mv_rule(B2, table), mv_rule(B1, table)) \ :- \ larger(B1, B2). \tag{27}$$

In our example, this makes unstacking blk1 and moving it to the table surface preferable to unstacking blk2, since the former is a smaller block. This blocks the opportunity to then move blk4 on top of blk2 and subsequently put blk1 on top of blk4. These preference rules can be applied to a pyramid-building program like this:

$$\begin{aligned}
&stack(0, Block).\\
&stack(N, X) \ :- \ N > 0 \otimes move(Y, _, X) \otimes stack(N-1, Y) \otimes on(Y, X).\\
&stack(N, X) \ :- \ (N > 0 \wedge on(Y, X)) \otimes unstack(Y) \otimes stack(N, X).\\
&\\
&unstack(X) \ :- \ on(Y, X) \otimes unstack(Y) \otimes unstack(X).\\
&unstack(X) \ :- \ isclear(X) \wedge on(X, table).\\
&unstack(X) \ :- \ (isclear(X) \wedge on(X, Y) \wedge Y \neq table) \otimes move(X, _, table).\\
&unstack(X) \ :- \ on(Y, X) \otimes unstack(Y) \otimes unstack(X).
\end{aligned} \tag{28}$$

Testing the above program on the tabled interpreter shows that the aforesaid rule preferences can significantly reduce the number of plans that need to be considered — sometimes to just one plan. □

Example 4.8 (Workflow modeling and execution example) *This example illustrates the use of defeasible reasoning for modeling business workflows. Transaction Logic have been used before for modeling concurrent workflows in [DKRR98, Dav02, DKR04]. Although, these works address a multitude of issues, including model checking for verifying workflows, integration of the data flow into the control flow by using transition conditions, sub - workflows, loops and iteration, and so on, priorities between different execution paths in the workflow haven't been considered before, leaving the task of implementing opposition and preferences between execution branches to the programmer. Although we don't talk in the* $\mathcal{TR}^{DA}$ *defeasible reasoning about various aspects of transaction logic used in modeling workflows, like concurrency and constraints on the interleaved execution, in this case of non - recursive workflows, this*

program can be systematically transformed into a purely sequential $\mathcal{TR}$ program.

Let's consider the following example of a workflow where various branches in the workflow execution oppose other branches. In this scenario depicted in Figure 16, a buy transaction is designed to make a financial transaction and a delivery of a product.

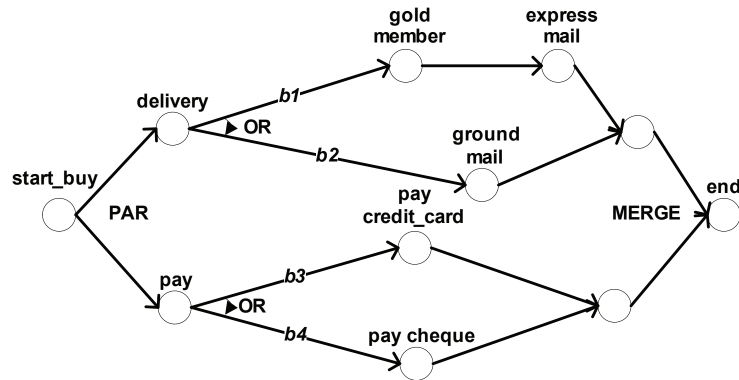

Figure 16: A transaction workflow example for defeasible reasoning in $\mathcal{TR}$

The following $\mathcal{TR}^{DA}$ program implements this simple workflow using sub - workflow actions defined in $\mathcal{TR}$. The execution process is non - deterministic and several different OR-branches of the workflow can be chosen to be executed. In this example, we represent the transaction buy as an interleaving of transactions, namely $pay|delivery$, where these transactions can non - deterministically choose various options: pay with credit card or with wire transfer from a bank account and deliver using express or ground mail. However, the policy of the store is that if the customer is a gold_member, then the delivery is done using express_mail, otherwise using ground_mail, or, that a wire transfer from a bank account is preferred to a credit card payment since the payment does not require a filling period. These actions are specified below by the opposes and overrides rules, respectively. The action delivery, defined by the second and the third rules, and the action pay, defined by the forth and the fifth rules, combine sub - workflows (actions) determined by what internal conditions are satisfied.

The key observation here is that at any given point certain workflow branches are preferred over other workflow branches. For instance, a successful branch @b4 *pay_cheque is preferred instead of the branch* @b3 *pay_credit_card although both*

might be applicable and several different combinations of the concurrent - branches in the workflow might be performed.

$$
\begin{array}{l}
buy\ :-\ pay|delivery. \\
@b1delivery\ :-\ gold_member \otimes express_mail. \\
@b2delivery\ :-\ ground_mail. \\
@b3pay\ :-\ pay_credit_card. \\
@b4pay\ :-\ pay_cheque. \\
\texttt{!opposes}(b1, b2). \\
\texttt{!overrides}(b1, b2). \\
\texttt{!opposes}(b4, b3). \\
\texttt{!overrides}(b4, b3). \\
gold_member. \\
express_mail\ :-\ insert(delivered_express_mail). \\
ground_mail\ :-\ insert(delivered_ground_mail). \\
pay_credit_card\ :-\ credit_card_credentials \otimes insert(credit_card_payment). \\
pay_cheque\ :-\ bank_account \otimes insert(bank_payment). \\
credit_card_credentials. \\
bank_account.
\end{array}
\qquad (29)
$$

□

4.4.2 $\mathcal{TR}^{DA}$ Evaluation

Table 13 shows how the preferential heuristic of Example 4.7 helps reduce the number of plans for pyramid construction (pruning away the plans for uninteresting pyramids), space, and time requirements. It shows that the number of plans and space requirements are reduced by an order of magnitude and time is reduced by a factor of about 5. The discrepancy between improvements in the runtime and the reduction in the number of plans can be explained by the fact that, even without the optimizing heuristics, out implementation of $\mathcal{TR}^{DA}$ takes advantage of sharing of partially constructed plans among the different searches. Therefore, the reduction in the runtime is not as dramatic compared to the reduction and space and the number of plans.

We conclude the evaluation section with the extreme case where we have a world of 10,000 blocks blk_1, blk_2, ..., $blk_{10,000}$ being on the *table* with blk_2 being *larger* than the block blk_1 and blk_3 being *larger* than both blocks blk_1 and blk_2, and so on, and an existential goal, $(\exists)stack(10,000, blk_{10,000})$ for stacking a pyramid of 9,999 blocks on the block $blk_{10,000}$ as a base. The original tabling algorithm presented in [FK10a] would try to try plan 9,999 different pyramids where one block blk_i, $1 \leq i \leq 9,999$, would sit separately on the *table* and easily fail because this requires a very large memory to store all reachable states. With the heuristic rules in Section 4.4.1, the new algorithm will return a single pyramid containing the blocks blk_2 to $blk_{10,000}$ with the block blk_1 sitting separately on the table, the rule being that on top of each clear block blk_i is preferred to stack the block blk_{i-1} since it's the largest clear block on the table. This will succeed in a short time because it requires only 1,000 steps and only 1,000 intermediate states to store in tables.

World size		No heuristics	With preferential heuristics
10 blocks	Plans	120	8
	Time(sec.)	0.078	0.016
	Space(kBs)	155	26
	Tabled states	296	36
	Transient states	165	17
	State comps.	605	89
20 blocks	Plans	1140	18
	Time(sec.)	0.563	0.109
	Space(kBs)	1162	60
	Tabled states	2491	76
	Transient states	1330	37
	State comps.	4410	189
30 blocks	Plans	4060	28
	Time(sec.)	2.390	0.438
	Space(kBs)	3730	90
	Tabled states	8586	116
	Transient states	4495	57
	State comps.	14415	289
40 blocks	Plans	9880	38
	Time(sec.)	7.000	1.219
	Space(kBs)	8562	120
	Tabled states	20581	156
	Transient states	10660	77
	State comps.	33620	389
50 blocks	Plans	19600	48
	Time(sec.)	17.109	2.938
	Space(kBs)	16347	150
	Tabled states	40476	196
	Transient states	20825	97
	State comps.	65025	489

Table 13: Time, space, tabled states and state comparisons for planning in the blocks world with and without preferential heuristics

Chapter 5

Conclusion and future work

In this thesis we focused on Transaction Logic, a language for specifying actions and state updates similar to Datalog rules with state changing elementary actions in the body of rules. We have addressed the following aspects: tabled definite Horn-Transaction Logic and defeasible reasoning in Transaction Logic.

Tabled Transaction Logic In the first part of the thesis we adapted the commonly used tabling technique [TS88, War92, SW94] from ordinary logic programs to Transaction Logic. We have shown that the proof theory of Transaction Logic modified with tabling is sound, complete and it terminates for programs satisfying certain conditions. We discussed a host of difficulties in implementing tabling for Transaction Logic and proposed various optimizations. The implementation was developed within the framework of XSB and it combines several different optimizations as plug - ins, enabling us to compare the different optimizations.

Defaults and defeasibility for Transaction Logic In the second part of the thesis we developed a well - founded semantics and a theory of defeasible reasoning for Transaction Logic. This extends our previous work on defeasible reasoning in logic programming using argumentation theories from static logics to a logic of state changes and transaction, which is capable to representing both declarative and procedural knowledge. We also extend the Courteous style of defeasible reasoning [Gro99] to incorporate actions, planning, and other dynamic aspects of knowledge representation. We believe that TRDA can become a rich platform for expressing

heuristics about actions. Along the way, we defined the well founded semantics for the $\mathcal{TR}^{DA}$ extension of $\mathcal{TR}$, an adaptation of the classical well founded semantics of [VRS91] for the $\mathcal{TR}$ dynamic logic. The primary advantages of this part of this work are: a direct model theory for defeasible Transaction Logic, a simple implementation for Courteous and other defeasible reasoning approaches as argumentation theories to this extension for $\mathcal{TR}$, and better control over edge case behavior.

The appendix outlines our contributions to complex event processing using $\mathcal{TR}$ with the goal of detecting event patterns of interest. We present a rule language for event processing with several event operators and temporal relationships, used in combining events into patterns, detecting complex events, and addressing issues like event filtering, routing, and consumption.

We outline here the possible future directions of the research in this thesis. In the short term, the following research problems are within grasp.

Formalisms for tabled evaluations developed for normal logic programs, such as, *SLG* [CW96] and *Extended SLG* (SLG_X) [Swi99], could be lifted to Transaction Logic Programming. For instance, we believe that in the partial deduction procedure for the *SLG* resolution, the *SLG* systems of the form $(A : \delta)$, where A is a subgoal and δ is a sequence of annotated rules for A, can be extended with the state in which the call to the goal A was made (two pairs would be considered different even if the two subgoals are identical but the calling states are different). The six fundamental *SLG* transformations can be modified with the above change for systems, so that each query in a calling state can be transformed step by step into a set of answers and return states. This new partial deduction technique would be a program transformation for Transaction logic that specializes the transaction logic program for a serial goal to produce a more efficient program equivalent to the original program as far as the serial goal is concerned. Similarly, the SLG_X algorithm from [Swi99] can be also reformulated to fit $\mathcal{TR}$. The *SLG* forest consisting of trees whose nodes have the form: $Answer_Template\ :-\ Delay_Set \mid GoaLList$ or $fail$, could include information about the states where these calls were made ($Call_State$). Moreover, the delay literals in the $Delay_Set$ should also be annotated with the calling state $Call_State$, while answers should be annotated with their return states. We suspect that both extensions are sound and search space complete for the existence of paths between

any two states with respect to the well-founded partial model for $\mathcal{TR}$ programs that address a finite number of states and all non-floundering queries.

The effects of tabling on a large number of optimizations have not been studied and it's another possible future direction. Such optimizations can include: the transformations of $\mathcal{TR}$ programs described in [Hun96b] and [F.S00], pushing fluents ahead of actions in the rule bodies and optimizations inherited from normal logic programming like heuristics for join - order optimization complemented by complexity analysis, and specialization.

In the long term, there are various interesting and highly challenging directions to pursue. The following are a few of these directions. We plan to further validate our results by incorporating an efficient implementation of B+ trees. We also believe that applications on the new tabled and defeasible $\mathcal{TR}$ extensions have important applications in security frameworks, semantic Web services composition and execution, and we plan to investigate these applications.

Bibliography

[ABGM01] G. Antoniou, D. Billington, G. Governatori, and M.J. Maher. Representation results for defeasible logic. *ACM Trans. Comput. Log.*, 2(2):255–287, 2001.

[AFR+10a] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stuehmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In *International Conference on Web Reasoning and Rule Systems (RR) (shortlisted for the Best Paper Award), Bressanone/Brixen, Italy*, September 2010.

[AFR+10b] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stuehmer, Nenad Stojanovic, and Rudi Studer. Etalis: Rule-based reasoning in event processing. In *Reasoning in Event-based Distributed Systems, Studies in Computational Intelligence series*, Springer Verlag, LNCS, 2010.

[AFRS11] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *20th International World Wide Web Conference (WWW)*, March 2011.

[AFSS09a] Darko Anicic, Paul Fodor, Roland Stuehmer, and Nenad Stojanovic. Event-driven approach for logic-based complex event processing. In *IEEE International Conference on Computational Science and Engineering (CSE)*, Vancouver, Canada, August 2009.

[AFSS09b] Darko Anicic, Paul Fodor, Roland Stühmer, and Nenad Stojanovic. An approach for data-driven logic-based complex event processing. In *The*

3rd ACM International Conference on Distributed Event-Based Systems (DEBS), 2009.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM 26, 11, 832-843*, 1983.

[AM02] G. Antoniou and M.J. Maher. Embedding defeasible logic into logic programs. In *Int'l Conference on Logic Programming*, pages 393–404, 2002.

[Bal04] Marcello Balduccini. Usa-smart: Improving the quality of plans in answer set planning. In *PADL*, 2004.

[Bal09] Marcello Balduccini. Solving the wise mountain man riddle with answer set programming. In *Ninth International Symposium on Logical Formalizations of Commonsense Reasoning*, 2009.

[BE99] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109:297–356, 1999.

[BE00] G. Brewka and T. Eiter. Prioritizing default logic. In *Intellectics and Computational Logic – Papers in Honour of Wolfgang Bibel*, pages 27–45. Kluwer Academic Publishers, 2000.

[BH95] F. Baader and B. Hollunder. Priorities on defaults with prerequisites, and their application in treating specificity in terminological default logic. *Journal of Automated Reasoning*, 15(1):41–68, 1995.

[BK93] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.

[BK94a] A.J. Bonner and M. Kifer. Applications of transaction logic to knowledge representation. In *Proceedings of the International Conference on Temporal Logic*, number 827 in Lecture Notes in Artificial Inteligence, pages 67–81, Bonn, Germany, July 1994. Springer-Verlag.

[BK94b] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133 (2):205–265, 1994.

[BK95] A.J. Bonner and Michael Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.

[BK96] Anthony J. Bonner and Michael Kifer. Concurrency and communication in transaction logic. In *JICSLP*, pages 142–156, 1996.

[BK98a] A.J. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.

[BK98b] A.J. Bonner and M. Kifer. The state of change: A survey. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.

[BK98c] Anthony J. Bonner and Michael Kifer. A logic for programming database transactions. In *Logics for Databases and Information Systems*, pages 117–166, 1998.

[BN07] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In *ESORICS*, 2007.

[Bon97] Anthony J. Bonner. Modular composition of transaction programs with deductive databases. In *DBPL*, pages 373–395, 1997.

[CKW93] W. Chen, Michael Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[Com79] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[CW96] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43 (1):20–74, 1996.

[Dav02] Hasan Davulcu. *A Game Logic for Workflows of Non-cooperative Services*. PhD thesis, State University of New York at Stony Brook, 2002.

[DKR04] Hasan Davulcu, Michael Kifer, and I. V. Ramakrishnan. Ctr-s: a logic for specifying contracts in semantic web services. In *WWW*, pages 144–153, 2004.

[DKRR98] Hasan Davulcu, Michael Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *PODS*, pages 25–33, 1998.

[DPR96] A. Dovier, A. Policriti, , and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. *Applied Logic*, 3:213–229, 1996.

[DS01] P.M. Dung and Tran Cao Son. An argument-based approach to reasoning with specificity. *Artificial Intelligence*, 133(1-2):35–85, 2001.

[DST03] James P. Delgrande, Torsten Schaub, and Hans Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 2:129–187, 2003.

[DST04] James P. Delgrande, Torsten Schaub, and Hans Tompits. Domain-specific preferences for causal reasoning and planning. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *KR*, pages 673–682, Whistler, Canada, 2004. AAAI Press.

[DST07a] James P. Delgrande, Torsten Schaub, and Hans Tompits. A general framework for expressing preferences in causal reasoning and planning. *J. Log. and Comput.*, 17:871–907, October 2007.

[DST07b] James P. Delgrande, Torsten Schaub, and Hans Tompits. A preference-based framework for updating logic programs. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 71–83, Tempe, AZ, USA, 2007. Springer.

[DSTW04] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(12):308–334, 2004.

[EFL+03] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *J. Artif. Int. Res.*, 19:25–71, August 2003.

[EFLP03] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming*, 3(4):463–498, 2003.

[FA] Paul Fodor and Darko Anicic. Event TransAction Logic Inference System (ETALIS). The ETALIS Web Site. http://code.google.com/p/etalis.

[FA09] Paul Fodor and Darko Anicic. The fast flowers delivery use case. In *ETALIS CEP system. Included in Languages for event processing, Event Processing Technical Society `http://www.ep-ts.com/content/view/79/109/` and the The Fast Flowers Delivery Use Case chapter, accompanying the book Event Processing In Action by Opher Etzion and Peter Niblett, Manning Publications*, 2009.

[FAR+10] Paul Fodor, Darko Anicic, Sebastian Rudolph, Roland Stuehmer, Nenad Stojanovic, and Rudi Studer. Processing out-of-order event streams in etalis. In *ACM International Conference on Distributed Event-Based Systems (DEBS), fast abstract*, Cambridge, United Kingdom, July 2010.

[FAR11] Paul Fodor, Darko Anicic, and Sebastian Rudolph. Results on out-of-order event processing. In *International Symposium on Practical Aspects of Declarative Languages (PADL), Austin, Texas, USA*, January 2011.

[FK10a] Paul Fodor and Michael Kifer. Tabling for transaction logic. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 199–208, New York, NY, USA, 2010. ACM.

[FK10b] Paul Fodor and Michael Kifer. Tabling for transaction logic. In *12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, Hagenberg, Austria, July 2010.

[FK11] Paul Fodor and Michael Kifer. Transaction logic with defaults and argumentation theories. In *27th International Conference on Logic Programming*, ICLP '11, July 2011.

[Fod09] Paul Fodor. Initial results on justification for the tabled transaction logic. In *AAAI Spring Symposium*, 2009.

[F.S00] Amalia F.Sleghel. An optimizing interpreter for concurrent transaction logic. Master's thesis, University of Toronto, 2000.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[GL98] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.

[GMS06] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *International Enterprise Distributed Object Computing Conference (EDOC)*, pages 221–232, 2006.

[GR10] Guido Governatori and Antonino Rotolo. Norm compliance in business process modeling. In *International Web Rule Symposium (RuleML)*, pages 194–209, 2010.

[Gro99] B.N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report Supplementary Update Follow-On to RC 21472, IBM, July 1999.

[GRS04] Guido Governatori, Antonino Rotolo, and Shazia Sadiq. A model of dynamic resource allocation in workflow systems. In *Klaus-Dieter Schewe*

and Hugh E. Williams, editors, Database Technology 2004, Dunedin, New Zealand. Conference Research and Practice of Information Technology, pages 197–206, 2004.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, 1978. IEEE Computer Society.

[GS98] Michael Gelfond and Tran Cao Son. Reasoning with prioritized defaults. In *Selected papers from the Third International Workshop on Logic Programming and Knowledge Representation*, pages 164–223, London, UK, 1998. Springer-Verlag.

[Hun96a] Samuel Hung. Transaction logic prototype, 1996.

[Hun96b] Samuel Y.K. Hung. Implementation and performance of transaction logic in prolog. Master's thesis, University of Toronto, 1996.

[Kif] Michael Kifer. FLORA-2: An object-oriented knowledge base language. The FLORA-2 Web Site. http://flora.sourceforge.net.

[KLW95] Michael Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.

[Liu98] M. Liu. Relationlog: a typed extension to datalog with sets and tuples. *Journal of Logic Programming*, 36, 1998.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1984.

[MN06] F. Maier and D. Nute. Relating defeasible logic to the well-founded semantics for normal logic programs. In *Int'l Workshop on Non-monotonic Reasoning*, 2006.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[Nut94] D. Nute. Defeasible logic. In *Handbook of logic in artificial intelligence and logic programming*, pages 353–395. Oxford University Press, 1994.

[Pon92] E. Pontelli. Logic programming with sets: Theory and implementation. Master's thesis, University of Houston, 1992.

[Pra93] H. Prakken. An argumentation framework in default logic. *Annals of Mathematics and Artificial Intelligence*, 9(1-2):93–132, 1993.

[Prz94] T.C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[RK07] Dumitru Roman and Michael Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *VLDB*, pages 627–638, 2007.

[RK08] Dumitru Roman and Michael Kifer. Semantic web service choreography: Contracting and enactment. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 550–566, Karlsruhe, 2008. Springer.

[SI00] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.

[Sle00] Amalia F. Sleghel. Concurrent transaction logic prototype, 2000.

[SP02] Tran Cao Son and Enrico Pontelli. Reasoning about actions in prioritized default theory. In *Logics in Artificial Intelligence*, pages 369–381, 2002.

[SP03] Tran Cao Son and Enrico Pontelli. Adding preferences to answer set planning. In *ICLP*, 2003.

[SP04] Tran Cao Son and Enrico Pontelli. Reasoning about actions and planning with preferences using prioritized default theory. In *Computational Intelligence 20(1)*, 2004.

[SP06] Tran Cao Son and Enrico Pontelli. Planning with preferences using logic programming. *Theory Pract. Log. Program.*, 6:559–607, September 2006.

[SRV01] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of automated reasoning*, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[SW94] T. Swift and D.S. Warren. An abstract machine for SLG resolution: Definite programs. In *Int'l Logic Programming Symposium*, Cambridge, MA, November 1994. MIT Press.

[Swi99] Terrance Swift. A new formulation of tabled resolution with delay. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, pages 163—177, 1999.

[TS86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Int'l Conference on Logic Programming*, pages 84–98, Cambridge, MA, 1986. MIT Press.

[TS88] Hisao Tamaki and Taisuke Sato. Old resolution with tabulation. In *ICLP*, pages 84–98, 1988.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[War92] David Scott Warren. Memoing for logic programs. *Communications of the ACM*, 35 (3):93–111, 1992.

[WGK+09a] H. Wan, B.N. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *ICLP*, pages 432–448, 2009.

[WGK+09b] Hui Wan, Benjamin Grosof, Michael Kifer, Paul Fodor, and Senlin Liang. Logic programming with defaults and argumentation theories. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 432–448, Berlin, Heidelberg, 2009. Springer-Verlag.

[WZL00] K. Wang, L. Zhou, and F. Lin. Alternating fixpoint theory for logic programs with priority. In *First Int'l Conference on Computational Logic (CL'00)*, number 1861 in Lecture Notes in Computer Science, pages 164–178. Springer, 2000.

[YKZ03] Guizhen Yang, Michael Kifer, and Chang Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)*, volume 2888 of *Lecture Notes in Computer Science*, pages 671–688. Springer, November 2003.

[ZWB01] Y. Zhang, C.M. Wu, and Y. Bai. Implementing prioritized logic programming. *AI Communications*, 14(4):183–196, 2001.

Appendix A

Application of Transaction Logic in CEP

In parallel to our work presented in the main part of the thesis we have applied $\mathcal{TR}$ to the area of Complex Event Processing (CEP). In this appendix, we outline this additional work. For more details the reader can consult several documents about the ETALIS language for event composition [AFSS09a, AFR$^+$10a, AFR$^+$10b, AFSS09b, FAR$^+$10, FAR11].

CEP has the task of processing streams of events with the goal of detecting event patterns of interest. An *event* represents something that occurs, happens or changes the current state of affairs. For example, an event may signify a problem, a threshold, an opportunity, an information becoming available or a deviation. An *atomic event* is defined as an instantaneous occurrence of interest at a point in time. In order to describe more complex dynamic matters that involve several atomic events, formalisms have been created which allow for combining atomic into *complex events* using event operators and temporal relationships. The goal of CEP is to detect complex events according to a set of event patterns, addressing other issues like event filtering, routing, consumption and transformation. It is typically assumed that events in an event stream are *totally ordered*, that is, the order in which events are received by the system is the same as their time stamp order.

We implemented such a CEP system in Prolog using $\mathcal{TR}$ named *ETALIS*[1]

[1]ETALIS: `http://code.google.com/p/etalis`

[AFSS09a, AFR+10a, AFR+10b, AFSS09b] with various extensions: support for garbage collection [FAR11], processing of out - of - order event streams [FAR+10], time - based and count - based windowing [FA], justification and debugging, support for streams of RDF triples in the Semantic Web domain [AFRS11] and extended applications [FA09].

In the ETALIS system, the events occur over time intervals, time instants as well as durations being modeled as nonnegative rational numbers $q \in \mathbb{Q}^+$. Events can be atomic or complex, while no distinction is made in their applicability to rules. In ETALIS, an *atomic event* refers to an instantaneous occurrence, i.e., the time interval length is zero. Although not a requirement, atomic events are ground (i.e. predicates followed by arguments which are terms not containing variables). Intuitively, the arguments of a ground atom describing an atomic event denote information items (i.e. event data) that provide additional information about the event.

Events participate in composition rules to trigger complex events. The syntax of *ETALIS Language for Events* allows for the description of *event* patterns as event rules of the form: $complexEvent \leftarrow eventPattern$. When an *event stream* of atomic events is fed into the system, all patterns are considered and complex events are triggered. A *variable assignment* is a mapping $\mu : Var \rightarrow Con$ assigning a value to every variable. The event stream is formalized as a mapping $\epsilon : Ground \rightarrow 2^{\mathbb{Q}^+}$ from ground predicates into sets of nonnegative rational numbers. It thereby indicates at what time instants what simple events occur. As a side condition, it is required that ϵ is free of accumulation points, i.e. for every $q \in \mathbb{Q}^+$, the set $\{q' \in \mathbb{Q}^+ \mid q' < q \text{ and } q' \in \epsilon(g) \text{ for some } g \in Ground\}$ is finite.

We define an interpretation $\mathcal{I} : Ground \rightarrow 2^{\mathbb{Q}^+ \times \mathbb{Q}^+}$ as a mapping from the ground atoms to sets of pairs of nonnegative rationals, such that $q_1 \leq q_2$ for every $\langle q_1, q_2 \rangle \in \mathcal{I}(g)$ for all $g \in Ground$. Given an event stream ϵ, an interpretation $\mathcal{I}$ is called a *model* for a rule set $\mathcal{R}$ – written as $\mathcal{I} \models_\epsilon \mathcal{R}$ – if the following conditions are satisfied:

C1 $\langle q, q \rangle \in \mathcal{I}(g)$ for every $q \in \mathbb{Q}^+$ and $g \in Ground$ with $q \in \epsilon(g)$

C2 for every rule $complexEvent \leftarrow eventPattern$ and every variable assignment μ , $\mathcal{I}_\mu(complexEvent) \subseteq \mathcal{I}_\mu(eventPattern)$ where $\mathcal{I}_\mu$ is inductively defined as follows:

pattern	$\mathcal{I}_\mu$(pattern)
q	$\{\langle q,q\rangle\}$ for all $q \in \mathbb{Q}^+$
p_1 SEQ p_2	$\{\langle q_1,q_4\rangle \mid \langle q_1,q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2,q_3 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
p_1 AND p_2	$\{\langle \min(q_1,q_3),\max(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2,q_3 \in \mathbb{Q}^+\}$
p_1 PAR p_2	$\{\langle \min(q_1,q_3),\max(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2,q_3 \in \mathbb{Q}^+$ with $\max(q_1,q_3) < \min(q_2,q_4)\}$
p_1 OR p_2	$\mathcal{I}_\mu(p_1) \cup \mathcal{I}_\mu(p_2)$
p_1 EQUALS p_2	$\mathcal{I}_\mu(p_1) \cap \mathcal{I}_\mu(p_2)$
p_1 MEETS p_2	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_2,q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+\}$
p_1 STARTS p_2	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_1,q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
p_1 FINISHES p_2	$\{\langle q_1,q_3\rangle \mid \langle q_2,q_3\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_1,q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_1 < q_2\}$

The intuitive meanings for the patterns presented above are the following (we assume that instances of two events, p_1 and p_2, are occurring):

- p_1 SEQ p_2 represents a sequence of two events, i.e. an occurrence of p_1 is followed by an occurrence of p_2; thereby p_1 must end before p_2 starts.
- p_1 AND p_2 is a pattern that is detected when instances of both p_1 and p_2 occur no matter in which order.
- p_1 PAR p_2 occurs when instances of both p_1 and p_2 happen, provided that their intervals have a non - zero overlap.
- p_1 OR p_2 is triggered for every instance of p_1 or p_2.
- p_1 EQUALS p_2 is triggered when the two events occur exactly at the same time interval.

- p_1 MEETS p_2 happens when the interval of an occurrence of p_1 ends exactly when the interval of an occurrence of p_2 starts.

- p_1 STARTS p_2 is detected when an instance of p_2 starts at the same time as an instance of p_1.

- p_1 FINISHES p_2 is detected when an instance of p_2 ends at the same time as an instance of p_1.

It is worth noting that the defined pattern language captures the set of all possible 13 relations on two temporal intervals as defined in [All83]. The set can also be used for rich temporal reasoning.

Given an interpretation $\mathcal{I}$ and some $q \in \mathbb{Q}^+$, we let $\mathcal{I}|_q$ denote the interpretation defined via $\mathcal{I}|_q(g) = \mathcal{I}(g) \cap \{\langle q1, q2 \rangle \mid q2 - q1 \leq q\}$.

Given two interpretations $\mathcal{I}$ and $\mathcal{J}$, we say that $\mathcal{I}$ is *preferred* to $\mathcal{J}$ if there exists a $q \in \mathbb{Q}^+$ with $\mathcal{I}|_q \subset \mathcal{J}|_q$.

A model $\mathcal{I}$ is called *minimal* if there is no other model preferred to $\mathcal{I}$. It is easy to show that for every event stream ϵ and rule set $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\epsilon,\mathcal{R}}$. Given an atom a and two rational numbers q_1, q_2, we say that the event $a^{[q_1,q_2]}$ is a *consequence* of the event stream ϵ and the rule base $\mathcal{R}$ (written $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$), if $\langle q_1, q_2 \rangle \in \mathcal{I}_\mu^{\epsilon,\mathcal{R}}(a)$ for some variable assignment μ. The behavior of the event stream ϵ beyond the time point q_2 is irrelevant for determining whether $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$ is the case. For any two event streams ϵ_1 and ϵ_2 with $\epsilon_1(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\} = \epsilon_2(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\}$ we have that $\epsilon_1, \mathcal{R} \models a^{[q_1,q_2]}$ exactly if $\epsilon_2, \mathcal{R} \models a^{[q_1,q_2]}$. This justifies to take the perspective of ϵ being only partially known (and continuously unveiled along a time line) while the task is to detect event - consequences as soon as possible.

An example of CEP rules is the transitive closure rules in (30). The event e of arity 2 is an atomic event, while the event tc of arity 2 is a composed event computing the transitive closure of the event stream composed of instances of the event e. In this example and in the semantics above, the event pattern is considered under the so - called *unrestricted policy.* In event processing, *consumption policies* deal with an issue of selecting particular events occurrences when there are more than one event instance applicable and consuming events after they have been used in patterns.

$$tc(X,Y) \leftarrow e(X,Y).$$
$$tc(X,Y) \leftarrow tc(X,Z) \text{ SEQ } e(Z,Y). \tag{30}$$

We now define how $\mathcal{TR}$ is used for the run - time detection of complex events in ETALIS. Lets consider the CEP rule $e \leftarrow a \text{ SEQ } b \text{ SEQ } c.$. The first step in the algorithm is *events coupling* or *binarization* of events, an operation that break event formulas into rules with one operand and at most two events. For example, now we can rewrite the rule $e \leftarrow a \text{ SEQ } b \text{ SEQ } c.$ as $ie_1 \leftarrow a \text{ SEQ } b$, and the $e \leftarrow ie_1 \text{ SEQ } c.$ Every monitored event (either atomic or complex), including intermediate events, will be assigned with one or more rules, fired whenever that event occurs. Using the binarization, it is more convenient to construct $\mathcal{TR}$ rules for three reasons. First, it is easier to implement an event operator when events are considered on "two by two" basis. Second, the binarization increases the possibility for *sharing* among events and intermediate events, when the granularity of intermediate patterns is reduced. Third, the binarization eases the *management* of rules. Each new use of an event (in a pattern) amounts to appending one or more rules to the existing rule set. However what is important for the management of rules, we don't need to *modify* existing rules when adding new ones.

The second step in the algorithm accepts binary rules and produces $\mathcal{TR}$ rules belonging to two different classes of rules: *goal inserting rules* and *checking rules.* The sequence operation in the event binary rule $ie_1 \leftarrow a$ SEQ b is converted into the following rules:

$a(T_1,T_2) : -trigger_all(a_s(T_1,T_2)).$
$a_s(T_1,T_2) : -insert(goal(b,a(T_1,T_2),e1)).$
$b(T_3,T_4) : -trigger_all(b_s(T_3,T_4)).$
$b_s(T_3,T_4) : -goal(b,a(T_1,T_2),ie_1),T_2 < T_3,ie_1(T_1,T_4).$

The first and the third rules call all the rules triggered by the event in an arbitrary order where a_s and b_s are the various definitions of a and b in the program. The second rule will fire when a occurs, and the meaning of the goal it inserts is as follows: "an event a has occurred at $[T_1,T_2]$, [2] and we are waiting for b to happen in order to detect ie_1". Obviously, the goal does not carry information about the times for

[2] Apart from the time stamp, an event may carry other data parameters. They are omitted here for the sake of readability.

b and ie_1, as we don't know when they will occur. The second event in the goal denotes the event that has just occurred, while the role of the first event in the goal is to specify what we are waiting for to detect an event that is on the third position in the goal. The forth rule belongs to the class of *checking rules*. It checks whether certain prerequisite goals already exist in the database, in which case it triggers the more complex event. The time occurrence of ie_1 (i.e. T_1, T_4) is defined based on the occurrence of constituting events (i.e. $a(T_1, T_2)$, and $b(T_3, T_4)$). Calling $ie_1(T_1, T_4)$, this event is effectively propagated either upward (if it is an intermediate event) or triggered as a finished complex event.

The algorithm sketched above was applied for all operands and the resulting set of event calls (and truth values in the final database if we would insert corresponding fluents for these event calls in the database) matches the events in the fixed point semantics of the ETALIS language defined above. Both the above translation and our fixed point semantics is defined for the the "unrestricted" consumption policy (we are still investigating if similar fixed point semantics and run - time detection algorithms can be developed for other consumption policies). For further details on ETALIS the reader is referred to [AFSS09a, AFR$^+$10a, AFR$^+$10b, AFSS09b, FAR$^+$10, FAR11].

Appendix B

Appendix: Tabled $\mathcal{TR}$ Soundness and Completeness

In this appendix, we prove the soundness and completeness of the inference system $\mathcal{F}^T$ developed in Section 3.1. For convenient reference, we reproduce the axioms and inference rules of system $\mathcal{F}^T$ below. If $\mathbf{P}$ is a transaction base and $\mathbf{D}_i$ $(1 \leq i)$ are database state identifiers, then $\mathcal{F}^T$ is the following system of axioms and inference rules.

Axioms: $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash state$

Rule 1a. *Applying transaction definitions for tabled predicates:*

Suppose b's predicate is tabled and there is no dominating pair $(c, \mathbf{D}_1)$ in the table space. Let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ whose variables have been renamed apart from $b \otimes rest$ (i.e., the rule shares no variables with the goal) and suppose that a and b unify with the most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\begin{array}{c}\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest) \\ (b, \mathbf{D}_1) \in \texttt{table space} \\ \forall\ \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\gamma,\ (b\gamma, \mathbf{D}_i) \in \texttt{answer table}(b, \mathbf{D}_1)\end{array}}$$

That is, given a sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma$, the rule allows us to derive $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)$. In addition, $(b, \mathbf{D}_1)$ is added to the table space, and for all γ such that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\gamma$ is derivable, the answer $(b\gamma, \mathbf{D}_i)$ is added to the answer table for $(b, \mathbf{D}_1)$.

Rule 1b. *Returning answers from answer tables:*

Suppose: (1) b's predicate symbol is declared as tabled, (2) there is a dominating pair $(c, \mathbf{D}_1)$ in the table space, (3) the answer table for $(c, \mathbf{D}_1)$ has an entry $(a, \mathbf{D}_i)$, and (4) a and b unify with most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_i \text{---} \vdash (\exists)\, (rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (b \otimes rest)}$$

Rule 1c. *Applying transaction definitions for non - tabled predicates:*

This rule is identical to Rule 1 in the proof theory of Section 2.1: let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ and a's predicate symbol is *not* tabled. Assume that this rule's variables have been renamed apart from $b \otimes rest$ and that a and b unify with most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (b \otimes rest)}$$

Rule 2. *Querying the database*:

If b is a fluent literal, $b\sigma$ and $rest\,\sigma$ share no variables, and $b\sigma$ is true in the database state $\mathbf{D}_1$, then:

$$\frac{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (b \otimes rest)}$$

Rule 3. *Performing elementary updates*:

If $b\sigma$ and $rest\,\sigma$ share no variables, and $b\,\sigma$ is an elementary action that changes state $\mathbf{D}_1$ to state $\mathbf{D}_2$, then:

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (b \otimes rest)}$$

Theorem 3.2 (Soundness and Completeness)

Suppose ϕ is a definite serial-Horn goal.

Soundness: *If there is a tabled deduction of the sequent* $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \phi$ *with the execution path* $\langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle$ *then the executional entailment* $\mathbf{P}, \mathbf{D_1} \ldots \mathbf{D_n} \models (\exists)\, \phi$ *holds.*

Completeness: *If the executional entailment* $\mathbf{P}, \mathbf{D_1 D_2} \ldots \mathbf{D_{n-1} D_n} \models (\exists)\, \phi$ *holds*

then there exists a tabled deduction of the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \phi$ *with an execution path* $\langle \mathbf{D}_1, \mathbf{D}'_2 \ldots \mathbf{D}'_m, \mathbf{D}_n \rangle$ *that starts in the database state* $\mathbf{D}_1$ *and ends in* $\mathbf{D}_n$*.*

Note that the path $\langle \mathbf{D}_1, \mathbf{D}'_2 \ldots \mathbf{D}'_m, \mathbf{D}_n \rangle$ has only the extremities $\mathbf{D}_1$ and $\mathbf{D}_n$, but does not have to be identical with the path $\langle \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{n-1}, \mathbf{D}_n \rangle$ as a result of the fact that the intermediate states $\mathbf{D}'_2, \ldots, \mathbf{D}'_m$ can be different from the intermediate states $\mathbf{D}_2, \ldots, \mathbf{D}_{n-1}$.

Proof:

Soundness: The inference system $\mathcal{F}^T$ is sound if all its axioms and inference rules are sound. First, we consider the axioms $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash state$. For any database state $\mathbf{D}_1$, we have $\mathbf{M}, \mathbf{D}_1 \models state$ by definition of the propositional constant *state* for all models $\mathbf{M}$ of $\mathbf{P}$. Thus, we have $\mathbf{P}, \mathbf{D}_1 \models state$.

Inference Rule 1: To prove the soundness of the inference rules 1a, 1b and 1c, suppose that $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed apart from $b \otimes rest$, a and b unify with the most general unifier σ.

Rules 1a and 1c: Suppose that b is a call to a tabled predicate encountered for the first time in a current state $\mathbf{D}_1$ (i.e., no dominating pair $(c, \mathbf{D}_1)$ is in the table space) and we apply Rule 1a, or b is is a call to a non-tabled predicate and we apply Rule 1c, and $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \models (\exists)\,(\phi \otimes rest)\sigma$ holds. For every model $\mathbf{M}$ of $\mathbf{P}$, $\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models (\exists)\,(\phi \otimes rest)\sigma$ holds by executional entailment. Therefore, $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle \models_\nu (\phi \otimes rest)\sigma$ for some variable assignment ν and, by Definition 2.8, we also have: $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \models_\nu \phi\sigma$ and $\mathbf{M}, \langle \mathbf{D}_i \ \ldots \mathbf{D}_n \rangle \models_\nu rest\,\sigma$ for some split $\langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \circ \langle \mathbf{D}_i \ \ldots \mathbf{D}_n \rangle$ of the path $\langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle$. Due to the fact that $\mathbf{M}$ is a model of $\mathbf{P}$, it is also a model of $a \leftarrow \phi$ and $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \models_\nu a\sigma$ holds. On account that a and b unify with the most general unifier σ, $a\sigma = b\sigma$, it results that $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \models_\nu b\sigma$ holds. By Definition 2.8, we also have that $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle \models_\nu (b \otimes rest)\sigma$ and $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle \models (\exists)\,(b \otimes rest)$ hold.

Rule 1b: There exists a dominating pair $(c, \mathbf{D}_1)$ in the table space such that $b\gamma = c\gamma$. Suppose a and c unify with the most general unifier σ: $a\sigma = b\sigma$. Hence, $a\sigma\gamma = b\gamma\sigma$. Since $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \models_\nu a\sigma$ holds, we obtain that $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_i \rangle \models_\nu b\gamma\sigma$ holds. Hence, by Definition 2.8, $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle \models_\nu (b \otimes rest)\gamma\sigma$ and $\mathbf{M}, \langle \mathbf{D}_1 \ \ldots \mathbf{D}_n \rangle \models (\exists)\,(b \otimes rest)$ hold.

The proof for the inference Rules 2 and 3 is identical to the classical $\mathcal{TR}$ case [BK95], so we skip it. Since all the axioms and the inference rules used to prove sequents of the form $\mathbf{P}, \mathbf{D}\text{---}\vdash (\exists)\,\phi$ are sound, it results that the executional entailment $\mathbf{P}, \mathbf{D_1} \ldots \mathbf{D_n} \models (\exists)\,\phi$ holds.

Completeness:

Lets consider a serial goal b. We will prove the following claim. Here we use $\vdash_{\mathbf{n\text{-}t}}$ to denote non-tabled inference in the proof theory $\mathcal{F}^I$ from [BK95] and $\vdash$ for the tabled inference system $\mathcal{F}^T$.

Claim B.1: For any given path $\langle \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{n-1}, \mathbf{D}_n \rangle$, if $\mathbf{P}, \mathbf{D}_1\text{---}\vdash_{n-t} b$ with some execution path $\langle \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{n-1},\ \mathbf{D}_n \rangle$, then $\mathbf{P}, \mathbf{D}_1\text{---}\vdash b$ with an execution path $\langle \mathbf{D}_1, \mathbf{D}'_2 \ldots \mathbf{D}'_m,\ \mathbf{D}_n \rangle$ with the same extremities $\mathbf{D}_1$ and $\mathbf{D}_n$.

Using Claim B.1 and the completeness of the non-tabled inference system $\mathcal{F}^I$, we can also deduce the completeness of the tabled inference system. Namely, suppose the executional entailment $\mathbf{P}, \mathbf{D}_1\mathbf{D}_2 \ldots \mathbf{D}_{n-1}\mathbf{D}_n \models b$ holds. By completeness of the non-tabled inference system, it follows that $\mathbf{P}, \mathbf{D}_1\text{---}\vdash_{n-t} b$ is derivable with the execution path $\langle \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{n-1}, \mathbf{D}_n \rangle$. By the above Claim B.1, it follows that $\mathbf{P}, \mathbf{D}_1\text{---}\vdash b$ with some execution path $\langle \mathbf{D}_1, \mathbf{D}'_2 \ldots \mathbf{D}'_m,\ \mathbf{D}_n \rangle$ that ends in the final state $\mathbf{D}_n$.

It remains to prove Claim B.1. We will prove it by induction on the number of applications of inference axioms and rules of the non-tabled inference system $N \geq 1$. We will use a stronger variant of the non-tabled inference system $\mathcal{F}^I$, denoted $\mathcal{F}^I_S$, where sequents have the form $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_{n-t} \psi$. The inference rules 1, 2 and 3 are changed by replacing $\mathbf{D}_1\text{---}$ with $\mathbf{D}_1 \ldots \mathbf{D}_n$ and $\mathbf{D}_2\text{---}$ with $\mathbf{D}_2 \ldots \mathbf{D}_n$ everywhere. This is a simple extension of the inference system Ground $\mathcal{F}^I$ in [BK95] which also applies to non-ground transactions (Ground $\Im^I$ assumes that all transaction invocations are ground, but we don't make this assumption in $\mathcal{F}^I_S$). We will also use a stronger variant of the proof theory $\mathcal{F}^T$, where sequents have the form $\mathbf{P}, \mathbf{D}_1, \ldots \mathbf{D}_n \vdash \psi$. We will denote the stronger theory with $\mathcal{F}^T_S$. Similarly to [BK95], it can be shown that any proof in this stronger theory can be converted to a proof in the original theory. Rule 1a is changed as follows:

$$\frac{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\,(b \otimes rest)}$$

$$(b, \mathbf{D}_1) \in \texttt{ table space}$$

$$\forall\ \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\gamma,\ (b\gamma, \mathbf{D}_i) \in \texttt{ answer table}(b, \mathbf{D}_1)$$

Rule 1b is changed as follows: suppose (1) b's predicate symbol is declared as tabled, (2) there is a dominating pair $(c, \mathbf{D}_1)$ in the table space, (3) the answer table for $(c, \mathbf{D}_1)$ has an entry $(a, \mathbf{D}_i)$, and (4) a and b unify with most general unifier σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}_i \ldots \mathbf{D}_n \vdash (\exists)\,(rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \ldots \mathbf{D}_n \vdash (\exists)\,(b \otimes rest)}$$

The rest of the inference rules (i.e., 1c, 2 and 3) are changed by replacing $\mathbf{D}_1$--- with $\mathbf{D}_1 \ldots \mathbf{D}_n$ and $\mathbf{D}_2$--- with $\mathbf{D}_2 \ldots \mathbf{D}_n$ everywhere.

We now return to proving the Claim B.1 using the theory $\mathcal{F}_S^T$. Suppose $N = 1$. Then b is a fluent, an elementary update, or the propositional constant *state*. These can be derived only via the inference rules 2, 3, or the axioms. Since these steps are identical for the tabled and the non-tabled inference systems, the claim follows.

For the induction step, consider the sequent $\Sigma = \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_{n-t} \exists b \otimes rest$ derived via $N + 1$ derivation steps in the non-tabled inference system $\mathcal{F}_S^I$. Consider the last derivation step $N + 1$ where the sequent Σ was derived from some sequent Σ' of the form $\mathbf{P}, \mathbf{D}'_1, \mathbf{D}'_2 \ldots \mathbf{D}'_k \vdash_{n-t} \exists\, \psi$. For the final step, we have an application of some inference rule in $\mathcal{F}_S^I$ as follows:

$$\frac{\Sigma' = \ \mathbf{P}, \mathbf{D}'_1, \mathbf{D}'_2 \ldots \mathbf{D}'_{k-1}, \mathbf{D}'_k \vdash_{n-t} \exists\, \psi}{\Sigma = \ \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash_{n-t} \exists\, b \otimes rest} \tag{31}$$

The sequent Σ' must have been derived in $\leq N$ steps, thus, by the inductive hypothesis, we have that some sequent Υ' of the form $\mathbf{P}, \mathbf{D}'_1, \mathbf{D}''_2 \ldots \mathbf{D}'_{k-1}, \mathbf{D}'_k \vdash \exists\, \psi$ can be derived in the tabled inference system $\mathcal{F}_S^T$. We will prove that some Υ of the form $\mathbf{P}, \mathbf{D}_1, \mathbf{D}'''_2 \ldots \mathbf{D}'''_m, \mathbf{D}_n \vdash \exists\, b \otimes rest$ can be derived in the tabled inference system.

We now consider the different possibilities how Σ could have been derived from Σ', i.e., where (31) is Rule 1, 2 or 3 of $\mathcal{F}_S^I$.

The cases of Rules 2 and 3 are trivial, since these rules are identical in $\mathcal{F}_S^I$ and in $\mathcal{F}_S^T$.

Suppose that (31) is rule $\mathcal{F}_S^I$. Then there is a rule $head \leftarrow body$ in the program $\mathbf{P}$ whose variables have been renamed apart from $b \otimes rest$ and $head$ unifies with b with the most general unifier σ. Thus, in the above sequent Σ', we have that $\psi = body \otimes rest$, $n = k$ and $\mathbf{D}_1 = \mathbf{D}'_1, \ldots, \mathbf{D}_n = \mathbf{D}'_k$, $\Upsilon' = \mathbf{P}, \mathbf{D}_1, \mathbf{D}''_2 \ldots \mathbf{D}''_{n-1}, \mathbf{D}_n \vdash \exists\, body \otimes rest$ and $\Upsilon = \mathbf{P}, \mathbf{D}_1, \mathbf{D}'''_2 \ldots \mathbf{D}'''_m, \mathbf{D}_n \vdash \exists\, b \otimes rest$. We need to show that Υ is derivable in $\mathcal{F}_S^T$.

Suppose the predicate symbol for the call b is tabled and there is no dominating pair $(c, \mathbf{D}_1)$ in the table space. Rule 1a of $\mathcal{F}_S^T$ is applicable and Υ can be derived from Υ'. In addition, for all γ such that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_i \vdash b\ \sigma\gamma$, the answer $(b\ \sigma\gamma, \mathbf{D}_i)$ is added to the answer table for $(b, \mathbf{D}_1)$.

Suppose the predicate symbol for the call b is tabled and there is a dominating pair $(c, \mathbf{D}_1)$ in the table space. Since there is a proof of $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{n-1}, \mathbf{D}_n \vdash_{n-t} (b \otimes rest)\sigma$, there must be a proof of $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_{i-1}, \mathbf{D}_i \vdash_{n-t} b\sigma$ and of $\Sigma'' = \mathbf{P}, \mathbf{D}_i, \mathbf{D}_{i+1} \ldots \mathbf{D}_{n-1}, \mathbf{D}_n \vdash_{n-t} rest\sigma$. These proofs have fewer than $N+1$ steps, so, by induction, there is a proof in $\mathcal{F}_S^T$ of $\mathbf{P}, \mathbf{D}_1, \overline{\mathbf{D}_2} \ldots \overline{\mathbf{D}_r}, \mathbf{D}_i \vdash b\sigma$ and a proof of $\Upsilon'' = \mathbf{P}, \mathbf{D}_i, \mathbf{D}''_{i+1} \ldots \mathbf{D}'_m, \mathbf{D}_n \vdash rest\sigma$. Therefore, there must be an answer $(b\sigma, \mathbf{D}_i)$ in the answer table for $(c, \mathbf{D}_1)$, by the definition of Rule 1a. Therefore, we can apply Rule 1b to the sequent Υ'' and derive Υ.

If b is non-tabled, then Rule 1c of $\mathcal{F}_S^T$ applies to Υ' in exactly th same way as Rule 1 of $\mathcal{F}_S^I$ applies to Σ'.

$\square$

Appendix C

Tabled $\mathcal{TR}$ Termination

In this Appendix we prove termination of the inference system $\mathcal{F}^T$ developed in Section 3.1. A transaction may have: an infinite number of different answers on various paths, a finite number of answer substitutions on infinitely many paths, or a finite number of answers on a finite number of paths. The case of an infinite number of answer substitutions appears due to function symbols and infinite recursive relation derivations in a similar manner to the classical Horn logic programming. The case of a finite number of answer substitutions on infinitely many paths occurs due to infinitely many database transformations. If the number of databases generated by the elementary updates in a program is finite, then there is a finite number of pairs of path extremities $(InitialState, FinalState)$. If the program has no function symbols with arity greater than 0, then the proof for a serial-Horn goal always terminates due to the fact that there is a finite number of tabled calls and a finite number of databases, resulting in a finite number of answer substitutions and final database states. We have that for any tabled atomic subgoal b a dominating call c will be solved with the Inference Rule 1a., all the other subsequent calls being fed with results obtained for this dominant goal. Naturally, from the Inference Rule $1b$, it can be inferred that for all goals $rest$, if $b \otimes rest$ is the current goal in some database $\mathbf{D}$ with b's predicate symbol declared as tabled and a dominating pair $(c, \mathbf{D})$ in the table space, then the inference rule $1.b$ is applied taking an answer for $(c, \mathbf{D})$.

Theorem 3.3 (Termination): *Let* $\mathbf{P}$ *be a program with no function symbols with*

arity greater than 0, that is, it allows only constants (i.e., 0-ary function symbols). Let us further assume that all recursive predicates in $\mathbf{P}$ *are marked as tabled. Then, for any definite serial-Horn goal* ϕ*, the* tabled *proof theory finds one or more proofs of* $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$ *and* terminates.

Proof: The inference Rules 1b, 2 and 3 cannot be applied an infinite number of times because they reduce the size of the transaction formula, resulting that either the Rule 1c or the Rule 1a are applied infinitely many times. The Rule 1c applies only to non-tabled predicates that are non-recursive (all recursive predicates are tabled by the theorem hypothesis). The Rule 1a is applicable only for dominating goals, but since there are no function symbols, the number of different dominating goals is finite. It follows that all inference rules are applied finitely many times.

□

Note that we can compute upper bounds for the derivation trees defined in Section 3.1.1. The tabled-$\mathcal{TR}$ derivation tree for any definite serial-Horn goal ϕ in a database $\mathbf{D}$ corresponds to the proofs of $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\, \phi$ because for every application of the inference rules a new child node and an arc are created (construction correspondence from the proof theory to the tabled derivation trees). Since there are no function symbols there are a finite number of queries and a finite number of elementary updates because each ground fluent can be queried, inserted or deleted and there is only a finite number of fluents. If m is an upper bound on the arity of fluent predicates and there are a finite number of C constants in the system, then there can be at most C^m different tuples for each fluent. Let F be this number of ground fluents in the system that can be constructed from the initial database and the transaction base. We can have at most 2^F different database states, because this is the set of all subsets of F, including the empty set and F itself.

Following a similar reasoning as above (C is the finite number of constants in the system, m is an upper bound on the arity of tabled predicates), there can be at most C^m different ground calls to tabled predicates. For each ground call, there can be 2^m non-ground calls because every argument position can be taken by a variable. As a consequence, under such conditions, there will be a limited number T of variant calls to tabled predicates (it is less than $2^m * C^m$ for each tabled predicate symbol). Additionally, we can also see that the elementary updates are also limited to $2 * F$

because each fluent can only be inserted or deleted. We can see that there can be a finite number of sequents ($T * 2^F$) in the proof for any serial goal. This is also the finite number of producer tabled left-most variant subgoals, and, from the non-repeating property for tabled derivation trees, we can also see that the number of nodes in the tree for these producer subgoals is finite.

The number of entries in the answer table for each of the producer subgoals are also finite because there is a finite number of substitutions and a finite number of databases. As a consequence, the number of arcs originating in dominated sequents in the derivation tree is also finite.

We can also see that the elementary updates are also limited to $2 * F$ because each fluent can only be inserted or deleted and there is a finite number of fluents. Finally, we have a finite number of dominated goals because if R is the number of rules in the program and B is the maximal number of literals in the bodies of all the clauses in the program, then each node in the tree has a serial goal with at most $R * B * D$ atoms (each rule can be applied for each database, with the maximum number of body literals). This gives us the limit on the *depth* of each branch in the tree. The result that the number of nodes in the tree is finite follows from the fact that we have a finite depth for each branch and a finite number of nodes (in effect, a finite branching factor for each node).

Appendix D

Unique Least Model for `not`-free $\mathcal{TR}$ Programs

In this appendix, we prove that any `not`-free $\mathcal{TR}$ program has a unique least partial model (see Section 4.1).

Theorem 4.4 (Unique Least Partial Model for serial `not`-free $\mathcal{TR}$ programs) *If* $\mathbf{P}$ *is a `not`-free* $\mathcal{TR}$ *program, then* $\mathbf{P}$ *has a least Herbrand model, denoted* $LPM(\mathbf{P})$*.*

Proof: Let $\mathbf{P}^+$ denote the positive program obtained from $\mathbf{P}$ by replacing all body literals of the form $\mathbf{u}^\pi$, where π is a path, with $\mathbf{t}^\pi$. (We will call such literals $\mathbf{u}$ - literals and $\mathbf{t}$ - literals, respectively.) Similarly, let denote $\mathbf{P}^-$ the positive program obtained by deleting the rules whose body includes $\mathbf{u}$ - literals. (This is equivalent to replacing all $\mathbf{u}$ - literals with a propositional constant $\mathbf{f}$ that is false on any paths). Note that both $\mathbf{P}^+$ and $\mathbf{P}^-$ have unique minimal Herbrand models, since they do not have the special literals $\mathbf{u}^\pi$ and thus are simply serial-Horn clauses; these minimal models are 2 - valued, as shown in [BK95].

Let $\mathbf{M}^+$ be the least model of $\mathbf{P}^+$ and $\mathbf{M}^-$ be the least model of $\mathbf{P}^-$. As noted above, both of these models are 2 - valued. Clearly, $\mathbf{P}^-$ is a subprogram of $\mathbf{P}^+$, so $\mathbf{M}^+$ is also a model of $\mathbf{P}^-$. Since $\mathbf{M}^-$ is the least model of $\mathbf{P}^-$, it follows that $\mathbf{M}^- \preceq \mathbf{M}^+$.

Thus, for any path π and any `not`-free literal L

$$\mathbf{M}^{-}(\pi)(L) \leq \mathbf{M}^{+}(\pi)(L) \quad \text{and} \quad \mathbf{M}^{+}(\pi)(\texttt{not}\, L) \leq \mathbf{M}^{-}(\pi)(\texttt{not}\, L) \tag{32}$$

This means that all `not`-free literals that are true in $\mathbf{M}^{-}(\pi)$ are also true in $\mathbf{M}^{+}(\pi)$ and all `not`- literals that are true in $\mathbf{M}^{+}(\pi)$ are also true in $\mathbf{M}^{-}(\pi)$.

We construct the least model $\mathbf{M}$ of $\mathbf{P}$ as a path structure such that, for any path π, $\mathbf{M}(\pi)$ is the classical Herbrand structure where

$$\begin{array}{llll} - & \mathbf{M}(\pi)(L) = \mathbf{t} & \text{iff} & \mathbf{M}^{-}(\pi)(L) = \mathbf{t} \\ & \mathbf{M}(\pi)(\texttt{not}\, L) = \mathbf{f} & \text{iff} & \mathbf{M}^{-}(\pi)(\texttt{not}\, L) = \mathbf{f} \\ - & \mathbf{M}(\pi)(L) = \mathbf{f} & \text{iff} & \mathbf{M}^{+}(\pi)(L) = \mathbf{f} \\ & \mathbf{M}(\pi)(\texttt{not}\, L) = \mathbf{t} & \text{iff} & \mathbf{M}^{+}(\pi)(\texttt{not}\, L) = \mathbf{t} \\ - & \text{otherwise,} \quad \mathbf{M}(\pi)(L) = \mathbf{M}(\pi)(\texttt{not}\, L) = \mathbf{u} & & \end{array} \tag{33}$$

for any ground `not`-free literal L. We will now prove that $\mathbf{M}$ is $LPM(\mathbf{P})$, the unique minimal model of $\mathbf{P}$.

By (32), $\mathbf{M}$ is well - defined, since it is not possible that $\mathbf{M}^{-}(\pi)(L) = \mathbf{t}$ and $\mathbf{M}^{+}(\pi)(L) = \mathbf{f}$ or that $\mathbf{M}^{-}(\pi)(\texttt{not}\, L) = \mathbf{f}$ and $\mathbf{M}^{+}(\pi)(\texttt{not}\, L) = \mathbf{t}$.

Next, we show that $\mathbf{M}$ is a partial model of $\mathbf{P}$. Suppose C is a rule in $\mathbf{P}$ of the form $H\ :-\ B_1 \otimes ... \otimes B_n$, such that none of the B_is is a $\mathbf{u}$ - literal. By definition, C belongs *both* to $\mathbf{P}^{-}$ and $\mathbf{P}^{+}$. If, for some path π, $\mathbf{M}(\pi)(B_1 \otimes ... \otimes B_n) = \mathbf{t}$, then $\mathbf{M}^{-}(\pi)(B_1 \otimes ... \otimes B_n) = \mathbf{t}$, by the construction of $\mathbf{M}$ in (33). Since $\mathbf{M}^{-}$ is a model of C, the head H of C must be true in $\mathbf{M}^{-}(\pi)$ hence also in $\mathbf{M}(\pi)$. Thus, $\mathbf{M}(\pi)$ makes C true. If $\mathbf{M}(\pi)(B_1 \otimes ... \otimes B_n) = \mathbf{f}$ then $\mathbf{M}(\pi)$ satisfies C trivially. If $\mathbf{M}(\pi)(B_1 \otimes ... \otimes B_n) = \mathbf{u}$, it means that, for some split $\pi = \pi_1 \circ ... \circ \pi_n$, $\mathbf{M}(\pi_i)(B_i)$ is either $\mathbf{u}$ or $\mathbf{t}$. By (33), this implies $\mathbf{M}^{+}(\pi_i)(L) = \mathbf{t}$, and since $\mathbf{M}^{+}$ is also a model of C it follows that H must be true in $\mathbf{M}^{+}(\pi)$. The definition of $\mathbf{M}$ then implies that H must have the truth value $\mathbf{u}$ or $\mathbf{t}$ in $\mathbf{M}(\pi)$, so $\mathbf{M}(\pi)$ satisfies C once again.

Next, suppose that C is a clause $H\ :-\ B_1 \otimes ... \otimes B_n$ in $\mathbf{P} \setminus \mathbf{P}^{-}$, and suppose π is a path with a split $\pi = \pi_1 \circ ... \circ \pi_n$ such that none of the $\mathbf{M}(\pi_i)(B_i) = \mathbf{f}$. Note that since C is not in $\mathbf{P}^{-}$, at least one of the B_is must be $\mathbf{u}^{\pi_i}$ for some subpath π_i. So, it must be the case that $\mathbf{M}(B_1 \otimes ... \otimes B_n) = \mathbf{u}$ (it cannot be $\mathbf{t}$ because of $\mathbf{u}^{\pi_i}$ and it

cannot be $\mathbf{f}$ because of the assumption that none of the $\mathbf{M}(\pi_i)(B_i)$s is $\mathbf{f}$). This implies (again by (33)) that none of the $\mathbf{M}^+(\pi_i)(B_i)$s equals $\mathbf{f}$. Therefore $\mathbf{M}(\pi_i)(B_i) = \mathbf{t}$ for all body literals in the corresponding clause C^+ in $\mathbf{P}^+$ (one that is obtained from C by changing each $\mathbf{u}^{\pi_i}$ to $\mathbf{t}^{\pi_i}$). Therefore, H (which is the head of both C and C^+) must be true in $\mathbf{M}^+(\pi)$. So $\mathbf{M}(\pi)(H)$ is either $\mathbf{t}$ or $\mathbf{u}$. Thus, $\mathbf{M}(\pi)$ models every rule in $\mathbf{P} \setminus \mathbf{P}^-$ either and, therefore, $\mathbf{M}$ is a model of $\mathbf{P}$.

To prove minimality and uniqueness of $\mathbf{M}$, let $\mathbf{N}$ be a model of $\mathbf{P}$. We will show that $\mathbf{M} \preceq \mathbf{N}$, which would imply that $\mathbf{M}$ is the least model. We need to establish the following properties:

Property D.1: $\mathbf{M}(\pi)(L) \leq \mathbf{N}(\pi)(L)$

Property D.2: $\mathbf{N}(\pi)(\mathtt{not}\, L) \leq \mathbf{M}(\pi)(\mathtt{not}\, L)$

The proof of these relies on the following claims, which will be proved at the end:

Claim D.1: If $\mathbf{M}^-(\pi)(L) = \mathbf{t}$ then $\mathbf{N}(\pi)(L) = \mathbf{t}$.

Claim D.2: If $\mathbf{M}^-(\pi)(\mathtt{not}\, L) = \mathbf{f}$ then $\mathbf{N}(\pi)(\mathtt{not}\, L) = \mathbf{f}$.

Claim D.3: If $\mathbf{M}^+(\pi)(L) = \mathbf{t}$ then $\mathbf{N}(\pi)(L) \geq \mathbf{u}$.

Claim D.4: If $\mathbf{M}^+(\pi)(\mathtt{not}\, L) = \mathbf{f}$ then $\mathbf{N}(\pi)(\mathtt{not}\, L) \leq \mathbf{u}$.

To establish Property D.1, suppose that $\mathbf{M}(\pi)(L) = \mathbf{t}$. By (33), this means that $\mathbf{M}^-(\pi)(L) = \mathbf{t}$ and, by Claim D.1, $\mathbf{N}(\pi)(L) = \mathbf{t}$. If $\mathbf{M}(\pi)(L) = \mathbf{u}$ then, by (33), this implies $\mathbf{M}^+(\pi)(L) = \mathbf{t}$ and, by Claim D.3, $\mathbf{N}(\pi)(L) \geq \mathbf{u} = \mathbf{M}(\pi)(L)$. This proved Property D.1.

For Property D.2, suppose $\mathbf{M}(\pi)(\mathtt{not}\, L) = \mathbf{u}$. By the definition of $\mathbf{M}$, this implies $\mathbf{M}^+(\pi)(\mathtt{not}\, L) = \mathbf{f}$ and, by Claim D.4, $\mathbf{N}(\pi)(\mathtt{not}\, L) \leq \mathbf{u} = \mathbf{M}(\pi)(\mathtt{not}\, L)$. Similarly, if $\mathbf{M}(\pi)(\mathtt{not}\, L) = \mathbf{f}$, then $\mathbf{M}^-(\pi)(\mathtt{not}\, L) = \mathbf{f}$ and, by Claim D.2, $\mathbf{N}(\pi)(\mathtt{not}\, L) = \mathbf{f}$.

It remains to prove claims D.1–D.4. Claims D.1 and D.2 follow directly from the fact that $\mathbf{N}$ is be a model of $\mathbf{P}^-$ for which $\mathbf{M}^-$ is the least model, so $\mathbf{M}^- \preceq \mathbf{N}$. Claim D.3 can be easily proved by induction on the number of inference rules that need to be used in order to prove that L is true on π with respect to the program $\mathbf{P}^+$. Claim D.4 follows from Claim D.3: If $\mathbf{M}^+(\pi)(\mathtt{not}\, L) = \mathbf{f}$ then $\mathbf{M}^+(\pi)(L) = \mathbf{t}$. By Claim D.3, $\mathbf{N}(\pi)(L) \geq \mathbf{u}$, which implies that $\mathbf{N}(\pi)(\mathtt{not}\, L) \leq \mathbf{u}$. □

Appendix E

$\mathcal{TR}^{DA}$ Fixpoint and Well-founded Model

In this appendix, we prove that any $\mathcal{TR}^{DA}$ program has a unique well-founded model. For convenient reference, we reproduce the Definition 4.25 from Section 4.1 below.

Definition 4.25 (**$\mathcal{TR}^{DA}$ Quotient**): *Let* $\mathbf{P}$ *be a set of* $\mathcal{TR}^{DA}$ *rules and* $\boldsymbol{I}$ *a path structure for* $\mathbf{P}$. *The* $\mathcal{TR}^{DA}$ ***quotient of*** $\mathbf{P}$ ***by*** $\boldsymbol{I}$, *written as* $\dfrac{\mathbf{P}}{\boldsymbol{I}}$, *is defined through the following sequence of steps:*

1. *First, each occurrence of every* $\texttt{not}$ *- literal of the form* $\texttt{not}\,L$ *in* $\mathbf{P}$ *is replaced by* $\mathbf{t}^{\pi}$ *for every path* π *such that* $\boldsymbol{I}(\pi)(\texttt{not}\,L) = \mathbf{t}$ *and with* $\mathbf{u}^{\pi}$ *for every path* π *such that* $\boldsymbol{I}(\pi)(\texttt{not}\,L) = \mathbf{u}$.

2. *For each labeled rule of the form* $@r\ L \,{:}{-}\, Body$ *obtained in the previous step, replace it with the rules of the form:*

$$L \,{:}{-}\, \mathbf{t}^{\langle \mathbf{D}_t \rangle} \otimes Body$$
$$L \,{:}{-}\, \mathbf{u}^{\langle \mathbf{D}_u \rangle} \otimes Body$$

for each database state $\mathbf{D}_t$ *such that*

$$\boldsymbol{I}(\langle \mathbf{D}_t \rangle)(\texttt{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L)))) \ = \ \mathbf{t}$$

and each database state $\mathbf{D}_u$ such that

$$\boldsymbol{I}(\langle \mathbf{D}_u \rangle)(\boldsymbol{not}\,(\Diamond\ \texttt{\$defeated(handle(}r, L\texttt{))))}\ =\ \mathbf{u}$$

3. *Remove the labels from the remaining rules.*

The resulting set of rules is the quotient $\frac{\mathbf{P}}{\boldsymbol{I}}$. □

Note that, the $\mathcal{TR}^{DA}$ quotient of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to an argumentation theory AT (denoted $(\mathbf{P}, AT)$) for any path structure $\mathbf{I}$, $\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}$, is a negation-free $\mathcal{TR}$ program, so, by Theorem 4.4, it has a unique least Herbrand model, $LPM(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}})$.

We will now give the definition for the immediate consequence operator Γ. We will use the set representation of Herbrand models: $\boldsymbol{I}^+ = \{L \mid\ L \in \boldsymbol{I}$ *is a* $\boldsymbol{not}$*-free literal*$\}$, $\boldsymbol{I}^- = \{L \mid\ L \in \boldsymbol{I}$ *is a* $\boldsymbol{not}$*-literal*$\}$ and $\boldsymbol{I}\ =\ \boldsymbol{I}^+ \cup \boldsymbol{I}^-$.

Definition 4.26 (**$\mathcal{TR}^{DA}$ immediate consequence operator**):
The incremental consequence operator, Γ, for a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ with respect to the argumentation theory AT takes as input a path structure $\boldsymbol{I}$ and generates a new path structure as follows:

$$\Gamma(\boldsymbol{I}) =^{def} LPM\left(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}\right)$$

Suppose $I_\emptyset$ is the path structure that maps each path π to the empty *Herbrand interpretation in which all propositions are undefined (i.e., for every path π and every literal L, we have $I_\emptyset(\pi)(L) = \mathbf{u}$.*
The ordinal powers of the immediate consequence operator Γ are defined inductively as follows:

- $\Gamma^{\uparrow 0}(I_\emptyset)\ =\ I_\emptyset$;
- $\Gamma^{\uparrow \alpha}(I_\emptyset)\ =\ \Gamma(\Gamma^{\uparrow \alpha - 1}(I_\emptyset))$, *for α a successor ordinal;*
- $\Gamma^{\uparrow \alpha}(I_\emptyset)(\pi)\ =\ \cup_{\beta < \alpha} \Gamma^{\uparrow \beta}(I_\emptyset)(\pi)$, *for every path π and α a limit ordinal.*

□

The following lemma states a basic result about the immediate consequence operator Γ.

Lemma E.1 (Γ is monotonic) *The operator Γ is monotonic with respect to the information order relation $\leq$ when $\mathbf{P}$ and AT are fixed, i.e.*

$$\Gamma(I) \ \leq \ \Gamma(I') \quad \textit{if} \quad I \ \leq \ I' \ .$$

Proof:

The proof relies on the following claim.

Claim E.1: if I and I' are two Herbrand path structures, $I \leq I'$, $\mathbf{D}$ is a database state and $\Gamma^{\uparrow n}(I)(\langle \mathbf{D} \rangle)(\texttt{not}\,(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ )))) = v$ with $v \in \{\mathbf{f}, \mathbf{t}\}$, then $\Gamma^{\uparrow n}(I')(\langle \mathbf{D} \rangle)(\texttt{not}\,(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ )))) = v$.

Proof of Claim E.1:
By hypothesis, $\Gamma^{\uparrow n}(I)(\langle \mathbf{D} \rangle)(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ ))) = \sim v$. If $v = \mathbf{t}$, then $\Gamma^{\uparrow n}(I)(\rho)(\texttt{\$defeated}(\texttt{handle}(r, B\ )) = \mathbf{f} = \sim v$ for every path ρ that starts at $\mathbf{D}$, and, by induction hypothesis, $\Gamma^{\uparrow n}(I')(\rho)(\texttt{\$defeated}(\texttt{handle}(r, B\ )) = \mathbf{f} = \sim v$. If $v = \mathbf{f}$, then $\Gamma^{\uparrow n}(I)(\rho)(\texttt{\$defeated}(\texttt{handle}(r, B\ )) = \mathbf{t}$ for some path ρ that starts at $\mathbf{D}$, and, by induction hypothesis, $\Gamma^{\uparrow n}(I')(\rho)(\texttt{\$defeated}(\texttt{handle}(r, B\ )) = \mathbf{t} = \sim v$. In both cases, by Definition 4.21, $\Gamma^{\uparrow n}(I')(\langle \mathbf{D} \rangle)(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ ))) = \sim v$. Hence, $\Gamma^{\uparrow n}(I')(\langle \mathbf{D} \rangle)(\texttt{not}\,(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ )))) = v$. Q.E.D.

Continuing with the proof of Lemma E.1, let I and I' be two Herbrand path structures, where $I \leq I'$. Thus, for any path π, $I(\pi)^{+} \ \subseteq \ I'(\pi)^{+}$ and $I(\pi)^{-} \ \subseteq \ I'(\pi)^{-}$. In order to show that $\Gamma(I) \ \leq \ \Gamma(I')$, we will prove that $\Gamma^{\uparrow n}(I) \ \leq \ \Gamma^{\uparrow n}(I')$, for all n. This is true for $n = 0$ (since $I \ \leq \ I'$).

Suppose $\Gamma^{\uparrow n}(I) \ \leq \ \Gamma^{\uparrow n}(I')$ holds true for some n, and $\Gamma^{\uparrow n+1}(I)(\pi)(A) = \mathbf{t}$ for some literal A. There must be a clause $@r \ B \ :- \ L_1 \otimes \ldots \otimes L_m$ in $\mathbf{P}$ and a ground substitution θ such that $A = B\theta$, $\Gamma^{\uparrow n}(I)(\pi)(L_1\theta \otimes \ldots \otimes L_m\theta) = \mathbf{t}$ and $\Gamma^{\uparrow n}(I)(\langle \mathbf{D}_0 \rangle)(\texttt{not}\,(\Diamond \ \texttt{\$defeated}(\texttt{handle}(r, B\ \theta)))) = \mathbf{t}$, where $\mathbf{D}_0$ is the initial database of π. By Definition 4.21, there exists a split $\pi = \langle \mathbf{D}_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$, such that $\Gamma^{\uparrow n}(I)(\pi_i)(L_i\theta) = \mathbf{t}$ for each $1 \leq i \leq m$. In $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{I}}$, we have rules of the form $B \ :- \ \mathbf{t}^{\langle \mathbf{D}_t \rangle} \ \otimes L'_1 \otimes \ldots \otimes L'_m$ and $B \ :- \ \mathbf{u}^{\langle \mathbf{D}_u \rangle} \ \otimes L'_1 \otimes \ldots \otimes L'_m$, where the literals L'_i $(1 \leq i \leq m)$ denote the results of Step 1 transformation, i.e., L'_i is either L_i, if L_i is a `not`-free literal, or $\mathbf{t}^{\rho}$ or $\mathbf{u}^{\rho}$, for some path ρ where the Step 1 conditions in Definition 4.25 are satisfied. By the induction hypothesis, if $\Gamma^{\uparrow n}(I)(\pi_i)(L_i\theta) = \mathbf{t}$, then $\Gamma^{\uparrow n}(I')(\pi_i)(L_i\theta) = \mathbf{t}$. For every $L_i\theta$ `not`-free literal, we have $L'_i\theta = L_i\theta$, so

$\Gamma^{\uparrow n}(I')(\pi_i)(L_i'\theta) = \mathbf{t}$. If $L_i\theta$ is a `not`-literal, then from $\Gamma^{\uparrow n}(I)(\pi_i)(L_i\theta) = \mathbf{t}$ follows that $L_i'\theta = \mathbf{t}\pi_i$ and, by Definition 4.21, $\Gamma^{\uparrow n}(I')(\pi_i)(L_i'\theta) = \Gamma^{\uparrow n}(I')(\pi_i)(\mathbf{t}\pi_i) = \mathbf{t}$. Since π can be split into $\pi_1 \circ \ldots \circ \pi_m$, it follows that $\Gamma^{\uparrow n}(I')(\pi)(L_1'\theta \otimes \ldots \otimes L_m'\theta) = \mathbf{t}$. By Claim E.1, $\Gamma^{\uparrow n}(I')(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\Diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, B\ \theta)))) = \mathbf{t}$. It follows that $\Gamma^{\uparrow n+1}(I')(\pi)(B\theta) = \Gamma^{\uparrow n+1}(I')(\pi)(A) = \mathbf{t}$.

Suppose $\Gamma^{\uparrow n+1}(I)(\pi)(A) = \mathbf{f}$ for some literal A. For any clause $@r\ B\ :-\ L_1 \otimes \ldots \otimes L_m$ in $\mathbf{P}$ such that $A = B\theta$ for some substitution θ, $\Gamma^{\uparrow n}(I)(\pi)(L_1\theta \otimes \ldots \otimes L_m\theta) = \mathbf{f}$ or $\Gamma^{\uparrow n}(I)(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\Diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, B\ \theta)))) = \mathbf{f}$ (and the rule is not present in the quotient $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{I}}$). By Definition 4.21, for any split $\pi = \langle \mathbf{D}_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$, we have $\Gamma^{\uparrow n}(I)(\pi_i)(L_i\theta) = \mathbf{f}$ for some $1 \leq i \leq m$. By induction hypothesis, if $\Gamma^{\uparrow n}(I)(\pi_i)(L_i\theta) = \mathbf{f}$, then $\Gamma^{\uparrow n}(I')(\pi_i)(L_i\theta) = \mathbf{f}$. If $L_i\theta$ is a `not`-free literal, then $L_i' = L_i$ and $\Gamma^{\uparrow n}(I')(\pi_i)(L_i'\theta) = \mathbf{f}$. Hence, $\Gamma^{\uparrow n}(I')(\pi)(L_1' \otimes \ldots \otimes L_m') = \mathbf{f}$. If $L_i\theta$ is a `not`-literal, then the corresponding rule is not present in the quotient. On the other hand, by Claim E.1, $\Gamma^{\uparrow n}(I')(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\Diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, B\ \theta)))) = \mathbf{f}$ and the rule is not present in the quotient corresponding to I'. In all cases, it follows that $\Gamma^{\uparrow n+1}(I')(\pi)(A) = \mathbf{f}$. □

Since Γ is monotonic, the sequence $\{\Gamma^{\uparrow n}(I_\emptyset)\}$ has a limit which is the unique least fixed point of Γ. It is computable via transfinite induction [Mos74, Llo84].

Definition 4.27 **(Well-founded model)**: *The* ***well - founded model*** *of a $\mathcal{TR}^{DA}$ transaction base* $\mathbf{P}$ *with respect to the argumentation theory AT, written as* $WFM(\mathbf{P}, AT)$*, is defined as the limit of the sequence* $\{\Gamma^{\uparrow n}(I_\emptyset)\}$. □

We will show that $WFM(\mathbf{P}, AT)$ is a model of $(\mathbf{P}, AT)$ by using the following lemma. This lemma states that the application of the immediate consequence operator Γ on models of the program $(\mathbf{P}, AT)$ results in smaller models with respect to the truth order $\preceq$. The reciprocal direction is also true.

Lemma E.2 *:* $\mathbf{N}$ *is a model of* $(\mathbf{P}, AT)$ ***iff*** $\Gamma(\mathbf{N}) \preceq \mathbf{N}$.

Proof:

The proof relies on the following claim.

Claim E.2: if a rule of the form:

$$B \ :- \ L'_0 \ \otimes \ L'_1 \otimes \ldots \otimes L'_m. \tag{34}$$

in $\dfrac{\mathbf{P} \cup AT}{\mathbf{N}}$ was obtained using the quotient Definition 4.25 from a rule of the form:

$$@r \ B \ :- \ L_1 \otimes \ldots \otimes L_m \tag{35}$$

in $(\mathbf{P}, AT)$, then $\mathtt{min}(\mathbf{N}(\pi)(L'_1), \ldots, \mathbf{N}(\pi)(L'_m)) \ = \ \mathtt{min}(\mathbf{N}(\pi)(L_1), \ldots, \mathbf{N}(\pi)(L_m))$ and $\mathbf{N}(\langle \mathbf{D}_0 \rangle)(L'_0) = \mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, L))))$, for any path $\pi = \langle \mathbf{D}_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$.

Proof of Claim E.2:

If L_i is a $\mathtt{not}$-free literal, then $L'_i = L_i$, so $\mathbf{N}(\rho)(L'_i) = \mathbf{N}(\rho)(L_i)$ for every path ρ. If L_i is a $\mathtt{not}$-literal $\mathtt{not}\, C_i$, then:

- if $\mathbf{N}(\rho)(\mathtt{not}\, C_i) = \mathbf{t}$, then $L'_i = \mathbf{t}^{\rho}$. Hence, $\mathbf{N}(\rho)(L'_i) = \mathbf{N}(\rho)(\mathbf{t}^{\rho}) = \mathbf{t}$,
- if $\mathbf{N}(\rho)(\mathtt{not}\, C_i) = \mathbf{u}$, then $L'_i = \mathbf{u}^{\rho}$. Hence, $\mathbf{N}(\rho)(L'_i) = \mathbf{N}(\rho)(\mathbf{u}^{\rho}) = \mathbf{u}$,
- if $\mathbf{N}(\rho)(\mathtt{not}\, C_i) = \mathbf{f}$, then the rule in $\dfrac{\mathbf{P} \cup AT}{\mathbf{N}}$ is not created,

for any path ρ. Hence, in all the above cases, when the rule (34) exists, we have that $\mathbf{N}(\rho)(L'_i) = \mathbf{N}(\rho)(L_i)$. Therefore, $\mathtt{min}(\mathbf{N}(\pi)(L'_1), \ldots, \mathbf{N}(\pi)\ (L'_m)) \ = \ \mathtt{min}(\mathbf{N}(\pi)(L_1), \ldots, \mathbf{N}(\pi)(L_m))$.

By the quotient Definition 4.25, the literal L'_0 in the Rule (34) must be either the propositional constant $\mathbf{t}^{\langle \mathbf{D}_0 \rangle}$, if $\mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, L)))) \ = \ \mathbf{t}$, or the propositional constant $\mathbf{u}^{\langle \mathbf{D}_0 \rangle}$, if $\mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, L)))) \ = \ \mathbf{u}$. However, the propositional constant $\mathbf{t}^{\langle \mathbf{D}_0 \rangle}$ is true only on the path $\langle \mathbf{D}_0 \rangle$, otherwise it is false, while the propositional constant $\mathbf{u}^{\langle \mathbf{D}_0 \rangle}$ is undefined only on the path $\langle \mathbf{D}_0 \rangle$, otherwise it is false. Hence, $\mathbf{N}(\langle \mathbf{D}_0 \rangle)(L'_0) = \mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not}\,(\diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, L))))$. Q.E.D.

We continue with the proof of Lemma E.2.

($\Rightarrow$):

In order to show that $\Gamma(\mathbf{N}) \preceq \mathbf{N}$, we have to prove that $\Gamma(\mathbf{N})(\pi)(A) \leq \mathbf{N}(\pi)(A)$ for all paths π and all literals A.

Since the path structure $\Gamma(\mathbf{N})$ is the least partial model of the program $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{N}}$, it follows that:

- $\Gamma(\mathbf{N})$ satisfies every rule in $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{N}}$, i.e., for every clause in $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{N}}$ of the form (34) we have:

$$\Gamma(\mathbf{N})(\pi)(B) \;\geq\; \Gamma(\mathbf{N})(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L'_m) \tag{36}$$

  for every path π;

- $\Gamma(\mathbf{N})$ is minimum model, i.e., for any other model M' of $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{N}}$, we have:

$$\Gamma(\mathbf{N}) \;\preceq\; M' \tag{37}$$

By Definition 4.25, each rule of $\dfrac{\mathbf{P} \cup AT}{\mathbf{N}}$ must correspond to a rule in $(\mathbf{P}, AT)$ of the form (35) where the L'_i literals $(1 \leq i \leq m)$ are the results of Step 1 transformation, i.e., L'_i is either L_i, if L_i is a `not`-free literal, or $\mathbf{t}^{\rho}$ or $\mathbf{u}^{\rho}$ for some path ρ, if L_i is a `not`-literal. By hypothesis, $\mathbf{N}$ models every rule in $(\mathbf{P}, AT)$:

$$\mathbf{N}(\pi)(B) \geq \mathtt{min}(\,\mathbf{N}(\pi)(L_1 \otimes \ldots \otimes L_m),\; \mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not} \diamond \mathtt{\$defeated}(\mathtt{handle}(r, B)))). \tag{38}$$

where $\mathbf{D}_0$ is the initial database in the path π.

Consider a split $\pi = \langle \mathbf{D}_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$. By Definition 4.21, $\mathbf{N}(\pi)(L_1 \otimes \ldots \otimes L_m) = \mathtt{min}(\mathbf{N}(\pi_1)(L_1), \ldots, \mathbf{N}(\pi_m)(L_m))$. Hence, $\mathbf{N}(\pi)(B) \geq \mathtt{min}(\mathbf{N}(\pi_1)(L_1), \ldots, \mathbf{N}(\pi_m)(L_m))$. We also have $\mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L'_m) = \mathtt{min}(\mathbf{N}(\langle \mathbf{D}_0 \rangle)(L'_0), \mathbf{N}(\pi_1)(L'_1), \ldots, \mathbf{N}(\pi_m)(L'_m))$. Hence, by Claim E.2, $\mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L'_m) = \mathtt{min}(\mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not} \diamond \mathtt{\$defeated}(\mathtt{handle}(r, B)))), \mathbf{N}(\pi_1)(L_1), \ldots, \mathbf{N}(\pi_m)(L_m))$. Hence, $\mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L'_m) = \mathtt{min}(\mathbf{N}(\langle \mathbf{D}_0 \rangle)(\mathtt{not} \diamond \mathtt{\$defeated}(\mathtt{handle}(r, B)))), \mathbf{N}(\pi)(L_1 \otimes \ldots \otimes L_m))$. By (38), $\mathbf{N}(\pi)(B) \;\geq\; \mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L'_m)$. Therefore, $\mathbf{N}$ models every rule (34). It follows that $\mathbf{N}$ is a model of $\dfrac{\mathbf{P} \cup AT}{\boldsymbol{N}}$, and, by (37), that $\Gamma(\mathbf{N}) \preceq \mathbf{N}$.

($\Leftarrow$):

In order to show that $\mathbf{N}$ is a model of $(\mathbf{P}, AT)$ we have to prove that $\mathbf{N}$ models

every rule in $(\mathbf{P}, AT)$.

By hypothesis, $\mathbf{N} \geq LPM(\frac{\mathbf{P} \cup AT}{\mathbf{N}})$. Hence, $\mathbf{N}$ is a model of the program $\frac{\mathbf{P} \cup AT}{\mathbf{N}}$. Therefore, for every rule $B \ :- \ L'_0 \ \otimes L'_1 \otimes \ldots \otimes L'_m$ and path π, we have $\mathbf{N}(\pi)(B) \ \geq \ \mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L_m)$. Consider a split $\pi = \langle D_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$. By Definition 4.21, $\mathbf{N}(\pi)(L'_0 \otimes L'_1 \otimes \ldots \otimes L_m) = \mathtt{min}(\mathbf{N}(\langle D_0 \rangle)\ (L'_0),\, \mathbf{N}(\pi_1)\ (L'_1),\, \ldots,\, \mathbf{N}(\pi_m)\ (L'_m))$.

By Claim E.2, $\mathbf{N}(\langle D_0 \rangle)\ (L'_0) = \mathbf{N}(\langle D_0 \rangle)\ (\mathtt{not}\,(\Diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, L))))$ and $\mathtt{min}(\mathbf{N}(\pi)(L_1), \ldots, \mathbf{N}(\pi)(L_m)) \ = \ \mathtt{min}(\mathbf{N}(\pi)(L'_1), \ldots, \mathbf{N}(\pi)(L'_m))$. Hence, $\mathbf{N}(\pi)(L_1 \otimes \ldots \otimes L_m) = \mathtt{min}(\mathbf{N}(\pi_1)\ (L'_1),\, \ldots,\, \mathbf{N}(\pi_m)\ (L'_m))$. Therefore, $\mathbf{N}(\pi)(B) \geq \mathtt{min}\ (\mathbf{N}(\pi)\ (L_1 \otimes \ldots \otimes L_m),\ \ \mathbf{N}(\langle \mathbf{D}_0 \rangle)\ (\mathtt{not}\,\Diamond\ \mathtt{\$defeated}(\mathtt{handle}(r, B))))$. It follows that $\mathbf{N}$ models every rule in $(\mathbf{P}, AT)$. Hence $\mathbf{N}$ is a model of $(\mathbf{P}, AT)$. □

The following corollary states that $WFM(\mathbf{P}, AT)$ is a model of the program $\mathbf{P}$ with respect to the argumentation theory AT.

Corollary E.3 *:* $WFM(\mathbf{P}, AT)$ *is a model of* $(\mathbf{P}, AT)$*.*

Proof: By Definition 4.27, $\Gamma(WFM(\mathbf{P}, AT)) \ = \ WFM(\mathbf{P}, AT)$. Hence, by Lemma E.2, it follows that $WFM(\mathbf{P}, AT)$ is a model. □

The next theorem states that our constructive computation of the least model of the program $(\mathbf{P}, AT)$ is correct.

Theorem 4.5 (Correctness of the Constructive $\mathcal{TR}^{DA}$ Least Model)

$WFM(\mathbf{P}, AT)$ *is the least model of* $(\mathbf{P}, AT)$*.*

Proof:

By Corollary E.3, $WFM(\mathbf{P}, AT)$ is a model of $(\mathbf{P}, AT)$. We will show by contradiction that $WFM(\mathbf{P}, AT)$ is the least model. Suppose that $\mathbf{N}$ is a model of $(\mathbf{P}, AT)$ such that $\mathbf{N} \preceq WFM(\mathbf{P}, AT)$. Hence, $\mathbf{N}(\pi)(A) \preceq WFM(\mathbf{P}, AT)(\pi)(A)$ for every path π and literal A. We will show that we must have that $WFM(\mathbf{P}, AT) \preceq \mathbf{N}$, i.e., $WFM(\mathbf{P}, AT)(\pi)^+ \subseteq \mathbf{N}(\pi)^+$ and $\mathbf{N}(\pi)^- \subseteq WFM(\mathbf{P}, AT)(\pi)^-$.

The set inclusion for positive literals can be shown using the monotonicity of the immediate consequence operator Γ since $I_\emptyset(\pi)^+ \subseteq \mathbf{N}(\pi)^+$: $\Gamma^{\uparrow 1}(I_\emptyset)(\pi)^+ \subseteq \Gamma(\mathbf{N})(\pi)^+$

after one step, and, $\Gamma^{\uparrow\alpha}(I_\emptyset)(\pi)^+ \subseteq \Gamma^{\uparrow\alpha}(\mathbf{N})(\pi)^+$, after α steps (where α is a limit ordinal), for any path π. By Lemma E.2, we also have: $\Gamma(\mathbf{N}) \preceq \mathbf{N}$, $\Gamma^{\uparrow 2}(\mathbf{N}) \preceq \Gamma(\mathbf{N})$, and $\Gamma^{\uparrow\alpha}(\mathbf{N}) \preceq \Gamma^{\uparrow\alpha-1}(\mathbf{N})$ after applying Γ a number of α times. Therefore, $\Gamma^{\uparrow\alpha}(I_\emptyset)(\pi)^+ \subseteq \mathbf{N}(\pi)^+$, for any path π. Hence, $WFM(\mathbf{P}, AT)(\pi)^+ \subseteq \mathbf{N}(\pi)^+$.

We will now prove by contradiction that $\mathbf{N}(\pi)^- \subseteq WFM(\mathbf{P}, AT)(\pi)^-$. Suppose $\mathbf{N}(\pi)(A) = \mathbf{f}$ and $WFM(\mathbf{P}, AT)(\pi)(A) > \mathbf{f}$, for some path π and a literal A. For any clause $@r\ B\ :-\ L_1 \otimes \ldots \otimes L_m$ in $(\mathbf{P}, AT)$, ground substitution θ such that $A = B\theta$, and any split $\pi = \langle \mathbf{D}_0 \rangle \circ \pi_1 \circ \ldots \circ \pi_m$, either $\mathbf{N}(\pi_i)(L_i\theta) = \mathbf{f}$ or $\mathbf{N}(\langle \mathbf{D}_0 \rangle)(\texttt{not}\,(\Diamond\ \ \texttt{\$defeated}(\texttt{handle}(r, B\ \theta)))) = \mathbf{f}$ for some i (where $i \leq m$). If $L_i\theta$ is a `not`-literal `not` C, then $\mathbf{N}(\pi_i)(C) = \mathbf{t}$, and thus, $WFM(\mathbf{P}, AT)(\pi_i)(C) = \mathbf{t}$, by the first part of this proof. Therefore, $WFM(\mathbf{P}, AT)(\pi_i)(L_i\theta) = \mathbf{f}$. If $L_i\theta$ is an atom, then $\mathbf{N}(\pi_i)(L_i\theta) = \mathbf{f}$. We will use the following property to show that $WFM(\mathbf{P}, AT)(\pi_i)(L_i\theta)$ must also be false.

Property E.1: a set $\mathbf{S}$ of atom/path pairs is $\mathbf{N}$-unsupported (analogous to unfounded sets in [VRS91]) if for every pair L/π in $\mathbf{S}$, $\mathbf{N}(\pi)(L) = \mathbf{f}$ and for every rule in $(\mathbf{P}, AT)$ that has L in the head, $@r'\ L\ :-\ L'_1 \otimes \ldots \otimes L'_k$, has a split of $\pi = \langle \mathbf{D}_0 \rangle \circ \pi'_1 \circ \ldots \circ \pi'_k$, so that for some body atom L'_i, the corresponding pair L'_i/π'_i also belongs to $\mathbf{S}$. It can be shown by induction that in every iteration of Γ all the pairs L/π in $\mathbf{S}$ are such that $\Gamma^{\uparrow n}(\pi)(L) = \mathbf{f}$.

By Property E.1, if $L_i\theta$ is an atom, then $WFM(\mathbf{P}, AT)(\pi_i)(L_i\theta) = \mathbf{f}$. Therefore, $WFM(\mathbf{P}, AT)(\pi)(L_1\theta \otimes \ldots \otimes L_m\theta) = \mathbf{f}$. Hence, $WFM(\mathbf{P}, AT)(\pi)(B\theta) = \mathbf{f}$. However, this is impossible since we started the proof by assuming $WFM(\mathbf{P}, AT)(\pi)(A) > \mathbf{f}$.
□

Appendix F

$\mathcal{TR}^{DA}$ Reduction to Transaction Logic

In this appendix, we prove that the well-founded model of any $\mathcal{TR}^{DA}$ program is identical with the well-founded model of a $\mathcal{TR}$ program obtained via a transformation of the original $\mathcal{TR}^{DA}$ program. Suppose a $\mathcal{TR}^{DA}$ program $(\mathbf{P}, AT)$ where $\mathbf{P}$ is a set of labeled $\mathcal{TR}^{DA}$ rules and AT is an argumentation theory.

Theorem 4.6 ($\mathcal{TR}^{DA}$ Reduction)

$WFM(\mathbf{P}, AT)$ coincides with the well - founded model of the $\mathcal{TR}$ program $\mathbf{P}' \cup AT$, where $\mathbf{P}'$ is obtained from $\mathbf{P}$ by changing every defeasible rule

$$@\texttt{r L :- Body} \tag{39}$$

in $\mathbf{P}$ to the plain rule

$$\texttt{L :- } \textit{\textbf{not}}\,(\Diamond\ \texttt{\$defeated(handle(r,L))}\ ) \otimes \texttt{Body} \tag{40}$$

and removing all the remaining tags.

Proof:

We will prove that the programs resulted after each quotient operation in the transfinite sequence during the computation of the well-founded model are the same

for both the original $\mathcal{TR}^{DA}$ program $\mathbf{P}$ and the transformed program $\mathbf{P}'$.

We split the first step of the Definition 4.25 into two steps:

Step 1a. Each occurrence of every `not`-literal of the form $\texttt{not}\, L$ in the bodies of the rules of $\mathbf{P}$, except the `$defeated`/1 literals, is replaced by $\mathbf{t}^\pi$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\, L) = \mathbf{t}$ and with $\mathbf{u}^\pi$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\, L) = \mathbf{u}$. This step applies identically both to 39 and to 40.

Step 1b. The literals in 40 of the form $\texttt{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L))\ )$ are replaced by $\mathbf{t}^\pi$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\,\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L))) = \mathbf{t}$ and with $\mathbf{u}^\pi$ for every path π such that $\boldsymbol{I}(\pi)(\texttt{not}\,\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L))) = \mathbf{u}$. Step 1b is precisely what Step 2 does to 39 except that instead of replacing the literals $\texttt{not}\,(\Diamond\ \texttt{\$defeated}(\texttt{handle}(r, L))\ )$ (which 39 does not have) Step 2 simply adds the appropriate $\mathbf{t}^\pi$ and $\mathbf{u}^\pi$.

□