Conflicts in Generalised Modular Logic Programming

Carlos Viegas Damásio and João Moura*

CENTRIA – Centre for Artificial Intelligence, Universidade Nova de Lisboa, Portugal

Abstract. Modularity has been studied extensively in conventional logic programming and incorporating modularity into Answer Set Programming has also become popular in the last few years. A major approach is Oikarinnen and Janhunen’s Gaifman-Shapiro-style architecture of program modules, which provides the composition of program modules. Recently one shortcoming to their approach, imposed in order to ensure the compatibility of their module system with the stable model semantics and that forcing output signatures of composing modules to be disjoint, has been lifted [14]. However, lifting such requirements implies that conflicts will occur in composed modules mostly because their design is error-prone and also due to the lack of automated conflict detection and resolution. We describe an improved modularity definition for logic programs, identify its basic conflict types and characterize these conflicts both in terms of Strong and Uniform Equivalence of logic programs and in terms of a novel Brave Equivalence. The conflicts are studied according to several dimensions, and a hierarchy of conflict types is proved to exist. These theoretical characterizations allow for the automatic identification of such conflicts, among other reasoning tasks.

1 Introduction

Recent work on the framework of modular answer set programming [15, 4] lifted some restrictive conditions that were disallowing logic programming modules with common outputs to be composed [14]. When one starts composing modules with this characteristic, conflicts in composed modules start to emerge. Some of these conflicts are obvious and take the form of inconsistencies but other are dormant in the specifications of composed modules and will only manifest in the presence of modules that actually instantiate these potential conflicts and turn them into inconsistencies. In this paper, we formally characterize these conflicts in terms of equivalence notions, effectively allowing their detection by automated systems that are available off the shelf.

Overview. We begin this Section by describing the relevant state of the art in answer set programming [10]. We also present an overview of the relevant work in the context of modular logic programs as well as, towards the logical characterization of conflicts in logic programming, an overview of strong and uniform equivalence in the context of logic programs due to [11, 6] and relativized versions of these notion due to [7]. We then

* The work of João Moura was supported by the grant SFRH/BD/69006/2010 from Fundação para a Ciência e Tecnologia (FCT) from the Portuguese MEC - Ministério do Ensino e da Ciência.
present a novel Brave interpretation of Strong Equivalence. In the following section we present a generalization of modular logic programming (GMLP). In Section 3 we present a composition semantics for the generalized version of GMLP. Next, in Section 4 we define different classes of program modules according to whether they contain intentional or extensional knowledge. In Section 5 we discuss possible characterizations of conflicts and end with conclusions and future work.

1.1 Answer set programming paradigm

First, we review the syntax of answer set programs:

Logic programs in the answer set paradigm are formed by finite sets of rules. A rule \( r \) has the syntax:

\[
L_1 \leftarrow L_2, \ldots, L_m, \sim L_{m+1}, \ldots, \sim L_n \quad (n \geq m \geq 0)
\]

where each \( L_i \) is either a logical atom \( A \) or the strong negation of an atom \( \neg A \) without occurrence of function symbols (arguments are either variables or constants of the logical alphabet).

- \( \text{Head}_P(r) = L_1 \): the literal in the head of (1).
- \( \text{Body}_P^+(r) = \{L_2, \ldots, L_m\} \): the set with all positive literals in the body of (1).
- \( \text{Body}_P^-(r) = \{L_{m+1}, \ldots, L_n\} \): the set with all negative literals in the body of (1).
- \( \text{Body}_P(r) = \{L_2, \ldots, L_n\} \): the set containing all literals in the body of (1).

If a program is positive we will omit the superscript in \( \text{Body}_P^+(r) \). Also, if the context is clear we will omit the subscript mentioning the program and write simply \( \text{Head}(r) \) and \( \text{Body}(r) \) as well as the argument mentioning the rule. The semantics of answer sets is defined via the reduct operation \([10]\). Given an interpretation \( M \) (a set of ground atoms), the reduct \( P^M \) of a program \( P \) with respect to \( M \) is the program

\[
P^M = \{ \text{Head}(r) \leftarrow \text{Body}^+(r) \mid r \in P, \text{Body}^-(r) \cap M = \emptyset \}.
\]

The interpretation \( M \) is an answer set of \( P \) iff \( M \) is the least model of program \( P^M \). The syntax of logic programs has been extended with other constructs, namely weighted and choice rules \([7]\). Besides choice facts \( \{c\} \), meaning that \( c \) may be present in the model or not. These constructs can be encoded as ordinary rules, and thus without loss of generality, we restrict the discussion to normal logic programs, not containing strongly negated literals nor these extensions.

1.2 Modular Logic Programming

The modular aspects of Answer Set Programming have been clarified in recent years \([15, 4]\) describing how and when can two program parts (modules) be composed together. In this paper, we will make use of Oikarinen and Janhunen’s logic program modules defined in analogy to \([9]\).

**Definition 1 (Module [15]).** A logic program module \( P \) is a tuple \( \langle R, I, O, H \rangle \) s.t. :

1. \( R \) is a finite set of rules;
2. \( I, O, \) and \( H \) are pair wise disjoint sets of input, output, and hidden atoms;
3. \( \text{At}(R) \subseteq \text{At}(P) \) defined by \( \text{At}(P) = I \cup O \cup H \); and
4. \( \text{Head}(R) \notin I \).
The set of atoms in $At_v(P) = I \cup O$ are considered to be visible and hence accessible to other modules composed with $P$ either to produce input for $P$ or to make use of the output of $P$. We use $At_i(P) = I$ and $At_o(P) = O$ to represent the input and output signatures of $P$, respectively. The hidden atoms in $At_h(P) = H = At(P) \setminus At_v(P)$ are used to formalize some auxiliary concepts of $P$ which may not be sensible for other modules but may save space substantially. The condition $\text{head}(R) \notin I$ ensures that a module may not interfere with its own input by defining input atoms of $I$ in terms of its rules. Thus, input atoms are only allowed to appear as conditions in rule bodies.

The answer set semantics is generalized to cover modules by introducing a generalization of the Gelfond-Lifschitz’s fixpoint definition. In addition to negative default literals (i.e., $\neg$), also literals involving input atoms are used in the stability condition.

**Definition 2 (Answer Sets of a Module [15])**. An interpretation $M \subseteq At(P)$ is an answer set of an ASP program module $P = \langle R, I, O, H \rangle$, if and only if $M = LM(R^M \cup \{a | a \in M \cap I\})^1$. The set of answer sets of $P$ is denoted by $AS(P)$.

### 1.3 Visible and Modular Equivalence

The notion of visible equivalence has been introduced in order to neglect hidden atoms when logic programs are compared on the basis of their models. The compositionality property from the module theorem enabled the authors to port this idea to the level of program modules—giving rise to modular equivalence of logic programs.

**Definition 3. [15]** Given two logic program modules $P$ and $Q$ we say that they are:

**Visibly equivalent** $P \equiv_v Q$ iff $At_v(P) = At_v(Q)$ and there is a bijection $f : AS(P) \rightarrow AS(Q)$ such that for all $M \in AS(P)$, $M \cap At_v(P) = f(M) \cap At_v(Q)$. And they are

**Modularly equivalent** $P \equiv_m Q$ iff $At_i(P) = At_i(Q)$ and $P \equiv_v Q$.

So, two modules are visibly equivalent if there is a bijection among their answer sets, and they coincide in their visible parts. If additionally, the two program modules have the same input and output atoms, then they are modularly equivalent.

### 1.4 Strong Equivalence of Logic Programs

Towards the logical characterization of conflicts in access control, we first refer the reader to an introduction to the logic of here-and-there and then to strong equivalence in the context of logic programs due to [11].

Basic concepts of equilibrium logic, an approach to non-monotonic reasoning developed by [16] as a generalization of the answer-set semantics for logic programs are also assumed. For more details, the reader is referred to [11, 16–19].

---

1 Note that $R^M$ is the version of the reduct operation allowing weighted and choice rules, and $LM$ is the operator returning the least model of the positive program argument.
**Strong Equivalence Theorem** In the statement of the theorem, formulas and rules are identified in the sense of nested logic programs[12] without strong negation and with propositional formulas. Accordingly, programs become a special case of theories, and we can talk about the equivalence of programs in the logic of here-and-there.

**Theorem 1.** For any programs $P$ and $Q$, the following conditions are equivalent:

(a) for every program $R$, programs $P \uplus R$ and $Q \uplus R$ have the same answer sets,

(b) $P$ is equivalent to $Q$ in the logic of here-and-there.

The fact that (a) is equivalent to (b) expresses the correspondence between the strong equivalence of logic programs and the equivalence of formulas in the logic of here-and-there.

### 1.5 Relativized Notions of Strong and Uniform Equivalence

In what follows, we revise the notion of relativized strong equivalence (RSE) and relativized uniform equivalence (RUE) due to [7].

**Definition 4 (Relativized Strong and Uniform Equivalence).** Let $P$ and $Q$ be programs and let $A$ be a set of atoms. Then,

(i) $P$ and $Q$ are strongly equivalent relative to $A$, denoted $P \equiv^A Q$, iff $P \cup R \equiv Q \cup R$, for all programs $R$ over $A$;

(ii) $P$ and $Q$ are uniformly equivalent relative to $A$, denoted $P \equiv^u_A Q$, if-and-only-if $P \cup F \equiv Q \cup F$, for all (non-disjunctive) facts $F \subseteq A$.

Observe that the range of applicability of these notions covers ordinary equivalence (by setting $A = \emptyset$) of two programs $P$, $Q$, and general strong (resp. uniform) equivalence (whenever $\text{Atm}(P \cup Q) \subseteq A$). Also the following relation holds: For any set $A$ of atoms, let $A' = A \cap \text{Atm}(P \cup Q)$. Then, $P \equiv^e_A Q$ holds, if-and-only-if $P \equiv^{A'}_e Q$ holds, for $e \in \{s, u\}$.

### 2 Generalised Modularity in Answer Set Programming

In this section we recall two versions of compositions as presented in [14]: (1) A relaxed composition operator ($\uplus$), aiming at maximizing information in the answer sets of modules. Unfortunately, we show that this operation is not compositional. (2) A conservative composition operator ($\otimes$), aiming at maximizing compatibility of atoms in the answer sets of modules. This version implies redefining the composition operator by resorting to a program transformation but uses the original join operator.

#### 2.1 Extra module operations

First we require a fundamental renaming operation for output atoms:
Definition 5 (Output renaming). Let $\mathcal{P}$ be the program module $\mathcal{P} = (R, I, O, H)$, $o \in O$ and $o' \not\in \text{At}(\mathcal{P})$. The renamed output program module $\rho_{o' \leftarrow o} (\mathcal{P})$ is the program module $\langle R' \cup \{ \bot \leftarrow o', \sim o \}, I \cup \{ o \}, \{ o' \} \cup (O \setminus \{ o \}), H \rangle$. The program part $R'$ is constructed by substituting the head of each rule $o \leftarrow \text{Body}$ in $R$ by $o' \leftarrow \text{Body}$. The heads of other rules remain unchanged, as well as the bodies of all rules.

Mark that, by making $o$ an input atom, the renaming operation can introduce extra answer sets. However, the original answer sets can be recovered by selecting the models where $o'$ has exactly the same truth-value of $o$. The constraint throws away models where $o'$ holds but not $o$. We will abuse notation and denote $\rho_{o' \leftarrow o_1} \ldots (\rho_{o'_n \leftarrow o_n} (\mathcal{P})) \ldots$ by $\rho_{\{ o'_1, \ldots, o'_n \} \leftarrow \{ o_1, \ldots, o_n \}} (\mathcal{P})$. Still before we dwell any deeper in this subject, we define operations useful to project or hide sets of atoms from a module.

Definition 6 (Hide and Project Operators). Let $\mathcal{P} = (R, I, O, H)$ be a module and $S$ an arbitrary set of atoms. If we want to hide $S$ from program module $\mathcal{P}$, then use $\mathcal{P} \setminus S = (R \cup \{ i \}, \{ i \in I \cap S \}, I \cap S, O \cap S, H \cup ((I \cup O) \cap S))$. Dually, we can project over $S$ by letting $\mathcal{P} |_S = (R \cup \{ i \}, \{ i \in I \setminus S \}, I \setminus S, O \setminus S, H \cup ((I \cup O) \setminus S))$.

Both operators do not change the answer sets of the original program, i.e. $AS(\mathcal{P}) = AS(\mathcal{P} \setminus S) = AS(\mathcal{P} |_S)$ but do change the set of visible atoms $At_v(\mathcal{P} \setminus S) = At_v(\mathcal{P} |_S)$ and $At_v(\mathcal{P} |_S) = At_v(\mathcal{P}) \cap S$

2.2 Relaxed Output Compatibility

For the reasons presented before, we start by defining a generalised version of the composition operator, by removing the condition enforcing disjointness of the output signatures of the two modules being combined.

Definition 7 (Relaxed Composition). Given two modules $\mathcal{P}_1 = (R_1, I_1, O_1, H_1)$ and $\mathcal{P}_2 = (R_2, I_2, O_2, H_2)$, their composition $\mathcal{P}_1 \uplus \mathcal{P}_2$ is defined when they respect each other’s hidden atoms, i.e., $H_1 \cap At(\mathcal{P}_2) = \emptyset$ and $H_2 \cap At(\mathcal{P}_1) = \emptyset$. Then their composition is $\mathcal{P}_1 \uplus \mathcal{P}_2 = (R_1 \cup R_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, H_1 \cup H_2)$.

Having defined the way to deal with common outputs in the composition of modules, we would like to redefine the operator $\bowtie$ for combining the answer sets of these modules. However, this has been shown to be impossible in cite. As we have motivated in the introduction, it is important to applications to be able to use $\uplus$ to combine program modules, and retain some form of compositionality. The following definition presents a construction that adds the required information in order to be able to combine program modules using the original natural join.

Definition 8 (Relaxed Compatibility Transformation). Consider the program modules $\mathcal{P}_1 = (R_1, I_1, O_1, H_1)$ and $\mathcal{P}_2 = (R_2, I_2, O_2, H_2)$. Let $O = O_1 \cap O_2$, and define the sets of newly introduced atoms $O' = \{ o' \mid o \in O \}$ and $O'' = \{ o'' \mid o \in O \}$. Construct program module $\mathcal{P}_3 = (R_3, O' \cup O'', O, \emptyset)$ where $R_3 = \{ o \leftarrow o' \mid o' \in O' \} \cup \{ o \leftarrow o'' \mid o'' \in O'' \}$. The relaxed compatibility composition is defined as the program module $(\mathcal{P}_1 \uplus^{\text{RT}} \mathcal{P}_2) = [\rho_{O' \leftarrow O}(\mathcal{P}_1) \cup \rho_{O'' \leftarrow O}(\mathcal{P}_2) \cup \mathcal{P}_3] \setminus (O' \cup O'')$. 
Intuitively, we rename the common output atoms in the original modules, and introduce an extra program module that unites the contributions of each module by a pair of rules for each common atom \( o \leftarrow o' \) and \( o \leftarrow o'' \). We then hide all the auxiliary atoms to obtain the original visible signature. If \( O = \emptyset \) then \( P_3 \) is empty, and all the other modules are not altered, falling back to the original definition.

**Theorem 2.** Let \( P_1 \) and \( P_2 \) be arbitrary program modules without positive dependencies among them. Then, \( P_1 \sqcup P_2 \equiv_m P_1 \sqcupRT P_2 \).

The important remark is that according to the original module theorem we have \( AS(\rho_{\nu} \cdot O(P_1) \sqcup \rho_{\nu} \cdot O(P_2) \sqcup P_3) = AS(\rho_{\nu} \cdot O(P_1)) \bowtie AS(\rho_{\nu} \cdot O(P_2)) \bowtie AS(P_3) \). Therefore, from a semantical point of view, users can always substitute module \( P_1 \sqcup P_2 \) by \( P_1 \sqcupRT P_2 \). This has an extra cost, since the models of the renamed program modules may increase. This is, however, essential to regain compositionality.

### 2.3 Conservative Output Compatibility

The following composition method aims at maximizing compatibility of atoms in the answer sets of their modules. This implies redefining the original composition operator (\( \oplus \)) resorting to a program transformation s.t. it remains compositional with respect to the join operator (\( \sqcup \)). The transformation we present next consists of taking Definition 8 and adding an extra module to guarantee that only compatible models are retained.

**Definition 9 (Compatibility Composition).** Let \( P_1 = (R_1, I_1, O_1, H_1) \) and \( P_2 = (R_2, I_2, O_2, H_2) \) be modules such that \( O = O_1 \cap O_2 \neq \emptyset \). Let \( O' = \{ o' \mid o \in O \} \) and \( O'' = \{ o'' \mid o \in O \} \) sets of newly introduced atoms. Construct program modules \( P_3 = (R_3, O' \cup O'', O, \emptyset) \) where \( R_3 = \{ o \leftarrow o' \mid o' \in O' \} \cup \{ o \leftarrow o'' \mid o'' \in O'' \} \), and \( P_4 = (\{ \bot \leftarrow o', \top \leftarrow o'' \mid o \in O \}, O' \cup O'', \emptyset, \emptyset) \). The conservative compatibility composition is defined as the program module: \( P_1 \otimes P_2 = [(\rho_{\nu} \cdot O(P_1) \sqcup \rho_{\nu} \cdot O(P_2) \sqcup P_3 \sqcup P_4) \setminus (O' \cup O'') \)

Note here that each clause not containing atoms that belong to \( O_1 \cap O_2 \) in \( P_1 \cup P_2 \) is included in \( P_1 \otimes P_2 \). So, if there are no common output atoms the original union based composition is obtained. Therefore, it is easy to see that this transformational semantics (\( \otimes \)) is a conservative extension to the existing one (\( \oplus \)).

**Theorem 3 (Conservative Module Theorem).** If \( P_1, P_2 \) are modules such that \( P_1 \otimes P_2 \) is defined, then \( M \in AS(P_1 \otimes P_2) \text{ iff } M \cap (At_u(P_1) \cup At_u(P_2)) \in AS(P_1) \bowtie AS(P_2) \).

### 3 Compositional Semantics for Modular Logic Programming

As we have seen, the existing compositional semantics does not work directly for modules with common outputs. However, it should be clear from the previous section that the effect of the extra renamed modules \( P_3 \) and \( P_4 \) is to impose extra conditions on the compatible models of the other two modules, which are the same for both composition forms. In fact, by changing the definition of answer sets of a module we can obtain a fully compositional semantics.
Definition 10 (Modular models of a generalised MLP). Given a program module \( P = \langle R, I, O, H \rangle \) its modular models are \( MMod(P) = AS(\rho_{O \leftarrow O}(P)) \).

Our MModels are models of the original program with extra annotations conveying if an atom is supported or not. This is a little more complex when hidden atoms are involved, but the interpretation of the extra atoms is the same: whenever \( o^* \) is true then there is a rule for \( o \) with true body, otherwise \( o^* \) is false.

We now define the join operators for the two forms of composition:

Definition 11. Given two modules \( P_1 = \langle R_1, I_1, O_1, H_1 \rangle \) and \( P_2 = \langle R_2, I_2, O_2, H_2 \rangle \) and sets of MModels \( A_1 \subseteq 2^{I_1 \cup O_1 \cup O_1^*} \) and \( A_2 \subseteq 2^{I_2 \cup O_2 \cup O_2^*} \), Let:

\[
\begin{align*}
A_1 \sqcup^+ A_2 &= \{ M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2, \text{and } M_1 \cap (I_2 \cup O_2) = M_2 \cap (I_1 \cup O_1) \} \\
A_1 \sqcup^\times A_2 &= \{ M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2, M_1 \cap (I_2 \cup O_2 \cup O_2^*) = M_2 \cap (I_1 \cup O_1 \cup O_1^*) \}
\end{align*}
\]

So, when joining two models we either look at visible atoms of the original modules for the case of the relaxed composition, or look at all visible atoms and extra annotations, thus discarding non-supported modular models. The main result is as follows:

Theorem 4. If \( P_1, P_2 \) are modules such that \( P_1 \uplus P_2 \) and \( P_1 \otimes P_2 \) is defined, then:

\[
\begin{align*}
(1) \ MMod(P_1 \uplus P_2) &= MMod(P_1) \sqcup^+ MMod(P_2) \\
(2) \ MMod(P_1 \otimes P_2) &= MMod(P_1) \sqcup^\times MMod(P_2)
\end{align*}
\]

4 Classes of Logic Programming Modules

We begin this section by presenting the three basic types of modules that can be arbitrarily composed. These reflect modules constructed from fundamental types of rules allowed in LP: normal rules, rules with empty heads (integrity constraints) and rules with empty bodies (facts). Using Definition 1, we respectively define abstract, restriction and instantiating modules as logic programming modules in the following way:

Definition 12 (Abstract module). We define an abstract module as tuple \( P = \langle Ra, Ia, Oa, Ha \rangle \), where:

1. \( Ra \) is a finite set of rules with non-empty bodies.
2. \( Ia \in \text{Body}(Ra) \), \( Oa \in \text{Head}(Ra) \) and \( Ha \) respectively as the set of configurable atoms, the set of decision atoms and the set of hidden atoms.

An abstract module has a set of interpretations \( I = \{ I_1, \ldots, I_n \} \), each containing atoms \( o \in Oa \), obtained through applicable rules in \( Ra \) by varying the possible values of atoms \( i \in Ia \) over their range. The following is an ACP in the form of an abstract module.

Example 1 (Abstract module). \( P = \{ \text{deny(download, Resource) \leftarrow public(Resource), allowRead(Resource), denyWrite(Resource), allow(download,Resource) \leftarrow public(Resource),} \}, \{\text{allowRead, denyWrite, public, deny, allow}\} \}

Definition 13 (Restriction module). We define a restriction module as tuple \( IC = \langle Rr, Ir, Or, Hr \rangle \), where:

7
1. $R_r$ is a finite set of rules (integrity constraints) of the form: $\bot \leftarrow A_1, \ldots, A_n, \sim B_1, \ldots, \sim B_m$.

2. $O_r = \emptyset$.

A restriction module has a set of interpretations $I = \{I_1, \ldots, I_n\}$, each containing atoms $o \in O_r$, obtained through applicable integrity constraint rules in $R_r$ by varying the possible values of atoms $i \in I_r$ over their range.

Example 2 (Restriction module). A model containing only integrity constraints: $P = \langle \{\leftarrow nota, b.\}, \{a, b\}, \emptyset, \emptyset \rangle$

Next we define a module composed of a set of facts.

Definition 14 (Instantiating module). Given a restriction module $R = \langle R_r, I_r, O_r, H_r \rangle$ and an abstract module $A = \langle R_a, I_a, O_a, H_a \rangle$, we define a set of facts as $F = \langle R_f, I_f, O_f, H_f \rangle$, where:

1. $R_f$ is a finite set of rules s.t. $At(R_f) \in I_a$ s.t. $AS(F \cup I_a) \neq \emptyset$.
2. $I_f = \emptyset$ and $O_f = At(R_f) H_f$.

An instantiating module has a set of interpretations $I = \{I_1, \ldots, I_n\}$, each containing atoms $o \in O_r$ over their range. The following is a set of instantiating facts in the form of an instantiating module.

Example 3 (Instantiating module). $P = \langle \{public(resource). allowRead(resource). denyWrite(resource). \}, \emptyset, \{allowRead, denyWrite, public\}, \emptyset \rangle$

5 Conflicts and Redundancies in ASP Modules

A conflict occurs when the objectives of two or more modules cannot be simultaneously met. In the literature (e.g., [13, 21]), authors have summarized three types of fundamental conflicts, defined as modality, redundancy and potential conflicts.

5.1 Basic Types of Conflicts and Redundancies

We start this subsection by giving intuitions of what are the different types of conflicts. Importantly note that we will always use the relativized notions of Strong and Uniform equivalence, even when we do not explicitly say so. In some cases we conjecture that this implies a small loss of generality that allows the definitions not to collapse into SE.

Modality Conflict Modality conflicts are inconsistencies in the specification of concrete modules which may arise when two or more modules have inconsistent outputs. Simply put, a modality conflict occurs when an integrity constraint is violated or when two modules are inconsistent. The module in Example 3 produces a conflict when joined with the one in Example 1.
**Redundancies** Modules that never apply, or modules that always produce the same answer sets, can be called redundant modules. Even though these redundancies have no influence in the composition of modules, they should be identified and dealt with. We distinguish these from policies that are compatible. These are modules that are always composable in a sense that if they both have answer sets then at least a pair will be joinable.

**Potential Conflict** A potential conflict is a type of conflict between two abstract policies having overlaps in their conditions. These abstract policies are such that when extensional knowledge is added, the resulting concrete policies contain rules s.t. their associated conditions are simultaneously satisfied and result in a redundancy or in a modality conflict. The abstract policy in Example 1 has a potential conflict because when instantiated with the facts of Example 3 we obtain a conflict between *allow* and *deny* literals.

We now have three different dimensions according to which we can characterize the identified conflicts: strong or uniform equivalence, brave or cautious reasoning and abstract or concrete policies. Orthogonally, we can also identify two forms of redundancies and compatibility that may be present in policies.

**Definition 15 (Incoherent Modules).** Let modules $P = (R_P, I_P, O_P, H_P)$, $R = (R_I, I_R, O_R, H_R)$ and $F = (R_F, I_F, O_F, H_F)$ respectively be abstract, restricting and instantiating modules, we define:

A1 **Incoherent module** Occurs when an abstract or instantiating module is incoherent, e.g., $P = \{(a \leftarrow \text{not } a), \{\}, \{a\}, \{\}\}$.

A2 **Incoherent composition** Occurs when the composition of modules produces or maintains an incoherence, being that explicit e.g., $F = (R=b, O=b)$, $P = (R=a \leftarrow b, \text{not } a, O = b, I = b)$ or implicit e.g., $F_1 = (R=b, O=b)$, $F_2 = (R=, O=b)$. The latter case can be solved by allowing only the composition of modules with disjoint outputs, while the former is solved by forcing an abstract module to have models for each possible input.

A2 **Recoverable Inconsistency** Occurs when a module $P$ is inconsistent but its composition with an instantiating module produces answer sets, i.e., $\forall F [AS(P \cup F) \neq \emptyset]$ and not (A1).

e.g., $P = \{-a \leftarrow \text{not } b, a \leftarrow \text{not } b\} \Rightarrow F = \{b\}$

A2 **Uniform Irrecoverable inconsistency** Occurs when the intentional part of a policy is inconsistent. Adding intentional facts will never produce a consistent answer set but there is a program s.t. adding it to the policy yields at least one consistent answer set: $\forall F [AS((P \cup \{a \leftarrow \text{allow, deny}\}) \cup F) = \emptyset]$ and not (C1).

**Definition 16 (Safe Abstract Modules).** Let $P$, $R$ and $F$ respectively be abstract, restricting and instantiating modules:

---

2 Due to non-monotonicity, programs may be incoherent, i.e., lack answer sets due to odd cyclic dependencies of an atom from its default negation being that dependency explicit or not.
S1 Safe Abstract Module  An abstract module that composed with consistent instantiating modules will have answer sets. i.e., \( \forall F [ AS(P \bowtie F) \neq \emptyset] \).

\( P = a \leftarrow \text{not} - b. \)

S2 Strongly Safe Abstract Module  An abstract module that composed with consistent instantiating modules and restricting modules will always have answer sets. \( \forall R [ AS(P \bowtie F \bowtie R) \neq \emptyset] \).

We also need to have here the notion of safe instantiating module.

Definition 17 (Conflicts in Instantiated Modules). Let \( P, R \) and \( F \) respectively be abstract, restricting and instantiating modules:

C1 Cautious Conflict Occurs when all answer sets from \( P \bowtie F \) become inconsistent when composed with some module \( R \): \( \forall SM \in AS(P \bowtie F) \supset SM \models \text{rule} \in R. \)

\( P = \{ a \leftarrow \text{not} - b. \}, F = \{ b. \}, R = \{ a. \}. \)

C2 Brave Conflict Occurs when some answer sets from \( P \) becomes inconsistent when composed with some modules \( F \) or \( R \): \( \exists SM \in AS(P \bowtie F) \supset SM \models \text{rule} \in R, \) and not (C1).

\( P = \{ a \leftarrow \text{not} b. b \leftarrow \text{not} a. \}, F = \{ \}, R = \{ a. \}. \)

WHAT ABOUT \( a \leftarrow \text{not} b. b \leftarrow \text{not} a. \bowtie - a. \)? For this to be considered, a notion of composition on the answer set level is necessary.

Definition 18 (Redundancies). Let \( P, Q \) and \( R \) be ACPs in the form of programs and \( F \) sets of facts:

R1 Redundancy (strong equivalence | cautious reasoning | _ _): Occurs between two policies that, even by adding any program \( R \), will always produce the same results: \( P \cup R \models \text{allow} \) and \( Q \cup R \models \text{allow} \) or: \( P \cup R \models \text{deny} \) and \( Q \cup R \models \text{deny} \).

R2 Compatibility ( uniform equivalence | brave | _ _): Occurs between two policies that, even by adding any set of facts \( F \), will always have one answer set in common (if both have answer sets): \( \forall F, \exists a \in AS(P \cup F), \exists b \in AS(Q \cup F) \) s.t. \( a = b \) if \( AS(P \cup F) \neq \emptyset \) and \( AS(Q \cup F) \neq \emptyset \).

Theorem 5 (Gradation of Conflicts). Let \( P \) be an Access Control Policy. Then, conflict dimensions are totally ordered, i.e., \( C1 \Rightarrow C2 \Rightarrow C3 \Rightarrow C4 \Rightarrow C5 \Rightarrow C6 \).

We present next an intuition for the conflict dimensions occurring in ACPs. The gradations in Theorem 5 form a chain where \( \top \) is (C1) and respectively \( \bot \) is (C6). The redundancy dimensions in Definition 18 are orthogonal to the ones in Definition ??

We will formally characterize these conflicts and link them with the gradation of conflicts presented (for now informally, so to favour intuition) in the following Theorem 5.

5.2 Characterisation of Conflicts

In this subsection we detail finer variants of the previously identified basic conflicts. We relate these dimensions with the gradation presented in Theorem 5.
Potential Conflict  We can find definitions of potential conflicts in the literature such as in [21] stating that a potential conflict occurs between two abstract policies $P$ and $Q$ if:

1. $P$ derives a permission (in the form of an $allow$ literal) and $Q$ derives a prohibition (in the form of an $deny$ literal) and,
2. There are overlaps such as: $body(P) \cap body(Q) \neq \emptyset$, and
3. There is no policy $R$ in the policy set such that $condition(P) \land condition(Q) \rightarrow condition(R)$ and $R$ derives a prohibition when $priority(P) \prec priority(R)$ or $R$ derives a permission when $priority(Q) \prec priority(R)$.

According to this definition, when some policies have the same condition literals it is possible to infer the existence of potential conflicts among these policies. Consequently, potential conflicts are highly pervasive in access control systems. This definition however does not consider for instance $\{a \leftarrow b. \neg a \leftarrow c.\}$ to be a potential conflict and furthermore takes in account a notion of ordering that turns out to be too restrictive. We present next a general definition of potential conflicts:

Definition 19 (Potential Conflict). Let $P$ be an abstract policy, there is a potential conflict if the policy neither contains an irreversible inconsistency in the intentional part:

$$P \cup \{\bot \leftarrow allow \land deny\} \equiv \{\bot \leftarrow \top\}$$

(C1)

nor the policy is completely safe (considering that allow and deny cannot be added):

$$P \cup \{\bot \leftarrow allow \land deny\} \equiv P$$

(C6)

These conflicts in Definition 19 do not correspond to any of the gradations presented before because there is not yet a concrete conflict. Still, safe and irreversibly inconsistent policies correspond respectively to (C6) and (C1).

Example 4 (Irrecoverable inconsistencies). The following logic program contains an inconsistency that cannot be repaired by adding extensional (C2) or even intentional (C1) knowledge, in this case $goodtester$ is the only extensional atom.

allow :- tester. deny :- coder.
tester :- coder. coder :- tester.
tester :- goodtester. coder :- not goodtester.

Potential conflicts have been analyzed in the extension of Lobo’s PDL with ordered disjunction by [1], where logic programming with ordered disjunction was used as a way to prioritize action execution in case conflicting actions were triggered.

Modality Conflict  Modality conflicts are generally defined in the literature (adapted to our context) as follows: Given a concrete modular logic program $P$, we say that there is a modality conflict if it derives a contradiction between atoms that are inconsistent in the presence of some integrity constraint ruling them:

$$P \models b' \land \neg b''$$

(C4)
The next conflict dimension is stronger since this conflict must be present in every answer set (using cautious reasoning):

\[ P \models_c o' \land \neg o'' \quad \text{(C3)} \]

**Redundancies and Compatibilities** Two policies \( P \) and \( Q \) are redundant if for every program \( R \), \( P \cup R \) and \( Q \cup R \) always have the same answer sets (filtering \textit{allow} and \textit{deny}). Two policies \( P \) and \( Q \) are compatible if for every set of facts \( F \), if \( P \cup R \) and \( Q \cup R \) both have answer sets, then they have one in common (filtering \textit{allow} and \textit{deny}).

**Definition 20 (Redundancy in terms of RSE).** Let \( P \) and \( Q \) be policies, they are Redundant if \( P \cup Q \) is strongly equivalent (filtering allow and deny) to \( Q \):

\[ P \cup Q \equiv Q \quad \text{(R1)} \]

The characterization of redundancy in terms of strong equivalence of logic programs has been noted in the past e.g., in [6, 8] but we adapt it to the context of access control by using strong equivalence. Furthermore, we introduce the notion of uniform brave equivalence to describe partial redundancies that can also be viewed as "compatible policies".

**Definition 21 (Compatibility in terms of BE).** Let \( P \) and \( Q \) be policies, they are Compatible if \( P \cup Q \) is Bravely Equivalent (filtering allow and deny) to \( Q \):

\[ P \cup Q \equiv_b Q \quad \text{(R2)} \]

A compatibility exists when (if both \( P \) and \( Q \) have answer sets) there is always one answer set (filtering allow and deny) that is the same for \( P \cup F \) and \( Q \cup F \) for every set of facts \( F \). It can also be seen the other way around meaning that when made Concrete by adding a set of extensional facts, they will never disagree. This means there will never be a grade \((C3)\) but a grade \((C4)\) conflict is still possible.

### 5.3 Discussion

In this section we dealt with three different dimensions that we used to finely describe and characterize conflicts in modular logic programming, namely: the notion of abstract, restriction and instantiating modules; whether brave or cautious reasoning is used; whether we consider changes on the intentional and extensional parts (related to strong equivalence) or on the extensional part only (related to uniform equivalence).

We then described potential conflicts as being conflicts occurring in rules on the intentional part of policies that are still abstract. In reality these are not yet conflicts and as such they are characterized as lying in the middle of uniformly safe \((C5)\) or strongly safe \((C6)\) and strongly inconsistent \((C1)\) or uniformly inconsistent \((C2)\). This is the case because they are not yet producing inconsistencies, and only when these policies are instantiated with extensional knowledge and become concrete do these potential problems manifest in the form of cautious modality conflicts \((C3)\) and brave modality conflicts \((C4)\).
In parallel we identify and characterize policies that are redundant as always producing the same results (R1) and policies that are compatible (R2) as being policies that never produce a cautious modality conflict no matter how they are instantiated but are not redundant in the sense that they do not always produce the same results.

6 Conclusions and Future Work

We identified different types of basic conflicts that occur in generalized modular answer set programs and characterize them in terms of the notions of relativized strong equivalence and relativized uniform equivalence of logic programs and in terms of a novel brave interpretation of uniform equivalence that we call brave equivalence. We also introduce a gradation of conflicts in modular answer set programming that can potentially be generalized to DLPs and extended logic programs, although more research is necessary.

These characterizations enables the detection of conflicts to be done automatically by using automatic theorem provers and most importantly the ones identified in [2] where it is stated that the relation they established between $\mathbf{S4F}$ and the logic of Here-and-There, allows using modal $\mathbf{S4F}$ provers for proving theorems in that intermediate logic. Because of the characterization of Strongly Equivalent programs as programs that are equivalent in the logic of HT, we can use these theorem provers to perform reasoning and automatically identify the conflicts we characterized before in terms of equivalence relations.

Overall, these characterizations are flexible enough to be extended to several types of conflicts, and can be used to detect which types of conflicts are generated, as well as trace them back to the source (potentially identifying justifications in the sense of provenance modules [20] for these problems in ASP).

Research must be done next on conflict resolution methods, formally defining rule combining algorithms. We also plan to study the implication of using paracoherent semantics such as Semi-Equilibrium models [5] for calculating answer sets of modules and [3].

References