Transaction Logic with Defaults and Argumentation Theories *

Paul Fodor and Michael Kifer

Stony Brook University
Stony Brook, NY 11794, USA
pfodor,kifer@cs.stonybrook.edu

Abstract

Transaction Logic is an extension of classical logic that gracefully integrates both declarative and procedural knowledge and has proved itself as a powerful formalism for many advanced applications, including modeling robot movements, actions specification, and planning in artificial intelligence. In a parallel development, much work has been devoted to various theories of defeasible reasoning. In this paper, we unify these two streams of research and develop Transaction Logic with Defaults and Argumentation Theories, an extension of both Transaction Logic and the recently proposed unifying framework for defeasible reasoning called Logic Programs with Defaults and Argumentation Theories. We show that this combination has a number of interesting applications, including specification of defaults in action theories and heuristics for directed search in artificial intelligence planning problems. We also demonstrate the usefulness of the approach by experimenting with a prototype of the logic and showing how heuristics expressed as defeasible actions can significantly reduce the search space as well as execution time and space requirements.

1998 ACM Subject Classification "I.2.3 Artificial Intelligence. Deduction and Theorem Proving and Knowledge Processing. D.3.3 Programming Languages. Language Constructs and Features frameworks"

Keywords and phrases Transaction Logic, Defeasible reasoning, Well-founded models

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Transaction logic (abbr., $\mathcal{TR}$) [5, 2] is a general logic for representing knowledge base dynamics. Its model and proof theories cleanly integrate declarative and procedural knowledge and the logic has been employed in domains ranging from reasoning about actions [3], to knowledge representation [1], AI planning [5], workflow management and Web services [14], and general knowledge base programming [4]. Defeasible reasoning is another important paradigm, which has been extensively studied in knowledge representation, policy specification, regulations, law, learning, and more [6, 11, 12].

In this paper we propose to combine $\mathcal{TR}$ with defeasible reasoning and show that the resulting logic language has many important applications. This new logic is called *Transaction Logic with Defaults and Argumentation Theories* (or $\mathcal{TR}^{DA}$) because it extends $\mathcal{TR}$ in the direction of the recently proposed unifying framework for defeasible reasoning called *logic programming with defaults and argumentation theories* (LPDA) [17]. Along the way we define a well-founded semantics [16] for $\mathcal{TR}$, which, to the best of our knowledge, has never been done before.

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 search-intensive applications such as planning. We also demonstrate the usefulness of the approach by experimenting with a

* This work is part of the SILK (Semantic Inference on Large Knowledge) effort within Project Halo, sponsored by Vulcan Inc. It was also partially supported by the NSF grant 0964196.

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–11

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

prototype of $\mathcal{TR}^{DA}$ and showing that heuristics expressed as defeasible actions can drastically prune the search space together with the execution time and space requirements.

This paper is organized as follows. Section 2 motivates reasoning with defaults in $\mathcal{TR}$ with an example. Section 3 provides background on Transaction Logic to make the paper self-contained. Section 4 extends $\mathcal{TR}$ by incorporating defeasible reasoning. Section 5 specializes the logic developed in Section 4 by defining a useful argumentation theory that extends Generalized Courteous Logic Programs (GCLP) [12] and Section 7 summarizes the paper and outlines future work.

2 Motivating Example

In this section, we give an example that illustrates the advantages of extending Transaction Logic with defeasible reasoning.

The syntax of $\mathcal{TR}^{DA}$ is similar to that of standard logic programming except for the fact that literals in the rule bodies are connected via the *serial conjunction*, $\otimes$, which specifies an order of action execution. For instance, $pickup(block1) \otimes puton(block1, block2)$ says that the action $pickup(block1)$ is to be executed first and the action $puton(block1, block2)$ second. The set of predicate symbols of the program is partitioned into:

- a set of *fluents*, which are facts stored in database states or derived propositions that do not change the state of the database; and
- a set of *actions*, which represent actions that change those states.

In addition to the user defined predicate symbols, there are built-in actions called *elementary transitions* for basic manipulation of states. These include *delete(f)* and *insert(f)* for every ground fluent f. Examples of such elementary transitions include $delete(on(block1, block0)$ and $insert(clear(block0)$.

As usual in defeasible reasoning, rules in $\mathcal{TR}^{DA}$ can be *tagged* with terms. For instance, the *move* rule in the example below is tagged with the term $mv_rule(Block, To)$. The predicate `!opposes` is used to specify that some rules are incompatible with others. The predicate `!overrides` specifies that some actions have higher priority than other actions.

Following the standard convention in Logic Programming, we will be using alphanumeric symbols that begin with an uppercase letter to denote variables. Alphanumeric symbols that begin with lowercase letters will denote constant, function, and predicate symbols.

▶ Example 2.1 (Block world planning). This example illustrates the use of defeasible reasoning for heuristic optimization of planning in the blocks world. 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 would tend to be constructed.[1]

In this example, we represent the blocks world using the fluents $on(x, y)$, which say that block x is on top of block y; $isclear(x)$, which says that nothing is on top of block x; and $larger(x, y)$, which says that the size of x is larger than the size of y. The action $pickup(X, Y)$ lifts the block X from the top of block Y 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 $move(X, From, To)$, specified by the first rule, 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 aforementioned actions $pickup$ and $putdown$, if certain preconditions are satisfied. The stacking action (not included in the program) 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 *move_action* might be applicable and several different moves might be performed. The predicate

[1] For more information on planning with $\mathcal{TR}$ see [5].

!**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).

```
@mv_rule(Block,To) move(Block,From,To) :-
          (on(Block,From) ∧ larger(To,Block)) ⊗
          pickup(Block,From) ⊗ putdown(Block,To).
pickup(X,Y) :- (isclear(X) ∧ on(X,Y)) ⊗
          delete(on(X,Y)) ⊗ insert(isclear(Y)).
putdown(X,table) :- (isclear(X) ∧ not on(X,Z))
          ⊗ insert(on(X,table)).
putdown(X,Y) :- (isclear(X) ∧ isclear(Y) ∧ not on(X,Z))
          ⊗ delete(isclear(Y)) ⊗ insert(on(X,Y)).
!opposes(move(B1,F1,T1),move(B2,F2,T2)) :- B1 ≠ B2.
```

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 to the table surface.

```
!overrides(mv_rule(B2,To), mv_rule(B1,To)) :- larger(B2,B1) ∧ To ≠ table.
```

Consider the following configuration of blocks:

```
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).
```

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:

```
!overrides(mv_rule(B2,table), mv_rule(B1,table)) :- larger(B1,B2).
```

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:

```
stack(0,Block).
stack(N,X) :- N>0 ⊗ move(Y,_,X) ⊗ stack(N-1,Y) ⊗ on(Y,X).
stack(N,X) :- (N>0 ∧ on(Y,X)) ⊗ unstack(Y) ⊗ stack(N,X).
unstack(X) :- on(Y,X) ⊗ unstack(Y) ⊗ unstack(X).
unstack(X) :- isclear(X) ∧ on(X,table).
unstack(X) :- (isclear(X) ∧ on(X,Y) ∧ Y≠ table) ⊗ move(X,_,table).
unstack(X) :- on(Y,X) ⊗ unstack(Y) ⊗ unstack(X).
```

Running this program by the interpreter described in [9] shows that the above preferences drastically reduce the number of plans that need to be considered—sometimes to just one plan. These experiments are described in Section 6. □

3 Serial-Horn Transaction Logic

In this section we describe a subset of Transaction Logic called serial-Horn $\mathcal{TR}$. This subset has been shown to be sufficiently expressive for many applications, including planning, workflow management, and action languages [5].

The syntax of $\mathcal{TR}$ is derived from that of standard logic programming. 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 may be inserted or deleted. 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* depending on whether p is a fluent or an action symbol. Furthermore, if p is a derived or base fluent symbol then $p(t_1, ..., t_n)$ is said to be a derived or base fluent atomic formula. An expression is *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 negated forms: `neg` α, `not` α, `not neg` α, where α is a fluent atomic formula. An *action literal* is an action atomic formula or has the form `not` α, where α is a action atomic formula. Literals of the form `neg` α, where α is an action, are not allowed. Atoms of the form `neg not` $alpha$ are also not allowed.

A *database state* is a set of ground base fluents. All database states are assumed to be *consistent*, meaning that they cannot have both f and `neg` f, 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 elementary transitions and fluents. In this paper, 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 `neg` g, where g is a ground base fluent. For any given database state $\mathbf{D}$,

- $insert(f)$ 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 and quantifiers, $\mathcal{TR}$ has new logical connectives:

- $\otimes$ - the sequential conjunction
- $\Diamond$ - the modal operator of hypothetical execution

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 actual state changes takes place. In procedural terms, executing $delete(on(blk1, table)) \otimes insert(on(blk1, blk2))$ means "first delete $on(blk1, table)$ from the database, and then insert $on(blk1, blk2)$." 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 regardless of 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 also explains our use of $\wedge$ in Example 2.1 where it could have been replaced with $\otimes$ without changing the meaning (but, the uses of $\otimes$ in the Example 2.1 *cannot* be replaced with $\wedge$ without changing the meaning).

▶ Definition 3.1 (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 $P_1 \wedge P_2$. □

▶ Definition 3.2 (Serial rules). A **serial rule** is an expression of the form: $H :- B.$ 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 of the form f or a fluent literal of the form $\texttt{neg}\ f$ 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. □

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: $\mathbf{P}, \mathbf{D_0}, \ldots \mathbf{D_n} \models \psi$. Further details on $\mathcal{TR}$ can be found in [5] and [2].

4 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 [17]. Language-wise, the only difference between $\mathcal{TR}^{DA}$ and serial $\mathcal{TR}$ is that the rules in $\mathcal{TR}^{DA}$ are tagged.

▶ Definition 4.1 (Tagged rules). A **tagged rule** in the language $\mathcal{TR}^{DA}$ is an expression of the form: $@r\ H\ :-\ B.$ 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 3.2.

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

▶ Definition 4.2 (Transaction formula). A **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 *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 (in this paper, we consider strict rules to be non-tagged rules, as in Definition 3.2).

▶ Definition 4.3 (Rule handle). Given a tagged rule, the term $\texttt{handle}(r, H)$ 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 4.4 (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 is usually done using special predicates defined in $\mathcal{TR}^{DA}$, such as **!opposes** and **!overrides** used in our example. 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 4.4 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 paper.

▶ Definition 4.5 (Herbrand universe and base). The **Herbrand universe** of $\mathcal{TR}^{DA}$, denoted $\mathcal{U}$, is the set of all ground terms built using the constants and function symbols of the language of $\mathcal{TR}^{DA}$. The **Herbrand base**, denoted $\mathcal{B}$, is the set of all ground `not` -free literals that can be constructed using the language of $\mathcal{TR}^{DA}$. □

The key concept underlying the semantics of $\mathcal{TR}$ and $\mathcal{TR}^{DA}$ is that of *execution paths*, which are sequences of database states. The truth assignment in $\mathcal{TR}$ is done using *path structures*, which are mappings from paths of states.

▶ Definition 4.6 (Path and Split). A **path** of length k, or a *k-path*, is a finite sequence of states, $\pi = \langle \mathbf{D}_1 \ ... \ \mathbf{D}_k \rangle$, where $k \geq 1$. A **split** of π is any pair of subpaths, 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 π_1, π_2 then we write $\pi = \pi_1 \circ \pi_2$. □

We extend the well-founded semantics for logic programing [16] to $\mathcal{TR}^{DA}$ using a definition in the style of [13]. In the following, we use the usual three truth values **t**, **f**, and **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 4.7 (Partial Herbrand interpretation). A **partial Herbrand interpretation** is a mapping $\mathcal{H}$ that assigns **f**, **u**, or **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 $\mathcal{H}(L) = \mathcal{H}(\mathtt{neg}\, L) = \mathbf{t}$. $\mathcal{H}$ is ***consistent*** if it is consistent relative to every formula. $\mathcal{H}$ is ***total*** if, for every ground `not` -free formula L (other than **u**), either $\mathcal{H}(L) = \mathbf{t}$ and $\mathcal{H}(\mathtt{neg}\, L) = \mathbf{f}$ or $\mathcal{H}(L) = \mathbf{f}$ and $\mathcal{H}(\mathtt{neg}\, L) = \mathbf{t}$. □

Partial Herbrand interpretations are used to define *path structures*, which tell which ground atoms (fluents or actions) are true on what paths. Path structures play the same role in $\mathcal{TR}^{DA}$ as that 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 4.8 (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. For every ground base fluent-literal d and every database state $\mathbf{D}$:
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{t}$, if $d \in \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{f}$, if $d \notin \mathbf{D}$;
   $\boldsymbol{I}(\langle \mathbf{D} \rangle)(d) = \mathbf{u}$, otherwise
2. $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(insert(p)) = \mathbf{t}$ *if* $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p\} \setminus \{\mathtt{neg}\, p\}$ 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* $\mathbf{D}_2 = \mathbf{D}_1 \setminus \{p\} \cup \{\mathtt{neg}\, p\}$ and $\mathbf{P}$ is a ground fluent-literal;
   $\boldsymbol{I}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle)(delete(p)) = \mathbf{f}$, otherwise. □

Without loss of generality, while 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 the set of all of their ground instantiations.

We assume that the language includes the 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 **u** over π and is false on all other paths.

▶ Definition 4.9 (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 $\boldsymbol{I}(\pi)(\phi \otimes \psi) = \mathtt{min}(\boldsymbol{I}(\pi_1)(p), \boldsymbol{I}(\pi_2)(q))$.
- If ϕ and ψ are serial goals then $\boldsymbol{I}(\pi)(\phi \wedge \psi) = \mathtt{min}(\boldsymbol{I}(\pi)(p), \boldsymbol{I}(\pi)(q))$.
- If ϕ is a serial goal then $\boldsymbol{I}(\pi)(\mathtt{not}\ \phi) = \sim \boldsymbol{I}(\pi)(\phi)$, 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
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \texttt{max}\{\boldsymbol{I}(\pi')(\phi) \mid \pi'$ is a path that starts at $\mathbf{D}\}$
  $\boldsymbol{I}(\pi)(\Diamond\phi) = \mathbf{f}$, otherwise.
- For a strict serial rule $F \texttt{ :- } G$,
  $\boldsymbol{I}(\pi)(F \texttt{ :- } G) = \mathbf{t}$ iff $\boldsymbol{I}(\pi)(F) \geq \boldsymbol{I}(\pi)(G)$.
- For a defeasible rule $@r\ F \texttt{ :- } G$,
  $\boldsymbol{I}(\pi)(\texttt{@r F :- G}) = \mathbf{t}$ iff
  $\boldsymbol{I}(\pi)(F) \geq \texttt{min}\,(\boldsymbol{I}(\pi)(G),\ \boldsymbol{I}(\langle D_0 \rangle)(\texttt{not } \Diamond \texttt{ \$defeated(handle}(r, F))))$,
  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 $\boldsymbol{I}(\pi)(\phi)$=**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 **t** or **f**. □

▶ Definition 4.10 (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 $\boldsymbol{I}$ 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 4.11 (Model of $\mathcal{TR}^{DA}$). A path structure $\boldsymbol{I}$ is a model of a $\mathcal{TR}^{DA}$ transaction base $\mathbf{P}$ if all rules in $\mathbf{P}$ are satisfied in $\boldsymbol{I}$ (i.e., $\boldsymbol{I} \models R$ 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-Horn $\mathcal{TR}$ can be formulated as a fixpoint theory [3]. In classical logic programming, given two Herbrand partial interpretations σ_1 and σ_2, we write $\sigma_1 \preceq \sigma_2$ 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, for partial interpretations, $\sigma_1 \leq \sigma_2$ 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 4.12 (Order on Path Structures). If $\mathbf{M}_1$ and $\mathbf{M}_2$ are partial 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. Similarly, we write $\mathbf{M}_1 \leq \mathbf{M}_2$ 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 $\mathbf{P}$, we have that $\mathbf{N} \preceq \mathbf{M}$ implies $\mathbf{N} = \mathbf{M}$. The **least** model of $\mathbf{P}$ is a minimal model that is unique (if it exists). □

It is well-known that in ordinary logic programming any set of Horn rules always has a least model. In [5], it is shown that every positive serial-Horn $\mathcal{TR}$ program has a unique least total model. Theorem 1, 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 rules.

▶ **Theorem 1** (Unique Least Partial Model for serial `not` -free $\mathcal{TR}$ programs)**.** *If* $\mathbf{P}$ *is a* `not` *-free* $\mathcal{TR}^{DA}$ *program, then* $\mathbf{P}$ *has a least Herbrand model, denoted* $LPM(\mathbf{P})$.[2] □

Next we define well-founded models for $\mathcal{TR}^{DA}$ by adapting the definition from [13]. 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 $\frac{\mathbf{P}}{\boldsymbol{I}}$. Despite what one might have been expecting, this adaptation is rather subtle.

[2] All proofs can be found in our technical report [10].

▶ Definition 4.13 (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 P by** $\boldsymbol{I}$, written as $\frac{\mathbf{P}}{\boldsymbol{I}}$, 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 $\boldsymbol{I}(\pi)(\texttt{not}\ L) = \mathbf{t}$ 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 $@r\ L\ \texttt{:-}\ Body$ obtained in the previous step, replace it with rules of the form: $L\ \texttt{:-}\ \mathbf{t}^{\langle D_t\rangle}\ \otimes\ Body$. for each database state D_t such that $\boldsymbol{I}(\langle D_t\rangle)(\texttt{not}\ (\Diamond\ \texttt{\$defeated(handle}(r, L)\texttt{)})) = \mathbf{t}$, and rules of the form $L\ \texttt{:-}\ \mathbf{u}^{\langle D_u\rangle} \otimes Body$ for each database state D_u such that $\boldsymbol{I}(\langle D_u\rangle)(\texttt{not}\ (\Diamond\ \texttt{\$defeated(handle}(r, L)\texttt{)})) = \mathbf{u}$.
3. Remove the labels from the remaining rules. The resulting set of rules is the quotient $\frac{\mathbf{P}}{\boldsymbol{I}}$. □

Note that in Step 1 of the above definition of the quotient each occurrence of `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 $\mathbf{I}$, $\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}$, is a negation-free $\mathcal{TR}$ program, so, by Theorem 1, 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. 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* `not` *-free literal*$\}$, $\boldsymbol{I}^{-} = \{L \mid\ L \in \boldsymbol{I}$ *is a* `not` *-literal*$\}$ and $\boldsymbol{I} = \boldsymbol{I}^{+} \cup \boldsymbol{I}^{-}$.

▶ **Definition 2** ($\mathcal{TR}^{DA}$ 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: $\Gamma(\boldsymbol{I}) =^{def} LPM\left(\frac{\mathbf{P} \cup AT}{\boldsymbol{I}}\right)$

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 [10]). 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.

▶ **Definition 3** (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 $\{\Gamma^{\uparrow n}(I_{\emptyset})\}$. □

▶ **Theorem 4** (Correctness of the Constructive $\mathcal{TR}^{DA}$ Least Model)**.** $WFM(\mathbf{P}, AT)$ *is the least model of the program* $(\mathbf{P}, AT)$.

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 5** ($\mathcal{TR}^{DA}$ 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* $(@\texttt{r}\ \texttt{L :- Body}) \in \mathbf{P}$ *to the plain rule* `L :- not (` $\Diamond$ `$defeated(handle(r,L)) )` $\otimes$ `Body` *and removing all the remaining tags.*

5 The $GCLP^{\mathcal{TR}}$ Argumentation Theory

We present here a particularly interesting argumentation theory which extends GCLP—*generalized courteous logic programs* [12]—to $\mathcal{TR}$ under the $\mathcal{TR}^{DA}$ framework. The inferences claimed in the discussion of the planning example in Section 2 assumed that particular argumentation theory. We will call this argumentation theory $GCLP^{\mathcal{TR}}$. As any argumentation theory in our framework, $GCLP^{\mathcal{TR}}$ defines a version of the predicate `$defeated` using various auxiliary concepts. We define these concepts first.

The user-defined predicates `!opposes` and `!overrides` are relations specified over rule handles. They tell the system what rule instances are in conflict with each and which rule instances are preferred over other rules.

The predicate `$defeated` is defined indirectly in terms of the predicates `!opposes` and `!overrides`. In the following definitions the variables R and S are assumed to range over rule handles, while the implicit current state identifier D is assumed to range over the possible database states. A rule is *defeated* if it is *refuted* or *rebutted* by some other rule, assuming that the first rule is defeasible and the second rule is not *compromised* or *disqualified*. We will define these notions shortly, but first we explain them informally. A rule is *refuted* if a higher-priority rule implies a conclusion that is incompatible with the conclusion implied by the refuted rule, A rule *rebuts* another rule if the two rules assert conflicting conclusions and there is no way to resolve the conflict. A rule is *compromised* if it is defeated by some other rule, and a rule *disqualified* if that rule defeats itself.

$$\begin{array}{lll}
\texttt{\$defeated}(R) & \texttt{:-} & \texttt{\$refutes}(S,R) \wedge \texttt{not \$compromised}(S). \\
\texttt{\$defeated}(R) & \texttt{:-} & \texttt{\$rebuts}(S,R) \wedge \texttt{not \$compromised}(S). \\
\texttt{\$defeated}(R) & \texttt{:-} & \texttt{\$disqualified}(R). \\
\texttt{\$refutes}(R,S) & \texttt{:-} & \texttt{\$conflict}(R,S) \wedge \texttt{!}\mathbf{\texttt{overrides}}(R,S). \\
\texttt{\$conflict}(R,S) & \texttt{:-} & \texttt{\$candidate}(R), \texttt{\$candidate}(S), \texttt{!}\mathbf{\texttt{opposes}}(R,S).
\end{array} \tag{1}$$

A rule R *rebuts* another rule S if the two rules assert conflicting conclusions, but neither rule is "more important" than the other, *i.e.*, no preference can be inferred between the two rules. This intuition can be expressed in several different ways, but we selected the one below, which mimics the definition in [17].

$$\begin{array}{lll}
\texttt{\$rebuts}(R,S) & \texttt{:-} & \texttt{\$candidate}(R) \wedge \texttt{\$candidate}(S) \wedge \\
 & & \texttt{!}\mathbf{\texttt{opposes}}(R,S) \wedge \texttt{not \$compromised}(R) \wedge \\
 & & \texttt{not \$refutes}(_,R) \wedge \texttt{not \$refutes}(_,S).
\end{array} \tag{2}$$

The important difference here compared to [17] is that we are dealing with state-changing actions and so all tests for refutation, rebuttal, and the like, must be hypothetical. This is reflected in the definition of a rule candidate. We say that a rule instance is a *candidate* if its body is *hypothetically* true in the current database state. The other two rules in the group below specify the symmetry of `!opposes` and the fact that literals H and `neg` H are in conflict with each other.

$$\texttt{\$candidate}(R) \texttt{ :- } \mathbf{\texttt{body}}(R,B) \otimes \Diamond\mathbf{\texttt{call}}(B). \tag{3}$$

$$\texttt{!}\mathbf{\texttt{opposes}}(X,Y) \texttt{ :- !}\mathbf{\texttt{opposes}}(Y,X). \tag{4}$$

$$\texttt{!}\mathbf{\texttt{opposes}}(\texttt{handle}(\_,H), \texttt{handle}(\_, \texttt{neg } H)). \tag{5}$$

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

$$\begin{array}{l}
\texttt{\$compromised}(R) \texttt{ :- \$refuted}(R) \wedge \texttt{\$defeated}(R). \\
\texttt{\$disqualified}(X) \texttt{ :- \$defeats}_{tc}(X,X). \\
\texttt{\$defeats}_{tc}(X,Y) \texttt{ :- \$defeats}(X,Y). \\
\texttt{\$defeats}_{tc}(X,Y) \texttt{ :- \$defeats}_{tc}(X,Z) \wedge \texttt{\$defeats}(Z,Y).
\end{array} \tag{6}$$

As in [17], 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 the main focus of the present paper.

6 Implementation, evaluation and related work

We implemented an interpreter for $\mathcal{TR}^{DA}$ in XSB [3] and tested it on a number of examples, including Example 2.1. 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. Table 1 shows how the preferential heuristics of Example 2.1 fare in our tests. We can see that the number of plans being searched decreases dramatically and so does the time and space. However, the time spent on generation of all those plans is not proportional to their number because our implementation takes advantage of sharing of partially constructed plans among the different searches due to tabling [9] even without the heuristics.

World size		No heuristics	Preferential heuristics
30 blocks	Plans	4060	28
	Time(sec.)	2.390	0.438
	Space(kBs)	3730	90
40 blocks	Plans	9880	38
	Time(sec.)	7.000	1.219
	Space(kBs)	8562	120
50 blocks	Plans	19600	48
	Time(sec.)	17.109	2.938
	Space(kBs)	16347	150

Table 1 Planing in blocks world with and without preferential heuristics

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 $\mathcal{TR}$. As far as the actual chosen approach to defeasible reasoning is concerned, this work is based on [17], and extensive comparison with other works on defeasible reasoning can be found there. Although our work is *not* about planning but rather about a general language for declarative programming with defeasible actions, the closest works that we can possibly compare with are the works on planning with preferences. $\mathcal{TR}^{DA}$ is quite different from [15] in that it is a full-fledged logic that combines both declarative and procedural elements, while [15] is geared towards specifying preferences over planning *solutions*.Whereas $\mathcal{TR}^{DA}$ deals with infinite domains and allows function symbols, the approach in [15] 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. Both the *temporal* and the *choice* preferences presented in [7]can be expressed in the $\mathcal{TR}^{DA}$ framework, although due to the difference in the semantics the exact relationship needs further study. The framework [8] for planning with cost preferences assigns a numeric cost to each action 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.

7 Conclusions

This paper proposes a theory of defeasible reasoning in Transaction Logic, an extension of classical logic for representing both declarative and procedural knowledge. This new logic, called $\mathcal{TR}^{DA}$, extends our prior work on defeasible reasoning with argumentation theories from static logic programming to a logic that captures the dynamics in knowledge representation. We also extend the Courteous style of defeasible reasoning [12] to incorporate actions, planning, and other dynamic aspects of knowledge representation. We believe that $\mathcal{TR}^{DA}$ can become a rich platform for expressing heuristics about actions. The paper also makes a contribution directly to the development of Transaction Logic itself by defining the well-founded semantics for it and for its $\mathcal{TR}^{DA}$ extension—a non-trivial adaptation of the classical well-founded semantics of [16].

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

References

1 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.

2 A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, Berlin, March 1998.

3 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.

4 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.

5 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`.

6 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.

7 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.

8 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.

9 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.

10 Paul Fodor and Michael Kifer. Transaction Logic with Defaults and Argumentation Theories. Technical report, Stony Brook University, May 2011. Available from `http://ewl.cewit.stonybrook.edu/trda.pdf`.

11 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.

12 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.

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

14 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.

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

16 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.

17 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.