Forgetting in Modular Answer Set Programming

Ricardo Gonçalves[1] and **Tomi Janhunen**[2] and **Matthias Knorr**[1] and **João Leite**[1] and **Stefan Woltran**[3]
[1]Universidade Nova de Lisboa
[2]Aalto University
[3]Vienna University of Technology

Abstract

Modular programming facilitates the creation and reuse of large software, and has recently gathered considerable interest in the context of Answer Set Programming (ASP). In this setting, forgetting, or the elimination of middle variables no longer deemed relevant, is of importance as it allows one to, e.g., simplify a program, make it more declarative, or even hide some of its parts without affecting the consequences for those parts that are relevant. While forgetting in the context of ASP has been extensively studied, its known limitations make it unsuitable to be used in Modular ASP. In this paper, we present a novel class of forgetting operators and show that such operators can always be successfully applied in Modular ASP to forget all kinds of atoms – input, output and hidden – overcoming the impossibility results that exist for general ASP. Additionally, we investigate conditions under which this class of operators preserves the module theorem in Modular ASP, thus ensuring that answer sets of modules can still be composed, and how the module theorem can always be preserved if we further allow the reconfiguration of modules.

1 Introduction

Modularity in Answer Set Programming (ASP) (Dao-Tran et al. 2009; Harrison and Lierler 2016; Baral, Dzifcak, and Takahashi 2006; Janhunen et al. 2009; Oikarinen and Janhunen 2008), just as in many other programming paradigms, is a fundamental concept to ease the creation and reuse of large programs. In one of the most significant general approaches to modularity – the so-called *programming-in-the-large* – compositional operators are provided for combining separate and independent modules, i.e., essentially answer set programs extended with well-defined input/output interfaces, based on standard semantics. The compositionality of the semantics of individual modules is ensured by the so-called *module theorem* (Janhunen et al. 2009).

The operation of *forgetting*, which aims at eliminating a set of variables from a knowledge base while preserving all relationships (direct and indirect) between the remaining variables, has recently gained a lot of attention, not only because it is *useful*, e.g., as a means to clean up a theory by eliminating all auxiliary variables that have no relevant declarative meaning, but also because it may be *necessary*, e.g., as a means to deal with privacy and legal issues such as to eliminate illegally obtained data, or to comply with the recently enacted *right to be forgotten* (European Union 2016).

Whereas *forgetting* in the context of classical logic is essentially a solved problem (Bledsoe and Hines 1980; Weber 1986; Middeldorp, Okui, and Ida 1996; Lang, Liberatore, and Marquis 2003; Moinard 2007; Gabbay, Schmidt, and Szalas 2008), new challenging issues arise when it is considered in the context of a non-monotonic logic based language such as ASP (Zhang and Foo 2006; Eiter and Wang 2008; Wong 2009; Wang, Wang, and Zhang 2013; Knorr and Alferes 2014; Wang et al. 2014; Delgrande and Wang 2015; Gonçalves, Knorr, and Leite 2016b). According to (Goncalves, Knorr, and Leite 2016a), forgetting in ASP is best captured by *strong persistence* (Knorr and Alferes 2014), a property inspired by *strong equivalence*, which requires that there be a correspondence between the answer sets of a program before and after forgetting a set of atoms, and that such correspondence be preserved in the presence of additional rules not containing the atoms to be forgotten. However, it has also been shown that, in ASP, it is not always possible to forget and satisfy *strong persistence* (Gonçalves, Knorr, and Leite 2016b).

What about *forgetting* in Modular ASP? Do the same negative results hold, and sometimes it is simply impossible to forget while satisfying *strong persistence*? Is *strong persistence* an adequate requirement in the case of Modular ASP? Can *forgetting* be reconciled with the *module theorem*?

Investigating forgetting in the context of Modular ASP is the central topic of this paper. Our main contributions are:

- We argue that, given that the input of a module is just a set of facts, strong persistence is too strong when forgetting in Modular ASP, and that it is more suitable to rely on *uniform equivalence* (Sagiv 1988; Eiter and Fink 2003) for a weaker form of persistence, say *uniform persistence*, which has not been considered before.
- We thoroughly investigate forgetting in ASP under uniform equivalence, including formalizing *uniform persistence* and showing that, unlike with strong persistence, it is always possible to forget under this new property.
- We show that no previously known class of forgetting operators satisfies *uniform persistence*, which leads us to introduce a new class of forgetting operators that satisfies

uniform persistence, and investigate its other properties.

- We employ the newly introduced class of operators to forget in a prominent approach of modular ASP, DLP-functions (Janhunen et al. 2009), and show how it can adequately be used to forget *input*, *output*, and *hidden* atoms from a module, while obeying uniform persistence.
- We also show that, not unexpectedly, the *module theorem* no longer holds in general after forgetting.
- To overcome the latter problem, we investigate ways to modify modules so that the *module theorem* can be preserved while forgetting under *uniform persistence* i.e., ways to reconfigure ASP modules by merging and splitting modules, so that we can properly forget while preserving the compositionality of stable models of modules.

2 Preliminaries

We start by recalling some notions about logic programs.

An *(extended) rule* r is an expression of the form

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_k, \\ not\ not\ d_1, ..., not\ not\ d_l\ , \qquad (1)$$

where $a_1, \ldots, a_n$, $b_1, \ldots, b_m$, $c_1, \ldots, c_k$, and $d_1, \ldots, d_l$ are atoms of a given propositional alphabet $\mathcal{A}$. Note that double negation is standard in the context of forgetting in ASP. We also write such rules as $A \leftarrow B, not\ C, not\ not\ D$ where $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$, $C = \{c_1, \ldots, c_k\}$, and $D = \{d_1, \ldots, d_l\}$. An *(extended) logic program* is a finite set of rules. By $\mathcal{A}(P)$ we denote the set of atoms appearing in P and by $\mathcal{C}_e$ the class of extended programs. We call r *disjunctive* if $D = \emptyset$; *normal* if, additionally, A has at most one element; *Horn* if on top of that $C = \emptyset$; and *fact* if also $B = \emptyset$. The classes of *disjunctive*, *normal* and *Horn programs*, $\mathcal{C}_d$, $\mathcal{C}_n$, and $\mathcal{C}_H$, are then defined as usual.

Given a program P and an *interpretation* I, i.e., a set $I \subseteq \mathcal{A}$, the *reduct* P^I is defined as $P^I = \{A \leftarrow B \mid A \leftarrow B, not\ C, not\ not\ D \in P, C \cap I = \emptyset, D \subseteq I\}$. An interpretation I is a *model* of a rule $A \leftarrow B$ if $A \cap I \neq \emptyset$ whenever $B \subseteq I$; I is a *model* of a reduct R if it satisfies every rule of R; I is a *minimal model* of the reduct R if I is a model of R and there is no model I' of R s.t. $I' \subset I$; and I is an *answer set* of an extended program P if it is a minimal model of the reduct P^I. The set of all answer sets of a program P is denoted by $\mathcal{AS}(P)$. Given a set of atoms V, the *V-exclusion* of a set of sets $\mathcal{M}$, denoted $\mathcal{M}_{\|V}$, is $\{X \backslash V \mid X \in \mathcal{M}\}$.

Two programs P_1 and P_2 are said to be *equivalent* if $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$, *strongly equivalent*, denoted by $P_1 \equiv P_2$, if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$ for any $R \in \mathcal{C}_e$, and *uniformly equivalent*, denoted by $P_1 \equiv_u P_2$, if $\mathcal{AS}(P_1 \cup R) = \mathcal{AS}(P_2 \cup R)$, for any set of facts R.

An *HT-interpretation* is a pair $\langle X, Y\rangle$ s.t. $X \subseteq Y \subseteq \mathcal{A}$. Given a program P, an HT-interpretation $\langle X, Y\rangle$ is an *HT-model of* P if $Y \models P$ and $X \models P^Y$, where $\models$ stands for the classical satisfaction relation for rules. The set of *all HT-models of* P is denoted by $\mathcal{HT}(P)$. Also, $Y \in \mathcal{AS}(P)$ iff $\langle Y, Y\rangle \in \mathcal{HT}(P)$ and there is no $X \subset Y$ s.t. $\langle X, Y\rangle \in \mathcal{HT}(P)$. Also, $\mathcal{HT}(P_1) = \mathcal{HT}(P_2)$ precisely when $P_1 \equiv P_2$ (Lifschitz, Pearce, and Valverde 2001). Given a set of atoms V, the *V-exclusion* of a set of HT-interpretations $\mathcal{M}$, $\mathcal{M}_{\|V}$, is $\{\langle X \backslash V, Y \backslash V\rangle \mid \langle X, Y\rangle \in \mathcal{M}\}$.

A *forgetting operator* over a class $\mathcal{C}$ of programs over $\mathcal{A}$ is a partial function $\mathsf{f} : \mathcal{C} \times 2^{\mathcal{A}} \to \mathcal{C}$ s.t. the *result of forgetting about V from P*, $\mathsf{f}(P, V)$, is a program over $\mathcal{A}(P) \backslash V$, for each $P \in \mathcal{C}$ and $V \subseteq \mathcal{A}$. We denote the domain of f by $\mathcal{C}(\mathsf{f})$ and usually we focus on $\mathcal{C} = \mathcal{C}_e$, and leave $\mathcal{C}$ implicit. The operator f is called *closed* for $\mathcal{C}' \subseteq \mathcal{C}(\mathsf{f})$ if $\mathsf{f}(P, V) \in \mathcal{C}'$, for every $P \in \mathcal{C}'$ and $V \subseteq \mathcal{A}$. A *class* F *of forgetting operators (over $\mathcal{C}$)* is a set of forgetting operators f s.t. $\mathcal{C}(\mathsf{f}) \subseteq \mathcal{C}$.

We recall notions of modules using *ELP-functions*, a generalization of *DLP-functions* (Janhunen et al. 2009).[1] An *ELP-function*, Π, is a quadruple $\langle P, I, O, H\rangle$, where I, O, and H are pairwise distinct sets of *input atoms*, *output atoms*, and *hidden atoms*, respectively, and P is a logic program s.t. for each rule $A \leftarrow B, not\ C, not\ not\ D$ of P,

1. $A \cup B \cup C \cup D \subseteq I \cup O \cup H$, and
2. if $A \neq \emptyset$, then $A \cap (O \cup H) \neq \emptyset$.

Input atoms and output atoms are also called *visible atoms*.

An *interpretation* for an ELP-function $\Pi = \langle P, I, O, H\rangle$ is an arbitrary set $M \subseteq \mathcal{A}(\Pi)$, where $\mathcal{A}(\Pi) = I \cup O \cup H$. We denote by $\mathcal{A}_i(\Pi)$, $\mathcal{A}_o(\Pi)$, $\mathcal{A}_h(\Pi)$, and by M_i, M_o, M_h the subsets of $\mathcal{A}(\Pi)$ and M restricted to elements in I, O, and H, respectively. Given ELP-function $\Pi = \langle P, I, O, H\rangle$ and interpretation M, the *reduct of* Π *w.r.t.* M is the ELP-function $\Pi^M = \langle P^M, I, O, H\rangle$, where P^M is the reduct of P w.r.t. M. An interpretation N is a *model of* Π^M iff N is a model of P^M. A model N of Π^M is *I-minimal* iff there is no model N' of Π^M such that $N'_i = N_i$ and $N' \subset N$. An interpretation M is a *stable model*[2] of Π iff M is an I-minimal model of Π^M. The set of all stable models of Π is denoted by $\mathcal{SM}(\Pi)$. We have $M \in \mathcal{SM}(\Pi)$ iff $M \in \mathcal{AS}(P \cup M_i)$ (Lierler and Truszczynski 2011).

Given a program P and a set of atoms S, the set of *defining rules* for S is $\mathsf{Def}_P(S) = \{A \leftarrow B, not\ C, not\ not\ D \in P \mid A \cap S \neq \emptyset\}$. Two ELP-functions $\Pi_1 = \langle P_1, I_1, O_1, H_1\rangle$ and $\Pi_2 = \langle P_2, I_2, O_2, H_2\rangle$ *respect the input/output interfaces* of each other iff (1) $(I_1 \cup O_1 \cup H_1) \cap H_2 = \emptyset$; (2) $(I_2 \cup O_2 \cup H_2) \cap H_1 = \emptyset$; (3) $O_1 \cap O_2 = \emptyset$; (4) $\mathsf{Def}_{P_1}(O_1) = \mathsf{Def}_{P_1 \cup P_2}(O_1)$, and (5) $\mathsf{Def}_{P_2}(O_2) = \mathsf{Def}_{P_1 \cup P_2}(O_2)$.

Let $\Pi_1 = \langle P_1, I_1, O_1, H_1\rangle$ and $\Pi_2 = \langle P_2, I_2, O_2, H_2\rangle$ be ELP-functions that respect the input/output interfaces of each other. The *composition* $\Pi_1 \oplus \Pi_2$ is defined as

$$\langle P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2\rangle.$$

The *join* $\sqcup$ of modules builds on this composition imposing further restrictions. The *positive dependency graph* of an ELP-function $\Pi = \langle P, I, O, H\rangle$ is the pair $DG^+(\Pi) = \langle O \cup H, \leq_1\rangle$, where $b \leq_1 a$ holds for $a, b \in (O \cup H)$ iff there is a rule $A \leftarrow B, not\ C, not\ not\ D \in P$ s.t. $a \in A$ and $b \in B$. The reflexive and transitive closure of $\leq_1$ provides the dependency relation $\leq$ over output and hidden atoms.

[1] While we limit our generalization to extended logic programs to the necessary notions for individual modules, we do not foresee major difficulties for other aspects left out of scope of this paper.

[2] We reserve the term "answer set" for programs and the term "stable model" for ELP-functions to ease the reading.

A *strongly connected component (SCC)* S of $DG^+(\Pi)$ is a maximal set $S \subseteq \mathcal{A}_o(\Pi) \cup \mathcal{A}_h(\Pi)$ s.t. $b \leq a$ for all pairs $a, b \in S$. If $\Pi_1 \oplus \Pi_2$ is defined, then Π_1 and Π_2 are *mutually dependent* iff $DG^+(\Pi_1 \oplus \Pi_2)$ has an SCC S s.t. $S \cap \mathcal{A}_o(\Pi_1) \neq \emptyset$ and $S \cap \mathcal{A}_o(\Pi_2) \neq \emptyset$, and *mutually independent* otherwise. Thus, given ELP-functions Π_1 and Π_2, if the composition $\Pi_1 \oplus \Pi_2$ is defined and Π_1 and Π_2 are mutually independent, then the *join* $\Pi_1 \sqcup \Pi_2$ of Π_1 and Π_2 is defined and coincides with $\Pi_1 \oplus \Pi_2$ (Janhunen et al. 2009).

3 Forgetting under Uniform Persistence

Arguably, among the many properties for forgetting in ASP, strong persistence is the one that should intuitively hold, since it imposes the preservation of all original direct and indirect dependencies between atoms not to be forgotten. Here and in the sequel, F is a class of forgetting operators.

(SP) F satisfies *Strong Persistence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all programs $R \in \mathcal{C}(\mathsf{f})$ with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$.

Essentially, **(SP)** requires that the answer sets of $\mathsf{f}(P, V)$ correspond to those of P, no matter what programs R over $\mathcal{A} \backslash V$ we add to both, which is closely related to the concept of strong equivalence. However, this property is rather demanding, witnessed by the fact that it cannot always be satisfied (Gonçalves, Knorr, and Leite 2016b). On the other hand, in the case of a module, i.e., an ELP-function, its program P is fixed, and we only vary the input, which is closely related to considering a fixed ASP program, encoding the declarative specification of a problem, and only varying the instances corresponding to the specific problem to be solved. This is captured by the notion of *uniform equivalence*, which weakens *strong equivalence* by considering that only facts can be added. To investigate *forgetting* in such cases, we introduce *Uniform Persistence*, **(UP)**, obtained from **(SP)** by restricting the varying programs R to sets of facts.

(UP) F satisfies *Uniform Persistence* if, for each $\mathsf{f} \in \mathsf{F}$, $P \in \mathcal{C}(\mathsf{f})$ and $V \subseteq \mathcal{A}$, we have $\mathcal{AS}(\mathsf{f}(P, V) \cup R) = \mathcal{AS}(P \cup R)_{\|V}$, for all sets of facts R with $\mathcal{A}(R) \subseteq \mathcal{A} \backslash V$.

Having introduced **(UP)** as the desired property for forgetting in ELP-functions, we now turn our attention to which forgetting operator to use. Unfortunately, none of the existing classes mentioned in the literature[3] satisfy **(UP)**.[4]

Theorem 1 *None of the classes* F *of forgetting operators studied in (Goncalves, Knorr, and Leite 2016a; Gonçalves et al. 2017) satisfy* **(UP)**.

Due to this negative result and the fact that it is not always possible to forget while satisfying **(SP)**, the question that arises is whether this is actually different for **(UP)**, given that it is less demanding in its requirements.

[3]Cf. the survey on forgetting in ASP (Goncalves, Knorr, and Leite 2016a), (Gonçalves et al. 2017), and references therein.

[4]Note that the result in (Goncalves, Knorr, and Leite 2016a) (Fig. 1) indicating that class F_{Sas} satisfies **(SP)**, the generalization of **(UP)**, is in fact not entirely accurate, since the only known operator in F_{Sas} is not defined for a class of programs, but rather for instances of forgetting.

Example 1 *Consider program* P *used in the impossibility result for* **(SP)** *(Gonçalves, Knorr, and Leite 2016b):*

$$a \leftarrow p \qquad b \leftarrow q \qquad p \leftarrow not\, q \qquad q \leftarrow not\, p$$

Adding program $R = \{a \leftarrow; b \leftarrow\}$*, it is shown there that any result of forgetting* $\{p, q\}$ *from* P, $\mathsf{f}(P, \{p, q\})$*, that satisfies* **(SP)** *is required to have an HT-model* $\langle ab, ab\rangle$[5]*. At the same time, since* $\{a, b\}$ *(modulo* $\{p, q\}$*) is not an answer set of* P*, we must have* $\langle X, ab\rangle \in \mathcal{HT}(\mathsf{f}(P, \{p, q\}))$ *for at least one* $X \subset \{a, b\}$*, to prevent* $\{a, b\}$ *from being an answer set of* $\mathsf{f}(P, \{p, q\})$*. It is then shown that due to different programs* R*,* $\langle X, ab\rangle \notin \mathcal{HT}(\mathsf{f}(P, \{p, q\}))$ *for any such* X*, thus causing a contradiction. However, in the case of* $X = \emptyset$*,* $R = \{a \leftarrow b; b \leftarrow a\}$ *is used, which is not a set of facts and thus not relevant w.r.t.* **(UP)***. In fact, given the only possible four sets of facts over* $\{a, b\}$ *to be considered for* R*, we can verify that* $P' = \{a \leftarrow not\, b;\ a \leftarrow not\, not\, a, b;\ b \leftarrow not\, a;\ b \leftarrow not\, not\, b, a\}$ *is a result of forgetting* $\{p, q\}$ *from* P *for which the condition of* **(UP)** *is satisfied.*

A naive approach to define a class of forgetting operators that satisfies **(UP)** would be to use relativized uniform equivalence (Eiter, Fink, and Woltran 2007), which is close in spirit to **(UP)**. However, this would not work, for the same reasons that a similar approach based on relativized strong equivalence fails to capture **(SP)** (Gonçalves et al. 2017).

Instead, we define a class of forgetting operators that satisfies **(UP)**, dubbed F_{UP}, whose more involved definition – that we will gently introduce in an incremental way – builds on the manipulation of HT-models given an input program P and a set of atoms $V \subseteq \mathcal{A}(P)$ to forget. To this end, we aim at devising a mapping from $\mathcal{HT}(P)$ to the set of HT-models of the result of forgetting, $\mathsf{f}(P, V)$, for any operator $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$. This mapping can be illustrated as follows.

Example 2 *The program* P *from Ex. 1 has 15 HT-models:*

$$\begin{array}{cccc} \langle bq, bq\rangle & \langle bq, abq\rangle & \langle b, abpq\rangle & \langle abp, abpq\rangle \\ \langle ap, ap\rangle & \langle abq, abq\rangle & \langle bq, abpq\rangle & \langle abq, abpq\rangle \\ \langle ap, abp\rangle & \langle \emptyset, abpq\rangle & \langle ap, abpq\rangle & \langle abpq, abpq\rangle \\ \langle abp, abp\rangle & \langle a, abpq\rangle & \langle ab, abpq\rangle & \end{array}$$

The HT-models for the proposed result P' *of forgetting are* $\langle a, a\rangle$*,* $\langle b, b\rangle$*,* $\langle \emptyset, ab\rangle$ *and* $\langle ab, ab\rangle$*.*

But how could we determine the latter set of HT-models for any P and V? Given the HT-models listed above, the set $\mathcal{HT}(P)_{\|V}$ contains extra tuples such as $\langle a, ab\rangle$ and $\langle b, ab\rangle$. Thus, a more involved analysis of HT-models is in order.

By the definition of **(UP)**, an answer set Y of $\mathsf{f}(P, V) \cup R$ corresponds to an answer set $Y \cup A$ of $P \cup R$, for some $A \subseteq V$. We will therefore collect all HT-models $\langle X, Y \cup A\rangle$ in $\mathcal{HT}(P)$ with the same Y and join them in blocks separated by the varying A. To this end, we first characterize all the different total HT-models of P, namely, for each $Y \subseteq \mathcal{A} \backslash V$:

$$Sel^Y_{\langle P,V\rangle} = \{A \subseteq V \mid \langle Y \cup A, Y \cup A\rangle \in \mathcal{HT}(P)\}.$$

Example 3 *Given the HT-models (Ex. 2) for* P *of Ex. 1 and* $V = \{p, q\}$*, we obtain* $Sel^{\emptyset}_{\langle P,V\rangle} = \emptyset$*,* $Sel^{\{a\}}_{\langle P,V\rangle} = \{\{p\}\}$*,* $Sel^{\{b\}}_{\langle P,V\rangle} = \{\{q\}\}$*, and* $Sel^{\{a,b\}}_{\langle P,V\rangle} = \{\{p\}, \{q\}, \{p, q\}\}$*.*

[5]We follow a common convention and abbreviate sets in HT-interpretations such as $\{a, b\}$ with the sequence of its elements, ab.

Clearly, the total models to be considered in the result of forgetting should be restricted to those Y s.t. $Sel^Y_{\langle P,V\rangle}$ is non-empty. But not all these sets should be considered.

Example 4 *Let P be a program over $\mathcal{A} = \{a, b, p, q\}$ s.t. its HT-models of the form $\langle X, \{a, b\} \cup A\rangle$ with $A \subseteq V = \{p, q\}$ are $\langle ab, abp\rangle$, $\langle abp, abp\rangle$, $\langle abp, abpq\rangle$, and $\langle abpq, abpq\rangle$. We have that $Sel^{\{a,b\}}_{\langle P,V\rangle} = \{\{p\}, \{p, q\}\}$. Nevertheless, the non-total models $\langle ab, abp\rangle$ and $\langle abp, abpq\rangle$ do not allow $\{a, b, p\}$ and $\{a, b, p, q\}$ to be answer sets of $P \cup R$, for any R over $\mathcal{A}\backslash V = \{a, b\}$. So, although $Sel^{\{a,b\}}_{\langle P,V\rangle} \neq \emptyset$, the set $\{a, b\}$ should not be a possible answer set of the forgetting result.*[6]

Taking this observation into account, we define the set of total models for the result of forgetting V from P:

$$T_{\langle P,V\rangle} = \{Y \subseteq \mathcal{A}\backslash V \mid \text{ there exists } A \in Sel^Y_{\langle P,V\rangle} \text{ s.t. } \langle Y \cup A', Y \cup A\rangle \notin HT(P) \text{ for every } A' \subset A\}.$$

Example 5 *Based on the HT-models of P listed in Ex. 2, the sets $Sel^Y_{\langle P,V\rangle}$ identified in Ex. 3, and $V = \{p, q\}$, we observe that $T_{\langle P,V\rangle} = \{\{a\}, \{b\}, \{a, b\}\}$. In each of the three cases, the condition in the definition of $T_{\langle P,V\rangle}$ is satisfied by some element of $Sel^Y_{\langle P,V\rangle}$. For $Y = \{a, b\}$ in particular, the set A can be either $\{p\}$ or $\{q\}$, but not $\{p, q\}$. Given $T_{\langle P,V\rangle}$, we expect three total HT-models for the result of forgetting $\{p, q\}$ from P, i.e., the ones indicated in Ex. 2 for P'.*

The crucial question now is how to extract the non-total HT-models for the result of forgetting in general. For this purpose, for each $A \in Sel^Y_{\langle P,V\rangle}$, we first consider the non-total HT-models of P of the form $\langle X, Y \cup A\rangle$:

$$N^{Y,A}_{\langle P,V\rangle} = \{X\backslash V \mid \langle X, Y \cup A\rangle \in \mathcal{HT}(P) \text{ and } X \neq Y \cup A\}.$$

Example 6 *Continuing Ex. 5, these non-total models, in particular those relevant for the desired result $\langle \emptyset, ab\rangle$, are:*

$$N^{\{a,b\},\{p\}}_{\langle P,V\rangle} = \{\{a\}\}, N^{\{a,b\},\{q\}}_{\langle P,V\rangle} = \{\{b\}\}, \text{ and } N^{\{a,b\},\{p,q\}}_{\langle P,V\rangle} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}.$$

Now, since HT-models of facts never include $\langle \emptyset, Y\rangle$ for any Y, we know that any HT-model $\langle \emptyset, Y\rangle$ of P will not occur in $\mathcal{HT}(P \cup R)$ for any (non-empty) set of facts R. Hence, either one of the $N^{Y,A}_{\langle P,V\rangle}$ is empty, in which case P itself has an answer set Y modulo V and the result of forgetting should have an answer set Y, or $\emptyset \in N^{Y,A}_{\langle P,V\rangle}$ for any A results in an HT-model $\langle \emptyset, Y\rangle$ for the result of forgetting, which is why $\langle \emptyset, ab\rangle \in \mathcal{HT}(P')$ in Ex. 2 holds. Generalizing this observation, whenever there is a set X s.t. each $N^{Y,A}_{\langle P,V\rangle}$ contains an element X' with $X \subseteq X'$, then adding X as facts to P cannot result in an answer set of P, and thus, $\langle X, Y\rangle$ should be part of the forgetting result. In Ex. 6, the only such set X is indeed $X = \emptyset$.

[6]Similar considerations have been used in the context of relativized equivalence (Eiter, Fink, and Woltran 2007) and in forgetting (Gonçalves, Knorr, and Leite 2016b).

We thus collect all sets $N^{Y,A}_{\langle P,V\rangle}$ for each Y, define tuples over this set of sets, and intersections over these tuples. The latter correspond to the maximal subsets X, which suffices for uniform equivalence (Eiter, Fink, and Woltran 2007).

Definition 1 *Let P be a program, $V \subseteq \mathcal{A}$, and $Y \subseteq \mathcal{A}\backslash V$. Consider the indexed family of sets $\mathcal{S}^Y_{\langle P,V\rangle} = \{N^{Y,i}_{\langle P,V\rangle}\}_{i\in I}$ where $I = Sel^Y_{\langle P,V\rangle}$. For each tuple $(X_i)_{i\in I}$ such that $X_i \in N^{Y,i}_{\langle P,V\rangle}$, we define the intersection of its sets as $\bigcap_{i\in I} X_i$. We denote by $SInt^Y_{\langle P,V\rangle}$ the set of all such intersections.*

The resulting intersections indeed correspond to sets X pointed out in the preceding discussion. Therefore, we obtain the definition of F_{UP} by combining the total models based on $T_{\langle P,V\rangle}$ and the non-total ones based on $SInt^Y_{\langle P,V\rangle}$, but naturally restricted to those cases where the corresponding total model exists.

Definition 2 (UP-Forgetting) *Let F_{UP} be the class of forgetting operators defined as:*

$$\{\mathsf{f} \mid \mathcal{HT}(\mathsf{f}(P, V)) = (\{\langle Y, Y\rangle \mid Y \in T_{\langle P,V\rangle}\} \cup \{\langle X, Y\rangle \mid Y \in T_{\langle P,V\rangle} \text{ and } X \in SInt^Y_{\langle P,V\rangle}\}) \text{ for all } P \in \mathcal{C}(\mathsf{f}) \text{ and } V \subseteq \mathcal{A}\}.$$

Example 7 *Recall P from Ex. 1. Following the discussion after Ex. 6, we can verify that the result of forgetting about $V = \{p, q\}$ from P according to F_{UP} has the expected HT-models (cf. Ex. 2): $\langle a, a\rangle$, $\langle b, b\rangle$, $\langle \emptyset, ab\rangle$, and $\langle ab, ab\rangle$.*

The definition of F_{UP} characterizes the HT-models of a result of forgetting for any $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$, but not an actual program. This may raise the question whether there actually is such an operator, and we can answer this question positively.

Theorem 2 *There exists f such that $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$.*

The construction underlying the result relies on the notion of counter-models in HT (Cabalar and Ferraris 2007), which has been used for defining forgetting operators before (Wang, Wang, and Zhang 2013; Wang et al. 2014).

While the definition of UP-Forgetting itself is certainly non-trivial, it turns out that for the case of Horn programs, a considerably simpler definition can be used.

Proposition 1 *Let f be in F_{UP}. Then, for every $V \subseteq \mathcal{A}$:*

$$\mathcal{HT}(\mathsf{f}(P, V)) = \mathcal{HT}(P)_{\|V} \text{ for each } P \in \mathcal{C}_H.$$

This result serves as further indication that UP-Forgetting is well-defined, given that essentially all classes of forgetting operators coincide with this definition for the class of Horn programs (Goncalves, Knorr, and Leite 2016a).

We are able to show that F_{UP} indeed satisfies **(UP)** which guarantees that, unlike for the property **(SP)**, it is always possible to forget satisfying **(UP)**.

Theorem 3 F_{UP} *satisfies* **(UP)**.

Despite **(SP)** being the property that best captures the essence of forgetting in ASP in general, of which **(UP)** is the weaker version that is sufficient when dealing with modules, other properties have been investigated in the literature (cf. (Goncalves, Knorr, and Leite 2016a)). We obtain that F_{UP} satisfies the following properties.

Proposition 2 F_{UP} *satisfies* **(sC)**, **(wE)**, **(SE)**, **(CP)**, **(wC)**, **($\mathbf{E}_{\mathcal{C}_e}$)**, **($\mathbf{E}_{\mathcal{C}_H}$)**, *but not* **(W)**, **(PP)**, **(SI)**, **(SP)**, **($\mathbf{E}_{\mathcal{C}_d}$)**, **($\mathbf{E}_{\mathcal{C}_n}$)**.

Given the close connection between the class F_{UP} and uniform equivalence (cf. Thm. 3), it is not surprising that some properties of forgetting that are closely connected to strong equivalence are not satisfied by F_{UP}, notably **(PP)** and **(SI)**, which are satisfied by the class of forgetting operators defined for forgetting w.r.t. **(SP)** when forgetting is possible (Gonçalves, Knorr, and Leite 2016b).

Finally, we obtain that deciding whether a program is the result of forgetting for $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$ is in Π_3^P.

Theorem 4 *Given programs P, Q, and $V \subseteq \mathcal{A}$, deciding whether $P \equiv \mathsf{f}(Q, V)$ for $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$ is in Π_3^P.*

Note that the same problem for the classes of forgetting operators that approximate forgetting under **(SP)** is Π_3^P-complete (Gonçalves et al. 2017). Also, by (Wang et al. 2014) and Prop. 1, if Q is Horn, then this problem is in Π_1^P.

4 Forgetting in Modules

We now turn our attention to the use of F_{UP} to forget in modules i.e., ELP-functions. Towards characterizing results of forgetting in modules, the notion of equivalence between ELP-functions – modular equivalence (Janhunen et al. 2009) – needs to first be adapted, since it is too strong as it requires the existence of a bijection between stable models of different ELP-functions, which is not possible in general when reducing the language, as illustrated by the next example.

Example 8 *Take $\Pi = \langle\{a \leftarrow; b \leftarrow not\, not\, b\}, \emptyset, \{a\}, \{b\}\rangle$ with $\mathcal{SM}(\Pi) = \{\{a\}, \{a, b\}\}$. Forgetting b should yield, e.g., $\Pi' = \langle\{a \leftarrow\}, \emptyset, \{a\}, \emptyset\rangle$ with $\mathcal{SM}(\Pi') = \{\{a\}\}$, but then no bijection between $\mathcal{SM}(\Pi)$ and $\mathcal{SM}(\Pi')$ is possible.*

Therefore, we introduce a novel notion of equivalence for program modules according to which two modules are V-equivalent if they coincide on I and O ignoring V, and if their stable models coincide ignoring V.

Definition 3 (V-Equivalence) *Let Π_1 and Π_2 be ELP-functions, and V a set of atoms. Then, Π_1 and Π_2 are V-*equivalent*, denoted by $\Pi_1 \equiv_V \Pi_2$, iff*

1. $\mathcal{A}_i(\Pi_1)\backslash V = \mathcal{A}_i(\Pi_2)\backslash V$ *and* $\mathcal{A}_o(\Pi_1)\backslash V = \mathcal{A}_o(\Pi_2)\backslash V$;
2. $\mathcal{SM}(\Pi_1)_{\|V} = \mathcal{SM}(\Pi_2)_{\|V}$.

Forgetting from each of the pairwise disjoint sets of atoms considered in a module – *input*, *output* and *hidden* – needs to be characterised in turn. Additionally, in the case of input and output atoms, we also consider *hiding* them – useful when atoms are not declaratively meaningful outside the module, or should not be shown – and discuss its difference with respect to forgetting them.

We start by showing that the hidden atoms of an ELP-function can be forgotten without affecting its behavior perceived in terms of visible atoms, ensuring that we can deal with cases when we are not allowed to express a certain piece of information in terms of our hidden atoms, or do not want to show it to someone who wants to visualize the program of a module.

Theorem 5 (Forgetting hidden atoms) *Given a set $V \subseteq H$ of hidden atoms to forget, an ELP-function $\Pi = \langle P, I, O, H\rangle$ is V-equivalent to any ELP-function $\Pi' = \langle \mathsf{f}(P, V), I, O, H\backslash V\rangle$ based on a uniformly persistent forgetting operator $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$.*

But forgetting is also applicable to the visible elements of a module. For instance, whenever output atoms are no longer used by other modules, they can effectively be removed without affecting the behavior of the module.

Theorem 6 (Forgetting output atoms) *Given a set $V \subseteq O$ of output atoms to forget, an ELP-function $\Pi = \langle P, I, O, H\rangle$ is V-equivalent to any ELP-function $\Pi' = \langle \mathsf{f}(P, V), I, O\backslash V, H\rangle$ based on a uniformly persistent forgetting operator $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$.*

An alternative to forgetting output atoms is hiding them. Given an ELP-function $\Pi = \langle P, I, O, H\rangle$ and a set $V \subseteq O$ of output atoms, we could create an ELP-function $\langle P, I, O\backslash V, H \cup V\rangle$ where the atoms of V are simply hidden. This would be computationally cheap since P would not change, but could be regarded insufficient under the strict interpretation of forgetting V, i.e., the elements of V should not appear in the result at all. Nevertheless, we derive the following counterpart to Thm. 6.

Theorem 7 (Hiding output atoms) *Given a set $V \subseteq O$ of output atoms to hide, an ELP-function $\Pi = \langle P, I, O, H\rangle$ is V-equivalent to the ELP-function $\Pi' = \langle P, I, O\backslash V, H \cup V\rangle$.*

Thus, both hiding and forgetting output atoms yields V-equivalent ELP-functions.

Turning to forgetting (or hiding) of input atoms, no analogous result exists without making changes to the program.

Example 9 *Take $\Pi = \langle\{a \leftarrow b\}, \{b\}, \{a\}, \emptyset\rangle$. Then, $\mathcal{SM}(\Pi) = \{\emptyset, \{a, b\}\}$, but moving b from I to H yields Π' with $\mathcal{SM}(\Pi') = \{\{\}\}$, which is not $\{b\}$-equivalent.*

Nevertheless, if we allow programs to change, such V-equivalent ELP-functions can be constructed using the idea of an *input generator* (cf. (Oikarinen and Janhunen 2006, Thm. 4)), easily encodable with extended rules, and we can forget about input atoms from ELP-functions as follows.

Theorem 8 (Forgetting input atoms) *Given a set $V \subseteq I$ of input atoms to forget, an ELP-function $\Pi = \langle P, I, O, H\rangle$ is V-equivalent to any*

$$\Pi' = \langle \mathsf{f}(P \cup \{a \leftarrow not\, not\, a \mid a \in V\}, V), I\backslash V, O, H\rangle$$

based on a uniformly persistent forgetting operator $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$.

This construction of Π' can also be used to hide input atoms.

Theorem 9 (Hiding input atoms) *Given a set $V \subseteq I$ of input atoms to hide, an ELP-function $\Pi = \langle P, I, O, H\rangle$ is V-equivalent to $\Pi' = \langle P \cup \{a \leftarrow not\, not\, a \mid a \in V\}, I\backslash V, O, H \cup V\rangle$.*

Combining these results, we can now define a general notion of a module resulting from forgetting elements of single parts of a module's interface. From now on, we assume that some forgetting operator $\mathsf{f} \in \mathsf{F}_{\mathsf{UP}}$ has been fixed.

Definition 4 *Given an ELP-function $\Pi = \langle P, I, O, H\rangle$ and a set V of atoms to forget, we denote by $\Pi\backslash V$ the resulting ELP-function $\langle \mathsf{f}(P \cup \{a \leftarrow not\ not\ a \mid a \in I \cap V\}, V), I\backslash V, O\backslash V, H\backslash V\rangle$.*

We can show that this notion indeed fits the expectations.

Corollary 1 *For an ELP-function Π and a set of atoms $V \subseteq \mathcal{A}(\Pi)$, we have $\mathcal{SM}(\Pi\backslash V) = \mathcal{SM}(\Pi)_{\|V}$.*

And it follows that we can forget sets of atoms iteratively.

Proposition 3 *Let Π be an ELP-function and $V \subseteq \mathcal{A}(\Pi)$. Then, if $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$, we have*

$$\mathcal{SM}(\Pi\backslash V) = \mathcal{SM}((\Pi\backslash V_1)\backslash V_2) = \mathcal{SM}((\Pi\backslash V_2)\backslash V_1).$$

In (Janhunen et al. 2009), it is shown, through the *module theorem*, that the stable model semantics of modules is fully compositional, which should be preserved under forgetting.

In the case of two modules $\Pi_1 = \langle P_1, I_1, O_1, H_1\rangle$ and $\Pi_2 = \langle P_2, I_2, O_2, H_2\rangle$ that do not mention each other's hidden atoms and their join $\Pi_1 \sqcup \Pi_2$ is defined (coincides with the composition $\Pi_1 \oplus \Pi_2$), the module theorem states that $\mathcal{SM}(\Pi) = \mathcal{SM}(\Pi_1) \bowtie \mathcal{SM}(\Pi_2)$ where the join of sets of stable models captured by the operator $\bowtie$ contains $M_1 \cup M_2$ whenever $M_1 \in \mathcal{SM}(\Pi_1)$, $M_2 \in \mathcal{SM}(\Pi_2)$, and M_1 and M_2 are compatible, i.e., $M_1 \cap (I_2 \cup O_2) = M_2 \cap (I_1 \cup O_1)$ so that M_1 and M_2 coincide on visible atoms.

Limited to forgetting atoms that are not shared by two modules, if we consider two modules whose join is defined, then the module theorem can be preserved while forgetting.

Theorem 10 *If Π is an ELP-function obtained as a join of two ELP-functions Π_1 and Π_2, and $V \subseteq \mathcal{A}(\Pi)$ is a set of atoms to forget s.t. $V \cap (I_1 \cup O_1) \cap (I_2 \cup O_2) = \emptyset$, then $\mathcal{SM}(\Pi\backslash V) = \mathcal{SM}(\Pi_1\backslash V) \bowtie \mathcal{SM}(\Pi_2\backslash V)$.*

We can generalize this result to deal with cases where atoms to be forgotten appear in more than two modules.

Theorem 11 *If Π is an ELP-function obtained as a join of n ELP-functions $\Pi_1, \ldots, \Pi_n$, and $V \subseteq \mathcal{A}(\Pi)$ is a set of atoms to forget s.t., for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, $V \cap (I_i \cup O_i) \cap (I_j \cup O_j) = \emptyset$, then $\mathcal{SM}(\Pi\backslash V) = \bowtie_{i=1}^{n} \mathcal{SM}(\Pi_i\backslash V)$.*

Yet, if we lift the restrictions on where the atoms to forget appear, we lose a full correspondent to the module theorem.

Theorem 12 *If Π is an ELP-function obtained as a join of two ELP-functions Π_1 and Π_2, and $V \subseteq \mathcal{A}(\Pi)$ is a set of atoms to forget, then $\mathcal{SM}(\Pi\backslash V) \subseteq \bowtie_{i=1}^{n} \mathcal{SM}(\Pi_i\backslash V)$.*

Only one of the two inclusions one would expect actually holds, and this is not by chance. In general, it is possible that modules $\Pi_1\backslash V$ and $\Pi_2\backslash V$ possess compatible stable models M_1 and M_2 such that $M = M_1 \cup M_2 \in \mathcal{SM}(\Pi_1\backslash V) \bowtie \mathcal{SM}(\Pi_2\backslash V)$ but $M \notin \mathcal{SM}(\Pi\backslash V)$ as illustrated next.

Example 10 *Let us consider ELP-functions $\Pi_1 = \langle\{a \leftarrow b\}, \{b\}, \{a\}, \emptyset\rangle$ and $\Pi_2 = \langle\{b \leftarrow not\ c\}, \{c\}, \{b\}, \emptyset\rangle$ and their join $\Pi = \langle P, \{c\}, \{a, b\}, \emptyset\rangle$ with $P = P_1 \cup P_2$ for the respective sets of rules P_1 and P_2 of Π_1 and Π_2.*

As regards forgetting $V = \{b\}$, we have $\Pi_1\backslash V = \langle\{a \leftarrow not\ not\ a\}, \emptyset, \{a\}, \emptyset\rangle$, $\Pi_2\backslash V = \langle\emptyset, \{c\}, \emptyset, \emptyset\rangle$, and $\Pi\backslash V = \langle\{a \leftarrow not\ c\}, \{c\}, \{a\}, \emptyset\rangle$. It remains to observe that $M_1 = \emptyset \in \mathcal{SM}(\Pi_1\backslash V)$, $M_2 = \emptyset \in \mathcal{SM}(\Pi_2\backslash V)$, and $M_1 \cup M_2 \notin \mathcal{SM}(\Pi\backslash V) = \{\{a\}, \{c\}\}$ although M_1 and M_2 are (trivially) compatible.

The example suggests that it is not safe to use $\mathsf{f} \in \mathsf{F_{UP}}$ to forget shared atoms that inherently change the I/O interface between the modules. The same is also true for hiding.

Example 11 *Consider again Ex. 10. We obtain the three modules in each of which b has been hidden as follows: $\Pi_1' = \langle\{a \leftarrow b; b \leftarrow not\ not\ b\}, \emptyset, \{a\}, \{b\}\rangle$, $\Pi_2' = \langle\{b \leftarrow not\ c\}, \{c\}, \emptyset, \{b\}\rangle$ and $(\Pi_1 \sqcup \Pi_2)' = \langle\{a \leftarrow b; b \leftarrow not\ c\}, \{c\}, \{a\}, \{b\}\rangle$. But then Π_1' and Π_2' do not respect the input/output interfaces of each other. We could circumvent this by renaming one of the occurrences of b, but we would also lose the prior dependency of a on c.*

5 Module Reconfiguration

Preserving the compositionality of stable models of modules while forgetting is desirable by the very idea of modular ASP: we want users to define ASP modules that can be composed into larger programs/modules. However, as we have seen, the module theorem no longer works entirely whenever some atom to be forgotten is shared by two modules.

In such cases, one alternative is to somehow modify the modules so that these atoms cease to occur in the visible components of different modules, i.e., reconfigure ASP modules by merging and splitting modules, so that we can forget while preserving the compositionality of stable models of modules. Of course, for this to be feasible, we must have access to the modules in question (by communication, or because we own the modules). This may require sharing some information about some module, which may not always be desirable, but, arguably, whenever possible, this is a reasonable trade-off for being able to forget atoms from modules while preserving **(UP)** and the module theorem.

One way to address the problem, provided all involved modules are mutually independent and their composition is defined, is to join all the modules that contain such atoms.

Let Π be an ELP-function obtained as a join of n ELP-functions $\Pi_1, \ldots, \Pi_n$, and $V \subseteq \mathcal{A}(\Pi)$ a set of atoms to forget. Consider the following relation on $N = \{1, \ldots, n\}$: $i \sim_V j$ iff $V \cap (I_i \cup O_i) \cap (I_j \cup O_j) \neq \emptyset$. This relation identifies those ELP-functions that share atoms to forget, i.e., that can cause problems with the module theorem. We denote by $\sim_V^*$ the reflexive and transitive closure of $\sim_V$ on N. Since $\sim_V$ is clearly a symmetric relation, its reflexive and transitive closure, $\sim_V^*$, is an equivalence relation on N. We can therefore consider the quotient set $N\backslash_{\sim_V^*}$, i.e., the set of equivalence classes defined by $\sim_V^*$ on N. We then consider, for each $e \in N\backslash_{\sim_V^*}$, the ELP-function $\Pi_e = \bigsqcup_{i \in e} \Pi_i$, the join of those ELP-functions corresponding to the considered equivalence class. This allows us to prove a relaxed version of the module theorem.

Theorem 13 *Let Π be an ELP-function obtained as a join of n ELP-functions $\Pi_1, \ldots, \Pi_n$, and $V \subseteq \mathcal{A}(\Pi)$ a set of atoms to forget. Let $N = \{1, \ldots, n\}$, and consider $\sim_V^*$ the equivalence relation on N as defined previously, and $N\backslash_{\sim_V^*} = \{e_1, \ldots, e_k\}$ the respective quotient set. Then, $\mathcal{SM}(\Pi\backslash V) = \bowtie_{i=1}^{k} \mathcal{SM}(\Pi_{e_i}\backslash V)$.*

This shows that joining those modules that share atoms to be forgotten allows for the preservation of the module theorem.

Joining entire modules is not ideal. However, it may happen that only part of a module is relevant to the shared atom to be forgotten, in which case we can use the operation of decomposing (or splitting) modules to do a more fine-grained recomposition of modules that still preserves the module theorem. Towards this end, we adapt the necessary notions to introduce module decomposition (Janhunen et al. 2009). Given an ELP-function $\Pi = \langle P, I, O, H\rangle$, let $SCC^+(\Pi)$ denote the set of strongly connected components of $DG^+(\Pi)$. The dependency relation $\leq$ can be lifted to $SCC^+(\Pi)$ by setting $S_1 \leq S_2$ iff there are atoms $a_1 \in S_1$ and $a_2 \in S_2$ s.t. $a_1 \leq a_2$. It is easy to check that $\leq$ is well-defined over $SCC^+(\Pi)$, i.e., it does not depend on the chosen $a_1 \in S_1$ and $a_2 \in S_2$, and that $\langle SCC^+(\Pi), \leq\rangle$ is a partially ordered set, i.e., $\leq$ is reflexive, transitive, and antisymmetric. For each $S \in SCC^+(\Pi)$ we consider the ELP-function $\Pi_S = \langle \mathsf{Def}_P(S), \mathcal{A}(\mathsf{Def}_P(S))\backslash S, S \cap O, S \cap H\rangle$.

Some of these modules Π_S, however, may share hidden atoms, and therefore cannot be joined. To overcome this, such components of $SCC^+(\Pi)$ need to be identified.

Definition 5 *Given an ELP-function $\Pi = \langle P, I, O, H\rangle$, components $S_1, S_2 \in SCC^+(\Pi)$ do not respect the hidden atoms of each other, denoted by $S_1 \leftrightsquigarrow_h S_2$, if and only if $S_1 \neq S_2$ and (at least) one of the following conditions holds:*

1. *there is $h \in \mathcal{A}_h(\Pi_{S_1})$ such that $h \in \mathcal{A}_i(\Pi_{S_2})$,*
2. *there is $h \in \mathcal{A}_h(\Pi_{S_2})$ such that $h \in \mathcal{A}_i(\Pi_{S_1})$,*
3. *there are $h_1 \in \mathcal{A}_h(\Pi_{S_1})$ and $h_2 \in \mathcal{A}_h(\Pi_{S_2})$ such that both occur in some integrity constraint of Π.*

It is clear that the relation $\leftrightsquigarrow_h$ is irreflexive and symmetric on $SCC^+(\Pi)$ for every ELP-function Π. If we consider the reflexive and transitive closure of $\leftrightsquigarrow_h$, denoted by $\leftrightsquigarrow_h^*$, we obtain an equivalence relation. A repartition of $SCC^+(\Pi)$ can then be obtained by considering the quotient set $SCC^+(\Pi)\backslash \leftrightsquigarrow_h^*$, i.e., the set of equivalence classes of $\leftrightsquigarrow_h^*$ over $SCC^+(\Pi)$, which can be used to decompose Π.

Definition 6 *Given an ELP-function $\Pi = \langle P, I, O, H\rangle$, the* decomposition induced by $SCC^+(\Pi)$ and $\leftrightsquigarrow_h^*$ *includes an ELP-function $\Pi_0 = \langle \mathsf{IC}_0(P), \mathcal{A}(\mathsf{IC}_0(P)) \cup (I \setminus \mathcal{A}(P)), \emptyset, \emptyset\rangle$ where $\mathsf{IC}_0(P) = \{\leftarrow B, not\, C, not\, not\, D \in P \mid (B \cup C \cup D) \cap H = \emptyset\}$ and, for each $\mathcal{S} \in SCC^+(\Pi)\backslash \leftrightsquigarrow_h^*$, an ELP-function $\Pi_{\mathcal{S}} = \langle \mathsf{Def}_P(S) \cup \mathsf{IC}_S(P), \mathcal{A}(\mathsf{Def}_P(S) \cup \mathsf{IC}_S(P)) \setminus S, S \cap O, S \cap H\rangle$, where $S = \bigcup \mathcal{S}$ and $\mathsf{IC}_S(P) = \{\leftarrow B, not\, C, not\, not\, D \in P \mid (B \cup C \cup D) \cap (S \cap H) \neq \emptyset\}$.*

The module Π_0 keeps track of integrity constraints as well as input atoms that are not mentioned by the rules of P. We can adapt straightforwardly (from (Janhunen et al. 2009)) that this decomposition of an ELP-function is valid.

Proposition 4 *Given an ELP-function $\Pi = \langle P, I, O, H\rangle$, then $\Pi = \Pi_0 \sqcup (\bigsqcup_{\mathcal{S} \in SCC^+(\Pi)\backslash \leftrightsquigarrow_h^*} \Pi_{\mathcal{S}})$.*

We now show that this decomposition can be used to allow forgetting while still preserving the module theorem.

Let $\Pi_1 = \langle P_1, I_1, O_1, H_1\rangle$ and $\Pi_2 = \langle P_2, I_2, O_2, H_2\rangle$ be two ELP-functions such that their join is defined. Since Prop. 3 shows that we can forget a set of atoms by forgetting iteratively every atom in the set, we focus on forgetting a single atom p. Suppose that p is shared by the two modules, i.e., $p \in (I_1 \cup O_1) \cap (I_2 \cup O_2)$, and recall that we cannot guarantee that forgetting p separately in Π_1 and Π_2 preserves the module theorem. We first consider the set of components of the decomposition of Π_1 that are relevant for atom p, i.e., $\mathcal{R}(\Pi_1, p) = \{\mathcal{S} \in SCC^+(\Pi_1)\backslash \leftrightsquigarrow_h^* \mid p \in \mathcal{A}_o(\Pi_{\mathcal{S}}) \cup \mathcal{A}_i(\Pi_{\mathcal{S}})\}$. We denote by Π_1^p the union of the ELP-functions in $\mathcal{R}(\Pi_1, p)$, i.e., $\Pi_1^p = \bigsqcup \mathcal{R}(\Pi_1, p)$, by $\overline{\mathcal{R}}(\Pi_1, p)$ the set of components of the decomposition of Π_1 that are not relevant for p, i.e., $\overline{\mathcal{R}}(\Pi_1, p) = \{\mathcal{S} \in SCC^+(\Pi_1)\backslash \leftrightsquigarrow_h^* \mid \mathcal{S} \notin \mathcal{R}(\Pi_1, p)\}$, and by $\overline{\Pi_1^p}$ the union of the ELP-functions in $\overline{\mathcal{R}}(\Pi_1, p)$, i.e., $\overline{\Pi_1^p} = \bigsqcup \overline{\mathcal{R}}(\Pi_1, p)$.

The decomposition of Π_1 can then be used to obtain a restricted version of the module theorem.

Theorem 14 (Reconfiguration) *Let Π be an ELP-function obtained as a join of two ELP-functions Π_1 and Π_2, and let $p \in (\mathcal{A}_i(\Pi_1) \cup \mathcal{A}_o(\Pi_1)) \cap (\mathcal{A}_i(\Pi_2) \cup \mathcal{A}_o(\Pi_2))$. Then,*

$$\mathcal{SM}(\Pi\backslash\{p\}) = \mathcal{SM}(\overline{\Pi_1^p}\backslash\{p\}) \bowtie \mathcal{SM}((\Pi_2 \sqcup \Pi_1^p)\backslash\{p\}).$$

Thus, to allow forgetting in modules and preserve the module theorem, we can essentially decompose certain modules and reconfigure them in such a way that all rules on the considered shared atom occur in a single module.

6 Conclusions

In this paper, we thoroughly investigated the operation of *forgetting* in the context of modular ASP.

We began by observing that *strong persistence* **(SP)** – the property usually taken to best characterize forgetting in ASP, which cannot always be guaranteed – is too strong when we consider modular ASP. Given the structure of modules in the context of modular ASP, namely their restricted interface, a weaker notion of persistence based on *uniform equivalence* is sufficient to properly characterise forgetting in this case, which led us to introduce *uniform persistence* **(UP)**.

We showed that, unlike with **(SP)**, it is always possible to to forget under **(UP)**. Perhaps surprisingly, we also showed that, in general, none of the operators defined in the literature satisfies this weaker form of persistence, which led us to introduce the class of forgetting operators F_{UP} that we proved to obey **(UP)**, as well as a set of other properties commonly discussed in the literature.

We then turned our attention to the application of this class of forgetting operators to forget *input*, *output*, and *hidden* atoms from modules, and related it with the operation of *hiding*. Despite showing that we can always forget atoms from modules under *uniform persistence*, we also showed that the important *module theorem* no longer holds in general, with negative consequences in the compositionality of stable models. Subsequently, after pinpointing the conditions under which the *module theorem* holds, we proceeded by investigating how the theorem could be "recovered" through a reconfiguration of the modules obtained by suitable decomposition and composition operations.

Possible avenues for future work include investigating *forgetting* in other existing ways to view modular ASP, such

as (Dao-Tran et al. 2009; Harrison and Lierler 2016), and the precise relationship of **(UP)** and UP-Forgetting to the notion of relativized uniform equivalence (Eiter, Fink, and Woltran 2007), and obtaining syntactic operators for UP-Forgetting.

Acknowledgments Authors R. Gonçalves, M. Knorr, and J. Leite were partially supported by FCT project FORGET (PTDC/CCI-INF/32219/2017) and by FCT project NOVA LINCS (UID/CEC/04516/2013 and UID/CEC/04516/2019). T. Janhunen was partially supported by the Academy of Finland project 251170. R. Gonçalves was partially supported by FCT grant SFRH/BPD/100906/2014. S. Woltran was supported by the Austrian Science Fund (FWF): Y698, P25521.

References

Baral, C.; Dzifcak, J.; and Takahashi, H. 2006. Macros, macro calls and use of ensembles in modular answer set programming. In Etalle, S., and Truszczynski, M., eds., *Procs. of ICLP*, volume 4079 of *LNCS*, 376–390. Springer.

Bledsoe, W. W., and Hines, L. M. 1980. Variable elimination and chaining in a resolution-based prover for inequalities. In Bibel, W., and Kowalski, R. A., eds., *Procs. of CADE*, volume 87 of *LNCS*, 70–87. Springer.

Cabalar, P., and Ferraris, P. 2007. Propositional theories are strongly equivalent to logic programs. *TPLP* 7(6):745–759.

Dao-Tran, M.; Eiter, T.; Fink, M.; and Krennwallner, T. 2009. Modular nonmonotonic logic programming revisited. In Hill, P. M., and Warren, D. S., eds., *Procs. of ICLP*, volume 5649 of *LNCS*, 145–159. Springer.

Delgrande, J. P., and Wang, K. 2015. A syntax-independent approach to forgetting in disjunctive logic programs. In Bonet, B., and Koenig, S., eds., *Procs. of AAAI*, 1482–1488. AAAI Press.

Eiter, T., and Fink, M. 2003. Uniform equivalence of logic programs under the stable model semantics. In Palamidessi, C., ed., *Procs. of ICLP*, volume 2916 of *LNCS*, 224–238. Springer.

Eiter, T., and Wang, K. 2008. Semantic forgetting in answer set programming. *Artif. Intell.* 172(14):1644–1672.

Eiter, T.; Fink, M.; and Woltran, S. 2007. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Trans. Comput. Log.* 8(3).

European Union. 2016. General Data Protection Regulation. *Official Journal of the European Union* L119:1–88.

Gabbay, D. M.; Schmidt, R. A.; and Szalas, A. 2008. *Second Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications.

Gonçalves, R.; Knorr, M.; Leite, J.; and Woltran, S. 2017. When you must forget: Beyond strong persistence when forgetting in answer set programming. *TPLP* 17(5-6):837–854.

Goncalves, R.; Knorr, M.; and Leite, J. 2016a. The ultimate guide to forgetting in answer set programming. In Baral, C.; Delgrande, J.; and Wolter, F., eds., *Procs. of KR*, 135–144. AAAI Press.

Gonçalves, R.; Knorr, M.; and Leite, J. 2016b. You can't always forget what you want: on the limits of forgetting in answer set programming. In Fox, M. S., and Kaminka, G. A., eds., *Procs. of ECAI*, 957–965. IOS Press.

Harrison, A., and Lierler, Y. 2016. First-order modular logic programs and their conservative extensions. *TPLP* 16(5-6):755–770.

Janhunen, T.; Oikarinen, E.; Tompits, H.; and Woltran, S. 2009. Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res. (JAIR)* 35:813–857.

Knorr, M., and Alferes, J. J. 2014. Preserving strong equivalence while forgetting. In Fermé, E., and Leite, J., eds., *Procs. of JELIA*, volume 8761 of *LNCS*, 412–425. Springer.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res. (JAIR)* 18:391–443.

Lierler, Y., and Truszczynski, M. 2011. Transition systems for model generators - A unifying approach. *TPLP* 11(4-5):629–646.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Trans. Comput. Log.* 2(4):526–541.

Middeldorp, A.; Okui, S.; and Ida, T. 1996. Lazy narrowing: Strong completeness and eager variable elimination. *Theor. Comput. Sci.* 167(1&2):95–130.

Moinard, Y. 2007. Forgetting literals with varying propositional symbols. *J. Log. Comput.* 17(5):955–982.

Oikarinen, E., and Janhunen, T. 2006. Modular equivalence for normal logic programs. In Brewka, G.; Coradeschi, S.; Perini, A.; and Traverso, P., eds., *Procs. of ECAI*, 412–416.

Oikarinen, E., and Janhunen, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *TPLP* 8(5-6):717–761.

Sagiv, Y. 1988. Optimizing datalog programs. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann. 659–698.

Wang, Y.; Zhang, Y.; Zhou, Y.; and Zhang, M. 2014. Knowledge forgetting in answer set programming. *J. Artif. Intell. Res. (JAIR)* 50:31–70.

Wang, Y.; Wang, K.; and Zhang, M. 2013. Forgetting for answer set programs revisited. In Rossi, F., ed., *Procs. of IJCAI*, 1162–1168. IJCAI/AAAI.

Weber, A. 1986. Updating propositional formulas. In *Expert Database Conf.*, 487–500.

Wong, K.-S. 2009. *Forgetting in Logic Programs*. Ph.D. Dissertation, The University of New South Wales.

Zhang, Y., and Foo, N. Y. 2006. Solving logic program conflict through strong and weak forgettings. *Artif. Intell.* 170(8-9):739–778.