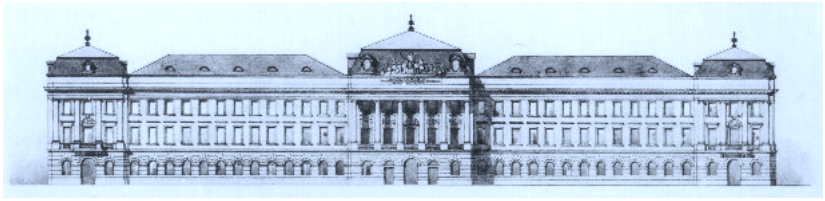
INFSYS RESEARCH REPORT



INSTITUT FÜR INFORMATIONSSYSTEME
ARBEITSBEREICH WISSENSBASIERTE SYSTEME

MANAGING INCONSISTENCY IN MULTI-CONTEXT SYSTEMS USING THE IMPL POLICY LANGUAGE

THOMAS EITER MICHAEL FINK
GIOVAMBATTISTA IANNI PETER SCHÜLLER

INFSYS RESEARCH REPORT 1843-12-05
APRIL 2012

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TU WIEN

kbs Knowledge-Based Systems Group

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-12-05, APRIL 2012

MANAGING INCONSISTENCY IN MULTI-CONTEXT SYSTEMS USING THE IMPL POLICY LANGUAGE

Thomas Eiter[1] Michael Fink[1] Giovambattista Ianni[2] Peter Schüller[1]

Abstract. Multi-context systems are a formalism for interlinking knowledge-based system (contexts) which interact via (possibly nonmonotonic) bridge rules. Such interlinking provides ample opportunity for unexpected inconsistencies. These are undesired and come in different categories: some are serious and must be inspected by a human operator, while some should simply be repaired automatically. However, no one-fits-all solution exists, as these categories depend on the application scenario. To tackle inconsistencies in a general way, we therefore propose a declarative policy language for inconsistency management in multi-context systems. We define syntax and semantics, discuss methodologies for applying the language in real world applications, and describe an implementation by rewriting to the ACTHEX formalism which is an extension of Answer Set Programs.

[1]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: {eiter,fink,schueller}@kr.tuwien.ac.at.

[2]Dipartimento di Matematica, Cubo 30B, Università della Calabria, 87036 Rende (CS), Italy; email: ianni@mat.unical.it.

Acknowledgements: This research has been supported by the Vienna Science and Technology Fund (WWTF) project ICT08-020. This report is an extended version of [Eiter et al., 2012].

Contents

1 Introduction

Powerful knowledge based applications can be built by interlinking smaller existing knowledge based systems. Multi-context systems (MCSs) [Brewka and Eiter, 2007], based on [Giunchiglia and Serafini, 1994, Brewka et al., 2007], are a generic formalism that captures heterogeneous knowledge bases (contexts) which are interlinked using (possibly nonmonotonic) bridge rules.

However, the advantage of building a system from smaller parts poses the challenge of unexpected inconsistencies due to unintended interaction of system parts. Such inconsistencies are undesired, as (under common principles) inference becomes trivial. Explaining reasons for inconsistency in MCSs has been investigated in [Eiter et al., 2010a]: several independent inconsistencies can exist in a MCS, and each inconsistency usually comes with more than one possibility to repair it.

For example, imagine a hospital information system which links several databases and suggests treatments for patients. A simple inconsistency which can be automatically ignored would be if a patient enters her birth date correctly at the front desk, but swaps two digits filling in a form at the X-ray department. An entirely different story is, if we have a patient who needs treatment, but all options are in conflict with some allergy of the patient. Attempting an automatic repair may not be a viable option in this case: a doctor should inspect the situation and make a decision.

In the light of such scenarios, tackling inconsistency requires individual strategies and targeted (re)actions, depending on the type of inconsistency and on the application. We thus propose the declarative *Inconsistency Management Policy Language* (IMPL), which provides a means to specify inconsistency management strategies for MCSs. Our contributions are as follows.

- We define the syntax of IMPL, inspired by Answer Set Programming (ASP) [Gelfond and Lifschitz, 1991]. In particular, we specify *input for policy reasoning*, in terms of reserved predicates. These predicates encode inconsistency analysis results as introduced in [Eiter et al., 2010a]. Furthermore, we specify *action predicates that can be derived by rules*. Actions provide a means to counteract inconsistency by modifying the MCS, and may involve interaction with a human operator.

- We define the semantics of IMPL in a three-step process which calculates models of a policy, then determines effects of actions which are present in such a model (this possibly involves user interaction), and finally applies these effects to the MCS.

- We provide methodologies for integrating IMPL into application scenarios, and discuss possible modes of reasoning and language extensions that could be useful in practical applications.

- We identify a fragment of IMPL, called *Core* IMPL , which is sufficient for realizing functionality of the full IMPL language. We give a rewriting from Core IMPL to the acthex formalism [Basol et al., 2010] which extends Answer Set Programs with external computations and actions.

- Finally we provide a method of rewriting IMPL to the Core IMPL fragment. This allows for using the acthex rewriting as an implementation for the full IMPL language.

The paper is organized as follows: first we introduce MCS and notions for explaining inconsistency in MCSs in Section 2, we define syntax and semantics of the IMPL policy language in Section 3, describe methodologies for applying IMPL in practice in Section 4, provide a possibility for realizing Core IMPL by rewriting to acthex in Section 5, give a rewriting from the full IMPL language to Core IMPL in Section 6, and conclude the paper in Section 7.

2 Preliminaries

We first introduce the MCS formalism and then describe notions for analyzing inconsistencies in MCSs.

2.1 Multi-context systems (MCSs)

A heterogeneous nonmonotonic MCS [Brewka and Eiter, 2007] consists of *contexts*, each composed of a knowledge base with an underlying *logic*, and a set of *bridge rules* which control the information flow between contexts.

A logic $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ is an abstraction which captures many monotonic and nonmonotonic logics, e.g., classical logic, description logics, or default logics. It consists of the following components, the first two intuitively define the logic's syntax, the third its semantics:

- $\mathbf{KB}_L$ is the set of well-formed knowledge bases of L. We assume each element of $\mathbf{KB}_L$ is a set of "formulas".
- $\mathbf{BS}_L$ is the set of possible belief sets, where a belief set is a set of "beliefs".
- $\mathbf{ACC}_L : \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ assigns to each KB a set of acceptable belief sets.

Since contexts may have different logics, this allows to model heterogeneous systems.

Example 2.1. *For* propositional logic L_{prop} *under the closed world assumption over signature* Σ, $\mathbf{KB}$ *is the set of propositional formulas over* Σ; $\mathbf{BS}$ *is the set of deductively closed sets of propositional* Σ*-literals; and* $\mathbf{ACC}(kb)$ *returns for each* kb *a singleton set, containing the set of literal consequences of* kb *under the closed world assumption.* □

A *bridge rule* models information flow between contexts: it can add information to a context, depending on the belief sets accepted at other contexts. Let $L = (L_1, \ldots, L_n)$ be a tuple of logics. An L_k-bridge rule r over L is of the form

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

where k and c_i are context identifiers, i.e., integers in the range $1, \ldots, n$, p_i is an element of some belief set of L_{c_i}, and s is a formula of L_k. We denote by $h_b(r)$ the formula s in the head of r and by $B(r) = \{(c_1 : p_1), \ldots, \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots\}$ the set of body literals (including negation) of r.

A multi-context system $M = (C_1, \ldots, C_n)$ is a collection of contexts $C_i = (L_i, kb_i, br_i)$, $1 \leq i \leq n$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ a knowledge base, and br_i is a set of L_i-bridge rules over $(L_1, \ldots, L_n)$. By $IN_i = \{h_b(r) \mid r \in br_i\}$ we denote the set of possible *inputs* of context C_i added by bridge rules. For each $H \subseteq IN_i$ it is required that $kb_i \cup H \in \mathbf{KB}_{L_i}$. We denote by $c(M) = \{C_1, \ldots, C_n\}$ the set of all contexts of M. By $br(M) = \bigcup_{i=1}^{n} br_i$ we denote the set of all bridge rules of M.

The following running example will be used throughout the paper.

Example 2.2 (generalized from [Eiter et al., 2010a])**.** *Consider a MCS* M_1 *in a hospital which comprises the following contexts: a patient database* C_{db}*, a blood and X-Ray analysis database* C_{lab}*, a disease ontology* C_{onto}*, and an expert system* C_{dss} *which suggests proper treatments. Knowledge bases are given below;*

initial uppercase letters are used for variables and description logic concepts.

$$
\begin{aligned}
kb_{db} &= \{person(sue, 02/03/1985), allergy(sue, ab1)\},\\
kb_{lab} &= \{customer(sue, 02/03/1985), test(sue, xray, pneumonia),\\
&\quad test(Id, X, Y) \rightarrow \exists D : customer(Id, D)),\\
&\quad customer(Id, X) \wedge customer(Id, Y) \rightarrow X = Y\},\\
kb_{onto} &= \{Pneumonia \sqcap Marker \sqsubseteq AtypPneumonia\},\\
kb_{dss} &= \{give(Id, ab1) \vee give(Id, ab2) \leftarrow need(Id, ab).\\
&\quad give(Id, ab1) \leftarrow need(Id, ab1).\\
&\quad \neg give(Id, ab1) \leftarrow not\ allow(Id, ab1), need(Id, Med).\}.
\end{aligned}
$$

Context C_{db} uses propositional logic (see Example 2.1) and provides the information that Sue is allergic to antibiotics '$ab1$'. Context C_{lab} is a database with constraints which stores laboratory results connected to Sue: $pneumonia$ was detected in an X-ray. Constraints enforce, that each test result is linked to a $customer$ record, and that each customer has only one birth date. C_{onto} specifies, that presence of a blood marker in combination with pneumonia indicates atypical pneumonia. C_{onto} is based on $\mathcal{AL}$, a basic description logic [Baader et al., 2003]: $\mathbf{KB}_{onto}$ is the set of all well-formed theories within that description logic, $\mathbf{BS}_{onto}$ is the powerset of the set of all assertions $C(o)$ where C is a concept name and o an individual name, and $\mathbf{ACC}_{onto}$ returns the set of all concept assertions entailed by a given theory. C_{dss} is an ASP program that suggests a medication using the $give$ predicate.

Schemas for bridge rules of M_1 are as follows:

$$
\begin{aligned}
r_1 &= (lab : customer(Id, Birthday)) &&\leftarrow (db : person(Id, Birthday)).\\
r_2 &= (onto : Pneumonia(Id)) &&\leftarrow (lab : test(Id, xray, pneumonia)).\\
r_3 &= (onto : Marker(Id)) &&\leftarrow (lab : test(Id, bloodtest, m1)).\\
r_4 &= (dss : need(Id, ab)) &&\leftarrow (onto : Pneumonia(Id)).\\
r_5 &= (dss : need(Id, ab1)) &&\leftarrow (onto : AtypPneumonia(Id)).\\
r_6 &= (dss : allow(Id, ab1)) &&\leftarrow \mathbf{not}\ (db : allergy(Id, ab1)).
\end{aligned}
$$

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

Equilibrium semantics [Brewka and Eiter, 2007] selects certain belief states of a MCS $M = (C_1, \ldots, C_n)$ as acceptable. A *belief state* is a list $S = (S_1, \ldots, S_n)$, s.t. $S_i \in \mathbf{BS}_i$. A bridge rule (1) is *applicable* in S iff for $1 \leq i \leq j$: $p_i \in S_{c_i}$ and for $j < l \leq m$: $p_l \notin S_{c_l}$. Let $app(R, S)$ denote the set of bridge rules in R that are applicable in belief state S. Then a belief state $S = (S_1, \ldots, S_n)$ of M is an *equilibrium* iff, for $1 \leq i \leq n$, the following condition holds: $S_i \in \mathbf{ACC}_i(kb_i \cup \{hd(r) \mid r \in app(br_i, S)\})$.

For simplicity we will disregard the issue of grounding bridge rules (see [Fink et al., 2011]), and only consider ground instances of bridge rules. In the following, with $r_1, \ldots, r_6$ we refer to the ground instances of the respective bridge rules in Example 2.2, where variables are replaced by $Id \mapsto sue$ and $Birthday \mapsto 02/03/1985$ (all other bridge rule instances are irrelevant).

Example 2.3 (ctd). *MCS M_1 has one equilibrium $S = (S_{db}, S_{lab}, S_{onto}, S_{dss})$, where*

$$\begin{aligned} S_{db} &= kb_{db}, \\ S_{lab} &= \{customer(sue, 02/03/1985), test(sue, xray, pneumonia)\}, \\ S_{onto} &= \{Pneumonia(sue)\}, \text{ and} \\ S_{dss} &= \{need(sue, ab), give(sue, ab2), \neg give(sue, ab1)\}. \end{aligned}$$

Moreover, bridge rules r_1, r_2, and r_4 are applicable under S. □

2.2 Explaining Inconsistency in MCSs.

Inconsistency in a MCS is the lack of an equilibrium [Eiter et al., 2010a]. Note that no equilibrium may exist even if all contexts are 'paraconsistent' in the sense that for all $kb \in \mathbf{KB}$, $\mathbf{ACC}(kb)$ is nonempty. No information can be obtained from an inconsistent MCS, e.g., inference tasks like brave or cautious reasoning on equilibria become trivial. To analyze, and eventually repair, inconsistency in a MCS, we use the notions of consistency-based *diagnosis* and entailment-based *inconsistency explanation* [Eiter et al., 2010a], which characterize inconsistency by sets of involved bridge rules.

Intuitively, a diagnosis is a pair (D_1, D_2) of sets of bridge rules which represents a concrete system repair in terms of removing rules D_1 and making rules D_2 unconditional. The intuition for considering rules D_2 as unconditional is that the corresponding rules should become applicable to obtain an equilibrium. One could consider more fine-grained changes of rules such that only some body atoms are removed instead of all. However, this increases the search space while there is little information gain: every diagnosis (D_1, D_2) as above, together with a witnessing equilibrium S, can be refined to such a generalized diagnosis.

The formal definition is as follows. Given a MCS M and a set R of bridge rules, by $M[R]$ we denote the MCS obtained from M by replacing its set of bridge rules $br(M)$ with R (in particular, $M[br(M)] = M$ and $M[\emptyset]$ is M with no bridge rules). By $M \models \bot$ we denote that M is inconsistent, by $M \not\models \bot$ the opposite. For any set of bridge rules A, we denote by $heads(A)$ the rules in A in unconditional form. For pairs $A = (A_1, A_2)$ and $B = (B_1, B_2)$ of sets, $A \subseteq B$ denotes the pointwise subset relation.

Definition 2.4 (Diagnosis [Eiter et al., 2010a]). *Given a MCS M, a* diagnosis *of M is a pair (D_1, D_2), $D_1, D_2 \subseteq br(M)$, s.t. $M[br(M) \setminus D_1 \cup heads(D_2)] \not\models \bot$. $D^{\pm}(M)$ is the set of all such diagnoses. $D^{\pm}_m(M)$ is the set of all pointwise subset-minimal diagnoses of a MCS M.*

Dual to that, inconsistency explanations (short: explanations) separate independent inconsistencies. An explanation is a pair (E_1, E_2) of sets of bridge rules, such that the presence of rules E_1 and the absence of heads of rules E_2 necessarily makes the MCS inconsistent. In other words, bridge rules in E_1 cause an inconsistency in M which cannot be resolved by considering additional rules that are already present in M, or by modifying rules in E_2 (in particular by making them unconditional).

Definition 2.5 (Inconsistency Explanation [Eiter et al., 2010a]). *Given a MCS M, an* inconsistency explanation *of M is a pair (E_1, E_2) s.t. for all (R_1, R_2) where $E_1 \subseteq R_1 \subseteq br(M)$ and $R_2 \subseteq br(M) \setminus E_2$, it holds that $M[R_1 \cup heads(R_2)] \models \bot$. By $E^{\pm}(M)$ we denote the set of all inconsistency explanations of M, and by $E^{\pm}_m(M)$ the set of all pointwise subset-minimal ones.*

See [Eiter et al., 2010a] for detailed motivation of these notions, relationships between them, and more background discussion.

Example 2.6 (ctd)**.** *Consider a MCS M_2 obtained from M_1 by modifying kb_{lab}: we replace $customer(sue,\ 02/03/1985)$ by the two facts $customer(sue, 03/02/1985)$ and $test(sue,\ bloodtest, m1)$, i.e., we change the birth date, and add a blood test result. Accordingly,*

$$\begin{aligned} kb_{lab} = \{ & customer(sue, 03/02/1985), \\ & test(sue, xray, pneumonia), test(sue, bloodtest, m1), \\ & test(Id, X, Y) \rightarrow \exists D : customer(Id, D)), \\ & customer(Id, X) \wedge customer(Id, Y) \rightarrow X = Y\}, \end{aligned}$$

M_2 is inconsistent with two minimal inconsistency explanations $e_1 = (\{r_1\}, \emptyset)$ and $e_2 = (\{r_2, r_3, r_5\}, \{r_6\})$: e_1 characterizes the problem, that C_{lab} does not accept any belief set because constraint $customer(Id, X) \wedge customer(Id, Y) \rightarrow X = Y$ is violated. Another independent inconsistency is pointed out by e_2: if e_1 is repaired, then C_{onto} accepts $AtypPneumonia(sue)$, therefore r_5 imports the need for $ab1$ into C_{dss} which makes C_{dss} inconsistent due to Sue's allergy. Moreover, the following minimal diagnoses exist for M_2: $(\{r_1, r_2\}, \emptyset)$, $(\{r_1, r_3\}, \emptyset)$, $(\{r_1, r_5\}, \emptyset)$, and $(\{r_1\}, \{r_6\})\}$. For instance, diagnosis $(\{r_1\}, \{r_6\})$ removes bridge rule r_1 from M_2 and adds r_6 unconditionally to M_2, which yields a consistent MCS. Formally, analysis of inconsistency as defined in [Eiter et al., 2010a] yields $E_m^{\pm}(M_2) = \{e_1, e_2\}$ and $D_m^{\pm}(M_2) = \big\{(\{r_1, r_2\}, \emptyset)$, $(\{r_1, r_3\}, \emptyset)$, $(\{r_1, r_5\}, \emptyset)$, $(\{r_1\}, \{r_6\})\big\}$. □

3 Policy Language IMPL

Dealing with inconsistency in an application scenario is difficult, because, even if inconsistency analysis provides information how to restore consistency, it is not obvious which choice of system repair is rational. It may not even be clear whether it is wise at all to repair the system by changing bridge rules.

Example 3.1 (ctd)**.** *Repairing e_1 by removing r_1 and thereby ignoring the birth date (which differs at the granularity of months) may be the desired reaction and could very well be done automatically. On the contrary, repairing e_2 by ignoring either the allergy or the illness is a decision that should be left to a doctor, as every possible repair could cause serious harm to Sue.* □

Therefore, managing inconsistency in a controlled way is crucial. To address these issues, we propose the declarative *Inconsistency Management Policy Language* IMPL, which provides a means to create policies for dealing with inconsistency in MCSs. Intuitively, an IMPL policy specifies (i) which inconsistencies are repaired automatically and how this shall be done, and (ii) which inconsistencies require further external input, e.g., by a human operator, to make a decision on how and whether to repair the system. Note that we do not rule out automatic repairs, but — contrary to previous approaches — automatic repairs are done only if a given policy specifies to do so, and only to the extent specified by the policy.

Since a major point of MCSs is to abstract away context internals, IMPL treats inconsistency by modifying bridge rules. For the scope of this work we delegate any potential repair by modifying the kb of a context to the user. The effect of applying an IMPL policy to an inconsistent MCS M is a *modification* (A, R) which is a pair of sets of bridge rules which are syntactically compatible with M. Intuitively, a modification specifies bridge rules A to be added to M and bridge rules R to be removed from M, similar as for diagnoses without restriction to the original rules of M.

An IMPL policy P for a MCS M is intended to be evaluated on a set of *system and inconsistency analysis* facts, denoted EDB_M, which represents information about M, in particular EDB_M contains atoms which describe bridge rules, minimal diagnoses, and minimal explanations of M.

The evaluation of P yields certain actions to be taken, which potentially interact with a human operator, and modify the MCS at hand. This modification has the potential to restore consistency of M.

In the following we formally define syntax and semantics of IMPL.

3.1 Syntax.

We assume disjoint sets C, V, $Built$, and Act, of constants, variables, built-in predicate names, and action names, respectively, and a set of ordinary predicate names $Ord \subseteq C$. Constants start with lowercase letters, variables with uppercase letters, built-in predicate names with $\#$, and action names with $@$. The set of *terms* T is defined as $T = C \cup V$.

An *atom* is of the form $p(t_1, \ldots, t_k)$, $0 \leq k$, $t_i \in T$, where $p \in Ord \cup Built \cup Act$ is an ordinary predicate name, built-in predicate name, or action name. An atom is ground if $t_i \in C$ for $0 \leq i \leq k$. The sets A_{Act}, A_{Ord}, and A_{Built}, called sets of *action atoms*, *ordinary atoms*, and *built-in atoms*, consist of all atoms over T with $p \in Act$, $p \in Ord$, respectively $p \in Built$.

Definition 3.2. *An* IMPL policy *is a finite set of rules of the form*

$$h \leftarrow a_1, \ldots, a_j, not\, a_{j+1}, \ldots, not\, a_k. \tag{2}$$

where h is an atom from $A_{Ord} \cup A_{Act}$, every a_i, $1 \leq i \leq k$, is from $A_{Ord} \cup A_{Built}$, and 'not' is negation as failure.

Given a rule r, we denote by $H(r)$ its head, by $B^+(r) = \{a_1, \ldots, a_j\}$ its positive body atoms, and by $B^-(r) = \{a_{j+1}, \ldots, a_k\}$ its negative body atoms. A rule is ground if it contains ground atoms only. A ground rule with $k = 0$ is a *fact*. As in ASP, a rule must be safe, i.e., variables in $H(r)$ or in $B^-(r)$ must also occur in $B^+(r)$. For a set of rules R, we use $cons(R)$ to denote the set of constants from C appearing in R, and $pred(R)$ for the set of ordinary predicate names and action names (elements from $Ord \cup Act$) in R.

We next describe how a policy represents information about the MCS M under consideration.

3.1.1 System and Inconsistency Analysis Predicates.

Entities, diagnoses, and explanations of the MCS M at hand are represented by a suitable finite set $C_M \subseteq C$ of constants which uniquely identify contexts, bridge rules, beliefs, rule heads, diagnoses, and explanations. For convenience, when referring to an element represented by a constant c we identify it with the constant, e.g., we write 'bridge rule r' instead of 'bridge rule of M represented by constant r'.

Reserved atoms use predicates from the set $C_{res} \subseteq Ord$ of *reserved predicates*, with $C_{res} = \{ruleHead, ruleBody^+, ruleBody^-, context, modAdd, modDel, diag, explNeed, explForbid\}$. They represent the following information.

- $context(c)$ denotes that c is a context.
- $ruleHead(r, c, s)$ denotes that bridge rule r is at context c with head formula s.
- $ruleBody^+(r, c, b)$ (resp., $ruleBody^-(r, c, b)$) denotes that bridge rule r contains body literal '$(c : b)$' (resp., '$\mathbf{not}\ (c : b)$').
- $modAdd(m, r)$ (resp., $modDel(m, r)$) denotes that modification m adds (resp., deletes) bridge rule r. Note that r is represented using $ruleHead$ and $ruleBody$.

- $diag(m)$ denotes that modification m is a minimal diagnosis in M.
- $explNeed(e, r)$ (resp., $explForbid(e, r)$) denotes that the minimal explanation (E_1, E_2) identified by constant e contains bridge rule $r \in E_1$ (resp., $r \in E_2$).
- $modset(ms, m)$ denotes that modification m belongs to the set of modifications identified by ms.

Example 3.3 (ctd)**.** *We can represent r_1, r_5, and the diagnosis $(\{r_1, r_5\}, \emptyset)$ as the set of reserved atoms $I_{ex} = \{ruleHead(r_1, c_{lab}, `customer(sue, 02/03/1985)'), ruleBody^+(r_1, c_{db}, `person(sue, 02/03/1985)'), ruleHead(r_5, c_{dss}, `need(sue, ab1)'), ruleBody^+(r_5, c_{onto}, `AtypPneumonia(sue)'), modDel(d, r_1), modDel(d, r_5), diag(d)\}$ where constant d identifies the diagnosis.* □

Further knowledge used as input for policy reasoning can easily be defined using additional (supplementary) predicates. Note that predicates over all explanations or bridge rules can easily be defined by projecting from reserved atoms. Moreover, to encode preference relations (e.g., as in [Eiter et al., 2010b]) between system parts, diagnoses, or explanations, an atom $preferredContext(c_1, c_2)$ could denote that context c_1 is considered more reliable than context c_2. The extensions of such auxiliary predicates need to be defined by the rules of the policy or as additional input facts (ordinary predicates), or they are provided by the implementation (built-in predicates), i.e., the 'solver' used to evaluate the policy. The rewriting to acthex given in Section 5.2 provides a good foundation for adding supplementary predicates as built-ins, because the acthex language has generic support for calls to external computational sources. A possible application would be to use a preference relation between bridge rules that is defined by an external predicate and can be used for reasoning in the policy.

Towards a more formal definition of a policy input, we distinguish the set B_M of ground atoms built from reserved predicates C_{res} and terms from C_M, called ***MCS input base***, and the *auxiliary input base* B_{Aux} given by predicates over $Ord \setminus C_{res}$ and terms from C. Then, the *policy input base* $B_{Aux,M}$ is defined as $B_{Aux} \cup B_M$. For a set $I \subseteq B_{Aux,M}$, $I|_{B_M}$ and $I|_{B_{Aux}}$ denote the restriction of I to predicates from the respective bases.

Now, given an MCS M, we say that a set $S \subseteq B_M$ is a *faithful representation* of M wrt. a reserved predicate $p \in C_{res} \setminus \{modset\}$ iff the extension of p in S exactly characterizes the respective entity or property of M (according to a unique naming assignment associated with C_M as mentioned). For instance, $context(c) \in S$ iff c is a context of M, and correspondingly for the other predicates. Consequently, S is a faithful representation of M iff it is a faithful representation wrt. all p in $C_{res} \setminus \{modset\}$ and the extension of $modset$ in S is empty.

A finite set of facts $I \subseteq B_{Aux,M}$ containing a faithful representation of all relevant entities and properties of an MCS qualifies as input of a policy, as defined next.

Definition 3.4. *A* policy input *I wrt. MCS M is a finite subset of the policy input base $B_{Aux,M}$, such that $I|_{B_M}$ is a faithful representation of M.*

In the following, we denote by EDB_M a policy input wrt. a MCS M. Note that reserved predicate $modset$ has an empty extension in a policy input (but corresponding atoms will be of use later in combination with actions).

Given a set of reserved atoms I, let c be a constant that appears as a bridge rule identifier in I. Then $rule_I(c)$ denotes the corresponding bridge rule represented by reserved atoms $ruleHead$, $ruleBody^+$, and $ruleBody^-$ in I with c as their first argument. Similarly we denote by $mod_I(m) = (A, R)$ (resp., by $modset_I(m) = \{(A_1, R_1), \ldots\}$) the modification (resp., set of modifications) represented in I by the respective predicates and identified by constant m.

Subsequently, we call a modification m that is projected to rules located at a certain context c 'the *projection* of m to context c'. (We use the same notation for sets of modifications.) Formally we denote by $mod_I(m)|_c$ (resp., $modset_I(m)|_c$) the projection of modification (resp., set of modifications) m in I to context c.

Example 3.5 (ctd)**.** *In the previous example I_{ex}, $rule_{I_{ex}}(r_1)$ refers to bridge rule r_1; moreover $mod_{I_{ex}}(d) = (\{r_1, r_5\}, \emptyset)$ and the projection of modification d to c_{dss} is $mod_{I_{ex}}(d)|_{c_{dss}} = (\{r_5\}, \emptyset)$.* □

Example 3.6 (ctd)**.** *A proper EDB_{M_2} of our running example is, e.g., as follows:*

$$\{context(c_{db}), context(c_{lab}), context(c_{onto}), context(c_{dss}),$$
$$ruleHead(r_1, c_{lab}, `customer(sue, 02/03/1985)'), ruleBody^+(r_1, c_{db}, `person(sue, 02/03/1985)'),$$
$$ruleHead(r_2, c_{onto}, `Pneumonia(sue)'), \quad ruleBody^+(r_2, c_{lab}, `test(sue, xray, pneumonia)'),$$
$$ruleHead(r_3, c_{onto}, `Marker(sue)'), \quad ruleBody^+(r_3, c_{lab}, `test(sue, bloodtest, m1)'),$$
$$ruleHead(r_4, c_{dss}, `need(sue, ab)'), \quad ruleBody^+(r_4, c_{onto}, `Pneumonia(sue)'),$$
$$ruleHead(r_5, c_{dss}, `need(sue, ab1)'), \quad ruleBody^+(r_5, c_{onto}, `AtypPneumonia(sue)'),$$
$$ruleHead(r_6, c_{dss}, `allow(sue, ab1)'), \quad ruleBody^-(r_6, c_{lab}, `allergy(sue, ab1)'),$$
$$diag(d_1), modDel(d_1, r_1), modDel(d_1, r_2),$$
$$diag(d_2), modDel(d_2, r_1), modDel(d_2, r_3),$$
$$diag(d_3), modDel(d_3, r_1), modDel(d_3, r_5),$$
$$diag(d_4), modDel(d_4, r_1), modAdd(d_4, r_6),$$
$$explNeed(e_1, r_1),$$
$$explNeed(e_2, r_2), explNeed(e_2, r_3), explNeed(e_2, r_5), explForbid(e_2, r_6)\}$$

Here, the two explanations and four diagnoses given in Ex. 2.6 are identified by constants e_1, e_2, d_1, ..., d_4, respectively. □

A policy can create representations of new rules, modifications, and sets of modifications, because reserved atoms are allowed to occur in heads of policy rules. However such new entities require new constants identifying them. To tackle this issue, we next introduce a facility for value invention.

3.1.2 Value Invention via Built-in Predicates '$\#id_k$'.

Whenever a policy specifies a new rule and uses it in some action, the rule must be identified with a constant. The same is true for modifications and sets of modifications. Therefore, IMPL contains a family of special built-in predicates which provide policy writers a means to comfortably create new constants from existing ones.

For this purpose, built-in predicates of the form $\#id_k(c', c_1, \ldots, c_k)$ may occur in rule bodies (only). Their intended usage is to uniquely (and thus reproducibly) associate a new constant c' with a tuple $c_1, \ldots, c_k$ of constants (for a formal semantics see the definitions for action determination in Section 3.2).

Note that this value invention feature is not strictly necessary, as new constants can be obtained via defining an order relation over all constants, a pool of unused constants, and auxiliary rules that use the next unused constant for each new constant that is required by the program. However, a dedicated value invention built-in, as introduced here, simplifies policy writing and improves policy readability.

Example 3.7. *Assume one wants to consider projections of modifications to contexts as specified by the extension of an auxiliary predicate $projectMod(M, C)$. The following policy fragment achieves this using*

a value invention built-in to assign a unique identifier with every projection (recorded in the extension of another auxiliary predicate $projectedModId(M', M, C)$*).*

$$\left\{\begin{array}{rl} projectedModId(M', M, C) \leftarrow & projectMod(M, C), \\ & \#id_3(M', pm_{id}, M, C); \\ modAdd(M', R) \leftarrow & modAdd(M, R), ruleHead(R, C, S), \\ & projectedModId(M', M, C); \\ modDel(M', R) \leftarrow & modDel(M, R), ruleHead(R, C, S), \\ & projectedModId(M', M, C) \end{array}\right\} \tag{3}$$

Intuitively, we identify new modifications by new ids $c_{pm_{id},M,C}$ *which are obtained via* $\#id_3$ *from* M*,* C*, and an auxiliary constant* $pm_{id} \notin C_M$*. The latter simply serves the purpose of disambiguating constants used for projections of modifications. This way link new identifiers to constant* pm_{id}*, therefore we can easily combine* (3) *with other policy fragments that use* $\#id_3$ *on modifications and contexts, and values invented in these fragments will not interfere with one another as long as every fragments uses its own auxiliary constant. (We therefore can think of* pm_{id} *as being 'reserved for value-invention in the projection of modifications'.)* □

Besides representing modifications of a MCS and reasoning about them, an important feature of IMPL is to actually apply them. Actions serve this purpose.

3.1.3 Actions.

Actions alter the MCS at hand and may interact with a human operator. According to the change that an action performs, we distinguish *system actions* which modify the MCS in terms of entire bridge rules that are added and/or deleted, and *rule actions* which modify a single bridge rule. Moreover, the changes can depend on external input, e.g., obtained by user interaction. In the latter case, the action is termed *interactive*. Accumulating the changes of all actions yields an overall modification of the MCS. We formally define this intuition when addressing the semantics in Section 3.2.2.

Syntactically, we use @ to prefix action names from Act. The predefined actions listed below are reserved action names. Let M be the MCS under consideration, then the following predefined actions are (non-interactive) system actions:

- $@delRule(r)$ removes bridge rule r from M.
- $@addRule(r)$ adds bridge rule r to M.
- $@applyMod(m)$ applies modification m to M.
- $@applyModAtContext(m, c)$ applies those changes in m to the MCS that add or delete bridge rules at context c (i.e., applies the projection of m to c).

Note that a policy might specify conflicting effects, i.e., the removal and the addition of a bridge rule at the same time. In this case the semantics gives preference to addition.

The predefined actions listed next are rule actions:

- $@addRuleCondition^{+}(r, c, b)$ (resp., $@addRuleCondition^{-}(r, c, b)$) adds body literal $(c : b)$ (resp., $\mathbf{not}\ (c : b)$) to bridge rule r.
- $@delRuleCondition^{+}(r, c, b)$ (resp., $@delRuleCondition^{-}(r, c, b)$) removes body literal $(c : b)$ (resp., $\mathbf{not}\ (c : b)$) from bridge rule r.

- $@makeRuleUnconditional(r)$ makes bridge rule r unconditional.

Since these actions can modify the same rule, this may also result in conflicting effects, where again addition is given preference over removal by the semantics. (Moreover, rule modifications are given preference over addition or removal of the entire rule.)

Eventually, the subsequent predefined actions are interactive (system) actions, i.e., they involve a human operator:

- $@guiSelectMod(ms)$ displays a GUI for choosing from the set of modifications ms. The modification chosen by the user is applied to M.
- $@guiEditMod(m)$ displays a GUI for editing modification m. The resulting modification is applied to M.[1]
- $@guiSelectModAtContext(ms, c)$ projects modifications in ms to c, displays a GUI for choosing among them and applies the chosen modification to M.
- $@guiEditModAtContext(m, c)$ projects modification m to context c, displays a GUI for editing it, and applies the resulting modification to M.

As we define formally in Section 3.2, changes of individual actions are not applied directly, but collected into an overall modification which is then applied to M (respecting preferences in case of conflicts as stated above). Before turning to a formal definition of the semantics, we give example policies.

Example 3.8 (ctd)**.** *Figure 1 shows three policies that can be useful for managing inconsistency in our running example. Their intended behavior is as follows. P_1 deals with inconsistencies at C_{lab}: if an explanation concerns only bridge rules at C_{lab}, an arbitrary diagnosis is applied at C_{lab}, other inconsistencies are not handled. Applying P_1 to M_2 removes r_1 at C_{lab}, the resulting MCS is still inconsistent with inconsistency explanation e_2, as only e_1 has been automatically fixed. P_2 extends P_1 by adding an 'inconsistency alert formula' to C_{lab} if an inconsistency was automatically repaired at that context. Finally, P_3 is a fully manual approach which displays a choice of all minimal diagnoses to the user and applies the user's choice. Note, that we did not combine automatic actions and user-interactions here since this would result in more involved policies (and/or require an iterative methodology; cf. Section 4).* □

We refer to the predefined IMPL actions $@delRule$, $@addRule$, $@guiSelectMod$, and $@guiEditMod$ as *core* actions, and to the remaining ones as *comfort* actions. Comfort actions exist for convenience of use, providing means for projection and for rule modifications. They can be rewritten to core actions as sketched in the following example.

Example 3.9. *To realize $@applyMod(M)$ and $@applyModAtContext(M, C)$ using the core language, we replace them by $applyMod(M)$ and $applyModAtContext(M, C)$, respectively, use rules* (3) *from Example 3.7, and add the following set of rules.*

$$\left\{\begin{array}{rl} @addRule(R) & \leftarrow applyMod(M),\ modAdd(M, R); \\ @delRule(R) & \leftarrow applyMod(M),\ modDel(M, R); \\ projectMod(M, C) & \leftarrow applyModAtContext(M, C); \\ applyMod(M') & \leftarrow applyModAtContext(M, C), \\ & \quad projectedModId(M', M, C) \end{array}\right\} \tag{4}$$

□

[1] It is suggestive to also give the human operator a possibility to abort, causing no modification at all to be made, however we do not specify this here because a useful design choice depends on the concrete application scenario.

Policies (sets of IMPL rules)	Intuitive meaning of rules in each set
$P_1 = \{ expl(E) \leftarrow explNeed(E,R);$	Define domain predicate
$\quad expl(E) \leftarrow explForbid(E,R);$	for explanations.
$\quad incNotLab(E) \leftarrow explNeed(E,R),$	Find out whether one explanation
$\quad\quad ruleHead(R,C,F), C \neq c_{lab};$	only concerns bridge rules at c_{lab}.
$\quad incNotLab(E) \leftarrow explForbid(E,R),$	
$\quad\quad ruleHead(R,C,F), C \neq c_{lab};$	
$\quad incLab \leftarrow expl(E), not\ incNotLab(E);$	
$\quad in(D) \leftarrow not\ out(D), diag(D), incLab;$	Guess a diagnosis.
$\quad out(D) \leftarrow not\ in(D), diag(D), incLab;$	
$\quad \bot \leftarrow in(A), in(B), A \neq B;$	Ensure that we guess exactly one
$\quad useOne \leftarrow in(D);$	diagnosis if there is a local
$\quad \bot \leftarrow not\ useOne, incLab;$	inconsistency at c_{lab}.
$\quad @applyModAtContext(D, c_{lab}) \leftarrow$	Apply the guessed diagnosis after
$\quad\quad useDiag(D)\}$	projecting it to context c_{lab}.
$P_2 = \{ ruleHead(r_{alert}, c_{lab}, alert) \leftarrow;$	Define new inconsistency alert rule r_{alert}.
$\quad @addRule(r_{alert}) \leftarrow incLab\}$	Add that new rule to c_{lab}.
$\cup\ P_1$	Reuse policy P_1.
$P_3 = \{ modset(md, X) \leftarrow diag(X);$	Create modification set with all diagnoses.
$\quad @guiSelectMod(md) \leftarrow\}$	Let the user choose from that set.

Figure 1: Sample IMPL policies for our running example.

This concludes our introduction of the syntax of IMPL, and we move on to a formal development of its semantics which so far has only been conveyed by accompanying intuitive explanations.

3.2 Semantics

The semantics of applying an IMPL policy P to a MCS M is defined in three steps:

- *Actions* to be executed are determined by computing a *policy answer set* of P wrt. policy input EDB_M.
- *Effects of actions* are determined by executing actions. This yields modifications (A, R) of M for each action. Action effects can be nondeterministic and thus only be determined by executing respective actions (which is particularly true for user interactions).
- Effects of actions are *materialized* by building the componentwise union over individual action effects and applying the resulting modification to M.

In the remainder of this section, we introduce the necessary definitions for a precise formal account of these steps.

3.2.1 Action Determination.

We define IMPL policy answer sets similar to the stable model semantics [Gelfond and Lifschitz, 1991]. Given a policy P and a policy input EDB_M, let id_k be a fixed (built-in) family of one-to-one mappings from k-tuples $c_1, \ldots, c_k$, where $c_i \in cons(P \cup EDB_M)$ for $1 \leq i \leq k$, to a set $C_{id} \subset C$ of 'fresh' constants, i.e., disjoint from $cons(P \cup EDB_M)$.[2] Then the *policy base* $B_{P,M}$ of P wrt. EDB_M is the set of ground IMPL atoms and actions, that can be built using predicate symbols from $pred(P \cup EDB_M)$ and terms from $U_{P,M} = cons(P \cup EDB_M) \cup C_{id}$, called policy universe.

The *grounding of* P, denoted by $grnd(P)$, is given by grounding its rules wrt. $U_{P,M}$ as usual. Note that, since $cons(P \cup EDB_M)$ is finite, only a finite amount of mapping functions id_k is used in P. Hence only a finite amount of constants C_{id} is required, and therefore $U_{P,M}$, $B_{P,M}$, and $grnd(P)$ are finite as well.

An *interpretation* is a set of ground atoms $I \subseteq B_{P,M}$. We say that I *models* an atom $a \in B_{P,M}$, denoted $I \models a$ iff (i) a is not a built-in atom and $a \in I$, or (ii) a is a built-in atom of the form $\#id_k(c, c_1, \ldots, c_k)$ and $c = id_k(c_1, \ldots, c_k)$. I models a set of atoms $A \subseteq B_{P,M}$, denoted $I \models A$, iff $I \models a$ for all $a \in A$. I models the body of rule r, denoted as $I \models B(r)$, iff $I \models a$ for every $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and for a ground rule r, $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. Eventually, I *is a model of* P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. The *FLP-reduct* [Faber et al., 2011] of P wrt. an interpretation I, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$.[3]

Definition 3.10 (Policy Answer Set)**.** *Given an MCS M, let P be an* IMPL *policy, and let EDB_M be a policy input wrt. M. An interpretation $I \subseteq B_{P,M}$ is a* policy answer set of P for EDB_M *iff I is a $\subseteq$-minimal model of $fP^I \cup EDB_M$.*

We denote by $\mathcal{AS}(P \cup EDB_M)$ the set of all policy answer sets of P for EDB_M.

3.2.2 Effect Determination.

We define the effects of action predicates $@a \in Act$ by nondeterministic functions $f_{@a}$. Nondeterminism is required for interactive actions. An action is evaluated wrt. an interpretation of the policy and yields an effect according to its type: the effect of a system action is a modification (A, R) of the MCS, in the following sometimes denoted *system modification*, while the effect of a rule action is a *rule modification* $(A, R)_r$ wrt. a bridge rule r of M, i.e., in this case A is a set of bridge rule body literals to be added to r, and R is a set of bridge rule body literals to be removed from r.

Definition 3.11. *Given an interpretation I, and a ground action α of form $@a(t_1, \ldots, t_k)$, the* effect of α wrt. *I is given by $eff_I(\alpha) = f_{@a}(I, t_1, \ldots, t_k)$, where $eff_I(\alpha)$ is a system modification if α is a system action, and a rule modification if α is a rule action.*

Action predicates of the IMPL core fragment have the following semantic functions.

- $f_{@delRule}(I, r) = (\emptyset, \{rule_I(r)\})$.
- $f_{@addRule}(I, r) = (\{rule_I(r)\}, \emptyset)$.

[2] Disjointness ensures finite groundings; without this restriction, e.g., the program $\{p(C') \leftarrow \#id_1(C', C);\ p(C)\}$ would not have finite grounding.

[3] We use the FLP reduct for compliance with acthex (used for realization in Section 5), but for the language considered, the Gelfond-Lifschitz reduct would yield an equivalent definition.

- $f_{@guiSelectMod}(I, ms) = (A, R)$ where (A, R) is the user's selection after being displayed a choice among all modifications in $\{(A_1, R_1), \ldots\} = modset_I(ms)$.
- $f_{@guiEditMod}(I, m) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is preloaded with modification $(A, R) = mod_I(m)$.

Note that the effect of any core action in I can be determined independently from the presence of other core actions in I, and rule modifications are not required to define the semantics of core actions. However, rule modifications are needed to capture the effect of *comfort* actions. Moreover, adding and deleting rule conditions, and making a rule unconditional can modify the same rule, therefore such action effects yield accumulated rule modifications.

More specifically, the semantics of IMPL comfort actions is defined as follows:

- $f_{@delRuleCondition^+}(I, r, c, b) = (\emptyset, \{(c : b)\})_r$.
- $f_{@delRuleCondition^-}(I, r, c, b) = (\emptyset, \{\mathbf{not}\ (c : b)\})_r$.
- $f_{@addRuleCondition^+}(I, r, c, b) = (\{(c : b)\}, \emptyset)_r$.
- $f_{@addRuleCondition^-}(I, r, c, b) = (\{\mathbf{not}\ (c : b)\}, \emptyset)_r$.
- $f_{@makeRuleUnconditional}(I, r) = (\emptyset, \{(c_1 : p_1), \ldots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (c_m : p_m)\})_r$ for r of the form (1).
- $f_{@applyMod}(I, m) = mod_I(m)$.
- $f_{@applyModAtContext}(I, m, c) = mod_I(m)|_c$.
- $f_{@guiSelectModAtContext}(I, ms, c) = (A', R')$ where (A', R') is the user's selection after being displayed a choice among all modifications in $\{(A'_1, R'_1), \ldots\} = modset_I(ms)|_c$.
- $f_{@guiEditModAtContext}(I, m, c) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is preloaded with modification $mod_I(m)_c$.

In practice, however, it is not necessary to implement action functions on the level of rule modifications, since a policy in the comfort fragment can equivalently be rewritten to the core fragment (which does not rely on rule modifications). Example 3.9 already sketched a rewriting for $@applyMod$ and $@applyModAtContext$. In Section 6 we provide a rewriting from IMPL to the IMPL core fragment.

The effects of user-defined actions have to comply to Definition 3.11.

3.2.3 Effect Materialization.

Once the effects of all actions in a selected policy answer set have been determined, an overall modification is computed by the componentwise union over all individual modifications. This overall modification is then materialized in the MCS.

Given a MCS M and a policy answer set I (for a policy P and a corresponding policy input EDB_M), let I_M, respectively I_R, denote the set of ground system actions, respectively rule actions, in I. Then, $M_{\mathit{eff}} = \{\mathit{eff}_I(\alpha) | \alpha \in I_M\}$ is the set of effects of system action atoms in I, and $R_{\mathit{eff}} = \{\mathit{eff}_I(\alpha) | \alpha \in I_R\}$ is the set of effects of rule actions in I. Furthermore, $Rules = \{r \mid (A, R)_r \in R_{\mathit{eff}}\}$ is the set of bridge rules modified by R_{eff}, and for every $r \in Rules$, let $\mathcal{R}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} R$, respectively $\mathcal{A}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} A$, denote the union of rule body removals, respectively additions, wrt. r in R_{eff}.

Definition 3.12. *Given a MCS M, and an* IMPL *policy P, let I be a policy answer set of P for a policy input EDB_M wrt. M. Then, the* materialization of I in M *is the MCS M' obtained from M by replacing its set of bridge rules $br(M)$ by the set*

$$(br(M) \setminus \mathcal{R} \cup \mathcal{A}) \setminus Rules \cup \mathcal{M},$$

where $\mathcal{R} = \bigcup_{(A,R)\in M_{eff}} R$, $\mathcal{A} = \bigcup_{(A,R)\in M_{eff}} A$, and $\mathcal{M} = \{(k{:}s) \leftarrow Body \mid r \in Rules,\ r \in br_k,\ h_b(r) = s,\ Body = B(r) \setminus \mathcal{R}_r \cup \mathcal{A}_r\}$. (Formally, $M' = M[(br(M) \setminus \mathcal{R} \cup \mathcal{A}) \setminus Rules \cup \mathcal{M}]$.)

Note that, by definition, the addition of bridge rules has precedence over removal, and the addition of body literals similarly has precedence over removal. There is no particular reason for this choice; one just has to be aware of it when specifying a policy. Apart from that, no order for evaluating individual actions is specified or required.

Eventually, we can define modifications of a MCS that are accepted by a corresponding IMPL policy.

Definition 3.13. *Given a MCS M, an* IMPL *policy P, and a policy input EDB_M wrt. M, a modified MCS M'* is an admissible modification of M wrt. P and EDB_M *iff M' is the materialization of some policy answer set $I \in \mathcal{AS}(P \cup EDB_M)$.*

Example 3.14 (ctd)**.** *Evaluating $P_2 \cup EDB_{M_2}$ yields four policy answer sets, one is*

$$I_1 = EDB_{M_2} \cup \{expl(e_1), expl(e_2), incNotLab(e_2), incLab, in(d_1), out(d_2), out(d_3), out(d_4), useOne, \\ ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_1, c_{lab})\}.$$

From I_1 we obtain a single admissible modification of M_2 wrt. P_2: add bridge rule r_{alert} and remove r_1.

The other policy answer sets are

$$EDB_{M_2} \cup \{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), in(d_2), out(d_3), out(d_4), useOne, \\ ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_2, c_{lab})\},$$

$$EDB_{M_2} \cup \{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), out(d_2), in(d_3), out(d_4), useOne, \\ ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_3, c_{lab})\},\ \textit{and}$$

$$EDB_{M_2} \cup \{expl(e_1), expl(e_2), incNotLab(e_2), incLab, out(d_1), out(d_2), out(d_3), in(d_4), useOne, \\ ruleHead(r_{alert}, c_{lab}, alert), @addRule(r_{alert}), @applyModAtContext(d_4, c_{lab})\}.$$

Evaluating $P_3 \cup EDB_{M_2}$ yields one policy answer set, which is $I_2 = EDB_{M_2} \cup \{modset(md, d_1), modset(md, d_2), modset(md, d_3), modset(md, d_4), @guiSelect\text{-}Mod(md)\}$. Determining the effect of I_2 involves user interaction; thus multiple materializations of I_2 exist. For instance, if the user chooses to ignore Sue's allergy and birth date (and probably imposes additional monitoring on Sue), then we obtain an admissible modification of M which adds the unconditional version of r_6 and removes r_1. □

4 Methodologies of Applying IMPL and Realization

Based on the simple system design shown in Figure 2, we next briefly discuss elementary methodologies of applying IMPL for the purpose of integrating MCS reasoning with potential user interaction in case of inconsistency.

We maintain a representation of the MCS together with a *store of modifications*. The *semantics evaluation* component performs reasoning tasks on the MCS and invokes the *inconsistency manager* in case of an

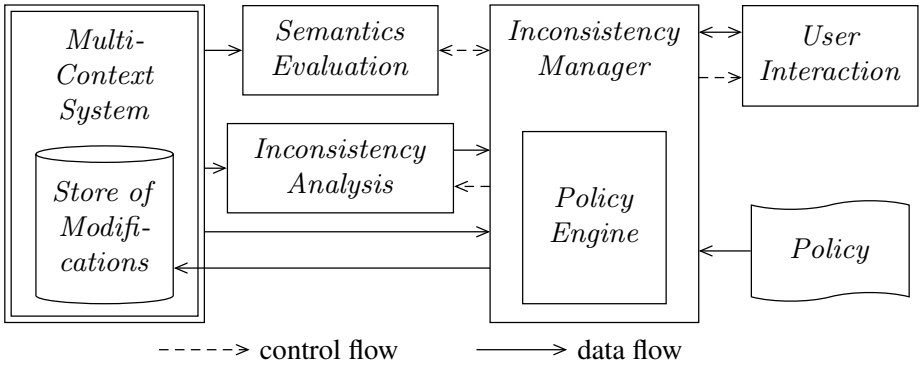


Figure 2: Policy integration data flow and control flow block diagram.

inconsistency. This inconsistency manager uses the *inconsistency analysis* component[4] to provide input for the *policy engine* which computes policy answer sets of a given IMPL *policy* wrt. the MCS and its inconsistency analysis result. This policy evaluation step results in action executions potentially involving user interactions and causes changes to the store of modifications, which are subsequently materialized. Finally the inconsistency manager hands control back to the semantics evaluation component.

4.1 Reasoning Modes

Principal modes of operation, and their merits, are the following.

Reason and Manage once. This mode of operation evaluates the policy once, if the effect materialization does not repair inconsistency in the MCS, no further attempts are made and the MCS stays inconsistent. While simple, this mode may not be satisfying in practice.

Manage-Once-Ranked-Repair-Attempts. In this strategy, the result of evaluating a policy wrt. an inconsistency does not yield a single attempt for restoring consistency, instead it yields multiple attempts, each with a separate set of actions.

This requires to augment actions of the policy language by an *attempt ranking* which specifies an order of actions to be applied: first only the highest-ranked modifications are used, if this repairs the system the process finishes. Otherwise the highest-ranked modifications are removed and the process restarts, looking for the set of actions with the second-best rank and so on. This is repeated until either the system becomes consistent (success), or until no lower rank exists (failure).

Example 4.1. *An inconsistency management strategy generates some sophisticated policy-generated set of modifications which is attempted first. If this first attempt fails to restore consistency, the policy uses an element of the set of minimal diagnoses as a fallback modification. This guarantees to restore inconsistency. Additionally, this second attempt adds a bridge rule to some context, to notify contexts (and thus users and operators of the MCS). This way reasoning with the system is never impossible due to inconsistency, however consistency may come at the cost of being a "fallback consistency".* □

[4]For realizations of this component we refer to [Bögl et al., 2010, Eiter et al., 2010a].

User interaction in this strategy demands special considerations: (i) user modifications could be the same for all attempt ranks, such that the user does not need to care about ranks, or (ii) the user can produce sets of modifications for multiple ranks. The first option seems easier to use, while the second provides more possibilities to the user and to inconsistency management as a whole.

Overall this mode of using IMPL requires only one reasoning step and easily guarantees termination of the inconsistency management process.

Reason and Manage iteratively. Another way to deal with failure to restore consistency is to simply invoke policy evaluation again on the modified but still inconsistent system. This is useful if user interaction may involve trial-and-error, especially if multiple inconsistencies occur: some might be more difficult to counteract than others.

Another positive aspect of iterative policy evaluation is, that it allows for policies to be structured, e.g., as follows: (a) classify inconsistencies into automatically versus manually repairable; (b) apply actions to repair one of the automatically repairable inconsistencies; (c) if such inconsistencies do not exist: perform user interaction actions to repair one (or all) of the manually repairable inconsistencies. Such policy structuring follows a divide-and-conquer approach, trying to focus on individual sources of inconsistency and to disregard interactions between inconsistencies as much as possible. If user interaction consists of trial-and-error bugfixing, fewer components of the system are changed in each iteration, and the user starts from a situation where only critical (i.e. not automatically repairable) inconsistencies are present in the MCS. Moreover, such policies may be easier to write and maintain. On the other hand, termination of iterative methodologies is not guaranteed. However, one can enforce termination by limiting the number of iterations, possibly by extending IMPL with a *control action* that configures this limit.

Manage-Iteratively-First-Auto-Then-User. This is a specialization of the above 'Manage-Iteratively' strategy, with the goal of adding more structure to the inconsistency management process. We accomplish this by deliberately using iterations as a procedural aspect controlled by the declarative policy language.As the name suggests, a policy following this strategy emits either only modification actions, or only user interactions.

This suggests to use the following structure for a policy: detected inconsistencies are categorized as automatically repairable or not, if there exist automatically repairable ones, actions to repair them are emitted, otherwise user interactions for the remaining inconsistencies are emitted. (Additionally, the policy could only emit repair actions for single automatically repairable inconsistencies in one iteration.)

This kind of a policy has the benefit that it does one thing at a time instead of doing everything at once. Therefore, identifying problems (i.e., debugging policies or the whole inconsistency management process) is likely to be easier than in the more general case.

Furthermore, if user interaction consists of trial-and-error bugfixing, fewer components of the system are changed in each iteration. This should have favorable effects on the performance and maintainability of inconsistency management.

4.2 Additional Considerations

Here we discuss additional properties and features that could be advantageous in practical applications. (And could easily be added to IMPL.)

Iteration-persistent Storage. In iterative mode it may be useful to access information from previous iterations. We call this persistent storage. For instance, a persistant storage (reminiscent of an RDF triplestore) can be added to IMPL as follows: (a) we add a (persistent) triplestore to the policy engine, (b) define actions $@kbAdd(S, P, O)$ and $@kbDel(S, P, O)$ s.t. $@kbAdd$ stores and $@kbDel$ removes triples, and finally (c) define a new ternary predicate $kbTriple(S, P, O)$ that is added to EDB_M for each stored triple.

Stable Identifiers. When an IMPL policy is applied to an MCS, it might remove, add, or change bridge rules. In an iterative mode of operation, it would be useful if changing a bridge rule did not change its identifier.

For example, the bridge rule r_{alert} might be added to M_2 by our example policy P_2 (see Example 3.14), which yields a new MCS M_2'. If we apply IMPL to M_2', the subsequent $EDB_{M_2'}$ should then use again r_{alert} to identify that bridge rule. If this is the case, we can reason about the existence of that rule in our policy.

When using iteration-persistant storage, we can store rule-identifiers across iterations; however this only makes sense if identifiers remain the same across iterations.

Therefore stable identifiers are a desireable property. This property can be added to IMPL as a simple condition on added and modified rules, namely that they have an associated identifier which remains the same for subsequently created EDB_M's. (In Section 6 we will take particular care to provide stable identifiers.)

Automatic Modifications vs User Interactions. In the current declarative semantics definition, a rule might be 'simultaneously' modified both by a user interation and by another action. However, this means that a modification done by a user can be undone by another action that was triggered by the policy. Therefore, to achieve a system with intuitively clear effects of a user's actions, user interaction actions should be limited to rules that are not modified by other actions.

5 Realizing IMPL in acthex

In this section, we demonstrate how IMPL can be realized using acthex. First we give preliminaries about acthex, which is a logic programming formalism that extends HEX programs with executable actions. We then show how to implement the core IMPL fragment by rewriting it to acthex in Section 5.2.

5.1 Preliminaries on acthex

The acthex formalism [Basol et al., 2010] generalizes HEX programs [Eiter et al., 2005] by adding dedicated action atoms to heads of rules. An acthex program operates on an *environment*; this environment can influence external sources in acthex, and it can be modified by the execution of actions.

5.1.1 Syntax.

By $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ we denote mutually disjoint sets whose elements are called constant names, variable names, external predicate names, and action predicate names, respectively. Elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ (resp., $\mathcal{A}$) are prefixed with "&" (resp. "#"). Names in $\mathcal{C}$ serve both as constant and predicate names, and we assume that $\mathcal{C}$ contains a finite subset of consecutive integers $\{0, \ldots, n_{max}\}$.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots Y_n)$, where $Y_0, Y_1, \ldots Y_n$ are terms, and $n \geq 0$ is the arity of the atom. Intuitively, Y_0 is the predicate name, and we

thus also use the more familiar notation $Y_0(Y_1 \dots Y_n)$. An atom is ordinary if Y_0 is a constant. An external atom is of the form $\&g[Y_1, \dots, Y_n](X_1, \dots, X_m)$ with $Y_1, \dots, Y_n$ and $X_1, \dots, X_m$ being lists of terms. An action atom is of the form $\#g[Y_1, \dots, Y_n]\{o, r\}[w : l]$ where $\#g$ is an action predicate name, $Y_1, \dots, Y_n$ is a list of terms (called input list), and each action predicate $\#g$ has fixed length $in(\#g) = n$ for its input list. Attribute $o \in \{b, c, c_p\}$ is called the *action option*; depending on o the action atom is called *brave, cautious, and preferred cautious*, respectively. Attributes r, w, and l are called *precedence*, *weight*,[5] and *level*[5] of $\#g$, denoted by $prec(a)$, $weight(a)$, and $level(a)$, respectively. They are optional and range over variables and positive integers.

A *rule* r is of the form $\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_1, \dots, \beta_n, not\ \beta_{n+1}, \dots, not\ \beta_m$, where $m, n, k \geq 0$, $m \geq n$, $\alpha_1, \dots, \alpha_k$ are atoms or action atoms, and $\beta_1, \dots \beta_m$ are atoms or external atoms. We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \dots, \beta_m\}$. An acthex *program* is a finite set P of rules.

Example 5.1. *The acthex program* $\{night \vee day \leftarrow ; \#robot[goto, charger]\{b, 1\} \leftarrow \&sensor[bat](low);$ $\#robot[clean, kitchen]\{c, 2\} \leftarrow night;$ $\#robot[clean, bedroom]\{c, 2\} \leftarrow day\}$ *uses action atom* $\#robot$ *to command a robot, and an external atom* $\&sensor$ *to obtain sensor information. Precedence 1 of action atom* $\#robot[goto, charger]\{b, 1\}$ *makes the robot recharge its battery before executing cleaning actions, if necessary.* □

5.1.2 Semantics.

We present several features of the semantics of acthex slightly different than [Basol et al., 2010]. Therefore we here first give an intuitive overview of the semantics and then precise formal definitions.

Intuitively, an acthex program P is evaluated wrt. an *external environment* E using the following steps: (i) *answer sets* of P are determined wrt. E, the set of *best models* is a subset of the answer sets determined by an objective function; (ii) one best model is selected, and one *execution schedule* S is generated for that model (although a model may give rise to multiple execution schedules); (iii) the *effects of action atoms* in S are applied to E in the order defined by S, yielding an updated environment E'; and finally (iv) the process may be iterated starting at (i), unless no actions were executed in (iii) which terminates an iterative evaluation process. Formally the acthex semantics is defined as follows.

Given an acthex program P, the *Herbrand base* HB_P of P is the set of all possible ground versions of atoms, external atoms, and action atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. Given a rule $r \in P$, the grounding $grnd(r)$ of r is defined accordingly, the grounding of program P is given as the grounding of all its rules. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ are implicitly given by P.

An *interpretation* I *relative to* P is any subset $I \subseteq HB_P$ containing ordinary atoms and action atoms. We say that I is a *model* of atom (or action atom) $a \in HB_P$, denoted by $I \models a$, iff $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n+m+2)$-ary Boolean function $f_{\&g}$, assigning each tuple $(E, I, y_1, \dots, y_n, x_1, \dots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $x_i, y_j \in \mathcal{C}$, $I \subseteq HB_P$, and E is an environment. Note that this slightly generalizes the external atom semantics such that they may take E into account, which was left implicit in [Basol et al., 2010]. We say that an interpretation I relative to P is a *model* of a ground external atom $a = \&g[y_1, \dots, y_n](x_1, \dots, x_m)$ wrt. environment E, denoted as $I, E \models a$, iff $f_{\&g}(E, I, y_1 \dots, y_n, x_1, \dots, x_m) = 1$. Let r be a ground rule. We define (i) $I, E \models H(r)$ iff there is some $a \in H(r)$ such that $I, E \models a$, (ii) $I, E \models B(r)$ iff $I, E \models a$ for all $a \in B^+(r)$ and $I, E \not\models a$ for all $a \in B^-(r)$, moreover (iii) $I, E \models r$ iff $I, E \models H(r)$ or $I, E \not\models B(r)$. We say that I is a *model* of P

[5] Weight and level have a similar intuition as the corresponding attributes of weak constraints in ASP [Buccafurri et al., 1997].

wrt. E, denoted by $I, E \models P$, iff $I, E \models r$ for all $r \in grnd(P)$. The *FLP-reduct* of P wrt. I and E, denoted as $fP^{I,E}$, is the set of all $r \in grnd(P)$ such that $I, E \models B(r)$. Eventually, I is an *answer set* of P wrt. E iff I is a $\subseteq$-minimal model of $fP^{I,E}$. Note that, as for HEX programs we need the FLP-reduct [Faber et al., 2011], which is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary programs, and in acthex ensures answer-set minimality in the presence of external atoms (see [Eiter et al., 2008b] for details). We denote by $\mathcal{AS}(P, E)$ the collection of all answer sets of P wrt. E.

The set of *best models* of P, denoted $\mathcal{BM}(P, E)$, contains those $I \in \mathcal{AS}(P, E)$ that minimize the objective function $H_P(I) = \Sigma_{a \in A}(\omega \cdot level(a) + weight(a))$, where $A \subseteq I$ is the set of action atoms in I, and ω is the first limit ordinal. (This definition using ordinal numbers is equivalent to the definition of weak constraint semantics in [Buccafurri et al., 1997].) Intuitively, an answer set I will be among the best models if no other answer set contains only actions with a lower level, and if no other answer set I' that contains only actions with the same level as I has a smaller weight of all contained actions.

An action $a = \#g[y_1, \ldots, y_n]\{o, r\}[w : l]$ with option o and precedence r is *executable in I wrt. P and E* iff (i) a is brave and $a \in I$, or (ii) a is cautious and $a \in B$ for every $B \in \mathcal{AS}(P, E)$, or (iii) a is preferred cautious and $a \in B$ for every $B \in \mathcal{BM}(P, E)$. An *execution schedule* of a best model I is a sequence of all actions executable in I, such that for all action atoms $a, b \in I$, if $prec(a) < prec(b)$ then a has a lower index in the sequence than b. We denote by $\mathcal{ES}_{P,E}(I)$ the set of all execution schedules of a best model I wrt. acthex program P and environment E; formally, let A_e be the set of action atoms that are executable in I wrt. P and E, then $\mathcal{ES}_{P,E}(I) = \big\{[a_1, \ldots, a_n] \mid prec(a_i) \leq prec(a_j)$, for all $1 \leq i < j \leq n$, and $\{a_1, \ldots, a_n\} = A_e\big\}$.

Example 5.2. *In Example 5.1, if the robot has low battery, then $\mathcal{AS}(P, E) = \mathcal{BM}(P, E)$ contains models*

$$I_1 = \{night, \#robot[clean, kitchen]\{c, 2\}, \#robot[goto, charger]\{b, 1\}\}, \text{ and}$$
$$I_2 = \{day, \#robot[clean, bedroom]\{c, 2\}, \#robot[goto, charger]\{b, 1\}\}.$$

We have $\mathcal{ES}_{P,E}(I_1) = \{\#robot[goto, charger]\{b, 1\}, \#robot[clean, bedroom]\{c, 2\}\}$. □

Given a model I, the *effect of executing a ground action $\#g[y_1, \ldots, y_m]\{o, p\}[w : l]$* on an environment E wrt. I is defined for each action predicate name $\#g$ by an associated $(m+2)$-ary function $f_{\#g}$ which returns an updated environment $E' = f_{\#g}(E, I, y_1, \ldots, y_m)$. Correspondingly, given an execution schedule $S = [a_1, \ldots, a_n]$ of a model I, the *execution outcome* of S in environment E wrt. I is defined as $EX(S, I, E) = E_n$, where $E_0 = E$, and $E_{i+1} = f_{\#g}(E_i, I, y_1, \ldots, y_m)$, given that a_i is of the form $\#g[y_1, \ldots, y_m]\{o, p\}[w : l]$. Intuitively the initial environment $E_0 = E$ is updated by each action in S in the given order. The set of possible execution outcomes of a program P on an environment E is denoted as $\mathcal{EX}(P, E)$, and formally defined by $\mathcal{EX}(P, E) = \{EX(S, I, E) \mid S \in \mathcal{ES}_{P,E}(I)$ where $I \in \mathcal{BM}(P, E)\}$.

In practice, one usually wants to consider a single execution schedule. This requires the following decisions during evaluation: (i) to select one best model $I \in \mathcal{BM}(P, E)$, and (ii) to select one execution schedule $S \in \mathcal{ES}_{P,E}(I)$. Finally, one can then execute S and obtain the new environment $E' = EX(S, I, E)$.

5.2 Rewriting the IMPL Core Fragment to acthex

Using acthex for realizing IMPL is a natural and reasonable choice because acthex already natively provides several features necessary for IMPL: external atoms can be used to access information from a MCS, and acthex actions come with weights for creating ordered execution schedules for actions occurring within the same answer set of an acthex program. Based on this, IMPL can be implemented by a rewriting to acthex, with acthex actions implementing IMPL actions, acthex external predicates providing information about the MCS to the IMPL policy, and acthex external predicates realizing the value invention built-in predicates.

We next describe a rewriting from the IMPL core language fragment to acthex. We assume that the environment E contains a pair $(\mathcal{A}, \mathcal{R})$ of sets of bridge rules, and an encoding of the MCS M (suitable for an implementation of the external atoms introduced below, e.g., in the syntax used by the MCS-IE system [Bögl et al., 2010], which provide the corresponding policy input). A given IMPL policy P wrt. the MCS M is then rewritten to an acthex program P^{act} as follows.

1. Each core IMPL action $@a(t)$ in the head of a rule of P is replaced by a brave acthex action $\#a[t]\{b, 2\}$ with precedence 2. These acthex actions implement semantics of the respective IMPL actions according to Def. 3.11: interpretation I and the original action's argument t are used as input, the effects are accumulated as $(\mathcal{A}, \mathcal{R})$ in E.

2. Each IMPL built-in $\#id_k(C, c_1, \ldots, c_k)$ in P is replaced by an acthex external atom $\&id_k[c_1, \ldots, c_k](C)$. The family of external atoms $\&id_k[c_1, \ldots, c_k](C)$ realizes value invention and has as semantics function $f_{\&id_k}(E, I, c_1, \ldots, c_k, C) = 1$ for one constant $C = auxc\_c_1\_\ldots\_c_k$ created from the constants in tuple $c_1, \ldots, c_k$.

3. We add to P^{act} a set P_{in} of acthex rules containing (i) rules that use, for every $p \in C_{res} \setminus \{modset\}$, a corresponding external atom to 'import' a faithful representation of M, and (ii) a preparatory action $\#reset$ with precedence 1, and a final action $\#materialize$ with precedence 3:

$$P_{in} = \{p(\mathbf{t}) \leftarrow \&p[](\mathbf{t}) \mid p \in C_{res} \setminus \{modset\}\} \cup \{\#reset[]\{b, 1\}; \#materialize[]\{b, 3\}\},$$

where $\mathbf{t}$ is a vector of different variables of length equal to the arity of p (i.e., one, two, or three).

The first two steps transform IMPL actions into acthex actions, and $\#id_k$-value invention into external atom calls. The third step essentially creates policy input facts from acthex external sources. External atoms in P_{in} return a representation of M and analyze inconsistency in M, providing minimal diagnoses and minimal explanations. Thus, the respective rules in P_{in} yield an extension of the corresponding reserved predicates which is a faithful representation of M. Moreover, action $\#reset$ resets the modification $(\mathcal{A}, \mathcal{R})$ stored in E to $(\emptyset, \emptyset)$.[6] Action $\#materialize$ materializes the modification $(\mathcal{A}, \mathcal{R})$ (as accumulated by actions of precedence 2) in the MCS M (which is part of E).

Example 5.3 (ctd)**.** *Policy P_3 from Ex. 3.8 translated to acthex contains the following rules*

$$P_3^{act} = P_{in} \cup \big\{modset(md, X) \leftarrow diag(X); \#guiSelectMod[md]\{b, 2\}\big\}$$

where

$$\begin{aligned} P_{in} = \big\{ & ruleHead(R, C, S) \leftarrow \&ruleHead[](R, C, S); \\ & ruleBody^+(R, C, S) \leftarrow \&ruleBody^+[](R, C, S); \\ & ruleBody^-(R, C, S) \leftarrow \&ruleBody^-[](R, C, S); \\ & \ldots \\ & \#reset[]\{b, 1\}; \#materialize[]\{b, 3\}\big\}. \end{aligned}$$

□

Note, that actions in the rewriting have no weights, therefore all answer sets are best models. For obtaining an admissible modification, any policy answer set can be chosen, and any execution schedule can be used.

[6] This reset is necessary if a policy is applied repeatedly, as discussed in Section 4.1, i.e., in iterative reasoning modes.

Proposition 5.4. *Given a MCS M, a core* IMPL *policy P, and a policy input EDB_M wrt. M, let P^{act} be as above, and consider an environment E containing M and $(\emptyset, \emptyset)$. Then, every execution outcome $E' \in \mathcal{EX}(P^{act} \cup EDB_M|_{B_{Aux}}, E)$ contains instead of M an admissible modification M' of M wrt. P and EDB_M.*

Proof. In this proof we denote by $\mathcal{AS}_I$ the IMPL policy answer set function, and by $\mathcal{AS}_A$ the acthex answer set function. Admissible modifications of IMPL are defined using $\mathcal{AS}_I$, and execution outcomes of acthex are defined using $\mathcal{AS}_A$, therefore we first establish a relationship between policy answer sets $I_I \in \mathcal{AS}_I(P \cup EDB_M)$ and answer sets $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M|_{B_{Aux}}, E)$. Let $P_{in}{}' = \{p(\mathbf{t}) \leftarrow \&p[](\mathbf{t}) \mid p \in C_{res} \setminus \{modset\}\}$ then semantics of external atoms $\&p[](\mathbf{t})$ are independent from E and from I (they depend only on M), and they are defined such that $\mathcal{AS}_A(P_{in}{}') = \{EDB_M|_{B_M}\}$. Therefore, and because $P_{in}{}' \subseteq P^{act}$, every I_A is such that $EDB_M|_{B_M} \subseteq I_A$ and we get that $\mathcal{AS}_A(P^{act} \cup EDB_M|_{B_{Aux}}, E) = \mathcal{AS}_A(P^{act} \cup EDB_M, E)$. We next show the following relationship between IMPL answer sets and acthex answer sets on the rewritten program: given a set A of action atoms $@\alpha(t)$ where $@\alpha \in Act$, $t \in C$, we show that $I_I \in \mathcal{AS}_I(P \cup EDB_M)$ iff $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M, E)$ where $I_I = I \cup A$ and $I_A = I \cup \{\#\alpha[t]\{b, 2\} \mid @\alpha(t) \in A\} \cup \{\#reset[]\{b, 1\}, \#materialize[]\{b, 3\}\}$ and I is a set of ground ordinary atoms (i.e., I neither contains IMPL actions nor acthex actions) with $EDB_M \subseteq I$. Item 2 (page 22) replaces all built-ins by an external computation that exactly realizes semantics of the replaced built-in (wlog. we assume that $id_k(c_1, \ldots, c_k) = auxc\_c_1\_\ldots\_c_k$, if this is not the case the answer sets coincide modulo auxiliary constant replacement). Rules in P_{in} are always satisfied by I_A as it contains the $\#reset$ and $\#materialize$ actions and as it contains EDB_M. Everything else (i.e., rule bodies and rule heads) in P^{act} is equal to P, and rule and rule body semantics are defined equally in IMPL and acthex, modulo action renaming. Furthermore, both semantics are defined as minimal models of the reduct, and (in the definition of IMPL policy answer sets) $I_I \models fP^{I_I} \cup EDB_M$ iff $I_I \models f(P \cup EDB_M)^{I_I}$. Therefore the following intermediate result holds: $I_I \in \mathcal{AS}_I(P \cup EDB_M)$ iff $I_A \in \mathcal{AS}_A(P^{act} \cup EDB_M, E)$, with I_I and I_A as introduced above. As actions in P^{act} have no weight and no level, all answer sets are best models. An execution schedule of an answer set I_A first executes $\#reset$, then executes actions that originated in IMPL actions, and finally executes $\#materialize$. The reset sets $(\mathcal{A}, \mathcal{R})$ in E to $(\emptyset, \emptyset)$ and actions created from IMPL actions by their definition realize semantics of the corresponding IMPL actions and accumulate the resulting sets of added and removed bridge rules in $(\mathcal{A}, \mathcal{R})$, before executing $\#materialize$ we have that $(\mathcal{A}, \mathcal{R}) = M_{\mathit{eff}}$ (for M_{eff} see 3.2.3) and $\#materialize$ modifies M in E to yield M' which is a materialization of I_I in M and therefore an admissible modification of M. Therefore the result holds. □

The results of this section can be used to realize the full IMPL language, using the rewriting technique described in the next section.

6 Rewriting IMPL to the IMPL Core fragment

In this section we provide a rewriting from the complete IMPL language to the IMPL core fragment. This allows us to realize the whole IMPL language using the acthex rewriting which realizes the IMPL core fragment.

Our rewriting will be 'identifier-neutral' in the sense that if the original policy would have created a rule with identifier r, the rewritten policy creates the rule exactly with the same identifier. Furthermore, rule modifications are realized by removing the original rule and adding a modified version. Here, again, the rewritten policy uses the original identifier to create the modified rule. As a consequence, our rewriting can be used if stable identifiers are required (see Section 4.2 for this property and its benefits).

For our rewriting, it is furthermore important that user interactions are limited to rules that are not modified by other actions. This restriction is useful in practice and has been discussed in Section 4.2.

For that purpose we introduce auxiliary predicates and constants which do not occur anywhere in a policy before rewriting. Given an IMPL policy P and a policy input EDB_M, we first define the set of *critical constants* which cannot be freely used by the rewriting: $critical(P \cup EDB_M) = C_M \cup C_{res} \cup cons(P \cup EDB_M)$.

Example 6.1 (ctd.)**.** *We have* $critical(P_1 \cup EDB_{M_2}) = C_{res} \cup \{c_{db}, c_{onto}, c_{lab}, c_{dss}, r_1, \ldots, r_6, d_1, \ldots, d_4, e_1, e_2, expl, incNotLab, incLab, in, out, useOne\}$*. (See Example 3.6 and Figure 3.8 for* EDB_{M_2} *and* P_1*).* □

W.l.o.g., we assume that the following sets of 'fresh' constants are disjoint with $critical(P \cup EDB_M)$: $\{c' \mid c \in C_M\}$, $\{ra_\alpha \mid \alpha \in Act\}$, and $\{map, modifiedRule, add^+, add^-, del^+, del^-, cm_{id}, csm_{id}, pm_{id}, psm_{id}, cleanMod, cleanedModId, cleanModSet, cleanedModSetId, projectMod, projectedModId, projectModSet, projectedModSetId\}$.

Given a set P of IMPL rules, we define the replacement function $tr_{repl}(P)$ which replaces every constant $c \in C_M$ in every rule in P by its corresponding constant c' and returns the resulting set of rules. Note that facts are also translated by tr_{repl}. The replacement of all constants with fresh constants is required to obtain an identifier-stable rewriting.

Given a set of IMPL rules P, we define the replacement function $tr_{act}(P)$ which replaces every action atom $\alpha(\mathbf{t})$ in every rule in P by an ordinary atom $ra_\alpha(\mathbf{t})$ and returns the resulting set of rules. (Again, facts are translated.)

Given an IMPL policy P and a policy input EDB_M, then $P' = tr_{repl}(tr_{act}(P \cup EDB_M))$ is an IMPL policy which does not contain any actions (therefore it is in the IMPL Core fragment), furthermore P' does not contain constants from C_M. Policy answer sets of P' correspond 1-1 to policy answer sets of $P \cup EDB_M$ such that the former contain a replacement atom ra_α iff the latter contain a corresponding action α.

We next describe the IMPL code fragment P_{Aux} which realizes semantics of IMPL actions by translating replacement atoms to IMPL core actions.

For mapping replaced constants back to their original value (to achieve stable identifiers) P_{Aux} contains the following facts:

$$map(c, c'). \qquad \text{for every constant } c \in C_M \tag{5}$$

We collect all rules which are modified in $modifiedRule$.

$$\begin{aligned}
&modifiedRule(R) \leftarrow ra_{addRuleCondition^+}(R, C, B).\\
&modifiedRule(R) \leftarrow ra_{addRuleCondition^-}(R, C, B).\\
&modifiedRule(R) \leftarrow ra_{delRuleCondition^+}(R, C, B).\\
&modifiedRule(R) \leftarrow ra_{delRuleCondition^-}(R, C, B).\\
&modifiedRule(R) \leftarrow ra_{makeRuleUnconditional}(R).
\end{aligned} \tag{6}$$

We accumulate effects of rule modification actions in add^+, add^-, del^+, and del^-.

$$\begin{aligned}
&add^+(R, C, B) \leftarrow ra_{addRuleCondition^+}(R, C, B).\\
&add^-(R, C, B) \leftarrow ra_{addRuleCondition^-}(R, C, B).\\
&del^+(R, C, B) \leftarrow ra_{delRuleCondition^+}(R, C, B).\\
&del^+(R, C, B) \leftarrow ra_{makeRuleUnconditional}(R), ruleBody^+(R, C, B).\\
&del^-(R, C, B) \leftarrow ra_{delRuleCondition^-}(R, C, B).\\
&del^-(R, C, B) \leftarrow ra_{makeRuleUnconditional}(R), ruleBody^-(R, C, B).
\end{aligned} \tag{7}$$

We represent rule bodies for modified rules in reserved predicates, using original rule, context, and belief identifiers. (We use primed variable names where primed identifiers will be grounded.)

$$\begin{aligned}
ruleBody^{+}(R,C,B) &\leftarrow add^{+}(R',C',B'),\\
&\quad modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleBody^{+}(R,C,B) &\leftarrow ruleBody^{+}(R',C',B'),\ not\ del^{+}(R',C',B'),\\
&\quad modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleBody^{-}(R,C,B) &\leftarrow add^{-}(R',C',B'),\\
&\quad modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleBody^{-}(R,C,B) &\leftarrow ruleBody^{-}(R',C',B'),\ not\ del^{-}(R',C',B'),\\
&\quad modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleHead(R,C,B) &\leftarrow ruleHead(R',C',B'),\\
&\quad modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').
\end{aligned} \tag{8}$$

We represent new rule bodies for unmodified rules in reserved predicates, using original identifiers.

$$\begin{aligned}
ruleBody^{+}(R,C,B) &\leftarrow ruleBody^{+}(R',C',B'),\ not\ modifiedRule(R'),\\
&\quad map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleBody^{-}(R,C,B) &\leftarrow ruleBody^{-}(R',C',B'),\ not\ modifiedRule(R'),\\
&\quad map(R,R'),\ map(C,C'),\ map(B,B').\\
ruleHead(R,C,B) &\leftarrow ruleHead(R',C',B'),\\
&\quad not\ modifiedRule(R'),\ map(R,R'),\ map(C,C'),\ map(B,B').
\end{aligned} \tag{9}$$

For actions that operate on modifications or sets of modifications, we must not use rules that have been changed by rule modifying actions. Therefore we next introduce an IMPL fragment that removes such rules from modifications specified by the extension of $cleanMod$, Identifiers for the changed modifications are created using auxiliary constant cm_{id}.

$$\begin{aligned}
cleanedModId(M',M) &\leftarrow cleanMod(M),\ \#id_2(M',cm_{id},M);\\
modAdd(M',R) &\leftarrow modAdd(M,R'),\ cleanedModId(M',M),\ map(R,R'),\\
&\quad not\ modifiedRule(R');\\
modDel(M',R) &\leftarrow modDel(M,R'),\ cleanedModId(M',M),\ map(R,R').
\end{aligned} \tag{10}$$

We trigger cleaning for every modification that is used by $@guiEditMod$ or $@applyMod$.

$$\begin{aligned}
cleanMod(M) &\leftarrow ra_{guiEditMod}(M).\\
cleanMod(M) &\leftarrow ra_{applyMod}(M).
\end{aligned} \tag{11}$$

The following fragment cleans sets of modifications similar as (10).

$$\begin{aligned}
cleanedModSetId(MS',MS) &\leftarrow cleanModSet(MS),\ \#id_2(MS',csm_{id},MS).\\
cleanMod(M) &\leftarrow modset(MS,M),\ cleanModSet(MS).\\
modset(MS',M') &\leftarrow cleanedModId(M',M), modset(MS,M),\\
&\quad cleanedModSetId(MS',MS).
\end{aligned} \tag{12}$$

We trigger cleaning of sets of modifications for all sets of modifications used by $@guiSelectMod$.

$$cleanModSet(MS) \leftarrow ra_{guiSelectMod}(MS). \tag{13}$$

For comfort actions that project modifications and sets of modifications, we need a projection feature in the rewriting. Additionally we must remove rules that have been changed by rule modifications.[7]

The following IMPL fragment projects modifications specified by the extension of $projectMod$, removes all bridge rules that have been modified from these modifications and maps rule identifier constants back to their original identifiers. We trigger this by actions $@guiEditModAtContext$ and $@applyModAtContext$.

$$\begin{aligned}
projectedModId(M', M, C) &\leftarrow projectMod(M, C),\ \#id_3(M', pm_{id}, M, C);\\
modAdd(M', R) &\leftarrow modAdd(M, R'),\ ruleHead(R', C, S),\\
&\quad projectedModId(M', M, C),\ map(R, R'),\\
&\quad not\ modifiedRule(R');\\
modDel(M', R) &\leftarrow modDel(M, R'), ruleHead(R', C, S),\\
&\quad projectedModId(M', M, C), map(R, R').\\
projectMod(M, C) &\leftarrow ra_{guiEditModAtContext}(M, C).\\
projectMod(M, C) &\leftarrow ra_{applyModAtContext}(M, C).
\end{aligned} \tag{14}$$

The next IMPL fragment achieves the same for sets of modifications, triggered by $@guiSelectModAtContext$.

$$\begin{aligned}
projectedModSetId(MS', MS, C) &\leftarrow projectModSet(MS, C),\ \#id_3(MS', psm_{id}, MS, C);\\
projectMod(M, C) &\leftarrow modset(MS, M), projectModSet(MS, C);\\
modset(MS', M') &\leftarrow projectedModId(M', M, C), modset(MS, M),\\
&\quad projectedModSetId(MS', MS, C).\\
projectModSet(MS, C) &\leftarrow ra_{guiSelectModAtContext}(MS, C).
\end{aligned} \tag{15}$$

Program fragments (5) to (15) prepared everything for executing core actions which realize the original comfort actions.

We trigger action $@delRule$ for every rule that was removed by $@delRule$ in the original program, for every rule that was removed by a cleaned $@applyMod$, for every rule that was removed by a projected $@applyModAtContext$, and for every rule that was modified by a rule modifying action. (We use the primed rule identifiers to remove the *original* rules.)

$$\begin{aligned}
@delRule(R') &\leftarrow ra_{delRule}(R').\\
@delRule(R') &\leftarrow ra_{applyMod}(M'),\ modDel(M', R').\\
@delRule(R') &\leftarrow ra_{applyModAtContext}(M', C'),\\
&\quad projectedModId(M'', M', C'),\ modDel(M'', R').\\
@delRule(R') &\leftarrow modifiedRule(R').
\end{aligned} \tag{16}$$

We trigger action $@addRule$ for every rule that was added and not modified, for every rule of an applied and cleaned modification, for every rule of an applied and projected modification, and for every rule that was modified. We map to the original rule identifiers to obtain an identifier stable rewriting. (This is achieved, because rules that are modified are removed with their primed identifiers, while their modified form is added using the original identifiers.)

$$\begin{aligned}
@addRule(R) &\leftarrow ra_{addRule}(R'),\ map(R, R'),\ not\ modifiedRule(R').\\
@addRule(R) &\leftarrow ra_{applyMod}(M'),\ cleanedModId(M'', M'),\ modAdd(M'', R).\\
@addRule(R) &\leftarrow ra_{applyModAtContext}(M', C'),\\
&\quad projectedModId(M'', M', C'),\ modAdd(M'', R).\\
@addRule(R) &\leftarrow modifiedRule(R'),\ map(R, R').
\end{aligned} \tag{17}$$

[7] Examples 3.7 and 3.9 already hinted at how to realize $@applyMod$ and $@applyModAtContext$, however these examples do not guarantee stable identifiers, therefore we here give extended rewritings.

Finally we realize cleaned and projected GUI actions by activating core GUI actions.[8]

$$\begin{aligned}
@guiSelectMod(M') &\leftarrow cleanedModSetId(M', M), ra_{guiSelectMod}(M).\\
@guiSelectMod(MS') &\leftarrow ra_{guiSelectModAtContext}(MS, C),\\
&\quad\ projectedModSetId(MS', MS, C).\\
@guiEditMod(M') &\leftarrow cleanedModId(M', M), ra_{guiEditMod}(M).\\
@guiEditMod(M') &\leftarrow ra_{guiEditModAtContext}(M, C), projectedModId(M', M, C).
\end{aligned} \tag{18}$$

This completes P_{Aux} (which consists of (5) to (18)). We formally define our rewriting as follows.

Definition 6.2. *Given an* IMPL *policy P and a policy input EDB_M the* rewritten policy $tr(P \cup EDB_M)$ *is defined as*

$$tr(P \cup EDB_M) = tr_{repl}(tr_{act}(P \cup EDB_M)) \cup P_{Aux}.$$

Using this rewriting, we can realize IMPL by implementing the IMPL core fragment.

Proposition 6.3. *Given an MCS M, an* IMPL *policy P, and a policy input EDB_M wrt. M, a MCS M' is an admissible modification of M wrt. P and EDB_M iff M' is an admissible modification of M wrt. $tr(P \cup EDB_M)$.*

Proof. We first investigate the internal structure of policy $tr(P \cup EDB_M) = P_{tr} \cup P_{Aux}$ where $P_{tr} = tr_{repl}(tr_{act}(P \cup EDB_M))$. P_{tr} contains no constants from C_M (they all have been replaced). P_{Aux} contains in its rule heads either actions, or atoms with predicates that are not critical and therefore disjoint with predicates in P_{tr}, or atoms with reserved predicates and constants from $C_M \cup C_{id}$. P_{Aux} contains no constraints and no cyclic dependencies (neither positive nor including default negation). Therefore P_{tr} does not depend on P_{Aux}, and we can split the policy and obtain $I_{tr} \cup I_{Aux} \in \mathcal{AS}(tr(P \cup EDB_M))$ iff $I_{tr} \in \mathcal{AS}(P_{tr})$ and $I_{tr} \cup I_{Aux} \in \mathcal{AS}(P_{Aux} \cup I_{tr})$. Due to the definition of tr_{repl} and tr_{act} we additionally have $I_{tr} \in \mathcal{AS}(P_{tr})$ iff $tr^{-1}_{repl}(tr^{-1}_{act}(I_{tr})) \in \mathcal{AS}(P \cup EDB_M)$. (I.e., translated policy answer sets directly correspond with policy answer sets of the translation.) I_{tr} contains no actions, because P_{tr} contains no actions (only replacements). To show the result, it remains to show that M' is a materialization of an answer set $I_C \in \mathcal{AS}(P \cup EDB_M)$ iff M' is a materialization of an answer set $I_{Aux} \cup tr_{repl}(tr_{act}(I_C)) \in \mathcal{AS}(P_{Aux} \cup tr_{repl}(tr_{act}(I_C)))$. As the translation removes actions, this amounts to showing that a materialization of actions in I_C is a materialization of actions in I_{Aux} where I_{Aux} contains atoms derived by P_{Aux} from $tr_{repl}(tr_{act}(I_C))$. Therefore we must show that $(br(M) \setminus \mathcal{R} \cup \mathcal{A}) \setminus Rules \cup \mathcal{M}$ (see Definition 3.12) yields the same result for I_C and for I_{Aux}. In the following we subscript sets in the above formula by the policy answer set that was used to create the respective set. Using this new notation, we need to show that

$$(br(M) \setminus \mathcal{R}_{I_{Aux}} \cup \mathcal{A}_{I_{Aux}}) \setminus Rules_{I_{Aux}} \cup \mathcal{M}_{I_{Aux}} = (br(M) \setminus \mathcal{R}_{I_C} \cup \mathcal{A}_{I_C}) \setminus Rules_{I_C} \cup \mathcal{M}_{I_C}.$$

As P_{Aux} contains only core actions, I_{Aux} contains only core actions, accordingly $Rules_{I_{Aux}} = \mathcal{M}_{I_{Aux}} = \emptyset$ and we need to show that $\mathcal{R}_{I_{Aux}} = \mathcal{R}_{I_C} \cup Rules_{I_C}$ and $\mathcal{A}_{I_{Aux}} = \mathcal{A}_{I_C} \setminus Rules_{I_C} \cup \mathcal{M}_{I_C}$.

We next show properties of answer sets of P_{Aux}. Given I_C, as P_{Aux} is stratified and contains no constraints, an answer set $I_{Aux} \cup tr_{repl}(tr_{act}(I_C)) \in \mathcal{AS}(P_{Aux} \cup tr_{repl}(tr_{act}(I_C)))$ always exists, is unique, and I_{Aux} has the following properties.

[8] The IMPL core actions $@guiEditMod$ and $@guiSelectMod$ cannot be realized by the simple rule $@\alpha(\mathbf{t}) \leftarrow ra_\alpha(\mathbf{t})$, because our usage of $@addRule$ and $@delRule$ for realizing rule modifying actions would lead to incorrect semantics.

(i) Due to (6), $modifiedRule(r') \in I_{Aux}$ iff $r \in Rules_{I_C}$.
(ii) Due to (7), $add^+(r', c', b') \in I_{Aux}$ iff $(c : b) \in \mathcal{A}_{I_C,r}$; $add^-(r', c', b') \in I_{Aux}$ iff **not** $(c : b) \in \mathcal{A}_{I_C,r}$; $del^+(r', c', b') \in I_{Aux}$ iff $(c : b) \in \mathcal{R}_{I_C,r}$; and $del^-(r', c', b') \in I_{Aux}$ iff **not** $(c : b) \in \mathcal{R}_{I_C,r}$.
(iii) Due to (ii) and (8), for every bridge rule $q \in \mathcal{M}_{I_C}$ identified by r in I_C, we have $q = rule_{I_{Aux}}(r)$, i.e., the modified bridge rule q is represented in I_{Aux} and identified by its original constant r.
(iv) Due to (9), for every bridge rule $q \in \mathcal{A}_{I_C} \setminus Rules_{I_C}$ we have $q = rule_{I_{Aux}}(r)$, i.e., q is represented in I_{Aux} and identified by constant r.
(v) Due to (10) and (11), for every modification $(A, R) = mod_{I_C}(m)$ such that $@guiEditMod(m) \in I_C$ or $@applyMod(m) \in I_C$, we have $(A \setminus Rules_{I_C}, R) = mod_{I_{Aux}}(c_{cm_{id},m})$ with $c_{cm_{id},m} \in \mathcal{I}_{id}$ and $cleanedModId(c_{cm_{id},m}, m) \in I_{Aux}$.
(vi) Due to (10), (12), and (13), for every modification set $\{(A_1, R_1), \ldots, (A_k, R_k)\} = modset_{I_C}(ms)$ such that $@guiSelectMod(ms) \in I_C$, we have $\{(A_1 \setminus Rules_{I_C}, R_1), \ldots, (A_k \setminus Rules_{I_C}, R_k)\} = modset_{I_{Aux}}(c_{csm_{id},ms})$ with $c_{csm_{id},ms} \in \mathcal{I}_{id}$ and $cleanedModSetId(c_{csm_{id},ms}, ms) \in I_{Aux}$.
(vii) Due to (14), for every modification $mod_{I_C}(m)$ and context identifier c such that $(A, R) = mod_{I_C}(m)|_c$ and $@guiEditModAtContext(m, c) \in I_C$ or $@applyModAtContext(m, c) \in I_C$, we have $(A \setminus Rules_{I_C}, R)|_c = mod_{I_{Aux}}(c_{pm_{id},m,c})$ with $c_{pm_{id},m,c} \in \mathcal{I}_{id}$ and $projectedModId(c_{pm_{id},m,c}, m, c) \in I_{Aux}$.
(viii) Due to (15), for every modification set $modset_{I_C}(ms)$ and context identifier c such that $\{(A_1, R_1), \ldots, (A_k, R_k)\} = modset_{I_C}(ms)|_c$ and $@guiSelectModAtContext(ms, c) \in I_C$, we have $\{(A_1 \setminus Rules_{I_C}, R_1)|c, \ldots, (A_k \setminus Rules_{I_C}, R_k)|c\} = modset_{I_{Aux}}(c_{psm_{id},ms,c})$ with $c_{psm_{id},ms,c} \in \mathcal{I}_{id}$ and $projectedModSetId(c_{psm_{id},ms,c}, ms, c) \in I_{Aux}$.

We first show correctness for non-GUI actions, indicated by superscript ng, and then for GUI actions, indicated by superscript gui. Due to (16), all rules in $\mathcal{R}^{ng}_{I_C}$ (from $@delRule$, $@applyMod$, and $@applyModAtContext$, see also (v) and (vii)) and all rules in $Rules_{I_C}$ (see also (i)) are deleted in I_{Aux} using $@delRule$, and no other rules are deleted due to (16). Therefore, $\mathcal{R}^{ng}_{I_{Aux}} = \mathcal{R}^{ng}_{I_C} \cup Rules_{I_C}$. Due to (17), those rules in $\mathcal{A}^{ng}_{I_C}$ which are not in $Rules_{I_C}$ (from $@addRule$, $@applyMod$, and $@applyModAtContext$, see also (v) and (vii)) and all rules in $\mathcal{M}_{I_C}$ (see also (ii) and (iii)) are added in I_{Aux} using $@addRule$, and no other rules are added due to (17). Therefore, $\mathcal{A}^{ng}_{I_{Aux}} = \mathcal{A}^{ng}_{I_C} \setminus Rules_{I_C} \cup \mathcal{M}_{I_C}$. Note that these rules are added using their original identifiers (see (iii) and (iv)) which makes our rewriting identifier-neutral wrt. created rules.

It remains to show, that also GUI actions are realized correctly by the rewriting, i.e., that $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and that $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus Rules_{I_C}$. As semantics of user interaction is nondeterministic, it is not possible (and makes no sense) to directly prove the above equalities. Instead, we split the rest of the proof into two directions: we prove that, given I_C and some effect of executing GUI actions in I_C, it is possible to achieve the same effect from executing GUI actions in I_{Aux}, and vice versa.

($\Rightarrow$) Given policy answer set $I_C \in \mathcal{AS}(P \cup EDB_M)$, and the accumulated effect $\mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_C}$ of GUI actions in I_C, the corresponding I_{Aux} (with $I_{Aux} \cup tr_{repl}(tr_{act}(I_C)) \in \mathcal{AS}(P_{Aux} \cup tr_{repl}(tr_{act}(I_C)))$ as above) contains due to (18) a set of GUI actions that corresponds to GUI actions in I_C as follows: $@guiEditMod(m) \in I_C$ is mapped to a modification editor over $(A \setminus Rules_{I_C}, R)$ (see (v)); $@guiSelectMod(ms) \in I_C$ is mapped to a modification selection over $\{(A_1 \setminus Rules_{I_C}, R_1), \ldots\}$ (see (vi)); $@guiEditModAtContext(m, c)$ and $@guiSelectModAtContext(ms, c)$ are mapped analogously, always removing $Rules_{I_C}$ from the first component of all modifications at hand. As GUI actions in I_{Aux} correspond to GUI actions in I_C with all rules from $Rules_{I_C}$ removed, it is clearly possible to obtain a GUI action effect such that $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus Rules_{I_C}$.

($\Leftarrow$) For every GUI action in I_{Aux} there is a corresponding GUI action in I_C which contains the same modification(s) as the action in I_{Aux} and probably contains some more rules from $Rules_{I_C}$. However, as

GUI actions accumulate in $\mathcal{R}_{I_C}$ and in $\mathcal{A}_{I_C}$ and $Rules_{I_C}$ is always subtracted from $\mathcal{R}_{I_C}$ and from $\mathcal{A}_{I_C}$ to obtain an admissible modification, a rule from $Rules_{I_C}$ which is added by an effect of a GUI actions in I_C is not added in the materialization of the overall accumulated action effects. Therefore here also $\mathcal{R}^{gui}_{I_{Aux}} = \mathcal{R}^{gui}_{I_C}$ and $\mathcal{A}^{gui}_{I_{Aux}} = \mathcal{A}^{gui}_{I_C} \setminus Rules_{I_C}$ and the result holds. □

7 Conclusion

We have motivated the need for an inconsistency management policy language for MCSs, have introduced the IMPL language, its syntax and semantics, discussed modes of reasoning with IMPL, and how to realize IMPL using acthex and a rewriting from the full IMPL language to a core fragment of IMPL.

Related Work

Related to IMPL is the action language *IMPACT* [Subrahmanian et al., 2000], which is a declarative formalism for actions in distributed and heterogeneous multi-agent systems. *IMPACT* is a very rich general purpose formalism, which however is more difficult to manage compared to the special purpose language IMPL. Furthermore, user interaction as in IMPL is not directly supported in *IMPACT*; nevertheless most parts of IMPL could be embedded in *IMPACT*.

In the fields of access control, e.g., surveyed in [Bonatti et al., 2009], and privacy restrictions [Duma et al., 2007], policy languages have also been studied in detail. As a notable example, $\mathcal{PDL}$ [Chomicki et al., 2000] is a declarative policy language based on logic programming which maps events in a system to actions. $\mathcal{PDL}$ is richer than IMPL concerning action interdependencies, whereas actions in IMPL have a richer internal structure than $\mathcal{PDL}$ actions. Moreover, actions in IMPL depend on the content of a policy answer set. Similarly, inconsistency analysis input in IMPL has a deeper structure than events in $\mathcal{PDL}$.

In the context of relational databases, logic programs have been used for specifying repairs for databases that are inconsistent wrt. a set of integrity constraints [Greco et al., 2003, Eiter et al., 2008a, Marileo and Bertossi, 2010]. These approaches may be considered fixed policies without user interaction, like an IMPL policy simply applying diagnoses in a homogeneous MCS. Note however, that an important motivation for developing IMPL is the fact that automatic repair approaches are not always a viable option for dealing with inconsistency in a MCS.

Active integrity constraints (AICs) [Caroprese et al., 2009, Caroprese and Truszczynski, 2008a,b] and *inconsistency management policies (IMPs)* [Martinez et al., 2008] have been proposed for specifying repair strategies for inconsistent databases in a flexible way. AICs extend integrity constraints by introducing update actions, for inserting and deleting tuples, to be performed if the constraint is not satisfied. On the other hand, an IMP is a function which is defined wrt. a set of functional dependencies mapping a given relation R to a 'modified' relation R' obeying some basic axioms.

Although suitable IMPL policy encodings can mimic database repair programs—AICs and (certain) IMPs—for specific classes of integrity constraints, there are fundamental conceptual differences between IMPL and the above approaches to database repair. Most notably, IMPL policies aim at restoring consistency by modifying bridge rules, which leaves knowledge bases unchanged; opposed to that, IMPs and AICs consider a set of fixed constraints and repair the database. Another difference is, that IMPL policies are able to operate on heterogeneous knowledge bases and may involve user interaction.

Ongoing and Future Work.

Regarding an actual prototype implementation of IMPL, we are currently working on improvements of acthex which are necessary for realizing IMPL using the rewriting technique described in Section 5.2. In particular, this includes the generalization of taking into account the environment in external atom evaluation. Other improvements concern the support for implementing model and execution schedule selection functions.

An important feature of IMPL is the user interface for selecting or editing modifications. There the number of displayed modifications might be reduced considerably by grouping modifications according to nonground bridge rules. This would lead to a considerable improvement of usability in practice.

Also, we currently just consider bridge rule modifications for system repairs, therefore an interesting issue for further research is to drop this convention. A promising way to proceed in this direction is to integrate IMPL with recent work on managed MCSs [Brewka et al., 2011], where bridge rules are extended such that they can arbitrarily modify the knowledge base of a context and even its semantics. Accordingly, IMPL could be extended with the possibility of using management operations on contexts in system modifications.

References

F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, 2003.

S. Basol, O. Erdem, M. Fink, and G. Ianni. HEX programs with action atoms. In *ICLP*, pages 24–33, 2010.

M. Bögl, T. Eiter, M. Fink, and P. Schüller. The MCS-IE system for explaining inconsistency in multi-context systems. In *JELIA*, pages 356–359, 2010.

P. A. Bonatti, J. L. D. Coi, D. Olmedilla, and L. Sauro. Rule-based policy representations and reasoning. In *REWERSE*, volume 5500, pages 201–232. Springer, 2009.

G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 385–390, 2007.

G. Brewka, F. Roelofsen, and L. Serafini. Contextual default reasoning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 268–273, 2007.

G. Brewka, T. Eiter, M. Fink, and A. Weinzierl. Managed multi-context systems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 786–791, 2011.

F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1997.

L. Caroprese and M. Truszczynski. Declarative semantics for active integrity constraints. In *ICLP*, volume 5366, pages 269–283, 2008a.

L. Caroprese and M. Truszczynski. Declarative semantics for revision programming and connections to active integrity constraints. In *JELIA*, volume 5293, pages 100–112, 2008b.

L. Caroprese, S. Greco, and E. Zumpano. Active integrity constraints for database consistency maintenance. *IEEE Trans. Knowl. Data Eng*, 21(7):1042–1058, 2009.

J. Chomicki, J. Lobo, and S. A. Naqvi. A logic programming approach to conflict resolution in policy management. In *KR*, pages 121–132, 2000.

C. Duma, A. Herzog, and N. Shahmehri. Privacy in the semantic web: What policy languages have to offer. In *POLICY*, pages 109–118, 2007.

T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 90–96, Denver, USA, 2005.

T. Eiter, M. Fink, G. Greco, and D. Lembo. Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.*, 33(2), 2008a.

T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495 – 1539, 2008b.

T. Eiter, M. Fink, P. Schüller, and A. Weinzierl. Finding explanations of inconsistency in nonmonotonic multi-context systems. In *KR*, pages 329–339, 2010a.

T. Eiter, M. Fink, and A. Weinzierl. Preference-based inconsistency assessment in multi-context systems. In *JELIA*, LNAI, pages 143–155, 2010b.

T. Eiter, M. Fink, G. Ianni, and P. Schüller. The IMPL policy language for managing inconsistency in multi-context systems. In *Postproceedings of the International Conference on Applications of Declarative Programming and Knowledge Management (INAP) and the Workshop on Logic Programming (WLP)*, 2012. To appear.

W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

M. Fink, L. Ghionna, and A. Weinzierl. Relational information exchange and aggregation in multi-context systems. In J. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning, 11th International Conference (LPNMR 2011)*, pages 120–133, 2011.

M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics, or: How we can do without modal logics. *Artificial Intelligence*, 65(1):29–70, 1994.

G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng*, 15(6):1389–1408, 2003.

M. C. Marileo and L. E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng*, 69(6):545–572, 2010.

M. V. Martinez, F. Parisi, A. Pugliese, G. I. Simari, and V. S. Subrahmanian. Inconsistency management policies. In *KR*, pages 367–377, 2008.

V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.