Intelligent Agents via Joint Tabling of Logic Program Abduction and Updating

Ammar Fathin Sabili
Faculty of Computer Science
Universitas Indonesia

Ari Saptawijaya
Faculty of Computer Science
Universitas Indonesia
Email: saptawijaya@cs.ui.ac.id

Luís Moniz Pereira
NOVA-LINCS, Laboratory for
Computer Science and Informatics
Universidade Nova de Lisboa

***Abstract* — Reasoning is an important aspect for an intelligent agent to come to a rational decision. With the same importance is the ability of such an agent to adapt itself to the environment by learning new knowledge from its observations. When the agent's knowledge base is represented by a logic program, goal-directed deliberative reasoning and the adaptive ability of such an agent can be achieved by abduction and updating on logic programs, respectively. Furthermore, the tabling feature in logic programming, which affords solutions reuse rather than recomputing them, enables an agent to make an immediate decision based on past reasoning, thus avoiding repetitive deliberative reasoning. Joint tabling of logic program abduction and updating is an approach first proposed by Pereira and Saptawijaya, motivated by its application in machine ethics, enabling an agent to make moral decisions, using their system QUALM. In this paper, we provide a complete program transformation which has not been detailed on that approach. We also resolve previously unidentified issues with respect to its implementation aspects. A prototype, QUALM*, is implemented as a proof of concept using XSB Prolog. Furthermore, an application is detailed, using QUALM*, emphasizing the importance of joint tabling of logic program abduction and updating, in the context of intelligent agents, specifically in ambient intelligence for eldercare.**

I. INTRODUCTION

Nowadays, the importance of artificial intelligence is evidenced by real-world applications for helping humans to achieve effectiveness and efficiency on many aspects. One such application is that of intelligent agents for eldercare [1]. An eldercare agent typically observes its environment through its sensors, making itself aware of the elder's needs (e.g., by being told, the agent is aware that a patient is hungry). Based on its observation, through *reasoning*, this agent makes an appropriate decision (e.g., preparing him/her some meal). At some point, this agent may also learn new knowledge, based on its further observations (e.g., the patient is actually wearing a denture) and consequently has to *update* its internal knowledge for achieving a more appropriate decision (e.g., preparing him/her a soft meal, given that he/she is still hungry).

Abduction is a form of reasoning that seeks out best explanations for supporting a given observation. When the knowledge base of an agent is represented as a logic program, abduction can be achieved via *abductive logic programming* [2]–[4]. In its application [3], [5], [6], abductive logic programming is often employed to determine reasonable actions of an agent to fulfill a given goal. For example, as for above eldercare agent, the observation that a patient is hungry (the goal of the agent to satisfy) leads to an action of preparing him/her some meal. Other reasonable alternative actions may also be available depending on the knowledge base of the agent. In that case, the agent picks an action that best fits the presented situation, e.g., by utility function.

As agents are typically situated in a dynamically changing environment, abductive reasoning should be complemented with the ability to update its knowledge base. This is illustrated in the above eldercare agent, where it further learns that the patient is actually wearing a denture, leading the agent to come up with a better decision (i.e., preparing him/her a soft meal). This knowledge updating part of an agent can be achieved by *logic program updating*, a non-monotonic form of reasoning which has been well studied in the field [7]–[9].

By now, research in logic programming is mature enough, with a number of Prolog systems and advanced features for facilitating diverse applications. One of these features is *tabling* for affording solutions reuse, by tabling solutions to a goal, rather than recomputing them. Interestingly, intelligent agents may benefit from this feature, as it allows an agent to arrive at an immediate decision (via tabled solutions) based on past reasoning, thus avoiding repetitive deliberative reasoning. While it is mainly used to speed up computation, such a dual-process model (the interaction between slow deliberative and rapid intuitive reasoning) in decision making has been well studied in psychology [10].

The combination of both logic program abduction and updating, via tabling mechanism, has been proposed and enacted by Pereira and Saptawijaya [11], known as *joint tabling of logic program abduction and updating*. This approach is particularly demonstrated for two applications in the burgeoning field of machine ethics [12]: moral updating and counterfactual moral reasoning. One of the challenges introduced by this approach is its complexity, due to the seamless integration of a tabling mechanism, abductive logic programming, and logic program updating. In this paper, we revisit this joint

tabling approach. The first contribution of our present research is therefore to provide a complete program transformation involved in this approach (Section III), which has not been fully detailed before in [12]. The second contribution is the further development of this approach, pertaining to its implementation aspect, by refining the involved transformation and resolving previously unidentified issues (Section IV). As a proof of concept, a prototype, QUALM*, in XSB Prolog [13] is implemented. Finally, in the third contribution, QUALM* is applied for knowledge representation and reasoning in intelligent agents, specifically in the context of ambient intelligence for eldercare (Section V). This application serves as a practical aspect of the agent's abductive and non-monotonic reasoning in a real-world use, realized by joint tabling of logic program abduction and updating, showing it as a solution to tackle the challenges of the dual-process model in decision making.

II. PRELIMINARIES

We start with basic notions in logic programming and proceed with logic program abduction and updating.

A. Logic Programs

We assume that alphabet $\mathcal{A}$ in a language $\mathcal{L}$ is given, denoting disjoint countable sets of constants (at least one), function symbols, and predicate symbols. A set of variable symbols is also assumed in $\mathcal{A}$. Underscore symbol (_) is a reserved variable stated an anonymous variable.

A *term* in $\mathcal{A}$ is defined recursively as a variable, a constant, or a form of $f(t_1, \cdots, t_n)$, where f is a function symbol in $\mathcal{A}$, $n \in \mathbb{N}$, and t_i are terms. An *atom* in $\mathcal{A}$ is of the form $p(t_1, \cdots, t_n)$, where p is a predicate symbol in $\mathcal{A}$, $n \in \mathbb{N}$, and t_i are terms. A form p/n states predicate p has an arity of n. A *literal* is an atom a or its negation (*default literal*) $not\ a$. They are *negation complements* to each other.

A logic program is a countable set of rules of the form: $H \leftarrow L_1, L_2, \cdots, L_m$, where H (head) is an atom, finite $m \in \mathbb{N}$, and L_i (body) are literals. Commas on a rule interpret conjunctions and symbol $\leftarrow$ is an entailment implication (not material implication). A rule without a body, simply written as H, denotes a *fact*.

Semantically, each atom is assigned a truth value, forming the so-called *interpretation* of a logic program. A *model* of a program is an interpretation that logically satisfies all rules (and facts) in the program.

B. Abduction in Logic Programs

Abduction, first introduced by Peirce [14], is a form of reasoning that seeks out best explanations to support a given observation. In logic programming, abduction (known as *abductive logic programming* [2]) amounts to finding explanations in the form of *abducibles* from an observation in the form of *query*, a goal to be satisfied in a program model where the abducibles are adopted. Abducibles in a logic program are atoms (or their negation) which initially are not assigned a truth value. Moreover, constraints, the so-called *integrity constraints* may be added. Integrity constraint is a rule in the denial form: $\bot \leftarrow L_1, \ldots, L_m$, where $\bot/0$ is a reserved predicate symbol (denoting $false$) in $\mathcal{L}$, $m \geq 1$, and $L_i s$ $(1 \leq i \leq m)$ are literals. Satisfying an integrity constraint therefore requires its body to be false in the *Well-Founded Semantics* [15], which we shall adopt.

Abductive framework is a triplet $\langle P, \mathcal{AB}, \mathcal{IC} \rangle$ where $\mathcal{AB}$ is a set of abducible predicates, P is a logic program over $\mathcal{L} \backslash \{\bot\}$ and does not contain rules whose head is a predicate in $\mathcal{AB}$,and $\mathcal{IC}$ is a set of integrity constraints. Given a query, abduction amounts to finding abductive solutions, i.e., finding *consistent* truth values of abducibles that make both the program rules satisfied and the query true, while satisfying the integrity constraints.

Abduction in logic programming can be accomplished by a top-down query-oriented procedure for finding its solution by need. The *Well-Founded Semantics*, which is the basis of XSB Prolog [13], allows finding only relevant abducibles and their truth value (true or false) through such a top-down query-oriented procedure. In this case, it is sufficient to consider only rules relevant to the query [3].

C. Updating in Logic Programs

As agents are typically situated in a dynamically changing environment, abduction has to be complemented with the capability to update its knowledge base. Updating in logic programs enables a logic program to represent knowledge dynamically, i.e., allowing knowledge to evolve over time. In logic programming, updating is enacted not only on facts, but can also be performed on rules.

Much research has been done on updating in logic programs, e.g., [7]–[9]. Therein, an evolution of logic programs is indicated by a series of program updates, where a rule may later be supervened by other rules subject to satisfying some conditions. In this paper, the implementation of updating in logic programs is based on EVOLP/R, which is a refinement of EVOLP in [8]. In EVOLP/R, updates are restricted to updates of dynamic literals (viz., *fluents*) rather than full-blown rules. In this case, a fluent F holds at some time T unless its negation complement fluent $not\ F$ supervenes it at a later time $T' > T$.

III. JOINT TABLING OF LOGIC PROGRAM ABDUCTION AND UPDATING

In logic programming, *tabling* is a technique for reusing solutions to a goal rather than recomputing them, by storing a goal and its answers resulting from a query evaluation. While it is similar to the dynamic programming approach, there are other consequences of employing tabling in a logic program, e.g., guaranteeing the termination of a logic program [13].

Currently tabling is featured in some Prolog systems, including XSB Prolog [13], to different extents. *Incremental tabling* is particularly supported by XSB Prolog. This feature warrants maintaining the consistency of tables in the presence of dynamic predicate updates. Both tabling and incremental tabling are useful for abduction and updating in logic programs, resp. Interestingly, intelligent agents may benefit from a tabling mechanism, as it permits immediate decision making (via tabled solutions) based on past reasoning, thus avoiding

repetitive deliberative reasoning.

The combination of logic program abduction and updating, via tabling mechanism, has been proposed by Pereira and Saptawijaya [11]. It essentially integrates two existing subsystems: (1) TABDUAL for *tabling in contextual abduction* [16], and (2) EVOLP/R for *fluent-restricted updating with incremental tabling* [17]. This technique of *joint tabling of logic program abduction and updating* consists of a program transformation and a collection of system predicates that are responsible for realizing the interplay between the abduction and updating parts in this joint tabling technique.

The program transformation for this joint tabling technique is based on that of TABDUAL, extended with an updating mechanism of EVOLP/R. More concretely, the tabling feature is employed for storing abductive solutions of a goal, whereas incremental tabling deals with storing fluent updates (i.e., tabling the *timestamp* of a fluent update).

We provide, for the first time, a complete program transformation involved in this joint tabling technique. The program transformation consists of five parts: tabling abductive solutions, dualized negation, transforming abducibles, initial updates, and transforming a query. The running example below illustrates all parts of the program transformation. The reader is referred to the online appendix of this paper (available from http://bit.ly/appendixJointTabling) for the formal definition of the program transformation. Given an abductive framework $\langle P, \{make\_meal/2\}, \mathcal{IC}\rangle$, where P consists of:

$$meal(porridge).$$
$$hungry(Patient) \leftarrow meal(Meal), \\ make\_meal(Patient, Meal).$$

and $\mathcal{IC}$ contains:

$$\bot \leftarrow hard\_meal(Meal), wearing\_denture(Patient), \\ make\_meal(Patient, Meal).$$

We now explain each part of the program transformation.

A. Tabling Abductive Solution

This section discusses how an abductive solution of a goal is tabled so it can be reused in other abductive contexts. For this purpose, each rule of predicate $p/1$ is transformed into another rule with *tabled* predicate $p_{ab}/3$. For instance, the rule of $hungry(Patient)$ is transformed into:

$$:- table\ hungry_{ab}/3\ as\ incremental.$$
$$hungry_{ab}(Patient, E_3, T) \leftarrow \\ \#r(hungry\_rule, [\,], E_1, T_r), \\ meal(Meal, E_1, E_2, T_m), \\ make\_meal(Patient, Meal, E_2, E_3, T_{mm}), \\ latest([(\#r(hungry\_rule), [\,], E_1, T_r), \\ (meal, Meal, E_1, E_2, T_m), \\ (make\_meal, Patient, Meal, E_2, E_3, T_{mm})], T).$$

This rule of $hungry_{ab}/3$ effectively tables its two extra arguments, viz., the abductive solution E_3 and the timestamp T. Note that the predicate $hungry_{ab}/3$ is tabled incrementally. A new literal (fluent) $\#r/4$, representing the name of this rule, is added into its rule's body. It is introduced in EVOLP/R [17] as a mechanism for rule updating albeit via this rulename fluent. The abductive solution E_3 is obtained by relaying the contents of abductive contexts, indicated by passing E_1, E_2 to E_3, from one subgoal to the subsequent ones in the rule's body. Finally, predicate $latest(Fs, T)$ is borrowed from EVOLP/R [17] to obtain the correct timestamp T of fluent $hungry$ by maintaining the timestamps T_r, T_m, and T_{mm} of each fluent in the body.

In addition to the above rule, an extra rule is defined for each predicate $p/1$ for facilitating abductive solution reuptake via $p_{ab}/3$. For predicate $hungry/1$, we thus add the rule below:

$$hungry(Patient, I, O, T) \leftarrow hungry_{ab}(Patient, E, T), \\ produce\_context(O, I, E).$$

This rule facilitates reusing previously obtained abductive solutions in another context. More precisely, given an *input abductive context* I, it returns an abductive solution of $hungry$ in the *output abductive context* O at *timestamp* T. This abductive solution in O is obtained from the tabled predicate $hungry_{ab}$ while maintaining the timestamp T through incremental tabling. Note that, $produce\_context/3$ warrants the consistency of the output O given the input context I and the already tabled solution E.

B. Dualized Negation

In abduction, a query may contain a negative goal $not\ G$. To facilitate abduction under a negative goal, the concept of dualized negation is employed. For each predicate F, its complement not_F is introduced and may appear in a rule's head, where not_F is true if and only if F is false, and vice versa. This dualized negation concept is realized with the so-called *dual program transformation* [3].

In the dual program transformation, the rule for not_F (called *dual rule*) is defined by falsifying each rule of F in the program. For so doing, two layers of dual rules are introduced. Consider the rule of predicate $hungry/1$. In the first layer, $not\_hungry/4$ is defined by falsifying the only rule of $hungry/1$ in the original program, viz., by invoking the new literal $hungry^{*1}/4$ in the body.

$$not\_hungry(X, E_0, E_1, T) \leftarrow \\ hungry^{*1}(X, E_0, E_1, (D_1, T_1)), \\ latest([(D_1, E_0, E_1, T_1)], T).$$

The definition of $hungry^{*1}/4$ is defined in the second layer by falsifying either subgoal in the body of the original rule of $hungry/1$, including the negation of $hungry/1$'s rulename, to permit abolishing the rule via an update to its name fluent.

$$hungry^{*1}(X, I, O, (not\_\#r(hungry\_rule), T)) \leftarrow \\ not\_\#r(hungry\_rule, I, O, T).$$
$$hungry^{*1}(X, I, O, (not\_meal, T)) \leftarrow \\ not\_meal(X, I, O, T).$$

$$hungry^{*1}(X, I, O, (not\_make\_meal, T)) \leftarrow$$
$$not\_make\_meal(X, Y, I, O, T).$$

C. Transforming Abducibles

Each abducible A is transformed assuring that abducing some Ab within the input abductive context I results in a consistent abductive solution in the output abductive context O. This consistency checking is performed by the system predicate $insert\_abducible(A, I, O)$. For example, the transformation for the abducible $make\_meal$ is:

$$make\_meal(\bar{X}, I, O, \_) \leftarrow$$
$$insert\_abducible(make\_meal(\bar{X}), I, O).$$
$$not\_make\_meal(\bar{X}, I, O, \_) \leftarrow$$
$$insert\_abducible(not\ make\_meal(\bar{X}), I, O).$$

where $\bar{X}$ represents its arguments (viz., $Patient$ and $Meal$). Note the timestamp on the head is an anonymous variable as it is not relevant in this transformation part, but only when the abducible is updated into the program.

D. Initial Updates

Whereas the program transformation, in Sections III.A-III.C, are derived from TABDUAL, the transformation concerning initial updates is additionally introduced in this joint tabling technique. The purpose of this transformation is to activate the rules in program P via the mechanism of rulename fluents.

In this transformation, a rule is activated by setting the timestamp of its corresponding rulename fluent (predicate name $\#r$)to 1. Additionally, the timestamp of its negation complement (predicate name $not\_\#r$) is set initially to 0. For instance, the initial updates concerning the only rule of predicate $hungry$ are as follows:

$$\#r(hungry\_rule, I, I, 1).$$
$$not\_\#r(hungry\_rule, I, I, 0).$$

E. Transforming a Query

Finally, a goal G in a query must also be transformed to adapt with previously discussed transformations. Several extra parameters are added, viz., the input abductive context I, the output abductive context O, the timestamp T when G holds, and the query time Qt when G is queried. The transformation of goal G is wrapped with the system predicate $holds(G, I, O, T, Qt)$, querying whether fluent G is true at query time Qt given input context I. It returns an abductive solution in the output context O and the timestamp T for which G is true. By the definition of $holds/5$, G is true with solution O if it is not supervened by its negation complement (by the same supplied I and O) at a later time $T' > T$.

As an example, query $hungry(alice)$ with empty input context and query time $Qt = 8$ is transformed as follows:

$$?{-}holds(hungry(alice), [\,], E, T, 8),\ not\_\bot(E, O, \_).$$

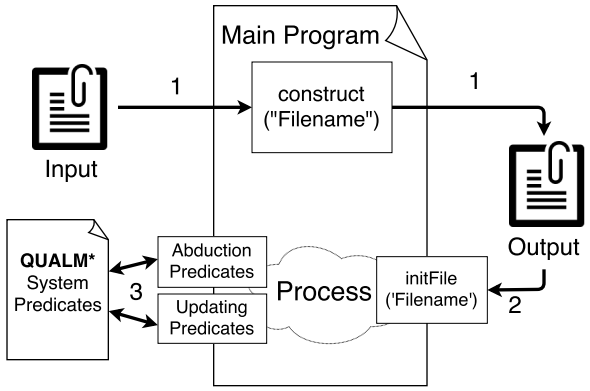

Fig. 1. The architecture of QUALM*

Note the addition of $not\_\bot(E, O, \_)$ to ensure that all integrity constraints are satisfied with respect to its input context E and obtaining an abductive solution in O.

IV. IMPLEMENTATION

The program transformation is implemented in a prototype: QUALM* (available from http://bit.ly/qualmStar). Several implementation aspects are identified to deal with technical issues in the implementation.

A. QUALM*

The concept of joint tabling of logic program abduction and updating (viz., the program transformation, in Section III) is implemented using XSB Prolog in a prototype, QUALM*, while taking into account the implementation aspects discussed in the subsequent section.

The architecture of QUALM* is depicted in Figure 1. In (1), an input file (a logic program) is initially transformed by predicate $construct/1$ and the output file is returned; (2) The output file is then loaded by predicate $initFile/1$; and (3) An agent may invoke queries pertaining to abduction and updating, via abduction predicate ($findAbds/2$) and updating predicates ($activateRule$, $deactivateRule$, $activate$, and $deactivate$), with respect to the output file. These predicates are provided by system predicates of QUALM*.

B. Implementation Aspects

There are three newly identified implementation aspects, which are not captured by the conceptual program transformation. These aspects are summarized below; they complement the implementation aspects introduced for tabling in contextual abduction [16].

1) Dealing with the Inertia of Fluents

Consider two rules: $p \leftarrow a$ and $p \leftarrow b$, where a and b are abducibles, referred to here by their rule names: p_1 and p_2, respectively. These rules are activated on timestamp 1, cf. Section III.D. Querying p on query time $Qt = 2$ will return two abductive solutions: $[a]$ and $[b]$. Suppose rule $p \leftarrow a$ is deactivated on timestamp 3. Querying p at $Qt = 4$ unintendedly returns those two solutions, instead of only $[b]$. In this

case, though invoking $deactivateRule$ for $p \leftarrow a$ successfully propagates the updating of fluent $not\_\#r(p_1, I, I, 3)$ to update fluent not_p (with timestamp 3) via its dual rule, the output context of not_p unfortunately does not match that of p. Therefore, fluent not_p fails to supervene fluent p at timestamp 3 (cf. the definition of $holds/5$, in Section III.E).

One solution for this issue is to store the *time of interest* for each predicate p, viz., a list of timestamps changing the status of a rule of p (from active to deactive, and vice versa). This list is employed to find out at which timestamps a rule of a predicate is innertially active (e.g., $p \leftarrow a$ is active on 1, while $p \leftarrow b$ is active on 1 and is still active on 3).

2) Dealing with Negative Goal

Two issues arise from the transformation for dualized negation when employed in the joint tabling technique.

First, *dealing with the empty set as abductive solution*. The empty set can become an abductive solution of a negative goal due to the deactivation of all rules that propagate fluent $not\_\#r$ for those rules via their dual rules. The issue notably occurs since all rules are initially inactive at timestamp 0 (cf. Section III.D), hence the empty set as an abductive solution is always returned. One solution for this issue is to guarantee that the empty solution is only returned when all rules of a predicate are indeed deactivated at timestamps greater than 0, ignoring the empty solution returned simply by the initial inactivation of rules at timestamp 0.

Second, *dealing with an abductive solution obtained from rules containing abducibles only in their body*. It is possible that all subgoals of a negative goal are abducibles only. Having them as the first argument of predicate $latest$ in the first layer of dual rule (cf. Section III.B) returns an undefined timestamp (in QUALM*, an abducible is not timestamped, unless it is updated as a fact). Consequently, no abductive solution is returned for such a negative goal, because QUALM* returns a solution when its timestamp is defined (and greater than 0). The first step to solve this issue is by introducing all literals preceding the dualized negated subgoal, similar to the solution on grounding dualized negated subgoals introduced for tabling in contextual abduction [16]. Introducing the literals before a subgoal consequently requires that predicate $latest$ now has to inspect all fluents in the body of involved dual rules. This can be done by redefining parameter (D_i, T_i) in the first layer of dual rule (cf. Section III.B) to include all body fluents in involved dual rules in the second layer.

3) Dealing with Updating Abducibles

In transforming abducibles, predicate $insert\_abducible$ (A, I, O) is introduced as a system predicate to insert abducible A into input context I resulting in output context O while keeping its consistency. In practice, an agent may commit itself to an abductive solution (an explanation/a decision). This commitment is realized by updating the abductive solution into the program. Consequently, the system predicate $insert\_abducible/3$ should be generalized. That is, alternatively to appending an abducible A for obtaining a consistent abductive solution, this system predicate should also facilitate updating A, as a fluent, into the program. A simple solution to generalize this system predicate is by storing the latest timestamp an abducible is updated as a fluent.

V. APPLICATION: AN ELDERCARE AGENT

We now discuss an application of QUALM* for knowledge representation and reasoning in intelligent agents, specifically in the context of ambient intelligence for eldercare. Developing an intelligent agent in this domain has gained increasing interest recently, as it requires an agent to make decisions in dynamically changing needs and to be constantly attentive to those needs [1]. The application serves as a practical aspect of the agent's abductive and non-monotonic reasoning in a real-world use, realized by joint tabling of logic program abduction and updating. It particularly aims at showing how this joint tabling technique becomes a solution to tackle the challenges of dual-process model in decision making in a fluctuating environment. The reader is referred to [16], [18] for experiments on evaluating the speed-up and other efficiency aspects of tabling abductive solutions.

In the following scenarios, an agent helps several patients in a nursing home on a rainy season, being attentive to their needs and satisfying them. The knowledge base of this agent is modeled in QUALM* by the code below. Note, predicate $abds/1$ lists all abducible predicates and their corresponding arity. The last two rules are integrity constraints, where $false$ represents $\perp$ (cf. Section II.B).

```
abds([accompany/2, make_meal/2]).

room(alice,101).          room(bob,102).
wearing_denture(bob).     raining.

relaxing_place(garden).
relaxing_place(terrace).
relaxing_place(window).

meal(porridge,soft).      meal(soup,soft).
meal(rice,hard).

bored_not(Patient) <- relaxing_place(Place),
    accompany(Patient, Place).

sleepy_not(Patient) <- bored_not(Patient).
sleepy_not(Patient) <- room(Patient, Number),
    accompany(Patient, room-Number).

hungry_not(Patient) <- meal(Meal, _),
    make_meal(Patient, Meal).

false <- raining, accompany(_, garden).
false <- wearing_denture(Patient),
         meal(Meal, hard),
         make_meal(Patient, Meal).
```

The scenarios are detailed below. In the first scenario, the eldercare agent observes that Alice is bored. The agent invokes a query of $findAbds(bored\_not(alice), X)$ to find out what to do (an action X) in this situation (make Alice to not bored). QUALM* successfully delivers two alternative

solutions: accompany her to terrace or to window. Note, the integrity constraint excludes the solution to accompany Alice to garden due to the fact $raining$ (it is raining).

In the second scenario, the agent observes that Alice is sleepy. Two possible rules are relevant to this state, thus query $findAbds(sleepy\_not(alice), X)$ leads to (a) actions to do when Alice is bored; or (b) accompany Alice to her bedroom. In case (a), the agent does not need to recompute the solutions. Thanks to tabling, (two) solutions obtained from the previous scenario can be reused. The agent only computes the other solution (accompanying Alice to her room) from the second rule of $sleepy\_not/1$ for case (b).

Continuing the second scenario, as the agent chooses to accompany Alice to her room, it turns out that she does not want to sleep. This can be modeled as an input context of not taking her to her room and so find a new action. Without recomputing the solution (because of tabling), the input context filters the possible actions to two candidates, leaving the option of accompanying Alice to her room.

As the time progresses, the agent finds out that Alice is hungry. By invoking query $findAbds(hungry\_not(alice), X)$, three possible actions are delivered: making her meal of porridge, soup, or rice. On the other hand, the agent subsequently learns that Bob is hungry. Invoking $findAbds(hungry\_not(bob), X)$ returns instead two alternative actions; making her meal of rice is discarded due to the last integrity constraint. It is discarded as Bob wears denture and rice is classified a hard meal. Several days have passed, the rainy season is now ended. The knowledge base of the agent is updated with this new observation. The agent invokes query $deactivate(raining, CurrentTime)$, where $CurrentTime$ is the current timestamp when the agent invokes this query (referring the time when the rainy season is ended). Subsequent to this updating query, the query $findAbds(bored\_not(alice), X)$ now additionally returns a solution to accompany Alice to the garden while still satifying the corresponding integrity constraint (because the fact $raining$ is no longer true).

VI. Conclusions and Future Work

Joint tabling of logic program abduction and updating is a knowledge representation and reasoning approach in logic programming that combines abductive reasoning and non-monotonic reasoning (through updating) by making use of an advanced feature in a Prolog system, viz., a tabling mechanism. In this paper, we illustrate a formal definition of program transformation involved in this approach by running examples. We also address the further development of this approach, pertaining to its implementation aspect, and provide a prototype QUALM*, implemented in XSB Prolog. QUALM* is then applied for modeling an intelligent agent for eldercare. The scenarios in this application serve as a solution to answer some challenges of the dual-process model in decision making, involving abductive and non-monotonic reasoning.

It is interesting to explore other applications of QUALM* in intelligent agents, particularly in the field of machine ethics which initially motivates this approach [12], e.g., agents in an interactive storytelling, game, or in counterfactual reasoning.

Acknowledgements

A. F. Sabili and A. Saptawijaya acknowledge the PITTA research grant 397/UN2.R3.1/HKP.05.00/2017 from Directorate Research and Community Services, Universitas Indonesia. L. M. Pereira acknowledges the support from Fundação para a Ciência e a Tecnologia (FCT/MEC) NOVA LINCS PEst UID/CEC/04516/2013.

References

[1] C. Caruso, "Grandma's little robot." Retrieved from https://www.scientificamerican.com/article/grandma-rsquo-s-little-robot/, May 2017.

[2] A. C. Kakas, R. A. Kowalski, and F. Toni, "Abductive logic programming," *Journal of Logic and Computation*, vol. 2, no. 6, pp. 719–770, 1992.

[3] J. J. Alferes, L. M. Pereira, and T. Swift, "Abduction in well-founded semantics and generalized stable models via tabled dual programs," *Theory and Practice of Logic Programming*, vol. 4, no. 4, pp. 383–428, 2004.

[4] L. M. Pereira, P. Dell'Acqua, and G. Lopes, "On preferring and inspecting abductive models," in *Procs. PADL 2009* (A. Gill and T. Swift, eds.), Springer, 2009.

[5] A. C. Kakas and P. Mancarella, "Database updates through abduction," in *16th Very large Database Conference*, pp. 650–661, 1990.

[6] A. C. Kakas and P. Mancarella, "Knowledge assimilation and abduction," in *International Workshop on Truth Maintenance*, vol. 515, pp. 54–71, Springer, 1990.

[7] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski, "Dynamic updates of non-monotonic knowledge bases," *The Journal of Logic Programming*, vol. 45, no. 1, pp. 43–70, 2000.

[8] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira, "Evolving logic programs," in *Procs. JELIA*, pp. 50–61, Springer, 2002.

[9] J. A. Leite, *Evolving Knowledge Bases: Specification and Semantics*. IOS Press, 2003.

[10] F. Cushman, L. Young, and J. D. Greene, "Multi-system moral psychology," in *The Moral Psychology Handbook* (J. M. Doris, ed.), Oxford University Press, 2010.

[11] L. M. Pereira and A. Saptawijaya, "Abduction and beyond in logic programming with application to morality," *IfCoLog Journal of Logics and their Applications*, vol. 3, no. 1, pp. 37–71, 2016.

[12] L. M. Pereira and A. Saptawijaya, *Programming Machine Ethics*, vol. 26 of *SAPERE*. Springer, 2016.

[13] T. Swift and D. S. Warren, "XSB: Extending prolog with tabled logic programming," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 157–187, 2012.

[14] C. Hartshorne and P. Weiss, eds., *Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1932.

[15] A. van Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs.," *Journal of ACM*, vol. 38, no. 3, pp. 620–650, 1991.

[16] A. Saptawijaya and L. M. Pereira, "TABDUAL: a tabled abduction system for logic programs," *IfCoLog Journal of Logics and their Applications*, vol. 2, no. 1, pp. 69–123, 2015.

[17] A. Saptawijaya and L. M. Pereira, "Incremental tabling for query-driven propagation of logic program updates," in *Procs. LPAR-19*, vol. 8312, pp. 694–709, Springer, 2013.

[18] S. M. A. Perkasa, A. Saptawijaya, and L. M. Pereira, "Tabling in contextual abduction with answer subsumption," in *Procs. 9th Intl. Conf. on Advanced Computer Science and Information Systems (ICACSIS)*, 2017.