Logic-Based Interpretation of Context:

Modelling and Applications

Second International Workshop
16 May 2011, Vancouver, Canada

Alessandra Mileo
Michael Fink (Eds.)

Preface

Context interpretation and context-based reasoning are key factors in the development of intelligent autonomous systems in a variety of applications. The ability to represent contextual factors, interpret them and combine them with other sources of knowledge are some of the challenges to enable intelligent systems achieve correct behavior. Much work has been done in application areas that make use of contextual information, such as pervasive computing, logic-based sensor fusion and data integration, distributed problem solving and societal issues in Multi-Agent Systems. As well, theoretical foundations for context-based reasoning have been studied.

However, there is still a great deal to do in context modeling, since generic context models for context-aware application development need to be further explored, as does the role of context reasoning in particular regarding distributed evaluation and in conjunction with more recently emerging areas such as ontologies, including Semantic Web data, social features and reasoning about mental states, as well as approaches to belief change.

Context-dependent data can arise from different sources; for example it may be gathered by sensors or collected from different knowledge sources in different formats. The incompleteness and heterogeneous nature of such data and the need for state-based context interpretation in dynamic systems suggest that non-monotonic reasoning techniques could be a powerful tool for effective context-dependent reasoning. Since in many applications the data stems from distributed sources, distributed reasoning mechanisms are a highly relevant subject of research. Likewise, declarative approaches to societal reasoning or agent coordination may provide the backbone for contextual reasoning in various application domains. Given the increasing interest in hybrid knowledge representation formalisms as basis of the Semantic Web, it is also very interesting to consider proposals that assume hybrid formalisms combining Description Logics and Logic Programming as the basic representation framework for reasoning with (distributed) contexts.

Log-IC 2011 provided a forum for researchers investigating context-aware applications and context-based or distributed reasoning to share and compare their views on the efficacy of different context representation and context interpretation frameworks. Like the first Log-IC workshop (in Potsdam, 2009), it was held in conjunction with LPNMR (organized in Vancouver, Canada, May 16-19, 2011) with the additional advantage of reaching out to the logic programming community, facilitating collaboration between different formalisms for context-based reasoning.

Besides regular and short papers accepted for presentation, the workshop program consisted of invited talks by Pedro Cabalar (Corunna University, Spain), Thomas Eiter (TU Wien, Austria), and Torsten Schaub (University of Potsdam, Germany). These proceedings contain abstracts of the invited talks and the four papers that were accepted for publication by our Programme Committee. Acceptance was based on a blind reviewing process where every submission had been evaluated by three reviewers. Research

topics covered by contributions in this volume include various aspects of context-based reasoning, for instance privacy preservation, model streaming, and inconsistency management, as well as issues of inter-context communication.

The organizers wish to thank the all the authors who submitted papers, our invited speakers, the members of the Programme Committee, the reviewers, all participants and everyone who contributed to the success of this workshop. We are also grateful to the LPNMR local Organization Chair Aaron Hunter and the people of EasyChair for making our lives easier in organizing the workshop.

May 2011

Alessandra Mileo
Michael Fink
Organizers
Log-IC 2011

ISSN 1613-0073

Organisation

Executive Committee

Workshop Chairs:	Alessandra Mileo (Digital Enterprise Research Institute, Galway, Ireland)
	Michael Fink (Vienna University of Technology, Austria)

Programme Committee

Sebastian Bader	University of Rostock, Germany
Marcello Balduccini	Kodak Research Labs, Rochester, NY, USA
Chitta Baral	Arizona State University, Tempe, AZ, USA
Leopoldo Bertossi	Carleton University, Canada
Roberto Bisiani	University of Milan-Bicocca, Italy
Gerhard Brewka	University of Leipzig, Germany
Pedro Cabalar Fernandez	Corunna University, Galicia, Spain
Marina de Vos	University of Bath, UK
James P. Delgrande	Simon Fraser University, Vancouver, Canada
Wolfgang Faber	University of Calabria, Italy
Stijn Heymans	SemanticBits LLC, Herndon, VA, USA
Joao Leite	Universidade Nova de Lisboa, Portugal
Jorge Lobo	IBM T. J. Watson Research Center, Hawthorne, NY, USA
Bernd Ludwig	University of Erlangen-Nuremberg, Germany
Wendy MacCaull	St. Francis Xavier University, Canada
Robert Mercer	University of Western Ontario, Canada
Tommie Meyer	Meraka Institute, Pretoria, South Africa
Axel Polleres	Digital Enterprise Research Institute, Galway, Ireland)
Enrico Pontelli	New Mexico State University, Las Cruces, NM, USA
Marie-Christine Rousset	University of Grenoble, France
Chiaki Sakama	Wakayama University, Japan
Torsten Schaub	University of Potsdam, Germany
Tran Cao Son	New Mexico State University, Las Cruces, NM, USA
Hans Tompits	Vienna University of Technology, Austria
Paolo Torroni	University of Bologna, Italy
Kewen Wang	Griffith University, Brisbane, Australia
Nic Wilson	University College, Cork, Ireland
Stefan Woltran	Vienna University of Technology, Austria

Table of Contents

Part I

Invited Talks

Temporal Equilibrium Logic*

Pedro Cabalar

Department of Computer Science
University of Corunna (Spain)
cabalar@udc.es

Abstract. This talk introduces the Temporal Equilibrium Logic, a combination of standard Linear Temporal Logic with a formalism called Equilibrium Logic, used to to characterise logic programming under the answer set semantics. The talk will explain the basic syntax and semantics together with some elementary properties and recent results. Some motivating examples will show the potential utility of nonmonotonic temporal reasoning for different application scenarios like action domains or context evolution.

Short Biography. Pedro Cabalar is an associate professor of the Department of Computer Science at the University of Corunna, Spain, and organizes the MSc and PhD degrees in Computing research offered by that department. He graduated in 1993 in Computer Science in the Politechnic University of Madrid and received his PhD degree from the University of Corunna in 2001, on the topic of causality in action domains. His research is mostly related to logical approaches for Knowledge Representation in Artificial Intelligence, being particularly interested in Nonmonotonic Reasoning and Logic Programming under the Answer Set semantics. He has both published and actively participated as Program Committee or reviewer in main conferences (ICLP, LPNMR, KR, AAAI, ECAI, JELIA), journals (TPLP, AIJ, AMAI) and specialized workshops (NMR, ASP, ASPOCP) of these areas. He has also conducted several research projects in Answer Set Programming, in coordination with other groups in Spain.

* I am especially grateful to David Pearce, Agustín Valverde, Felicidad Aguado, Gilberto Pérez, Conchi Vidal and Martín Diéguez for their direct implication in many of the results related to this talk, and to the workshop organisers Alessandra Mileo and Michael Fink for their kind invitation.

Nonmonotonic Multi-Context Systems in Dynamic Environments*

Thomas Eiter

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
eiter@kr.tuwien.ac.at

Abstract. Multi-context systems (MCS) have been developed as a means for interlinking stand-alone knowledge bases, called contexts, via bridge rules for information exchange. Expressive MCS can host heterogeneous components with different (possibly nonmonotonic) semantics, and allow to capture a range of application logics, providing a versatile framework for interlinking heterogenous knowledge bases. A underlying assumption of MCS is, however, that the underlying collection of knowledge bases and their interlinkage is fixed. This hinders, however, to usage of MCS in an open or dynamic environment, where the available knowledge bases might change. To improve on this aspect, recent work at TU Wien developed Dynamic MCS, which consist of schematic contexts where part of the information interlinkage can remain open at design time; a concrete linkage is established by a configuration step at run time. In this talk, we present dynamic MCS, methods for distributed configuration, and some results of an experimental implementation.

Short Biography. Thomas Eiter is a full professor (since 1998) in the Faculty of Informatics at Vienna University of Technology (TU Wien), Austria, where he leads the Knowledge Based Systems Group. His current research interests include knowledge representation and reasoning, logic programming, complexity in AI, knowledge-based agents, database foundations, and logic in computer science. He serves on the boards of several international journals and program committees in his fields (e.g., co-chair of KR 2012). He is a Fellow of the European Coordinating Committee for Artificial Intelligence (ECCAI), and a Corresponding Member of the Austrian Academy of Sciences.

* This research is partially supported by Austrian Science Fund (FWF) grant P20841, Vienna Science and Technology Fund (WWTF) grant ICT08-020, and the FP7 ICT Project Ontorule (FP7 231875).

Knowledge-intensive Stream Reasoning

Roberto Bisiani[2], Martin Gebser[3], Holger Jost[3], Davide Merico[2], Alessandra Mileo[1], Philippe Obermeier[1], Orkunt Sabuncu[3], and Torsten Schaub[3]

[1] DERI, Gallway
[2] University of Milan-Bicocca
[3] University of Potsdam
torsten@cs.uni-potsdam.de

Abstract. Nonmonotonic reasoning is context-dependent [1]. For instance, Reiter-style defaults capture patterns of inference of the form *"in the absence of information to the contrary conclude"* [2]. Thus, conclusions are tentative, and they may become retracted in view of further information (or changing contexts). In other words, conclusions are context-dependent and contexts change over time. Unlike this, today's ASP systems focus on problem solving and thus disregard changing contexts.

On the other hand, there is a the practically highly significant area of stream processing (or stream reasoning) that lacks complex reasoning tasks [3]. Given that a *Data Stream* may be regarded as a *Changing Context*, stream reasoning constitutes a highly promising application area of Nonmonotonic Reasoning and in particular it's computational embodiment, viz. Answer Set Programming.

To underpin this claim, we report upon an extensive work on indoor position estimation [4]. Although there are well established quantitative methods in robotics and neighboring fields for addressing this problems, they lack knowledge representation and reasoning capacities. Such capabilities are not only useful in dealing with heterogeneous and incomplete information but moreover they allow for a better inclusion of semantic information and more general homecare and patient-related knowledge. We address this problem and investigate how state-of-the-art localization and tracking methods can be combined with Answer Set Programming. We report upon a case-study and provide a first experimental evaluation of knowledge-based position estimation both in a simulated as well as in a real setting.

Moreover, we illustrate by means of the problem of Online Job Scheduling a new reactive approach to Answer Set Programming, introduced in [5]. This approach aims at reasoning about real-time dynamic systems running online in changing environments. Moreover, we decsribe the first genuine implementation of a reactive ASP solver, *oclingo*, freely available at http://potassco.sourceforge.net/labs.html.

This is joint work with Roberto Bisiani, Martin Gebser, Holger Jost, Davide Merico, Alessandra Mileo, Philippe Obermeier, and Orkunt Sabuncu.

Speaker's Short Biography. Torsten Schaub received his diploma and dissertation in informatics in 1990 and 1992, respectively, from the Technical University of Darmstadt, Germany. He received his habilitation in informatics in 1995 from the University of Rennes I, France. From 1990 to 1993 he was a Researcher at the

Technical University at Darmstadt. From 1993 to 1995, he was a Research Associate at IRISA/INRIA at Rennes. From 1995 to 1997, he was University Professor at the University of Angers. At Angers he founded the research group FLUX dealing with the automatisation of reasoning from incomplete, contradictory, and evolutive information. Since 1997, he is University Professor for knowledge processing and information systems at the University of Potsdam. In 1999, he became Adjunct Professor at the School of Computing Science at Simon Fraser University, Canada; and since 2006 he is also an Adjunct Professor in the Institute for Integrated and Intelligent Systems at Griffiths University, Australia. His research interests range from the theoretic foundations to the practical implementation of methods for reasoning from incomplete and/or inconsistent information, in particular Answer set programming.

References

1. Marek, V., Truszczyński, M.: Nonmonotonic logic: context-dependent reasoning. Artifical Intelligence. Springer-Verlag (1993)
2. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13**(1-2) (1980) 81–132
3. Ceri, S., Daniel, F., Matera, M., Raffio, A.: Providing flexible process support to project-centered learning. IEEE Transactions on Knowledge and Data Engineering **21**(6) (2009) 894–909
4. Mileo, A., Schaub, T., Merico, D., Bisiani, R.: Knowledge-based multi-criteria optimization to support indoor positioning. Annals of Mathematics and Artificial Intelligence (2011) To appear.
5. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Volume 6645 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2011) 54–66

Part II

Full Papers

Model Streaming for Distributed Multi-Context Systems*

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner

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

Abstract. Multi-Context Systems (MCS) are instances of a nonmonotonic formalism for interlinking heterogeneous knowledge bases in a way such that the information flow is in equilibrium. Recently, algorithms for evaluating distributed MCS have been proposed which compute global system models, called equilibria, by local computation and model exchange. Unfortunately, they suffer from a bottleneck that stems from the way models are exchanged, which limits the applicability to situations with small information interfaces. To push MCS to more realistic and practical scenarios, we present a novel algorithm that computes at most $k \geq 1$ models of an MCS using asynchronous communication. Models are wrapped into packages, and contexts in an MCS continuously stream packages to generate at most k models at the root of the system. We have implemented this algorithm in a new solver for distributed MCS, and show promising experimental results.

1 Introduction

During the last decade, there has been an increasing interest, fueled by the rise of the world wide web, in interlinking knowledge bases to obtain comprehensive systems that access, align and integrate distributed information in a modular architecture. Some prominent examples of such formalisms are MWeb [1] and distributed ontologies in different flavors [12], which are based on a uniform format of the knowledge bases.

Nonmonotonic multi-context systems (MCS) [5], instead, are a formalism to interlink heterogeneous knowledge bases, which generalizes the seminal multi-context work of Giunchiglia, Serafini et al. [10, 13]. Knowledge bases, called *contexts*, are associated with belief sets at a high level of abstraction in a logic, which allows to model a range of common knowledge formats and semantics. The information flow between contexts is governed by so called *bridge rules*, which may – like logics in contexts – be nonmonotonic. The semantics of an MCS is given by models (also called *equilibria*) that consist of belief sets, one for each context where the information flow is in equilibrium.

For MCS where the contexts are distributed and accessible by semantic interfaces while the knowledge base is not disclosed (a predominant setting), algorithms to compute the equilibria by local computation and model exchange were given in [6] and [2]. They

* This research has been supported by the Austrian Science Fund (FWF) project P20841 and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

incorporate advanced decomposition techniques, but still, these algorithms suffer from a bottleneck that stems from the way in which models are exchanged and that limits the applicability to situations with small information interfaces.

For example, suppose context C_1 accesses information from several other contexts $C_2, \ldots, C_m$, called its neighbors. Consider a simple setting where the information flow is acyclic, meaning that none of the neighbors (directly or indirectly) accesses information from C_1. Furthermore, assume that $n_2, \ldots, n_m$ are the numbers of partial equilibria that exist at the neighbors, respectively. Intuitively, a partial equilibrium at a context is an equilibrium of the subsystem induced by information access. By the current approach for distributed evaluation, all the partial equilibria are returned to the parent context C_1.

Before any local model computation can take place at the parent, it needs to join, i.e., properly combine the partial equilibria obtained from its neighbors. This may result in $n_2 \times n_3 \times \cdots \times n_m$ partial models to be constructed (each one providing a different input for local model computation) which may not only require considerable computation time but also exhaust memory resources. If so, then memory is exhausted before local model computation at C_1 has even been initiated, i.e., before any (partial) equilibrium is obtained.

Note however that, if instead of the above procedure, each neighbor would transfer back a just a portion of its partial equilibria, then the computation at C_1 can avoid such a memory blow up; in general it is indispensable to trade more computation time, due to recomputations, for less memory if eventually *all* partial equilibria at C_1 shall be computed. This is the idea underlying a new *streaming* evaluation method for distributed MCS. It is particularly useful when a user is interested in obtaining just *some* instead of all answers from the system, but also for other realistic scenarios where the current evaluation algorithm does not manage to output under resource constraints in practice any equilibrium at all.

In this paper we pursue the idea sketched above and turn it into a concrete algorithm for computing partial equilibria of a distributed MCS in a streaming fashion. Its main features are briefly summarized as follows:

- the algorithm is *fully distributed*, i.e., instances of its components run at every context and communicate, thus cooperating at the level of peers;
- upon invocation at a context C_i, the algorithm streams, i.e. computes, $k \geq 1$ partial equilibria at C_i at a time; in particular setting $k = 1$ allows for consistency checking of the MCS (sub-)system.
- issuing follow-up invocations one may compute the next k partial equilibria at context C_1 until no further equilibria exist; i.e., this evaluation scheme is complete.
- local buffers can be used for storing and exchanging local models (partial belief states) at contexts, avoiding the space explosion problem.

We have implemented this algorithm yielding a new solver for distributed MCS, and report promising experimental results.

To the best of our knowledge, a similar streaming algorithm has neither been developed for the particular case of computing equilibria of a MCS, nor more generally for computing models of distributed knowledge bases. Thus, our results are not only of interest in the setting of heterogeneous MCS, but they are relevant in general for model

computation and reasoning over distributed (potentially homogeneous) knowledge bases like e.g. distributed SAT instances.

The rest of the paper is organized as follows. Section 2 recalls some background on multi-context systems, while basic details on the current DMCS algorithm(s) for evaluating MCS are summarized in Section 3. The new streaming algorithm is presented in detail in Section 4 including a discussion on parallelization, and Section 5 reports some initial experimental results obtained with a corresponding prototype implementation. Finally, we conclude in Section 6.

2 Multi-Context Systems

We first recall some basic notions of nonmonotonic MCS [5]. For simplicity and in order to get to the essence of the distributed algorithm, we focus here on a subclass of MCS in which knowledge bases consist of propositional clause sets (under different semantics).

A *logic* is a tuple $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$, where

- $\mathbf{KB}_L$ is a set of admissible knowledge bases, each being a finite set of clauses

$$h_1 \vee \cdots \vee h_k \leftarrow b_1 \wedge \cdots \wedge b_n \tag{1}$$

  where all h_i and b_j are literals (i.e., atoms or negated atoms) over a propositional signature Σ_L;
- $\mathbf{BS}_L$ is the set of possible belief sets, where a *belief set* is a satisfiable set of literals; it is *total*, if it is maximal under $\subseteq$; and
- $\mathbf{ACC}_L \colon \mathbf{KB}_L \to 2^{\mathbf{BS}_L}$ assigns each $kb \in \mathbf{KB}_L$ a set of *acceptable belief sets*.

In particular, classical logic results if $\mathbf{ACC}_L(kb)$ are the total belief sets that satisfy kb as usual, and Answer Set Programming (ASP) if all h_i are positive literals and $\mathbf{ACC}_L(kb)$ are the answer sets of kb.

A *multi-context system (MCS)* $M = (C_1, \ldots, C_n)$ consists 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, $kb_i \in \mathbf{KB}_i$ is a knowledge base, and br_i is a set of L_i*-bridge rules* of the form

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathrm{not}\,(c_{j+1} : p_{j+1}), \ldots, \mathrm{not}\,(c_m : p_m) \tag{2}$$

where $1 \leq c_k \leq n$, p_k is an element of some belief set of L_{c_k}, $1 \leq k \leq m$, and $kb \cup \{s\} \in \mathbf{KB}_i$ for each $kb \in \mathbf{KB}_i$.

Informally, bridge rules allow to modify the knowledge base by adding s, depending on the beliefs in other contexts.

Example 1. Let $M = (C_1, \ldots, C_n)$ be an MCS such that for a given integer $m > 0$, we have $n = 2^{m+1} - 1$ contexts, and let $\ell > 0$ be an integer. For $i < 2^m$, a context $C_i = (L_i, kb_i, br_i)$ of M consists of ASP logics L_i,

$$kb_i = \{a_i^1 \vee \cdots \vee a_i^\ell \leftarrow t_i\} \text{ and} \tag{3}$$
$$br_i = \left\{ \begin{array}{llll} t_i \leftarrow (2i : a_{2i}^1) & \cdots & t_i \leftarrow (2i : a_{2i}^\ell) \\ t_i \leftarrow (2i+1 : a_{2i+1}^1) & \cdots & t_i \leftarrow (2i+1 : a_{2i+1}^\ell) \end{array} \right\},$$

and for $i \geq 2^m$, we let C_i have

$$kb_i = \{a_i^1 \vee \cdots \vee a_i^\ell \leftarrow t_i\} \cup \{t_i\} \text{ and } br_i = \emptyset \ . \tag{4}$$

Intuitively, M is a binary tree-shaped MCS with depth m and $\ell + 1$ is the size of the alphabet in each context. Fig. 1a shows such an MCS with $n = 7$ contexts and depth $m = 2$; the internal contexts have knowledge bases and bridge rules as in (3), while the leaf contexts are as in (4). The directed edges show the dependencies of the bridge rules.

The semantics of an MCS M is defined in terms of particular *belief states*, which are tuples $S = (S_1, \ldots, S_n)$ of belief sets $S_i \in \mathbf{BS}_i$. Intuitively, S_i should be a belief set of the knowledge base kb_i; however, also the bridge rules br_i must be respected. To this end, kb_i is augmented with the conclusions of all $r \in br_i$ that are applicable in S. Formally, for any r of form (2) let $head(r) = s$ and $B(r) = \{(c_k : p_k) \mid 1 \leq k \leq m\}$. We call r *applicable in* $S = (S_1, \ldots, S_n)$, if $p_i \in S_{c_i}$, for $1 \leq i \leq j$, and $p_k \notin S_{c_k}$, for $j < k \leq m$; for any set of bridge rules R, let $app(R, S)$ denote the set of all $r \in R$ applicable in S. Then S is an *equilibrium* of M, iff for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$.

Example 2. The multi-context system M from Ex. 1 has equilibria $S = (S_1, \ldots, S_n)$ with $S_i = \{a_i^k, t_i\}$, for $1 \leq k \leq \ell$.

Without loss of generality, we assume the signatures Σ_i of the contexts C_i are pairwise disjoint, and let $\Sigma = \bigcup_i \Sigma_i$.

Partial Equilibria. For a context C_k, the equilibria of the sub-MCS it generates are of natural interest, which are partial equilibria of the global MCS [6].

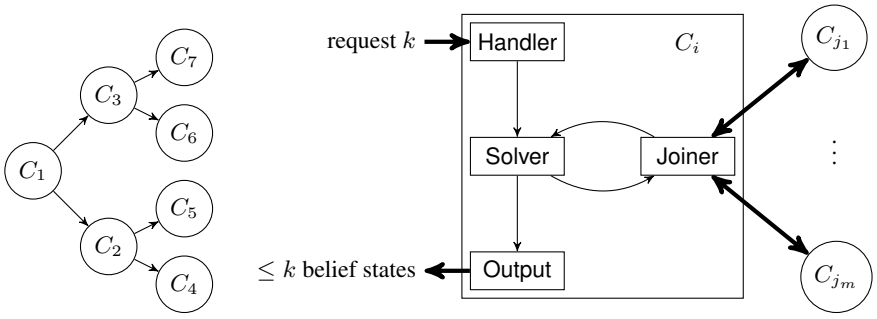
The sub-MCS is generated via recursive belief access. The *import neighborhood of a context* C_k in an MCS $M = (C_1, \ldots, C_n)$ is the set $In(k) = \{c_i \mid (c_i : p_i) \in B(r), r \in br_k\}$, and its *import interface* is $V(k) = \{p \mid (c : p) \in B(r), r \in br_k\}$, Moreover, the *import closure* $IC(k)$ *of* C_k is the smallest set I such that (i) $k \in I$ and (ii) for all $j \in I$, $In(j) \subseteq I$. and its *recursive import interface* is $V^*(k) = V(k) \cup \{p \in V(j) \mid j \in IC(k)\}$.

Based on the import closure, we define partial equilibria. Let $\epsilon \notin \bigcup_{i=1}^n \mathbf{BS}_i$. A *partial belief state* of M is a tuple $S = (S_1, \ldots, S_n)$, such that $S_i \in \mathbf{BS}_i \cup \{\epsilon\}$, for $1 \leq i \leq n$; S is a *partial equilibrium of* M *w.r.t. a context* C_k iff $i \in IC(k)$ implies $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, and if $i \notin IC(k)$, then $S_i = \epsilon$, for all $1 \leq i \leq n$.

Example 3. Continuing with Ex. 1, for $m = 2$ and $\ell = 3$, we get an MCS with seven contexts $M = (C_1, \ldots, C_7)$. A partial equilibrium of M w.r.t. context C_2 is the partial belief state $S = (\epsilon, \{a_2^1, t_2\}, \epsilon, \{a_4^3, t_4\}, \{a_5^2, t_5\}, \epsilon, \epsilon)$.

If one is only interested in consistency, (partial) equilibria of C_k may be projected to $V^*(k)$, hiding all literals outside. In accordance with this only belief sets projected to $V^*(k)$ would need to be considered.

For combining partial belief states $S = (S_1, \ldots, S_n)$ and $T = (T_1, \ldots, T_n)$, their *join* $S \bowtie T$ is given by the partial belief state $(U_1, \ldots, U_n)$, where (i) $U_i = S_i$, if $T_i = \epsilon \vee S_i = T_i$, and (ii) $U_i = T_i$, if $T_i \neq \epsilon \wedge S_i = \epsilon$, for all $1 \leq i \leq n$. Here $S \bowtie T$ is void, if some S_i, T_i are from $\mathbf{BS}_i$ but different.



(a) Binary tree MCS (b) DMCS-STREAMING Architecture

Fig. 1

3 DMCS Algorithms

There are two algorithms for computing (partial) equilibria of Multi-Context systems: (i) DMCS, a basic algorithm that needs explicit cycle-breaking during evaluation [6]; and (ii) DMCSOPT, an algorithm that operates on a pre-processed query plan [2].

Both DMCS and DMCSOPT aim at computing *all* partial equilibria starting from a particular context in an MCS. They run independently at each context in an MCS, and allow to project belief states to a relevant portion of the signature of the import closure. Upon request, they compute local models based on partial equilibria from neighboring contexts. The difference between the algorithms is that DMCS has no knowledge about the dependencies in an MCS and needs to detect cycles during evaluation using a history of visited contexts, whereas DMCSOPT assumes that there exists an acyclic query plan that has been created prior to evaluation-time using a decomposition technique based on biconnected components. Additionally, DMCSOPT includes information in the query plan to project partial equilibria to a minimum during evaluation. On the other hand, DMCS needs to get this information as input; this projection information in form of a set of relevant interface variables cannot be changed while the overall computation proceeds along the dependencies of the input MCS.

The algorithms use lsolve, an algorithm that completes combined partial equilibria from neighboring context with local belief sets. Formally, given a partial equilibrium S, lsolve(S) imports truth values of bridge atoms from neighboring contexts, solves the local theory, and returns all models. In the next section, we show a new algorithm that removes the restriction of DMCS and DMCSOPT to compute *all* partial equilibria to computing up to k partial equilibria, and if at least k partial equilibria exist, this algorithm will compute k of them.

4 Streaming Algorithm for DMCS

Given an MCS M, a starting (root) context C_r, and an integer k, we aim at finding at most k partial equilibria of M w.r.t. C_r in a *distributed* and *streaming* way. While the distributed aspect has been investigated in the DMCS and DMCSOPT algorithms,

adding the streaming aspect is not easy as one needs to take care of the communication between contexts in a nontrivial way. In this section, we present our new algorithm DMCS-STREAMING which allows for gradual streaming of partial equilibria between contexts. Section 4.1 describes a basic version of the algorithm, which concentrates on transferring packages of k equilibria with one return message. The system design is extendable to a parallel version, whose idea is discussed in Section 4.2.

4.1 Basic Streaming Procedure

In DMCSOPT, the reply to a request of a parent contains *all* partial equilibria from one context. This means that communication between contexts is synchronous—one request gets exactly one answer. While this is the easiest way to send solutions, it is very ineffective with larger MCS instances, as a small increase in the size of the alphabet may force the creation of many (partial) equilibria, which in turn may exceed memory limitations. The goal of this work is to develop an algorithm which allows for asynchronous communication for belief state exchange, i.e., one request for a bounded number of k (partial) equilibria may result in *at most* k solutions. This way we can restrict memory needs and evaluate multi-context systems that could not be handled by previous algorithms.

The basic idea is as follows: each pair of neighboring contexts can communicate in multiple rounds, and each request has the effect to receive at most k partial equilibria. Each communication window of k partial equilibria ranges from the k_1-th partial equilibria to the k_2-th ($= k_1 + k - 1$). A parent context C_i requests from a child context C_j a pair (k_1, k_2), and then receives at a future time point a package of at most k partial equilibria. Receiving ϵ indicates that C_j has fewer than k_1 models.

Important subroutines of the new algorithm DMCS-STREAMING take care of receiving the requests from parents, receiving and joining answers from neighbors, local solving and returning results to parents. They are reflected in four components: Handler, Solver, Output, and Joiner (only active in non-leaf contexts); see Fig. 1b for an architectural overview.

All components except Handler communicate using message queues: Joiner has j queues to store partial equilibria from j neighbors, Solver has one queue to hold joined equilibria from Joiner, and Output has a queue to carry results from Solver. As our purpose is to bound space usage, each queue has a limit on the number of entries. When a queue is full (resp., empty), the enqueuing writer (resp., dequeuing reader) is automatically blocked. Furthermore, getting an element also removes it from the queue, which makes room for other partial equilibria to be stored in the queue later. This property frees us from synchronization technicalities.

Algorithms 2 and 3 show how the two main components Solver and Joiner work. They use the following primitives:

- lsolve(S): works as lsolve in DMCSOPT, but in addition may return only one answer at a time and be able to tell whether there are models left.
- *get_first*(ℓ_1, ℓ_2, k): send to each neighbor from c_{ℓ_1} to c_{ℓ_2} a query for the first k partial equilibria, i.e., $k_1 = 1$ and $k_2 = k$; if all neighbors in this range return some models then store them in the respective queues and return $true$; otherwise, return $false$.

Algorithm 1: Handler(k_1, k_2: package range) at C_i

Output.$k_1 := k_1$, Output.$k_2 := k_2$, Solver.$k_2 := k_2$, Joiner.$k := k_2 - k1 + 1$
call Solver

Algorithm 2: Solver() at C_i

Data: Input queue: q, maximal number of models: k_2
$count := 0$
while $count < k_2$ **do**
(a) **if** C_i *is a leaf* **then** $S := \emptyset$
(b) **else** call Joiner and pop S from q
if $S = \epsilon$ **then** $count := k_2$
(c) **while** $count < k_2$ **do**
pick the next model $S^\star$ from Isolve(S)
if $S^\star \neq \epsilon$ **then** push $S^\star$ to Output.q and $count := count + 1$
else $count := k_2$
refresh() and push ϵ to Output.q

- *get_next*(ℓ, k): pose a query asking for the next k equilibria from neighbor C_{c_ℓ}; if C_{c_ℓ} sends back some models, then store them in the queue q_ℓ and return $true$; otherwise, return $false$. Note that this subroutine needs to keep track of which range has been already asked for to which neighbor by maintaining a set of counters. When a counter wrt. a neighbor C_{c_ℓ} is set to value t, then the latest request to C_{c_ℓ} asks for the t'th package of k models, i.e., models in the range given by $k_1 = (t-1) \times k + 1$ and $k_2 = t \times k$. For simplicity, we do not go into further details.
- *refresh*(): reset all counters and flags of Joiner to their starting states, e.g., $first\_join$ to $true$, all counters to 0.

The process at each context C_i is triggered when a message from a parent arrives at the Handler. Then Handler notifies Solver to compute up to k_2 model, and Output to collect the ones in the range from k_1 to k_2 and return them to the parent. Furthermore, it sets the package size at Joiner to $k = k_2 - k_1 + 1$ in case C_i needs to query further neighbors (cf. Algorithm 1).

When receiving a notification from Handler, Solver first prepares the input for its local solver. If C_i is a leaf context then the input S simply is the empty set assigned in Step (a); otherwise, Solver has to trigger Joiner (Step (b)) for input from neighbors. With input fed from neighbors, the subroutine Isolve is used in Step (c) to compute at most k_2 results and send them to the output queue.

The Joiner, only activated for intermediate contexts as discussed, gathers partial equilibria from the neighbors in a fixed ordering and stores the joined, consistent input to a local buffer. It communicates just one input at a time to Solver upon request. The fixed joining order is guaranteed by always asking the first package of k models from all neighbors at the beginning in Step (d). In subsequent rounds, we just query the first neighbor that can return further models (Step (e)). When all neighbors run out of models in Step (f), the joining process reaches its end and sends ϵ to Solver.

Algorithm 3: Joiner() at C_i

Data: Queue $q_1, \ldots,$ queue q_j for $In(i) = \{c_1, \ldots, c_j\}$, buffer for partial equilibria: buf, flag $first\_join$

while $true$ **do**
- **if** buf *is not empty* **then** pop S from buf, push S to Solver.q and **return**
- **if** $first\_join$ **then**
  - (d) **if** *get_first*$(1, j, k) = false$ **then** push ϵ to Solver.q and **return**
  - **else** $first\_join := false$
- **else**
  - $\ell := 1$
  - (e) **while** *get_next*$(\ell, k) = false$ *and* $\ell \leq j$ **do** $\ell := \ell + 1$
  - **if** $1 < \ell \leq j$ **then** *get_first*$(1, \ell - 1, k)$
  - (f) **else if** $\ell > j$ **then** push ϵ to Solver.q and **return**
- **for** $S_1 \in q_1, \ldots, S_j \in q_j$ **do** add $S_1 \bowtie \cdots \bowtie S_j$ to buf

Algorithm 4: Output() at C_i

Data: Input queue: q, starting model: k_1, end model: k_2

$buf := \emptyset$ and $count := 0$

while $count < k_1$ **do**
- pick an S from Output.q
- **if** $S = \epsilon$ **then** $count := k_2 + 1$
- **else** $count := count + 1$

while $count < k_2 + 1$ **do**
- wait for an S from Output.q
- **if** $S = \epsilon$ **then** $count := k_2 + 1$
- **else** $count := count + 1$ and add S to buf

if buf *is empty* **then** send ϵ to parent **else** send content of buf to parent

Note that while proceeding as above guarantees that no models are missed, it can in general lead to multiple considerations of combinations (inputs to Solver). Using a fixed size cache might mitigate these effects of recomputation, but since limitless buffering again quickly exceeds memory limits, recomputation is an unavoidable part of trading computation time for less memory.

The Output component simply reads from its queue until it receives ϵ (cf. Algorithm 4). Upon reading, it throws away the first $k_1 - 1$ models and only keeps the ones from k_1 onwards. Eventually, if fewer than k_1 models have been returned by Solver, then Output will return ϵ to the parent.

Example 4. Consider an instance of the MCS in Example 1 with $m = 1, \ell = 5$, i.e., $M = (C_1, C_2, C_3)$. Querying C_1 with a package size of $k = 1$, first causes the query to be forwarded to C_2 in terms of a pair $k_1 = k_2 = 1$. As a leaf context, C_2 invokes the local solver and eventually gets five different models. However, it just returns one partial equilibrium back to C_1, e.g., $(\epsilon, \{a_2^1\}, \epsilon)$. Note that t_2 is projected away since it does not appear among the atoms of C_2 accessed in bridge rules of C_1. The same happens at C_3 and we assume that it returns $(\epsilon, \epsilon, \{a_3^2\})$ to C_1. At the root context C_1, the two

single partial equilibria from its neighbors are consistently combined into $(\epsilon, \{a_2^1\}, \{a_3^2\})$. Taking this as an input to the local solving process, C_1 can eventually compute 5 answers, but in fact just returns one of them to the user, e.g., $S = (\{a_1^1, t_1\}, \{a_2^1\}, \{a_3^2\})$.

The following proposition shows the correctness of our algorithm.

Proposition 1. *Let $M = (C_1, \ldots, C_n)$ be an MCS, $i \in \{1, \ldots, n\}$ and let $k \geq 1$ be an integer. On input $(1, k)$ to C_i.*Handler*, C_i.*Output *returns up to k different partial equilibria with respect to C_i, and in fact k if at least k such partial equilibria exist.*

4.2 Parallelized Streaming

As the reader may have anticipated, the strategy of ignoring up to k_1 models and then collecting the next k is not likely to be the most effective. The reason is that each context uses only one Solver, which in general has to serve more than one parent, i.e., requests for different ranges of models of size k. When a new parent context requests models, we have to refresh the state of Solver and Joiner and redo from scratch. This is unavoidable, unless a context satisfies the specific property that only one parent can call it.

Another possibility to circumvent this problem is parallelization. The idea is to serve each parent with a set of the Handler, Joiner, Solver and Output components. In this respect, the basic interaction between each unit is still as shown in Fig. 1b, with the notable difference that each component now runs in an *individual thread*. The significant change is that Solver does not control Joiner but rather waits at its queue to get new input for the local solving process. The Joiner independently queries the neighbors, combines partial equilibria from neighbors, and puts the results into the Solver queue.

The effect is that we do not waste recomputation time for unused models. However, in practice, unlimited parallelization also faces a similar problem of exhausting resources as observed in DMCSOPT. While DMCSOPT runs out of memory with instances whose local theories are large, unlimited parallel instances of the streaming algorithm can exceed the number of threads/processes that the operating system can support, e.g., in topologies that allow contexts to reach other context using alternative paths such as the diamond topology. In such situations, the number of threads generated is exponential in the number of pairs of connected contexts, which prohibits scaling to large system sizes.

A compromise between the two extreme approaches is to have a *bounded parallel* algorithm. The underlying idea is to create a fixed-size pool of multiple threads and components, and when incoming requests cannot be served with the available resources, the algorithm continues with the basic streaming procedure, i.e., to share computational resources (the components in the system) between different parents at the cost of recomputation and unused models. This approach is targeted for future work.

5 Experimental Results

We present initial experimental results for a SAT-solver based prototype implementation of our streaming algorithm DMCS-STREAMING written in C++.[1] The host system was

[1] Available at `http://www.kr.tuwien.ac.at/research/systems/dmcs/`

topology / parameter	#	DMCSOPT	DMCS-STREAMING		
			$k = 0$	$k = 10$	$k = 100$
T_1 / $(10, 10, 5, 5)$	18	1.21	0.24	0.39	6.80
T_2 / $(10, 10, 5, 5)$	308	7.46	0.34	0.27	0.65
T_3 / $(50, 10, 5, 5)$	32	8.98	2.28	3.22	139.83
T_4 / $(50, 10, 5, 5)$	24	5.92	2.27	1.80	156.18
T_5 / $(100, 10, 5, 5)$	16	24.87	9.10	5.43	—
T_6 / $(100, 10, 5, 5)$	4	13.95	6.94	86.26	—
T_7 / $(10, 40, 20, 20)$	—	—	—	0.15	0.76
T_8 / $(10, 40, 20, 20)$	—	—	—	0.14	0.68
T_9 / $(50, 40, 20, 20)$	—	—	—	1.66	8.45
T_{10} / $(50, 40, 20, 20)$	—	—	—	1.64	8.04
T_{11} / $(100, 40, 20, 20)$	—	—	—	5.04	27.84
T_{12} / $(100, 40, 20, 20)$	—	—	—	5.00	26.30
R_1 / $(10, 10, 5, 5)$	12	2.17	0.99	2.44	—
R_2 / $(10, 10, 5, 5)$	16	3.11	3.31	0.17	143.83
R_3 / $(50, 10, 5, 5)$	21	15.49	13.43	—	—
R_4 / $(50, 10, 5, 5)$	12	10.30	6.43	—	—
R_5 / $(10, 40, 20, 20)$	—	—	—	0.86	6.27
R_6 / $(10, 40, 20, 20)$	—	—	—	0.51	5.66
R_7 / $(50, 40, 20, 20)$	—	—	—	3.06	29.81
R_8 / $(50, 40, 20, 20)$	—	—	—	4.29	43.94

Table 1: Runtime in secs, timeout 180 secs (—)

using two 12-core AMD Opteron 2.30GHz processors with 128GB RAM running Ubuntu Linux 10.10. We compare the basic version of the algorithm DMCS-STREAMING with DMCSOPT.

We used *clasp* 2.0.0 [9] and *relsat* 2.02 [3] as back-end SAT solvers. Specifically, all generated instantiations of multi-context systems have contexts with ASP logics. We use the translation defined in [6] to create SAT instances at contexts C_k, *clasp* to compute all models in case of DMCSOPT, and *relsat*[2] to enumerate models in case of DMCS-STREAMING.

For initial experimentation, we created random MCS instances with fixed topologies that should resemble the context dependencies of realistic scenarios. We have generated instances with binary tree (T) and ring (R) topologies. Binary trees grow balanced, i.e., every level is complete except for the last level, which grows from the left-most context.

A parameter setting (n, s, b, r) specifies (i) the number n of contexts, (ii) the local alphabet size $|\Sigma_i| = s$ (each C_i has a random ASP program on s atoms with 2^k answer sets, $0 \leq k \leq s/2$), (iii) the maximum interface size b (number of atoms exported), and (iv) the maximum number r of bridge rules per context, each having ≤ 2 body literals.

[2] The use of *relsat* is for technical reasons, and since *clasp* and *relsat* use different enumeration algorithms, the subsequent results are to be considered preliminary.

Table 1 shows some experimental results for parameter settings $(n, 10, 5, 5)$ and $(n, 40, 20, 20)$ with system size n ranging from 10 to 100. For each setting, running times on two instances are reported. Each row X_i ($X \in \{T, R\}$) displays pure computation time (no output) for rings (R) and binary trees (T), where the $\#$ columns show the number of projected partial equilibria computed at C_1 (initiated by sending the request to C_1 for the respective algorithms with the optimized query plan mentioned in [2]). With respect to DMCS-STREAMING, we run the algorithm with three request package sizes, namely $k = \{0, 10, 100\}$. The package size $k = 0$ amounts to an unbounded package, which means that the DMCS-STREAMING model exchange strategy is equivalent to DMCSOPT. For $k > 0$, we reported the running time until the first k unique models are returned, or all answers in case the total number of models is smaller than k.

We have observed several improvements. The new implementation appears to be faster than DMCSOPT when computing all partial equilibria ($k = 0$). This can be explained by the fact that in DMCSOPT we call the *clasp* binary and parse models using I/O streams, while DMCS-STREAMING tightly integrates *relsat* into the system, hence saving a significant amount of time used just for parsing models.

Getting the first k answers also runs faster in general. When we increase the size of the local theories, DMCSOPT and DMCS-STREAMING with $k = 0$ are stuck. However, DMCS-STREAMING can, with a reasonable small package size k, still return up to k answers in an acceptable time. When the package size increases, it usually takes longer or even timeouts. This can be explained by recomputation of models when requesting the next package of k models. We also observed this behavior in the ring topology with parameter setting $(50, 10, 5, 5)$, where DMCS-STREAMING timed out with $k \in \{10, 100\}$.

For the same reason, one would expect that asking for the next packages of k unique models might take more than linear amount of time compare to the time required to get the first package.

Comparing the two topologies, observe that rings are cyclic and thus the algorithm makes guesses for the values of the bridge atoms at the cycle-breaking context, and eventually checks consistency of the guess with the models computed locally at the same context. Hence, the system size that our algorithm can evaluate ring topology is smaller than that for the tree topology, which is acyclic.

6 Conclusion

Our work on computing equilibria for distributed multi-context systems is clearly related to work on solving constraint satisfaction problems (CSP) and SAT solving in a distributed setting; Yokoo et al. [14] survey some algorithms for distributed CSP solving, which are usually developed for a setting where each node (agent) holds exactly one variable, the constraints are binary, communication is done via messages, and every node holds constraints in which it is involved. This is also adopted by later works [15, 8] but can be generalized [14]. The predominant solution method are backtracking algorithms. In [11], a suite of algorithms was presented for solving distributed SAT (DisSAT), based on a random assignment and improvement flips to reduce conflicts. However, the algorithms are geared towards finding a single model, and an extension to streaming multiple

(or all) models is not straightforward; for other works on distributed CSP and SAT, this is similar. A closer comparison, in which the nature of bridge rules and local solvers as in our setting is considered, remains to be done.

Very recently, a model streaming algorithm for HEX-programs (which generalize answer set programs by external information access) has been proposed [7]. It bares some similarities to the one in this paper, but is rather different. There, monolithic programs are syntactically decomposed into modules (akin to contexts in MCS) and models are computed in a modular fashion. However, the algorithm is not fully distributed and allows exponential space use in components. Furthermore, it has a straightforward strategy to combine partial models from lower components to produce input for the upper component.

Currently, we are working on a conflict driven version of model streaming, i.e., in which clauses (nogoods) are learned from conflicts and exploited to reduce the search space, and on an enhancement by parallelization. We expect that this and optimizations tailored for the local solver and context setting (e.g., aspects of privacy) will lead to further improvements.

References

1. Analyti, A., Antoniou, G., Damasio, C.V.: Mweb: A principled framework for modular web rule bases and its semantics. ACM Trans. Comput. Logic 12(2), 17:1–17:46 (2011)
2. Bairakdar, S., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Decomposition of Distributed Nonmonotonic Multi-Context Systems. In: JELIA'10. LNAI, Springer (2010)
3. Bayardo, R.J., Pehoushek, J.D.: Counting models using connected components. In: AAAI'00. pp. 157–162. AAAI Press (2000)
4. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T.: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI'07. pp. 385–390. AAAI Press (2007)
6. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed Nonmonotonic Multi-Context Systems. In: KR'10. pp. 60–70. AAAI Press (2010)
7. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing Efficient Evaluation of HEX Programs by Modular Decomposition. In: LPNMR'11. pp. 93–106. Springer (2011)
8. Gao, J., Sun, J., Zhang, Y.: An improved concurrent search algorithm for distributed csps. In: Australian Conference on Artificial Intelligence. pp. 181–190. Springer (2007)
9. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Commun. 24(2), 107–124 (2011)
10. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics or: How we can do without modal logics. Artif. Intell. 65(1), 29–70 (1994)
11. Hirayama, K., Yokoo, M.: The distributed breakout algorithms. Artif. Intell. 161(1–2), 89–115 (2005)
12. Homola, M.: Semantic Investigations in Distributed Ontologies. Ph.D. thesis, Comenius University, Bratislava, Slovakia (2010)
13. Roelofsen, F., Serafini, L.: Minimal and absent information in contexts. In: IJCAI'05 (2005)
14. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems 3(2), 185–207 (2000)
15. Zivan, R., Meisels, A.: Concurrent search for distributed CSPs. Artif. Intell. 170(4-5), 440–461 (2006)

ISSN 1613-0073

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 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

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

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.

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 $kb_i \cup H \in \mathbf{KB}_{L_i}$. 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 $E_m^{\pm}(M_2) = \{e_1, e_2\}$ and $D_m^{\pm}(M_2) = \{d_1, d_2, d_3, d_4\}$, 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 $r'_j = r_j[Id|sue]$, $j \in \{2, 3, 5, 6\}$. $E_m^{\pm}(M_2)$ 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 $p \in Built$.

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

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

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 '$\mathbf{not}\ (c : b)$').
- $modAdd(m,r)$ (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 $(E_1, E_2) \in E_m^{\pm}(M)$ 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 $I_{e_1} = \{ruleHead(r'_1, c_{lab}, \text{`}customer(sue, 03/02/1985)'), ruleBody^+(r'_1, c_{db}, \text{`}person(sue, 03/02/1985)'), expl\text{-}Need(e_1, r'_1)\}$. 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

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

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

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

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 α wrt. *I, denoted $eff_I(\alpha)$, 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 $eff_I(\alpha)$ for every α in I.

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, $D_m^{\pm}(M)$, and $E_m^{\pm}(M)$ 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. □

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].

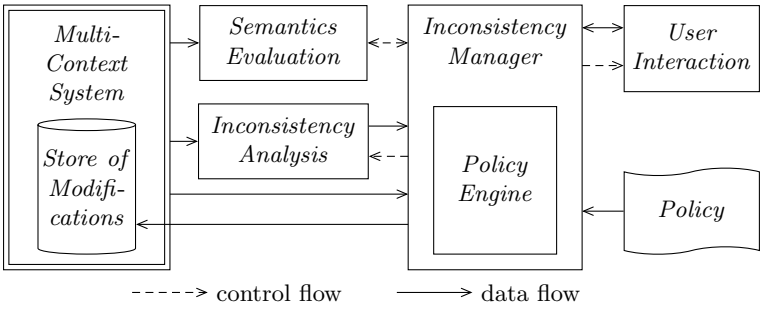


Fig. 2: Policy integration data flow and control flow block diagram.

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.

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)

Part III

Short Papers

Lightweight Communication Platform for Heterogeneous Multi-context Systems: A Preliminary Report *

Vladimír Dziuban, Michal Čertický, Jozef Šiška, and Michal Vince

Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava,
Slovakia{dziuban,certicky,siska,vince}@ii.fmph.uniba.sk

Abstract. This paper describes an ongoing implementation of a lightweight communication platform that uses RESTful TCP/IP requests to transfer FIPA ACL based messages between agents. The platform is intended for heterogeneous systems composed of numbers of simple agents as opposed to usual FIPA implementations. Agents can be written in any language and can communicate with each other without the need for a central platform.

1 Introduction

Multi-context systems describe information from different viewpoints (contexts) and the relationship between them in the form of bridge rules. Heterogeneous multi-context systems [1] allow different logics or completely different formalisms to be used in different contexts.

When building truly heterogeneous multi-context systems in a decentralised fashion, components built on different formalism can be implemented in different languages and must cooperate and communicate with each other. FIPA [5] defines standards for such interoperability of heterogeneous agents. However most implementations involve complex, mostly java-based, platforms, usually hosting multiple agents. To facilitate simpler and more agile development of small agents in different languages, a more lightweight approach is needed.

We present an ongoing implementation of a de-centralised lightweight communication framework for heterogeneous agents that implements a subset of FIPA specifications, while allowing simple development of small standalone agents. Agents in the framework are standalone processes, that communicate using FIPA ACL messages[2], transported through RESTful[7], peer-to-peer TCP/IP connections.

There is no full featured platform with its associated services. However, the framework provides a simple agent management system [3] in the form of a

* We acknowledge support from VEGA project of Slovak Ministry of Education and Slovak Academy of Sciences no. 1/0688/10 and Comenius University grant no. UK/486/2010.

discovery service that allows each agent to identify other agents on the local network along with the services they provide. This allows creation of systems consisting of multitude of small heterogeneous agents scattered through the local network without the need for a central server / registry. Communication with agents on remote networks or with other FIPA platforms / implementations with different message transport systems can be achieved through the use of gateway agents that forward messages and agent information. Implementations in Python and C++ are available and Java implementation is planned.

Example 1. Consider a heterogeneous multi-context system used to organize seminars of a research group. It consists of three types of agents:

- *Personal* Java-based agents running on the mobile phone/PDA of every group member, providing information about him and his schedule,
- *sensoric* C++ agents that use webcameras to observe certain rooms at the university,
- *timetable* agent that provides access to the university timetable and room reservations,
- and a logic based *scheduling* agent (written in Python as a frontend to a logic based formalism such as an ASP solver.)

Now imagine, that one of the group members wants to organize a meeting with his colleagues. He simply orders his *personal agent* to arrange this meeting. This agent then contacts all the other *personal agents*, and finds out (using the GPS on their devices) which of them are currently out of town. If the sufficient number of them are available, he asks the *scheduling agent* for the most appropriate time and place/room for the meeting. The *scheduling agent* collects information from the *sensoric agents* to find out which rooms are empty and also checks if they aren't reserved for the next 2 hours. After receiving this information, the invitation can be sent to all the *personal agents* of available members of the group.

The rest of this paper is structured as follows: in the next section we describe the presented communication platform; the third section describes our implementation; the fourth section contains a short comparison to other FIPA platforms; and the last section presents future plans and concluding remarks.

2 Communication Platform

This section describes a lightweight communication platform. Such a platform consists of *agents* that can send *messages* to each other. Agents are usually standalone processes / programs and can run on different computers. Messages conform to FIPA ACL specification[2] and are transported through a stateless TCP/IP connection to the recipient.

Each agent has a globally unique *agent name* that is used to identify the agent and a list of *services* that the agent provides.

An agent has access to two basic services: discovery service that serves as a very simple AMS and a message transport service. These are implemented per process/program and are thus shared between multiple agents running in a single process.

Discovery service monitors the local network and notifies the agent of the appearance or disappearance of other agents. It maintains a list of known agents and their locators.

2.1 Message Transport Service

A transport service with a single protocol is used. The protocol uses RESTful TCP/IP connections to another agent (not necessarily the final recipient of the message) to deliver requests. There are two types of requests defined: a POST request that delivers a single message, or a POLL requests, that ask the other agent for any pending messages for the connecting agent.

The *transport specific address* of an agent is an IP address and port tuple and there are two transport-specific properties: polling mode and level of indirection.

Level of indirection represents how many times would the message be forwarded, would it be sent to this address. Each agent that acts as a proxy/gateway for another increases this number when it announces the target agent on a local network. If the discovery service reports multiple addresses for a single agent, one of those with lowest level of indirection should be used when sending messages.

If the polling mode is enabled for an address, then the message should not be send, but kept in a queue at the sending agent until the receiving agent asks for messages with the POLL request or the message expires (either through a pre-set timeout or because of a size limit of the queue.)

This can be used by agents that don't have a stable or accessible address on the network, such as on a mobile device that roams between different network connections. Such an agent would normally register with a gateway agent, which would announce it on the local network, collecting all messages and delivering them later through the connection created by the first agent's POLL requests.

2.2 Gateway agents

A discovery service, as described in the previous section, can reliably work only on a local network. Similarly the message transport can deliver messages only to agents on the local network or with a publicly accessible IP address (e.g. not behind NAT.) These restrictions can be worked around by the introduction of special *gateway agents*.

A gateway agent (GA) is a special agent that acts as a proxy for agents from other networks. GA maintains a list of registered remote agents. When a remote agent registers, GA announces remote agent's presence on the local network with its own transport specific address. Thus any message sent to the remote agent from the local network is sent to GA, which looks up the remote agent in its database and delivers (forwards) it.

There are two basic ways to use gateway agents:

- remote agents register directly with gateway agents
- a *bridge* agent registers all agents on his local network with a gateway agent on a remote network (and vice versa).

Similarly, a gateway agent that acts as a bridge to other message transport systems can be created, thus enabling interoperability with other FIPA compliant platforms.

3 Implementation

This section presents the implementation of the communication platform.

The aim of our implementation is a simple and lightweight framework, that can be used also on devices with little memory and computational power, e.g. cell pohones, embedded devices, etc.

Our architecture is currently implemented in Python and C++, which were selected for their simplicity and speed respectively. In the future we plan to provide an implementation in JAVA, as a language most commonly associated with multi-agent system development, that might encourage further development of heterogenous agent applications. In the current implementation, single or multiple instances of the agent can be used per process, being served by the same discovery and message transport service

In the discovery service, multicast protocol (there are also plans to add Avahi/Zeroconf support) is used to announce the arrival of a new agent to the network. Following registration routine is then handled by message transport service by sending a request message in ACL to the new agent and inform message with agent's properties as an answer to this request.

Message transport service is responsible for marshalling and demarshalling messages, sending and receiving over TCP/IP protocol and posting them to the main loop of the corresponding agent.

Agent's mainloop is event driven, with four main types of events:

- agentAdded triggered when a new agent registers a service
- agentRemoved triggered when the agent leaves the network
- agentChanged triggered when the agent changes his serivces
- messageReceived triggered when message is reveived.

MessageReceived is then further marshalled by communicative act[4] to InformMessage, RequestMessage, QueryIfMessage, etc or to SystemMessage.

We are currently working on the implementation of gateway agents.

4 Comparison to Other Work

There are many different FIPA compliant frameworks, most of them are implemented in JAVA and their platforms offer many services. This makes them computationally and memory intensive, therefore running them on small devices with little memory and slow processors is very difficult, often even impossible.

Our solution does not implement agent platform or platform services. The latter can be however implemented in form of standalone agents. This approach is needed to ensure, that agents are more lightweight and can be deployed easily and with as few preliminary arrangements as possible.

SPADE[8, 6] might be an example of a more lightweight approach, although SPADE agents run on a platform with standard FIPA AMS and DF components. It is written in Python and uses XMPP protocol for message transportation. Each agent is a client with a registered Jabber ID and all communication is conducted by sending Jabber messages that contain FIPA ACL expressions. The platform however requires a central jabber/XMPP server.

5 Conclusion and Future Work

In this paper we have presented a lightweight communication platform designed especially to allow easy implementation of heterogeneous multi-context systems. The platform uses FIPA ACL messages transported through simple RESTful peer to peer TCP/IP connections. Each agent also has a discovery service that acts as a simple agent management system.

The platform also allows the creation of gateway agents that can be used to interconnect agents from different networks or based on other FIPA platforms using different message transports.

Each agent keeps a local agent directory via his discovery service, which might not scale well for larger systems. Gateway agents could take the role of directory services, especially since they can be dynamically created/registered. Ordinary agents will use local network discovery services only to find gateway agents and use them as a full featured agent management system/directory facilitator.

A possible improvement of the dynamic aspects of the platform is the ability to preserve its structural integrity by automatic reorganization of local-area components. Basic idea behind this is making every agent capable of starting a gateway service whenever one is needed (i.e. when the number of local gateway agents is critically low) and obtaining the needed database from other remaining local gateway agents. This makes it possible to maintain the connectivity with external networks or platforms even after the loss of several gateway agents.

References

1. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI. pp. 385–390. AAAI Press (2007)
2. Foundation for Intelligent Physical Agents: FIPA ACL Message Structure Specification. http://www.fipa.org/specs/fipa00061/index.html (2000)
3. Foundation for Intelligent Physical Agents: FIPA Agent Management Specification. http://www.fipa.org/specs/fipa00023/index.html (2000)
4. Foundation for Intelligent Physical Agents: FIPA Communicative Act Library Specification. http://www.fipa.org/specs/fipa00037/index.html (2000)
5. Foundation for Intelligent Physical Agents: FIPA Standard Specification. http://www.fipa.org/repository/standardspecs.html (2000)

6. Gregori, M.E., Cámara, J.P., Bada, G.A.: A jabber-based multi-agent system platform. In: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. pp. 1282–1284. AAMAS '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1160633.1160866
7. Representational state transfer. http://en.wikipedia.org/wiki/Representational_State_Transfer
8. Spade2: Smart python agent development environment. http://code.google.com/p/spade2/

Privacy Preservation Using Multi-Context Systems*

Wolfgang Faber

University of Calabria, Italy
wf@wfaber.com

Abstract. Preserving the privacy of sensitive data is one of the major challenges which the information society has to face. Traditional approaches focused on the infrastructure for identifying data which is to be kept private and for managing access rights to these data. However, while these efforts are useful, they do not address an important aspect: While the sensitive data itself can be protected nicely using these mechanisms, related data, which is deemed insensitive per se may be used to infer sensitive data. This can be achieved by combining insensitive data or by exploiting specific background knowledge of the domain of discourse. In this note, we show that resolving this problem can be achieved in a simple and elegant way by using multi-context systems.

1 Introduction

The privacy of individuals has become one of the most important and most discussed issues in modern society. With the advent of the Internet and easy access to a lot of data, keeping sensitive data private has become a priority for distributed information systems. An example area in which privacy is at stake are medical information systems.

Most databases have privacy mechanisms which are comparatively simple – by and large they boil down to keeping certain columns of the database hidden from certain types of users. There is a huge body of literature that deals with formalisms for this kind of authorization problem, which we cannot discuss in detail in this short note. As an example, see [6] for a work that discusses aspects of the authorization problem in non-monotonic knowledge bases. What we are interested in this short paper is a somewhat different issue, namely that users can infer information that is designated private by asking queries that do not involve private information and then making "common sense" inferences from the answers to infer private information.

In an earlier paper [4], we have given a formal definition of the *Privacy Preservation Problem* and shown how this can be addressed by using default logic (we also refer to this paper for discussions on related work). In that paper, however, there were several restrictions on the knowledge bases that can be used. Effectively, they had to be first-order theories, because in this way it is easily possible to build a default theory around them.

In order to lift this restriction, in this note we propose using multi-context systems as defined by Brewka and Eiter in [3] instead of default logic. By switching to that formalism, it is possible to use heterogeneous knowledge bases to which users may

* This work was supported by M.I.U.R. within the PRIN project LoDeN.

have access or which model user knowledge. The unifying framework are then contexts and bridge rules that link contexts instead of default rules in [4]. Apart from the greater flexibility concerning the types of "participating" knowledge bases, another advantage is that efficient systems for reasoning with multi-context systems begin to emerge [1].

In the following, we will first provide an adapted definition of the privacy preservation problem in section 2. This definition is slightly different from the one of [4] in order to allow for more heterogeneous knowledge bases to be involved. In section 3 we will then show how to construct a multi-context system for computing answers for a privacy preservation problem. In section 4 we conclude and outline future work.

2 Privacy Preservation Problem

In this section, we provide a simple formulation of the privacy preservation problem (**P3** for short), which is a generalization of the definition in [4]. For simplicity, we will not consider any evolution in time of the systems, as it was done in [4].

We consider a *logic* L as in [3] to be a triple $(\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ where $\mathbf{KB}_L$ is the set of well-formed knowledge bases of L (each of which is a set as well), $\mathbf{BS}_L$ is the set of possible belief sets, and $\mathbf{ACC}_L$ is a function $\mathbf{KB}_L \rightarrow \mathbf{2}^{\mathbf{BS}_L}$ describes the semantics of each knowledge base. In the following, when mentioning knowledge bases, we will usually not specify the underlying logic, intending that it can be any logic in the sense just described.

Let the finite set $\mathbf{U}$ contain one user ID for each user in the system under consideration. Moreover, we consider the *main knowledge base* $\mathbf{MKB}$ which the users will be querying. Furthermore, the function $\mathbf{BK}$ associates each user $u \in \mathbf{U}$ with a *background knowledge base* $\mathbf{BK}(u)$, while the function $\mathbf{Priv}$ associates each user $u \in \mathbf{U}$ with a belief set $\mathbf{Priv}(u)$ that should be kept private. Note that the various knowledge bases need not be of the same logic, but for practical reasons one would assume the belief sets to be homogeneous.

It should be pointed out that $\mathbf{BK}(u)$ is not necessarily the user's own knowledge base, but rather a model of the user's knowledge, maintained by the information system.

Example 1. Consider a small medical knowledge base MedKB containing information about the symptoms and diseases of some patients. Let this knowledge base describe two predicates symptom and disease and let the following be its only belief set S_{MedKB}:

$\mathsf{symptom}(john, s_1)$	$\mathsf{symptom}(jane, s_1)$	$\mathsf{disease}(jane, aids)$
$\mathsf{symptom}(john, s_2)$	$\mathsf{symptom}(jane, s_4)$	$\mathsf{disease}(john, cancer)$
$\mathsf{symptom}(john, s_3)$		$\mathsf{disease}(ed, polio)$

Note that MedKB could very well be just a database. Assume that $john$ and $jane$ are also users of the system and want to keep their diseases private, so $\mathbf{Priv}(john) = \{\mathsf{disease}(john, cancer)\}$, while $\mathbf{Priv}(jane) = \{\mathsf{disease}(jane, aids)\}$. Consider another user $acct$ (an accountant). This person may have the following background knowledge base $\mathbf{BK}(acct)$ in the form of rules (so the underlying logic might be answer set programming).

$$\mathsf{disease}(X, aids) \leftarrow \mathsf{symptom}(X, s_1), \mathsf{symptom}(X, s_4)$$
$$\mathsf{disease}(X, cancer) \leftarrow \mathsf{symptom}(X, s_2), \mathsf{symptom}(X, s_3)$$

Let a query be a construct to which for every semantics of a knowledge base a belief set is associated, which is referred to as the *answer* $\mathbf{Ans}(Q)$ to Q. A *privacy preserving answer* to a query Q over $\mathbf{MKB}$ posed by $u_o \in \mathbf{U}$ with respect to $\mathbf{BK}$ and $\mathbf{Priv}$ is $X \subseteq \mathbf{Ans}(Q)$ such that for all $u \in \mathbf{U} \setminus \{u_0\}$ and for all $p \in \mathbf{Priv}(u)$, if $p \notin \mathbf{ACC}(\mathbf{BK}(u_0))$ then $p \notin \mathbf{ACC}(X \cup \mathbf{BK}(u_0))$. A *maximal privacy preserving answer* is a subset maximal privacy preserving answer.

Note that here we assume that elements of belief sets can be added to knowledge bases, yielding again a knowledge base of the respective logic.

A privacy preservation problem $\mathbf{P3}$ is therefore a tuple $(\mathbf{MKB}, \mathbf{U}, \mathbf{BK}, \mathbf{Priv}, Q, u_0)$ and solving it amounts to finding the (maximal) privacy preserving answers to Q posed by u_0 over $\mathbf{MKB}$ with respect to $\mathbf{BK}$ and $\mathbf{Priv}$.

Example 2. Returning to our MedKB example, posing the query $\mathsf{disease}(john, X)$, we would get as an answer the set $\{\mathsf{disease}(john, cancer)\}$. Likewise, the answer to the query $\mathsf{symptom}(john, X)$ is the set $\{\mathsf{symptom}(john, s_1), \mathsf{symptom}(john, s_2), \mathsf{symptom}(john, s_3)\}$.

We assumed that John and Jane want their diseases kept private. However, the accountant can violate John's privacy by asking the query $\mathsf{symptom}(john, X)$. The answer that $acct$ would get from the system is $\{\mathsf{symptom}(john, s_1), \mathsf{symptom}(john, s_2), \mathsf{symptom}(john, s_3)\}$. However, recall that the accountant has some background knowledge including the rule

$$\mathsf{disease}(X, cancer) \leftarrow \mathsf{symptom}(X, s_2), \mathsf{symptom}(X, s_3)$$

which, with the answer of the query, would allow $acct$ to infer $\mathsf{disease}(john, cancer)$. Thus the privacy preserving answers to $\mathsf{symptom}(john, X)$ are

$$\begin{aligned}
Ans_1 &= \{\mathsf{symptom}(john, s_1), \mathsf{symptom}(john, s_2)\} \\
Ans_2 &= \{\mathsf{symptom}(john, s_1), \mathsf{symptom}(john, s_3)\} \\
Ans_3 &= \{\mathsf{symptom}(john, s_1)\} \\
Ans_4 &= \{\mathsf{symptom}(john, s_2)\} \\
Ans_5 &= \{\mathsf{symptom}(john, s_3)\} \\
Ans_6 &= \emptyset
\end{aligned}$$

None of these answers allows $acct$ to infer the private knowledge $\mathsf{disease}(john, cancer)$. However, except for the answers Ans_1 and Ans_2, which are maximal, all answers yield fewer information than could be disclosed without infringing privacy requirements. Any system should also provide only one of these answers to the user, because getting for instance both Ans_1 and Ans_2 would again violate John's privacy requirements.

In a practical system, upon disclosing an answer the system should update the respective user's knowledge model in order to avoid privacy infringements by repeated querying. For example, when the system returns Ans_1 to user $acct$, it should modify $\mathbf{BK}(acct)$ in order to reflect the fact that $acct$ now knows $\mathsf{symptom}(john, s_1)$ and $\mathsf{symptom}(john, s_2)$, such that asking the same query again it is made sure that $\mathsf{symptom}(john, s_3)$ will not be disclosed to $acct$. This however is part of the dynamic aspect of a privacy preserving information system, which we will not address in this paper.

3 Solving Privacy Preservation Problems Using Multi-Context Systems

The definitions in Section 2 were already slightly geared towards multi-context systems. We recall that a multi-context system in the sense of [3] is a tuple $(C_1, \ldots, C_n)$ where for each i, $C_i = (L_i, kb_i, br_i)$ where L_i is a logic, kb_i is a knowledge base of L_i and br_i is a set of L_i bridge rules over $\{L_1, \ldots, L_n\}$.

An L_i bridge rule over $\{L_1, \ldots, L_n\}$ is a construct

$$s \leftarrow (r_1 : p_1), \ldots, (r_j : p_j), \text{not } (r_{j+1} : p_{j+1}), \ldots, \text{not } (r_m : p_m)$$

where $1 \leq r_k \leq n$, p_k is an element of a belief set for L_{r_k} and for each $kb \in \mathbf{KB}_i$ $kb \cup \{s\} \in \mathbf{KB}_i$.

The semantics of a multi-context system is defined by means of equilibria. A *belief state* for a multi-context system $(C_1, \ldots, C_n)$ is $S = (S_1, \ldots, S_n)$, where $S_i \in \mathbf{BS}_i$ for $1 \leq i \leq n$. An L_i bridge rule of the form above is applicable in S iff for $1 \leq k \leq j$ $p_k \in S_{r_k}$ holds and for $j < k \leq m$ $p_k \notin S_{r_k}$ holds. Let $app(br, S)$ denote the set of all bridge rules in br which are applicable in a belief state S. A belief state $S = (S_1, \ldots, S_n)$ is an equilibrium of a multi-context system $(C_1, \ldots, C_n)$ iff for all $1 \leq i \leq n$, $S_i \in \mathbf{ACC}_i(kb_i \cup \{hd(r) \mid r \in app(br_i, S)\})$, where $hd(r)$ is the head of a bridge rule r, viz. s in the bridge rule schema given earlier.

Given a $\mathbf{P3}$ $(\mathbf{MKB}, \mathbf{U}, \mathbf{BK}, \mathbf{Priv}, Q, u)$, with $\mathbf{U} = \{u_1, \ldots, u_{|\mathbf{U}|}\}$, in order to identify privacy preserving answers, we build a multi-context system $M_{\mathbf{P3}} = (C_1, C_2, C_3, C_4, \ldots, C_{|\mathbf{U}|+3})$, where $C_1 = (L_{\mathbf{MKB}}, \mathbf{MKB}, \emptyset)$, $C_2 = (L_{\mathbf{MKB}}, \emptyset, br_2)$, $C_3 = (L_{\mathbf{MKB}}, \emptyset, br_3)$, $C_4 = (L_{\mathbf{BK}(u_1)}, \mathbf{BK}(u_1), br_4) \ldots$, $C_{|\mathbf{U}|+3} = (L_{\mathbf{BK}(u_{|\mathbf{U}|})}, \mathbf{BK}(u_{|\mathbf{U}|}), br_{|\mathbf{U}|+3})$. Here L_{kb} is the logic of the knowledge base kb. The meaning is that C_1 provides just the belief sets for $\mathbf{MKB}$ (no bridge rules), C_2 and C_3 are used to identify those belief sets which are privacy preserving, while $C_4, \ldots, C_{|\mathbf{U}|+3}$ represent the user information, that is, the background knowledge base of the querying user and the privacy requirements of the other users. The important part are the bridge rules, which we will describe next. In many cases, we will create one rule for each symbol that can occur in some belief set of $\mathbf{Ans}(Q)$, so for convenience let $\mathcal{D} = \{p \mid p \in B, B \in \mathbf{Ans}(Q)\}$.

The set br_2 contains one bridge rule $p \leftarrow (1 : p), \text{not } (3 : p)$ for each $p \in \mathcal{D}$. Symmetrically, br_3 contains one bridge rule $p \leftarrow (1 : p), \text{not } (2 : p)$ for each $p \in \mathcal{D}$. The intuition is that the belief sets of C_2 will be subsets of the belief set of C_1 in any equilibrium, and hence possible privacy preserving answers. C_3 exists only for technical reasons.

For i such that $u_{i-2} = u$, thus for the context C_i of the querying user, we add one bridge rule $p \leftarrow (2 : p)$ for each $p \in \mathcal{D}$. This means that in any equilibrium the belief set for i will contain all consequences of the privacy preserving answer with respect to u's knowledge base.

For each i where $3 \leq i \leq |\mathbf{U}|+2$ such that $u_{i-2} \neq u$, thus for contexts representing non-querying users, br_i contains one bridge rule $p_1 \leftarrow (j : p_1), \ldots, (j : p_l), \text{not } (i : p_1)$ for $u_j = u$ and $\{p_1, \ldots, p_l\} \in \mathbf{Priv}(u_{i-2})$. The idea is that no belief state can be an equilibrium, in which the querying user derives information which u_{i-2} wants to keep private.

Proposition 1. *Given a* $\mathbf{P3}$ $(\mathbf{MKB}, \mathbf{U}, \mathbf{BK}, \mathbf{Priv}, Q, u)$*, each equilibrium belief state* $(S_1, S_2, S_3, S_4, \ldots, S_{|\mathbf{U}|+3})$ *for* $M_{\mathbf{P3}}$ *is such that* S_2 *is a privacy preserving answer to* $\mathbf{P3}$*. Also, each privacy preserving answer* S *to* $\mathbf{P3}$ *is the second component of an equilibrium for* $M_{\mathbf{P3}}$*.*

Example 3. In the example examined above, consider the $\mathbf{P3}$ $(\mathsf{MedKB}, \{john, jane, acct\}, \mathbf{BK}, \mathbf{Priv}, \mathsf{symptom}(john, X), acct)$. Note that we did not define background knowledge bases for users $john$ and $jane$, but their nature is not important for the example, just assume that they exist. We also have not defined any privacy statement for $acct$, but also this is not important for our example and we will assume that it is empty, that is, $acct$ does not require anything to be kept private. We construct a multi-context system $(C_1, C_2, C_3, C_4, C_5, C_6)$ where $C_1 = (L_{\mathsf{MedKB}}, \mathsf{MedKB}, \emptyset)$, $C_2 = (L_{\mathsf{MedKB}}, \emptyset, br_2)$ with bridge rules br_2 being

$$\mathsf{symptom}(john, s_1) \leftarrow (1 : \mathsf{symptom}(john, s_1)), \mathrm{not}\ (3 : \mathsf{symptom}(john, s_1))$$
$$\mathsf{symptom}(john, s_2) \leftarrow (1 : \mathsf{symptom}(john, s_2)), \mathrm{not}\ (3 : \mathsf{symptom}(john, s_2))$$
$$\mathsf{symptom}(john, s_3) \leftarrow (1 : \mathsf{symptom}(john, s_3)), \mathrm{not}\ (3 : \mathsf{symptom}(john, s_3))$$

then $C_3 = (L_{\mathsf{MedKB}}, \emptyset, br_3)$ with bridge rules br_3 being

$$\mathsf{symptom}(john, s_1) \leftarrow (1 : \mathsf{symptom}(john, s_1)), \mathrm{not}\ (2 : \mathsf{symptom}(john, s_1))$$
$$\mathsf{symptom}(john, s_2) \leftarrow (1 : \mathsf{symptom}(john, s_2)), \mathrm{not}\ (2 : \mathsf{symptom}(john, s_2))$$
$$\mathsf{symptom}(john, s_3) \leftarrow (1 : \mathsf{symptom}(john, s_3)), \mathrm{not}\ (2 : \mathsf{symptom}(john, s_3))$$

then $C_4 = (L_{\mathbf{BK}(john)}, \mathbf{BK}(john), br_4)$ with bridge rules br_4 being

$$\mathsf{disease}(john, cancer) \leftarrow (6 : \mathsf{disease}(john, cancer)), \mathrm{not}\ (4 : \mathsf{disease}(john, cancer))$$

then $C_5 = (L_{\mathbf{BK}(jane)}, \mathbf{BK}(jane), br_5)$ with bridge rules br_5 being

$$\mathsf{disease}(jane, aids) \leftarrow (6 : \mathsf{disease}(jane, aids)), \mathrm{not}\ (5 : \mathsf{disease}(jane, aids)$$

and finally $C_6 = (L_{\mathbf{BK}(acct)}, \mathbf{BK}(acct), br_6)$ with bridge rules br_6 being

$$\mathsf{symptom}(john, s_1) \leftarrow (2 : \mathsf{symptom}(john, s_1))$$
$$\mathsf{symptom}(john, s_2) \leftarrow (2 : \mathsf{symptom}(john, s_2))$$
$$\mathsf{symptom}(john, s_3) \leftarrow (2 : \mathsf{symptom}(john, s_3))$$

$M_{\mathbf{P3}}$ has six equilibria

$$E_1 = (S_{\mathsf{MedKB}}, Ans_1, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_1, Ans_1, \emptyset, \emptyset)$$
$$E_2 = (S_{\mathsf{MedKB}}, Ans_2, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_2, Ans_2, \emptyset, \emptyset)$$
$$E_3 = (S_{\mathsf{MedKB}}, Ans_3, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_3, Ans_3, \emptyset, \emptyset)$$
$$E_4 = (S_{\mathsf{MedKB}}, Ans_4, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_4, Ans_4, \emptyset, \emptyset)$$
$$E_5 = (S_{\mathsf{MedKB}}, Ans_5, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_5, Ans_5, \emptyset, \emptyset)$$
$$E_6 = (S_{\mathsf{MedKB}}, Ans_6, \mathbf{Ans}(\mathsf{symptom}(john, X)) \setminus Ans_6, Ans_6, \emptyset, \emptyset)$$

where S_{MedKB} is as in Example 1 and the second belief set of each E_i is exactly the respective Ans_i of Example 2 and the third belief set is the complement of Ans_i with respect to $\mathbf{Ans}(\mathsf{symptom}(john, X)) = \{\mathsf{symptom}(john, s_1), \mathsf{symptom}(john, s_2), \mathsf{symptom}(john, s_3)\}$.

We would like to point out that in this construction the original knowledge bases are not changed, we only create contexts and bridge rules. All of the background knowledge bases could be multi-context systems themselves; for instance, if the user model for $acct$ foresees that $acct$ is aware of SNOMED and PEPID, then $acct$'s background knowledge base could be a multi-context system comprising these two medical knowledge bases.

In order to obtain maximal privacy preserving answers using the described construction, the simplest way is to postprocessing all privacy preserving answers. More involved solutions would have to interfere with the underlying multi-context system reasoner, for instance by dynamically changing the multi-context system. It is not clear to us at the moment whether it is possible to modify the construction such that the equilibria of the obtained multi-context system correspond directly to the maximal privacy preserving answers.

4 Conclusion and Future Work

We have presented a definition of the privacy preservation problem, which allows for using knowledge bases of different kinds. Finding privacy preserving answers can then be accomplished by building an appropriate multi-context system and computing one of its belief states. Since systems for solving multi-context systems begin to emerge, for example DMCS [1], this also implies that these privacy preserving answers can be effectively computed.

However, usually one is interested in maximal privacy preserving answers. It is unclear to us whether a similar construction as the one presented in this paper can be used for finding privacy preserving answers which are maximal, by just creating appropriate contexts and bridge rules and without modifying the involved knowledge bases or adding new knowledge bases of particular logics. One possible line of investigation would be to examine work on diagnosing inconsistent multi-context systems [5, 2], since in diagnosis tasks there is an implicit minimization criterion, which could be exploited for encoding maximality.

References

1. Bairakdar, S.E., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The dmcs solver for distributed nonmonotonic multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010). Lecture Notes in Computer Science, vol. 6341, pp. 352–355. Springer Verlag (2010)
2. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The mcs-ie system for explaining inconsistency in multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010). Lecture Notes in Computer Science, vol. 6341, pp. 356–359. Springer Verlag (2010)
3. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-2007). pp. 385–390. AAAI Press (2007)
4. Dix, J., Faber, W., Subrahmanian, V.: The Relationship between Reasoning about Privacy and Default Logics. In: Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005. Lecture Notes in Computer Science, vol. 3835, pp. 637–650. Springer Verlag (Dec 2005)

5. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In: Lin, F., Sattler, U., Truszczyński, M. (eds.) Proceedings of the Twelfth International Conference on Knowledge Representation and Reasoning (KR 2010). AAAI Press (2010)
6. Zhao, L., Qian, J., Chang, L., Cai, G.: Using ASP for knowledge management with user authorization. Data & Knowledge Engineering 69(8), 737–762 (2010)

ISSN 1613-0073

Author Index