# Towards a Policy Language for Managing Inconsistency in Multi-Context Systems*

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

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

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

**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 thus propose a declarative policy language for inconsistency management in multi-context systems. We define syntax and semantics, give methodologies for applying the language in real world applications, and discuss a possible implementation.

## 1 Introduction

Powerful knowledge based applications can be built by interlinking smaller existing knowledge based systems. Multi-context systems (MCSs) [5], based on [18, 6], are a generic formalism that captures heterogeneous knowledge bases (contexts) which are interlinked using (possibly nonmonotonic) bridge rules.

The advantage of building a system from smaller parts however 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 [13]: 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 conflict with some allergy of the patient. Here attempting an automatic repair may not be a viable option: a doctor should inspect the situation and make a decision.

* This research has been supported by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

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 IMPL, a declarative policy language providing a means to specify inconsistency management strategies for MCSs. Briefly, our contributions are as follows.

- We define the syntax of IMPL, inspired by answer set programs (ASPs) [17]. In particular, we specify *input for policy reasoning*, in terms of reserved predicates. These predicates encode inconsistency analysis results in terms of [13]. Furthermore, we specify *action predicates that can be derived in rules*. Actions provide a means to counteract inconsistency by modifying the MCS, and may involve interaction with a human operator.
- We define IMPL semantics in terms of a three-step process which calculates models of a policy, then determines effects of actions which are present in such 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 useful language extensions and a potential realization using the acthex formalism [2].

## 2 Preliminaries

**Multi-context systems (MCSs).** A heterogeneous nonmonotonic MCS [5] 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 1.* For *propositional logic* $L_{prop}$ under the closed world assumption over signature $\Sigma$, $\mathbf{KB}$ is the set of propositional formulas over $\Sigma$; $\mathbf{BS}$ is the set of deductively closed sets of propositional $\Sigma$-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 $L = (L_1, \dots, L_n)$ be a tuple of logics. An $L_k$-bridge rule $r$ over $L$ is of the form

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

where $k$ and $c_i$ are context identifiers, i.e., integers in the range $1, \dots, n$, $p_i$ 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 $r$.

A multi-context system $M = (C_1, \dots, C_n)$ 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, $kb_i \in \mathbf{KB}_i$

ISSN 1613-0073 © 2011 for the individual papers by the papers' authors. 

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 $kb_i \cup H \in \mathbf{KB}_{L_i}$. Similar to $IN_i$, $OUT_i$ denotes *output beliefs* of context $C_i$, which are those beliefs $p$ in $\mathbf{BS}_i$ that occur in some bridge rule body in $br_M$ as "$(i{:}p)$" or as "$\mathbf{not}\ (i{:}p)$" (see also [13]). By $br_M = \bigcup_{i=1}^{n} br_i$ we denote the set of all bridge rules of $M$, and by $ctx_M = \{C_1, \ldots, C_n\}$ the set of all contexts of $M$.

*Example 2 (generalized from [13]).* 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.

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

Context $C_{db}$ uses propositional logic (see Example 1) and provides 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 must be 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. This context is based on $\mathcal{AL}$, a basic description logic [1]: $\mathbf{KB}_{onto}$ 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 that suggests a medication using the $give$ predicate.

We next give schemas for bridge rules of $M_1$.

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

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* [5] selects certain belief states of a MCS $M=(C_1,\ldots,C_n)$ as acceptable. A *belief state* is a sequence $S=(S_1,\ldots,S_n)$, 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: $S_i \in \mathbf{ACC}_i(kb_i \cup \{hd(r) \mid r \in app(br_i,S)\})$.

In our running example we use bridge rules with variables, here we disregard the issue of instantiating these rules [16]. We denote by $r[X|c]$ the ground version of bridge rule $r$ where variable $X$ has been substituted by constant $c$.

*Example 3 (ctd).* MCS $M_1$ has one equilibrium $S=(S_{db},S_{lab},S_{onto},S_{dss})$, where $S_{db}=kb_{db}$, $S_{lab}=\{customer(sue, 02/03/1985), test(sue, xray, pneumonia)\}$, $S_{onto}=\{Pneumonia(sue)\}$, and $S_{dss}=\{need(sue, ab), give(sue, ab2), \neg give(sue, ab1)\}$. Moreover, the following bridge rules of $M_1$ are applicable under $S$: $r_1[Id|sue, Birthday|02/03/1985]$, $r_2[Id|sue]$, and $r_4[Id|sue]$. □

**Explaining Inconsistency in MCSs.** *Inconsistency* in a MCS is the lack of an equilibrium[13]. Note that no equilibrium may exist even if all contexts are 'paraconsistent' in the sense that $\mathbf{ACC}$ is never empty. 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* [13], 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. 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 already present in $M$ or by modifying rules in $E_2$ (in particular making them unconditional). See [13] for formal definitions of these notions, relationships between them, and more background discussion.

*Example 4 (ctd).* Consider a MCS $M_2$ given by our example MCS $M_1$ with different knowledge bases $kb_{db}=\{person(sue, 03/02/1985), allergy(sue, ab1)\}$ (i.e., month and day of the birth date are swapped) and $kb_{lab}$ containing an additional finding $test(sue, bloodtest, m1)$. $M_2$ is inconsistent with $E^{\pm}_m(M_2)=\{e_1,e_2\}$ and $D^{\pm}_m(M_2)=\{d_1,d_2,d_3,d_4\}$, where $e_1=(\{r'_1\},\emptyset)$, $e_2=(\{r'_2,r'_3,r'_5\},\{r'_6\})$, $d_1=(\{r'_1,r'_2\},\emptyset)$, $d_2=(\{r'_1,r'_3\},\emptyset)$, $d_3=(\{r'_1,r'_5\},\emptyset)$, $d_4=(\{r'_1\},\{r'_6\})$, and where $r'_1=r_1[Id|sue, Birthday|03/02/1985]$, and $r'_j=r_j[Id|sue]$, $j \in \{2,3,5,6\}$. $E^{\pm}_m(M_2)$ characterizes two inconsistencies in $M_2$, namely $e_1$: $C_{lab}$ does not accept any belief set because constraint

$customer(Id, X) \wedge customer(Id, Y) \rightarrow X = Y$ is violated; furthermore $e_2$: if we assume $e_1$ is repaired, then $C_{onto}$ accepts $AtypPneumonia(sue)$ in its unique accepted belief set, therefore $r_5[Id|sue]$ imports the need for $ab1$ into $C_{dss}$ which makes $C_{dss}$ inconsistent due to Sue's allergy. □

## 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 5 (ctd).* Repairing $e_1$ by 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 a declarative *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 can be (but not necessarily are) part of $br_M$, and which are syntactially 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 $M$.

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

### 3.1 Syntax.

We assume disjoint sets $C$, $V$, $Ord$, $Built$, and $Act$, of constants, variables, ordinary predicate names, built-in predicate names, and action names, resp. An *atom* is of the form $p(t_1, \ldots, t_k)$, $0 \leq k$, $t_i \in T$, where the set of *terms* $T$ is defined as $T = C \cup V$, and $p$ is a (built-in or ordinary) predicate name or an action name. As usual, an atom is ground if $t_i \in C$ for $0 \leq i \leq k$. The set $A_{Act}$ of *action atoms* has $p \in Act$, while $A_{Ord}$, respectively $A_{Built}$, denotes the set of *ordinary atoms* having $p \in Ord$, respectively the set of *built-in* atoms with $p \in Built$.

ISSN 1613-0073 © 2011 for the individual papers by the papers' authors. 

**Definition 1.** *An* IMPL policy *is a set of rules of the form*

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

*where $h$ is an atom from $A_{Ord} \cup A_{Act}$ or $\bot$, every $a_i$ is from $A_{Ord} \cup A_{Built}$, for $1 \leq i \leq k$, 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*.

An IMPL policy $P$ is intended to be evaluted provided some input $I_M$ respresenting relevant information about a given MCS $M$. The idea is that its evaluation yields certain actions to be taken upon inconsistency, which effect modifications of the MCS with the goal of restoring consistency. For representing basic inputs and actions, we consider specific predicate and action names. Corresponding atoms follow a particular syntax and semantics. We first present their syntax and provide intuitions of their semantics.

**Reserved Predicates.** Atoms with reserved predicate names provide a policy with information about the system at hand, and about results of inconsistency analysis on that system. Essentially, these atoms describe bridge rules, diagnoses, and explanations of a given MCS. Note that elements of these descriptions, like contexts, bridge rules, beliefs, etc., are assumed to be represented by suitable constants. For brevity, when referring to an element represented by a constant $c$, we identify it with the constant (omitting 'represented by constant').

- $ruleHead(r, c, s)$ denotes that the head of bridge rule $r$ at context $c$ is the formula $s$.
- $ruleBody^+(r, c, b)$ (resp., $ruleBody^-(r, c, b)$) denotes that bridge rule $r$ contains body literal '$(c : b)$' (resp., body literal '**not** $(c : b)$').
- $modAdd(m,r)$ (resp., $modDel(m,r)$) denotes that modification $m$ adds (resp., deletes) bridge rule $r$ ($r$ 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 $(E_1, E_2) \in E^{\pm}_m(M)$ identified by constant $e$ contains bridge rule $r \in E_1$ (resp., $E_2$).
- $member(ms, m)$ denotes that modification $m$ belongs to a set of modifications $ms$.
- $\#id(t, c, i)$ is a builtin predicate with the intention to handle the assignment of 'fresh' constants (not appearing in the input to a policy) as identifiers more easily (i.e., it facilitates limited value invention). Intuitively, $\#id(t, c, i)$ is true iff $c$ is a constant, $i$ is a non-negative integer constant (from a fixed, finite range, see also Sec. 3.2), and $t = c_i$, for a particular constant $c_i$.

Further knowledge used as input for policy reasoning can easily be defined using additional (auxiliary) predicates. For instance, to encode preference relations (e.g., as in [14]) 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 (ordinary predicates) or provided by the implementation (built-in predicates), i.e., the 'solver' used to evaluate the policy.

As for notation, given a set of ground atoms $I_M$ and a constant $c$, we denote by $rule_I(c)$ the *bridge rule identified by* $c$ *and characterized in* $I$ by $ruleHead$ and $ruleBody^{\pm}$ atoms. Similarly, we denote by $modification_I(c) = (A, R)$ the modification $c$ characterized in $I$ by $modAdd$, $modDel$, $ruleHead$ and $ruleBody^{\pm}$ atoms.

*Example 6 (ctd).* Explanation $e_1$ in $M_2$ can be represented as $I_{e_1} = \{ruleHead(r'_1, c_{lab}, \text{`}customer(sue, 03/02/1985)\text{'}), ruleBody^+(r'_1, c_{db}, \text{`}person(sue, 03/02/1985)\text{'}), explNeed(e_1, r'_1)\}$. Then, $rule_{I_{e_1}}(r'_1)$ denotes bridge rule $r'_1$. □

Note, that reserved predicates, except for the built-in $\#id$, are also allowed to occur in the head of policy rules.

**Actions.** Let us turn to the syntax and intuitive semantics of *action atoms* built from predefined action names. (We prefix action names from $Act$ with @). We assume that actions are independent from one another and each action yields a modification of the MCS. This modification can depend on external input, e.g., obtained by user interaction.

We distinguish three categories of actions: (a) actions that affect individual bridge rules, (b) actions that affect multiple bridge rules, and (c) actions that involve user interaction (and affect individual or multiple bridge rules).

The following actions affect individual bridge rules.

- $@delRule(r)$ removes bridge rule $r$ from the MCS ($r$ is represented using $ruleHead$ and $ruleBody$).
- $@addRule(r)$ adds bridge rule $r$ to the MCS.
- $@addRuleCondition^+(r, c, b)$ (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.

The following actions (potentially) affect multiple bridge rules.

- $@applyMod(m)$ applies modification $m$ to the MCS.
- $@applyModAtContext(m, c)$ applies those modifications of $m$ to the MCS which modify bridge rules at context $c$. Subsequently, we call this the projection of modification $m$ to context $c$.

The following actions involve user interaction.

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

The *core fragment* of IMPL consists of actions $@delRule$, $@addRule$, $@guiSelectMod$, and $@guiEditMod$, which are sufficient for realizing all actions described above. Actions not in the core fragment exist for convenience of use: they provide a means for projection and modifying only parts of rules which otherwise would need to be encoded using auxiliary predicates and core actions.

*Example 7.* Given a set of ground atoms $I_M$, making bridge rules $r$ with $foo(r) \in I_M$ unconditional can be achieved using a single rule with the action:

$@makeRuleUnconditional(R) \leftarrow foo(R).$

The following IMPL policy fragment achieves the same using only core actions:

% associate new constant with R to get identifier for rule derived from R
$aux(Rid, R) \leftarrow foo(R), \#id(Rid, R, 1).$
% copy existing rule heads (don't copy body literals)
$ruleHead(Rid, C, S) \leftarrow ruleHead(R, C, S), aux(Rid, R).$
% trigger actions
$@delRule(R) \leftarrow aux(Rid, R).$
$@addRule(Rid) \leftarrow aux(Rid, R).$
(Here and in the following, lines starting with % indicate comments.) □

*Example 8 (ctd).* Figure 1 shows three policies for managing inconsistency in $M_2$. Let us briefly illustrate their intended behaviour (before turning to a formal definition of semantics next). $P_1$ 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. Intuitively, applying $P_1$ to $M_2$ yields $M_3$ (any chosen diagnosis removes exactly $r'_1$ at $C_{lab}$): $M_3$ is still inconsistent due to $e_2$ but no longer due to $e_1$. $P_2$ extends $P_1$ by adding an 'inconsistency alert formula' $alert$ to $C_{lab}$ iff an inconsistency was automatically repaired at $C_{lab}$. Finally, $P_3$ is a different approach which displays a choice of all minimal diagnoses to the user if at least one diagnosis exists.

Note, that we did not combine automatic actions and user-interactions; this would require more involved policies or an iterative methodology (cf. Sec. 4). □

### 3.2 Semantics

The semantics of IMPL is defined in three steps: (i) *policy answer sets* are defined for a policy together with some input, each answer set represents a set of actions (to be executed); (ii) every action has associated effects in terms of a modification $(A, R)$, they can be nondeterministic (and thus only determined by executing the action). Finally (iii) materializing the effects of a set of (executed) actions is defined by combining their effects, i.e., modifications (componentwise union), and applying the respective changes to the MCS at hand. We next describe these steps more formally.

**Action Determination.** We define IMPL policy answer sets similar to the stable model semantics [17]. Given a MCS $M$, we associate with it a finite (nonempty) set of constants $C_M \subset C$, used as identifiers for representing its elements by means of the (ordinary) reserved predicates, as well as a finite set of integer constants $C_N \subset C$. Furthermore, let $C_{id}$ be a set of 'fresh' constants, i.e., disjoint from $C_M \cup C_N$, containing exactly one constant $c_i \in C$ for every (pair of constants) $c \in C_M$ and $i \in C_N$, and let $id$ be a fixed (built-in) one-to-one mapping from $C_M \times C_N$ to $C_{id}$.

For a policy $P$, let $cons(P) \subset C$ denote the set of constants appearing in $P$. The *policy base* $B_P$ of $P$ (given $M$) is the set of ground atoms that can be built using ordinary reserved predicate and action names, as well as any auxiliary ordinary predicate and action names appearing in $P$, and constants from $C_{P,M} = cons(P) \cup C_M \cup C_N \cup C_{id}$.

The grounding of $P$, $grnd(P)$ is given by grounding its rules wrt. $C_{P,M}$ in the usual way. An *interpretation* is a set of ground atoms $I \subseteq B_P$.

For an atom $a \in B_P$, as usual $I \models a$ iff $a \in I$, and for a ground built-in atom $a$ of the form $\#id(t_1, t_2, t_3)$, it holds that $I \models a$ iff $(t_2, t_3)$ is in the domain of $id$ and $t_1 = id(t_2, t_3)$. For a ground rule $r$, (i) $I \models B(r)$ iff $I \models a$ for every $a \in B^+(r)$ and

ISSN 1613-0073 © 2011 for the individual papers by the papers' authors. 

$$P_1 = \left\{\begin{array}{l}
\text{\% domain predicate for eplanations} \\
expl(E) \leftarrow explNeed(E,R). \quad expl(E) \leftarrow explForbid(E,R). \\
\text{\% find out whether one explanation only concerns bridge rules at } C_{lab} \\
incNotLab(E) \leftarrow explNeed(E,R), ruleHead(R,C,F), C \neq c_{lab}. \\
incNotLab(E) \leftarrow explForbid(E,R), ruleHead(R,C,F), C \neq c_{lab}. \\
incLab \leftarrow expl(E), not\ incNotLab(E). \\
\text{\% guess a unique diagnosis to apply} \\
in(D) \leftarrow not\ out(D), diag(D), incLab.\ out(D) \leftarrow not\ in(D), diag(D), incLab. \\
useOne \leftarrow in(D).\ \bot \leftarrow in(A), in(B), A \neq B.\ \bot \leftarrow not\ useOne, incLab. \\
\text{\% apply diagnosis projected to } C_{lab} \text{ if one was selected} \\
@applyModAtContext(D, c_{lab}) \leftarrow useDiag(D).
\end{array}\right\}$$

$$P_2 = \left\{\begin{array}{l}
\text{\% inconsistency alert} \\
ruleHead(r_{alert}, c_{lab}, alert). \\
@addRule(r_{alert}) \leftarrow incLab.
\end{array}\right\} \cup P_1$$

$$P_3 = \left\{\begin{array}{l}
\text{\% let the user choose from all diagnoses if there is a diagnosis} \\
member(md, X) \leftarrow diag(X). \\
@guiSelectMod(md).
\end{array}\right\}$$

Fig. 1: Example IMPL policies for managing inconsistency in $M_2$.

$I \not\models a$ for all $a \in B^-(r)$, and (ii) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. Then, $I$ is a *model* of $P$, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. The ***FLP-reduct*** [15] of $P$ wrt. an interpretation $I$, denoted $fP^I$, is the set of all $r \in grnd(P)$ such that $I \models B(r)$.

**Definition 2 (Policy Answer Sets).** *Let $P$ be an* IMPL *policy $P$, and let $I_M \subseteq B_P$ be an input for $P$. An interpretation $I \subseteq B_P$ is a* policy answer set of $P$ for $I_M$ *iff $I$ is a $\subseteq$-minimal model of $fP^I \cup I_M$. By $\mathcal{AS}(P, I_M)$ we denote the set of all policy answer sets of $P$ for $I_M$.*

**Effect Determination.** We define the effects of action predicates $@a \in Act$ by nondeterministic functions $f_{@a}$. Nondeterminism is required to due to external input, resp. user interaction. An action is evaluated wrt. a policy answer set.

**Definition 3.** *Given a policy answer set $I$, and an action $\alpha = @a(t_1, \ldots, t_k)$ in $I$, an* effect of $\alpha$ wrt. $I$*, denoted $\mathit{eff}_I(\alpha)$, is a modification $(A, R) = f_{@a}(I, t_1, \ldots, t_k)$.*

Action predicates of the IMPL core fragment have the following semantic functions. (For brevity we omit semantics of actions that are not in the core fragment.)

- $f_{@delRule}(I, r) = (\emptyset, \{rule_I(r)\})$.
- $f_{@addRule}(I, r) = (\{rule_I(r)\}, \emptyset)$.
- $f_{@guiSelectMod}(I, ms) = modification_I(m)$, for some modification $m$ such that $m \in \{m \mid member(ms, m) \in I\}$ (the user's selection after being displayed the choice among modifications $\{modification_I(m) \mid member(ms, m) \in I\}$).
- $f_{@guiEditMod}(I, m) = (A', R')$, where $(A', R')$ is some modification (resulting from the user interaction after being displayed an editor for the modification $(A, R) = modification_I(m)$).

Note, that no order of evaluating actions is specified or required. We say that a set $E_I$ of modifications is an effect set for a policy answer set $I$, iff it contains exactly one effect $\mathit{eff}_I(\alpha)$ for every $\alpha$ in $I$.

ISSN 1613-0073 © 2011 for the individual papers by the papers' authors.

**Effect Materialization** Once the effects of actions have been determined, an overall modification is calculated by componentwise union over all individual modifications. Finally, this overall modification is materialized in the MCS.

**Definition 4.** *Given a MCS $M$ and a policy answer set $I$, a* materialization of $I$ in *$M$ is a MCS $M'$ obtained from $M$ by replacing its set of bridge rules $br_M$ by the set $br_M \setminus \mathcal{R} \cup \mathcal{A}$, where $(\mathcal{A}, \mathcal{R}) = \bigcup_{(A,R) \in E_I} (A, R)$, for an effect set $E_I$ for $I$.*

Note that by definition addition of bridge rules has precedence over removal.[3]

Eventually, we can define modifications of a MCS that are accepted by a corresponding policy for managing inconsistency. Skipping a straightforward formal definition, let us say that a set of ground atoms $I_M$ is a proper input for an IMPL policy $P$ wrt. a MCS $M$, if it properly encodes $M$, $D^{\pm}_m(M)$, and $E^{\pm}_m(M)$ using reserved predicates.

**Definition 5.** *Given a MCS $M$, an* IMPL *policy $P$, and a proper input $I_M$ for $P$ wrt. $M$, then a modified MCS $M'$* is an admissible modification of $M$ wrt. policy $P$ *iff $M'$ is the materialization of some policy answer set $I \in \mathcal{AS}(P \cup I_M)$.*

*Example 9 (ctd).* For brevity we do not give a full account of a proper $I_{M_2}$. Intuitively $I_{M_2} = \bigcup_{a \in \{e_1,e_2,d_1,d_2,d_3,d_4\}} I_a$ where $I_{e_i}$ represents explanation $e_i$ and $I_{d_i}$ represents diagnosis $d_i$, e.g., $I_{e_1}$ is described in Ex. 6. Evaluating $P_2 \cup I_{M_2}$ yields four policy answer sets; one is $I_1 = I_{M_2} \cup \{expl(e_1),\ expl(e_2),\ incNotLab(e_2),\ incLab,\ in(d_1),\ out(d_2),\ out(d_3),\ out(d_4),\ useOne,\ ruleHead(r_{alert}, c_{lab}, alert),\ @addRule(r_{alert}),\ @applyModAtContext(d_1, c_{lab})\}$. Evaluating $P_3 \cup I_{M_2}$ yields exactly one policy answer set, which is $I_2 = I_{M_2} \cup \{@guiSelectMod(diag)\}$.

From $I_1$ we obtain a single admissible modification of $M$ wrt. $P_2$: add bridge rule $r_{alert}$ and remove $r'_1$. 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) by selecting $d_4$ (and probably imposing additional monitoring for Sue), we obtain an admissible modification of $M$ which adds bridge rule $r'_6$ and removes $r'_1$. $\square$

## 4 Methodologies of Applying IMPL and Realization

For applying IMPL and integrating it with a MCS and user interaction, we next develop methodologies, based on a simple system design shown in Figure 2. Due to space constraints, we only give an informal discussion.

We represent the MCS as containing a *store of modifications*. The *semantics evaluation* component performs reasoning tasks on the MCS and invokes the *inconsistency manager* in case of an inconsistency. This inconsistency manager uses the *inconsistency analysis* component[4] to provide input for the *policy engine* which calculates 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. Let us discuss principal modes of operation and their merits next.

[3] There is no particular reason for this choice of precedence; one just has to be aware of it when specifying a policy.

[4] For realizations of this component we refer to [3, 13].

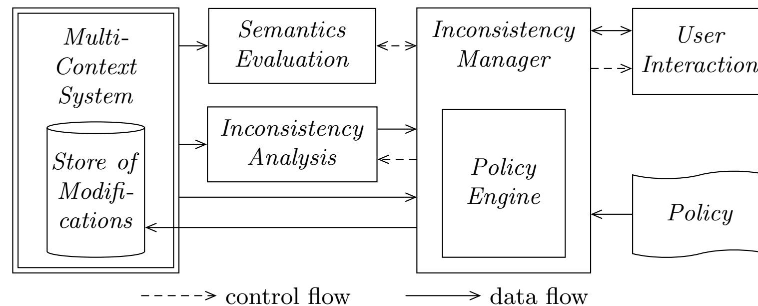


Fig. 2: Policy integration data flow and control flow block diagram.

**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. This mode while simple may not be satisfying in practice.

We can improve on the approach by extending actions with priority: the result of a single policy evaluation step then is a sequence of sets of actions, corresponding to several *attempts* for repairing the MCS. This can be exploited for writing policies that ensure repairs, by first attempting a 'sophisticated' repair possibly involving user interaction, and — if this fails — to simply apply some diagnosis to ensure consistency while the problem may be further investigated.

**Reason and Manage iteratively.** We now consider a mode where failure to restore consistency simply invokes policy evaluation again on the modified but still inconsistent system. This is useful if user interaction involves 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 policies may be structured, e.g., as follows: (a) classify inconsistencies into automatically vs manually repairable; (b) apply actions to repair one of the automatically repairable inconsistencies; if such inconsistencies do not exist (c) apply 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.

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.

In iterative mode, passing information from one iteration to the next can be useful. This can be accomplished by extending IMPL with add and delete actions which modify an iteration-persistent *knowledge base*, which is given to a policy as further input facts, represented using an additional dedicated predicate.

ISSN 1613-0073 © 2011 for the individual papers by the papers' authors. 

**Realization in acthex** The acthex formalism [2] extends answer set programs with external computations in bodies of rules, and action in heads of rules. Actions have an effect on an *environment* and this effect can use information from the answer set in which the action is present. Using acthex for realizing IMPL is a good choice because acthex already natively provides several features necessary for IMPL: external atoms can be used to access external information, and acthex actions come with weights for creating ordered execution schedules for actions occurring withinin 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, and acthex external predicates providing information about the MCS to the IMPL policy.

Using acthex to implement IMPL facilitates further extensions of IMPL, since new actions and external atoms can be added to acthex with little effort.

## 5 Conclusion

A language related to IMPL is the action language *IMPACT* [22], a declarative formalism for actions in distributed and heterogeneous multi-agent systems. While most parts of IMPL could be embedded in *IMPACT*, the latter is a very rich general purpose formalism, which is difficult to manage compared to the special purpose language IMPL. Furthermore, user interaction is not directly supported in *IMPACT*.

Policy languages have been studied in detail in the fields of access control, e.g., surveyed in [4], and privacy restrictions [11]. Notably, $\mathcal{PDL}$ [10] is a declarative policy language based on logic programming which maps events in a system to actions. It is richer than IMPL wrt. action interdependencies, whereas actions in IMPL have a richer internal structure and 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 [19, 12, 20]. Thus, they may be considered fixed policies without user interaction akin to an IMPL policy simply applying diagnoses in a homogenous MCS. Note however that one motivation for developing IMPL is that attempting automatic repair may not always be a viable option to deal with inconsistency in a MCS. In order to specify repair strategies for inconsistent databases in a more flexible way, *active integrity constraints (AICs)* [7–9] and *inconsistency management policies (IMPs)* [21] have been proposed. AICs extend the notion of integrity constraints by introducing update actions, for inserting and deleting tuples, to be performed if the constraint is not satisfied; whereas an IMP is a function defined wrt. a set of functional dependencies that maps a given relation $R$ to a 'modified' relation $R'$ obeying some basic axioms.

A conceptual difference between IMPL and the above approaches to database repair and inconsistency management is that IMPL policies aim at restoring consistency by modifying bridge rules leaving the knowledge bases unchanged rather than considering a (fixed) set of constraints and repairing the database. Moreover, IMPL policies operate on heterogeneous knowledge bases and may involve user interaction. Nevertheless, database repair programs, AICs and (certain) IMPs may by mimicked for particular classes of integrity constraints by corresponding IMPL policies given suitable encodings.

Ongoing work comprises the actual implementation of IMPL. Recalling that we currently just consider bridge rule modifications for system repairs, an interesting issue for further research is to drop this convention. This would mean to also allow modifications of knowledge bases of (some) contexts for repair, and to extend IMPL accordingly.

## References


1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge (2003)
2. Basol, S., Erdem, O., Fink, M., Ianni, G.: HEX programs with action atoms. In: ICLP. pp. 24–33 (2010)
3. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The MCS-IE system for explaining inconsistency in multi-context systems. In: JELIA. pp. 356–359 (2010)
4. Bonatti, P.A., Coi, J.L.D., Olmedilla, D., Sauro, L.: Rule-based policy representations and reasoning. In: Bry, F., Maluszynski, J. (eds.) REWERSE, vol. 5500, pp. 201–232 (2009)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI Conference on Artificial Intelligence (AAAI). pp. 385–390 (2007)
6. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: IJCAI. pp. 268–273 (2007)
7. Caroprese, L., Greco, S., Zumpano, E.: Active integrity constraints for database consistency maintenance. IEEE Trans. Knowl. Data Eng 21(7), 1042–1058 (2009)
8. Caroprese, L., Truszczynski, M.: Declarative semantics for active integrity constraints. In: ICLP. vol. 5366, pp. 269–283 (2008)
9. Caroprese, L., Truszczynski, M.: Declarative semantics for revision programming and connections to active integrity constraints. In: JELIA. vol. 5293, pp. 100–112 (2008)
10. Chomicki, J., Lobo, J., Naqvi, S.A.: A logic programming approach to conflict resolution in policy management. In: KR. pp. 121–132 (2000)
11. Duma, C., Herzog, A., Shahmehri, N.: Privacy in the semantic web: What policy languages have to offer. In: POLICY. pp. 109–118 (2007)
12. Eiter, T., Fink, M., Greco, G., Lembo, D.: Repair localization for query answering from inconsistent databases. ACM Trans. Database Syst. 33(2) (2008)
13. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in nonmonotonic multi-context systems. In: KR. pp. 329–339 (2010)
14. Eiter, T., Fink, M., Weinzierl, A.: Preference-based inconsistency assessment in multi-context systems. In: JELIA. pp. 143–155. LNAI (2010)
15. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
16. Fink, M., Ghionna, L., Weinzierl, A.: Relational information exchange and aggregation in multi-context systems. In: LPNMR (2011), to appear.
17. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
18. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: How we can do without modal logics. Artificial Intelligence 65(1), 29–70 (1994)
19. Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE Trans. Knowl. Data Eng 15(6), 1389–1408 (2003)
20. Marileo, M.C., Bertossi, L.E.: The consistency extractor system: Answer set programs for consistent query answering in databases. Data Knowl. Eng 69(6), 545–572 (2010)
21. Martinez, M.V., Parisi, F., Pugliese, A., Simari, G.I., Subrahmanian, V.S.: Inconsistency management policies. In: KR. pp. 367–377 (2008)
22. Subrahmanian, V., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., Ross, R.: Heterogeneous Agent Systems: Theory and Implementation. MIT Press (2000)


ISSN 1613-0073 © 2011 for the individual papers by the papers' authors. 