Answer Set Programming with Resources

Stefania Costantini
Dipartimento di Informatica, Università di L'Aquila
via Vetoio, I-67010 L'Aquila, Italy
stefcost@di.univaq.it

Andrea Formisano
Dipartimento di Matematica e Informatica, Università di Perugia
via Vanvitelli, 1, I-06123 Perugia, Italy
formis@dipmat.unipg.it

Abstract

In this paper, we propose an extension of Answer Set Programming (ASP) to support declarative reasoning on consumption and production of resources. We call the proposed extension RASP, standing for "Resourced ASP". Resources are modeled by introducing special atoms, called *amount-atoms*, to which we associate *quantities* that represent the available amount of a certain resource. The "firing" of a RASP-rule involving amount-atoms can both consume and produce resources. A RASP-rule can be fired several times, according to its definition and to the available quantities of required resources. We define the semantics for RASP programs by extending the usual answer set semantics. Different answer sets correspond to different possible allocations of available resources. We then propose an implementation based on standard ASP-solvers. The implementation consists of a standard translation of each RASP-rule into a set of plain ASP rules and of an inference engine that manages the firing of RASP-rules.

Key words: Answer set programming, non-monotonic logic programming, quantitative reasoning, language extensions.

Introduction

Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [27], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [41, 47]. Rich literature exists on applications of ASP in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [9, 3, 40, 60, 26] and the references therein).

The ASP formulations in these and other fields might take profit from the possibility of performing (at least to some extent) forms of *quantitative* reasoning like those that are possible in Linear Logics [29] and Description Logics [6]. In particular, in linear logics hypotheses are considered as resources: every hypothesis can be consumed exactly once in a proof. This differs from usual logics such as classical or intuitionistic logic where a true statement may be freely used as many times as required. In description logics instead, cardinality restrictions can express conditions on the number of elements that must exist for an existentially quantified property (or "role"). In both cases, one can deal with numbers that express significant aspects of whether a property is verified or a rule can be applied. Concepts similar to cardinalities of Description Logics can be found in some extensions of ASP such as Weight Constraint Rules [51]. However, this is not the case (at least in a direct way) for resource-based reasoning.

In this paper, we propose an extension of ASP to support declarative reasoning on consumption and production of resources. We call the proposed extension RASP, standing for Resourced ASP. Resources are modeled by specific atoms to which we associate *quantities* that represent the available amount. Thus, an atom of the form $q{:}n$ indicates a quantity n of resource q (where n stands for any specific quantity to be chosen). The firing of a RASP-rule can both consume and produce resources. If several tasks require the same resource, different allocation choices may be possible.

After briefly recalling the basic ASP paradigm (Section 1), in Section 2 we provide syntax of RASP programs. In RASP, we allow amount-atoms to occur both in the head and in the body of rules. Amount-atoms in the body of a rule (if present) indicate which resources are *consumed* by that rule, and in which quantity. Amount-atoms that occur in the head of a rule indicate instead which resources are *produced* by that rule, and in which quantity.

Let us focus on a simple example that should clarify the motivations of the approach. Consider a situation where, if one has three eggs, three hundred grams of flour and three hundred grams of sugar, then one can cook a cake. Similarly, if one has three eggs, two hundred grams of sugar and two hundred milliliters of milk, then one can prepare an ice-cream. Depending on the amount of available ingredients, it might be the case that they are sufficient to prepare both, one, or none of the desserts. Program P_1 below could be a RASP program encoding such a situation:

$$\begin{array}{ll}
\gamma_1: & \mathit{have\_cake} \leftarrow \mathit{egg}{:}3,\ \mathit{flour}{:}3,\ \mathit{sugar}{:}3. \\
\gamma_2: & \mathit{have\_ice\_cream} \leftarrow \mathit{egg}{:}3,\ \mathit{sugar}{:}2,\ \mathit{milk}{:}2. \\
\gamma_3: & \mathit{egg}{:}4. \\
\gamma_4: & \mathit{flour}{:}8. \\
\gamma_5: & \mathit{sugar}{:}6. \\
\gamma_6: & \mathit{milk}{:}3.
\end{array}$$

The amounts occurring in the body of each rule correspond to resources which are consumed whenever the rule is used. Facts $\gamma_3, \ldots, \gamma_6$ describe which resources are initially available. Notice that the heads of rules γ_1 and γ_2 do not involve resources, i.e., the two atoms *have_cake* and *have_ice_cream* are plain ASP atoms that simply model the fact of having a cake or an ice-cream available, respectively. We will see in the following that resources can explicitly occur in heads, to model an actual production of some amount of resources. Clearly, one could be interested in finding all possible alternative allocations of resources, possibly subject to further specific constraints or consumption policies (cf., Section 7).

Semantics for RASP programs is provided in Section 3, by combining usual answer set semantics with an interpretation of resource amounts. In particular, different allocation choices will correspond to different answer sets. In Section 4 we provide an encoding of RASP into ASP compatible with the semantics given in Section 3. Completeness and soundness of such encoding are also assessed. The complexity issue is dealt with in Section 5. Later on, we will discuss extensions to the basic RASP paradigm (Section 6). In particular, we allow the same rule to be (optionally) fired more than once, if the available quantities of consumed resources are sufficient for constructing more than one instance of the produced resources. Also, we introduce various kinds of constraints on quantities and the possibility of defining negative quantities to model byproducts of resources consumption/production.

In Section 7 we discuss how to introduce a filter on the answer sets of a RASP program, to represent different *policies* for production and consumption of resources. We identify three basic possibilities, among many which are in principle possible. A general policy can be defined in terms of these three. An implementation of RASP based on standard ASP-solvers is proposed in Section 8. The implementation consists of a standard translation of each RASP-rule into a set of plain ASP rules and of an inference engine that manages the firing of RASP-rules. Related work is discussed in Section 9.

1 ASP in nutshell

In this section, we briefly recall the basics about Answer Set Programming [9, 41, 47]. In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of properties and constraints which describe the (candidate) solutions. More specifically, an *ASP-program* is a collection of *rules* of the form

$$H \leftarrow\ L_1, \ldots, L_m,\ not\ \ L_{m+1}, \ldots, not\ \ L_{m+n}.$$

where H is an atom $m \geqslant 0$, $n \geqslant 0$ and each L_i is an atom. The symbol *not* stands for negation-as-failure. Various extensions to the basic paradigm exist, that we do not consider here all of them as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty head is a *constraint*. Actually, a constraint $\leftarrow\ L_1, \ldots, L_m,\ not\ \ L_{m+1}, \ldots, not\ \ L_{m+n}$ can be seen as a shorthand for a rule of the form $p\ \leftarrow\ L_1, \ldots, L_m,\ not\ \ p,\ not\ \ L_{m+1}, \ldots, not\ \ L_{m+n}$ (being p is a fresh atom). Such a rule imposes that the literals in the body cannot be all true, otherwise the whole rule would be falsified.

The semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, [27]). Consider first the case of a ground[1] ASP-program P which does not involve negation-as-failure (i.e., $n = 0$). In this case, a set of atoms X is said to be an answer set for P if it is the (unique) least model of P. Such a definition is extended to any ground program P containing negation-as-failure by considering the *reduct* P^X (of P) w.r.t. a set of atoms X. P^X is defined as the set of rules of the form $H \leftarrow L_1, \ldots, L_m$ for all rules of P such that X does not contain any of the literals $L_{m+1}, \ldots, L_{m+n}$. Clearly, P^X does not involve negation-as-failure. The set X is an answer set for P if it is an answer set for P^X.

Once a problem is described as an ASP-program P, its solutions (if any) are represented by the answer sets of P. Unlike other semantics, a logic program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set: $\{a \leftarrow \mathit{not}\ b.\ b \leftarrow \mathit{not}\ c.\ c \leftarrow \mathit{not}\ a.\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of answer sets. For a survey of this and other semantics of logic programs with negation, the reader may refer to [4].

Let us consider the program P consisting of the three rules

$$r \leftarrow p. \qquad p \leftarrow \mathit{not}\ q. \qquad q \leftarrow \mathit{not}\ p.$$

Such a program has two answer sets: $\{p, r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q.$ to P, then we rule-out the second of these answer sets, because it violates the new constraint.

This simple example reveals the core of the usual approach followed in formalizing/solving a problem with ASP. Intuitively speaking, the programmer adopts a "generate-and-test" strategy: first (s)he provides a set of rules describing the collection of (all) potential solutions. Then, the addition of constraints rules-out all those answer sets that are not desired real solutions.

Given a rule γ in a language $\mathcal{L}$, the *grounding* of γ w.r.t. $\mathcal{L}$ is the set of all ground rules obtainable from γ through (ground) instantiation using the constant symbols of $\mathcal{L}$. Usually, given a program P and a rule $\gamma \in P$, we will consider the grounding of γ w.r.t. the language underlying P. The grounding of a set of rules is defined similarly. Given a (not necessarily ground) program P, a set of atoms is an answer set for P if it is an answer set for the grounding of P.

To find the solutions of an ASP-program, an ASP-solver is used. Several solvers have became available [5], each of them being characterized by its own prominent valuable features. As it is well-known, ASP solvers produce the

[1]As customary, a term (atom, literal, rule, ...) is ground if no variable occurs in it. A ground program is a program that does not contain variables.

grounding of the given program as a first step, as they are able to find the answer sets of ground programs only.[2]

The expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer, for instance, to [20]. The reader can also see [9, 21], among others, for a presentation of ASP as a tool for declarative problem-solving.

A deeply investigated extension of the ASP language involves the use of *aggregate functions* in defining *aggregate literals* [36, 58, 23]. A simple form of aggregate literal is the following:

$$n_1 \; relop_1 \; Agg\{Var : Conj\} \; relop_2 \; n_2$$

where Var is a variable, $Conj$ is a conjunction of literals, each $relop_i$ is a relational operator (e.g., $<$,$\leqslant$,$\geqslant$, etc.), n_1 and n_2 are terms (called *guards*, usually numbers), and Agg is an aggregate function. (One among the guards might be absent.) Typical aggregate functions are *sum*, *min*, *max*, *count*, etc. The intended meaning is as follows: given an interpretation, the aggregate function Agg is applied to the multiset of all the values for Var that satisfy the conjunction of literals $Conj$. Variables in $Conj$, different from Var, are considered as being existentially quantified. The result of the function is then compared, through $relop_i$, with n_1 and n_2 to determine the truth value of the aggregate literal. As an example, the program

$$\begin{array}{ll} t(1). & t(2). \\ p(3). & p(X) \leftarrow t(X). \\ r(N) \leftarrow N = sum\{Z : p(Z)\}. & \end{array}$$

has answer set $\{t(1), t(2), p(3), p(1), p(2), r(6)\}$.

Following [36], a notion of *stratification* can be introduced for programs involving aggregate literals. Intuitively speaking, a program is *aggregate stratified* if there is no predicate recursively defined through aggregate functions. For example, the previous program is aggregate stratified, while a program containing the rule $q(X) \leftarrow X = sum\{Z : q(Z)\}.$ would be not.

Detailed treatment of the semantics of aggregates in ASP goes beyond the scope of this brief description. The interested reader can refer, among others, to [36, 58, 23]. For the purposes of this paper, it suffices to remark that the notion of reduct of a program P w.r.t. a set of atoms X can be plainly generalized to the case of aggregate stratified programs. This yields a definition of answer set for aggregate stratified programs in complete analogy to the case of plain ASP programs. In this paper, we deal with aggregate stratified programs only.

[2]Work is under way both theoretically and practically to overcome at least partially this limitation (cf., [19, 39], for instance). However, at present almost all ASP solvers perform the grounding.

2 Syntax of RASP-programs

The first step of the definition of RASP syntax is that of partitioning the symbols of the underlying language into $\mathcal{P}$rogram symbols and $\mathcal{R}$esource symbols. Precisely, let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be a structure where $\Pi = \Pi_{\mathcal{P}} \cup \Pi_{\mathcal{R}}$ is a set of predicate symbols such that $\Pi_{\mathcal{P}} \cap \Pi_{\mathcal{R}} = \emptyset$, $\mathcal{C} = \mathcal{C}_{\mathcal{P}} \cup \mathcal{C}_{\mathcal{R}}$ is a set of constant symbols such that $\mathcal{C}_{\mathcal{P}} \cap \mathcal{C}_{\mathcal{R}} = \emptyset$, and $\mathcal{V}$ is a set of variable symbols.

The elements of $\mathcal{C}_{\mathcal{R}}$ are said *amount-symbols*, while the elements of $\Pi_{\mathcal{R}}$ are said *resource-predicates.* A *program-term* is either a variable or a constant symbol. An *amount-term* is either a variable or an amount-symbol.

The second step is that of introducing amount-atoms in addition to plain ASP atoms, called program-atoms. Let $\mathcal{A}(X, Y)$ denote the collection of all atoms of the form $p(t_1, \ldots, t_n)$, with $p \in X$ and $\{t_1, \ldots, t_n\} \subseteq Y$. Then, a *program-atom* is an element of $\mathcal{A}(\Pi_{\mathcal{P}}, \mathcal{C} \cup \mathcal{V})$. An *amount-atom* is an expression of the form $q{:}a$ where $q \in \Pi_{\mathcal{R}} \cup \mathcal{A}(\Pi_{\mathcal{R}}, \mathcal{C} \cup \mathcal{V})$ and a is an amount-term. Let $\tau_{\mathcal{R}} = \Pi_{\mathcal{R}} \cup \mathcal{A}(\Pi_{\mathcal{R}}, \mathcal{C})$. We call elements of τ_R *resource-symbols.*

Example 2.1 *In the two expressions $p{:}3$ and $q(2){:}b$, p and $q(2)$ are resource-symbols (with $p, q \in \Pi_{\mathcal{R}}$ and $2 \in \mathcal{C}$) aimed at defining two resources which are available in quantity 3 and b, resp., (with $3, b \in \mathcal{C}_{\mathcal{R}}$ amount-symbols).*

Expressions such as $p(X){:}V$ where V, X are variable symbols are also allowed, as resources can be either directly specified as constants or derived. Notice that the set of variables is not partitioned, as the same variable may occur both as a program term and as an amount-term. *Ground* amount- or program-atoms contain no variables. As usual, a *program-literal* L is a program-atom A or the negation *not* A of a program-atom (intended as negation-as-failure).[3] If $L = A$ (resp., $L = \mathit{not}\ A$) then $\overline{L}$ denotes *not* A (resp., A). A *resource-literal* (r-literal) is either a program-literal or an amount-atom. Notice that we do not admit negation of amount-atoms. The reasons will be discussed later.

Finally, we distinguish between plain rules and rules that involve amount-atoms. In particular, a *program-rule* is defined as a regular ASP rule. Besides program-rules we introduce resource-rules which differ from program rules in that they may contain amount-atoms.

Definition 2.1 *A* resource-proper-rule *has the form*

$$H \leftarrow B_1, \ldots, B_k.$$

where $B_1, \ldots, B_k$, $k > 0$, are r-literals and H is either a program-atom or a (non-empty) list of amount-atoms.

Definition 2.2 *A* resource-fact *(*r-fact*, for short) has the form $H \leftarrow .$ where H is an amount-atom $q{:}a$ and a is an amount symbol.*

[3]In this paper we will only deal with negation-as-failure. Nevertheless, classical negation of program literals could be used in RASP programs and treated as usually done in ASP.

According to the definition, the amount of an initially available resource has to be explicitly stated. Thus, the amount term a cannot be a variable. (As usual, we often denote the r-fact $H \leftarrow$ simply by writing H.)

Definition 2.3 *A* resource-rule *(*r-rule*, for short) can be either a resource-proper-rule or a resource-fact.*

A RASP program may involve both program rules and resource-rules:

Definition 2.4 *A* RASP-rule *(*rule*, for short)* γ *is either a program-rule or a resource-rule. An* r-program *is a finite set of RASP-rules.*

Example 2.2 *Let us consider the r-program* P_1 *mentioned in the Introduction. All the rules are r-rules. We identify* $\mathcal{C}_{\mathcal{R}}$ *with the set of integer numbers* $\mathbb{Z}$ *and we have* $\Pi_{\mathcal{R}} = \{egg, flour, sugar, milk\}$ *and* $\Pi_{\mathcal{P}} = \{have\_cake, have\_ice\_cream\}$*.*

Remark 2.1 *As Definition 2.1 states, in general, we admit several amount-atoms in the head of a rule where the case in which a rule* γ *has an empty head is admitted only if* γ *is a program-rule (i.e.,* γ *is an ASP* constraint*).*

Notice that the list of amount-atoms composing the head of an r-rule has to be understood conjunctively, i.e., as a collection of those resources that are all produced at the same time by firing the rule.

Example 2.3 *The combustion reaction of methane can be described by the following equation:*

$$CH_4 + 2O_2 \quad \Rightarrow \quad CO_2 + 2H_2O$$

where starting from a molecule of methane (CH_4) *and two molecules of oxygen* (O_2)*, we obtain a molecule of carbon dioxide* (CO_2) *and two molecules water* (H_2O)*. The following r-rule encodes such a reaction:*

$$carbDioxide{:}1, water{:}2 \;\leftarrow\; methane{:}1, oxygen{:}2.$$

where two amount-atoms in the head represent the simultaneous production of carbon dioxide and water.

The grounding of an r-program P is the set of all ground instances of rules of P, obtained through ground substitutions over the constants occurring in P.

Remark 2.2 *Notice that in any r-program only a finite number of amount-symbols of* $\mathcal{C}_{\mathcal{R}}$ *occurs, also because all r-facts must be ground. Hence, as far as amount-atoms are concerned, a finite number of ground instances can be generated by the grounding process. This is because all instances of amount-terms are among the instances of the terms occurring in program atoms. A "smart" grounder for RASP would avoid generating instances of r-rule where variables occurring as amount-terms are instantiated to constants of* $\mathcal{C}_{\mathcal{P}}$ *instead of constants of* $\mathcal{C}_{\mathcal{R}}$*. Such "wrong" instances are however both semantically and practically irrelevant (apart from the waste of space).*

3 Semantics of RASP-programs

The semantics of a (ground) RASP program is determined by interpreting usual literals as in ASP and amount-atoms in an auxiliary algebraic structure that supports operations and comparisons. The rationale behind the proposed semantic definition is the following. On the one hand, we translate each r-rule into a fragment of a plain ASP program, so that we do not have to modify the definition of stability which remains the same: this is of some importance in order to make the several theoretical and practical advances in ASP still available for RASP. However, an answer set of a RASP program will support the firing of an r-rule only if: the rule is satisfied (in the usual way) as concerns its program-literals; and the requested amounts are allocated for all the resource-atoms. Hence, an interpretation (and consequently an answer set) for an r-program has two components: a set of program atoms and an allocation of actual quantities to amount-atoms.

In the semantics that we present below we do not consider negation-as-failure applied to amount-atoms. The reason is not technical, as there might be several possible approaches to assess this semantics. The problem is that it is not clear to us which is the most intuitive meaning of saying, for instance, that "an amount-atom $q{:}a$ (occuring in the head) cannot be assumed to be true". Should it mean that the resource q is not produced at all? Or should it mean that q can be produced in any amount different from a? Moreover, what should be considered as the *scope* of this constraint? A single rule or the whole program? Actually, some form of constrains conceptually involving negation are expressible in RASP: one can use negated program atoms to impose that an r-rule can produce some amount of the resource q, but this amount cannot be equal to a specific value a. For instance, as in the following r-rule:

$$q{:}X \leftarrow q_1{:}Y,\ X \neq a, p_1(X,Y),\ not\ p_2(Y).$$

where, similarly, further restrictions on the values of the consumed amount Y of q_1 can be imposed through the program predicates p_1 and p_2. Moreover, one can impose constraints on the global balance of a resource (for example, to impose that, globally, a certain amount of a resource has to be used/left). This topic is discussed at some length in the next sections.
Below is an example of how negation can be (though in an indirect way) exploited in RASP programs without committing to a specific meaning of negation of amount-atoms.

$$\begin{array}{ll}
\gamma_1: & happy\_wife \leftarrow cinema.\\
\gamma_2: & cinema \leftarrow money{:}3.\\
\gamma_3: & happy\_husband \leftarrow restaurant.\\
\gamma_4: & restaurant \leftarrow money{:}6.\\
\gamma_5: & trouble\_at\_home \leftarrow happy\_wife,\ not\ \ happy\_husband.\\
\gamma_6: & trouble\_at\_home \leftarrow not\ \ happy\_wife,\ happy\_husband.
\end{array}$$

Rules γ_2 and γ_4 compete for the assignment of the resource *money* (and may also compete with other rules that may be present in the rest of the program).

If enough money is available or is produced, e.g., by the rule

$$\gamma_7 : \qquad \mathit{money}{:}10 \leftarrow \mathit{hour\_of\_work}{:}3.$$

then there will be an answer set where both husband and wife are happy, and there is no trouble at home. If the available money is insufficient for either uses, none of them will be concluded to be happy. If the money is sufficient just for one, only that one will be concluded to be happy by the above program fragment. Answer sets where only one is happy can be ruled-out by the constraint:

$$\gamma_8 : \qquad \leftarrow \mathit{trouble\_at\_home}.$$

In describing the semantics of an r-program P we will proceed as follows. First we fix an algebraic structure to represent quantities and support operations on them. Then, we develop a representation for collections of quantities with positive balance. (Intuitively, negative amounts represent consumed resources, while positive amounts denote productions.) Each of these collections will be a potential allocation of quantities to all the amount-atoms relative to a single resource symbol in P. Then, we introduce the notion of r-interpretation of P by selecting an allocation of amounts for each resource symbol in P.

As mentioned, amounts are modeled by choosing a collection Q of *quantities*, the operations to combine and compare quantities, and a mapping $\kappa : \mathcal{C}_{\mathcal{R}} \to Q$ that associates quantities to amount-symbols. A simple natural choice for Q is the set of integer numbers. In what follows, unless differently specified, we will assume that $Q = \mathbb{Z}$.

Remark 3.1 *Plainly, alternative options for Q are possible. For instance, one could choose Q to be the set $\mathbb{Q}$ of rational numbers.*

In general, amounts can be interpreted by choosing Q to be any set as carrier of a structure $\mathcal{Q} = \langle Q, \oplus_{\mathcal{Q}}, \preccurlyeq_{\mathcal{Q}}, \emptyset_{\mathcal{Q}} \rangle$ that is a totally-ordered Abelian group, where $\emptyset_{\mathcal{Q}}$ denotes the identity element in $\mathcal{Q}$ w.r.t. the additive operation $\oplus_{\mathcal{Q}}$, and $\preccurlyeq_{\mathcal{Q}}$ is a translation-invariant total order over Q (i.e., such that for all $x, y, z \in Q$, $x \preccurlyeq_{\mathcal{Q}} y \to x \oplus_{\mathcal{Q}} z \preccurlyeq_{\mathcal{Q}} y \oplus_{\mathcal{Q}} z$).

Example 3.1 *Consider the r-program P_1 used in the Introduction and in Example 2.2. By choosing Q to be $\mathbb{Z}$ (with the usual sum and total order relation), since we have identified $\mathcal{C}_{\mathcal{R}}$ with $\mathbb{Z}$, κ is the identity function over $\mathbb{Z}$.*

As will be discussed in Section 8, a solver for RASP can be obtained by exploiting existing ASP-solvers through compilation into ASP. Once κ has been chosen, it can be encoded in r-programs through program-predicates and usual ASP-rules. Alternatively, one can directly use numerical terms and rely on the built-in features of the specific ASP-solver. For the sake of simplicity, in the rest of the presentation whenever it will be clear from the context (e.g., in the examples), we will adopt a simplification by identifying $\mathcal{C}_R$ with $\mathbb{Z}$ (and κ being the identity).

Before going on, we introduce some useful notation. Given two sets X, Y, let $\mathcal{FM}(X)$ denote the collection of all finite multisets[4] of elements of X, and let Y^X denote the collection of all (total) functions having X and Y as domain and codomain, respectively. Given a (multi)set Z of integers, $\sum(Z)$ denotes their sum (e.g., $\sum(\{\!\!\{ 2, 5, 3, 3, 5 \}\!\!\}) = 18$).

We introduce now the notion of r-interpretation for r-programs, by considering the ground case in the first place. For any fixed resource-symbol, an interpretation of a (ground) r-program P must determine an allocation of amounts for all occurrences of such a symbol in rules of P. Since amounts and resource-symbols are used to model production and consumption of "real-world" objects, we must take into account the obvious constraint that any resource cannot be consumed if it is not produced. In other words, for each resource symbol q, the overall sum of quantities allocated to amount-atoms of the form $q{:}a$ must not be negative. The collection $\mathbb{S}_P$ of all potential allocations (i.e., those having a non-negative global balance)—for any single resource-symbol occurring in P (considered as a set of rules)—is the following collection of mappings:

$$\mathbb{S}_P = \{F \in (\mathcal{FM}(Q))^P \mid 0 \leqslant \sum \big(\bigcup_{\gamma \in P} F(\gamma) \big) \} \tag{1}$$

The rationale behind the definition of $\mathbb{S}_P$ is as follows: Let q be a fixed resource-symbol. Each element $F \in \mathbb{S}_P$ is a function that associates to each rule $\gamma \in P$ a (possibly empty) multiset $F(\gamma)$ of quantities, assigning certain quantities to each occurrence of amount-atoms of the form $q{:}a$ in γ. All such F must satisfy, by definition of $\mathbb{S}_P$, the requirement that, considering the entire P, the global sum of all the quantities F assigns must be non-negative. As we will see later, only some of these allocations will actually be acceptable as a basis for a model.

To interpret an r-program, we select a collection of elements of $\mathbb{S}_P$, one element for each amount-symbol in $\tau_{\mathcal{R}}$. More formally, an r-interpretation of a ground r-program P is defined by providing a mapping

$$\mu : \tau_{\mathcal{R}} \to \mathbb{S}_P.$$

Such a function μ determines, for each resource-symbol $q \in \tau_{\mathcal{R}}$, a mapping $\mu(q) \in \mathbb{S}_P$. In turn, each mapping $\mu(q)$ assigns to each rule $\gamma \in P$ a multiset $\mu(q)(\gamma)$ of quantities, as explained above. As we will see, the function μ is one component of an interpretation $\mathcal{I}$. The interpretation is allowed to be an answer set only if the quantities assigned by μ are consistent with the firing of the rules.

The use of multisets allows us to handle multiple copies of the same amount-atom: each of these copies must be taken into account, since it corresponds to a different amount of resource consumed or produced. Notice that, it is possible that the grounding of an r-rule actually produces multiple copies of the same amount-atom within the same instance of the r-rule. Consider for example the

[4]A multiset (or bag) is a generalization of a set, where repeated elements are allowed. Then, a member of a multiset can have more than one membership.

r-rule $q{:}X \leftarrow p(a){:}X,\ p(B){:}Y,\ r(B,X,Y)$. Depending on how $r(B,X,Y)$ is instantiated, the two amount-atoms in the body might become identical.

Recall that in characterizing the notion of r-interpretation for an r-program we have to take into account two kinds of literals: resource-atoms and program literals. Namely, two aspects have to be considered: the truth of program literals and the allocation of resources. Let $\mathcal{B}(X,Y)$ denote the collection of all ground atoms built up from predicate symbols in X and terms in Y. We have the following definition.

Definition 3.1 *An* r-interpretation *for a (ground) r-program P is a pair $\mathcal{I} = \langle I, \mu\rangle$, with $I \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C})$ and $\mu : \tau_{\mathcal{R}} \to \mathbb{S}_P$.*

Intuitively: I plays the role of a usual answer set assigning truth values to program-literals.

Example 3.2 *Let us consider again the r-program P_1 of Examples 2.2 and 3.1. An r-interpretation for P_1 is $\langle I, \mu\rangle$ with $I = \{have\_cake\}$ and μ such that*

$$\begin{array}{lcl}
\mu(egg)(\gamma_1) & = & \{\!\![\, -3\,]\!\!\} \\
\mu(egg)(\gamma_3) & = & \{\!\![\, 4\,]\!\!\} \\
\mu(egg)(\gamma_i) & = & \emptyset \;\; \textit{for } i \in \{2,4,5,6\} \\
\mu(flour)(\gamma_1) & = & \{\!\![\, -3\,]\!\!\} \\
\mu(flour)(\gamma_4) & = & \{\!\![\, 8\,]\!\!\} \\
\mu(flour)(\gamma_i) & = & \emptyset \;\; \textit{for } i \in \{2,3,5,6\} \\
\mu(sugar)(\gamma_1) & = & \{\!\![\, -3\,]\!\!\} \\
\mu(sugar)(\gamma_5) & = & \{\!\![\, 6\,]\!\!\} \\
\mu(sugar)(\gamma_i) & = & \emptyset \;\; \textit{for } i \in \{2,3,4,6\} \\
\mu(milk)(\gamma_6) & = & \{\!\![\, 3\,]\!\!\} \\
\mu(milk)(\gamma_i) & = & \emptyset \;\; \textit{for } i \in \{1,2,3,4,5\}.
\end{array}$$

The firing of an r-rule (which may involve consumption/production of resources) can happen only if the truth values of the program-literals satisfy the rule. We reflect the fact that the satisfaction of an r-rule γ depends upon the truth of its program-literals by introducing a suitable fragment of ASP program $\widehat{\gamma}$. As we will see below, this fragment is taken into consideration only if, in the interpretation $\mathcal{I}$ at hand, rule γ is fired (i.e., $\mathcal{I}$ assigns the correct quantities to the amount-atoms in γ). The set of ASP rules $\widehat{\gamma}$ is used to state that $\mathcal{I}$ is allowed to be an answer set only if, whenever γ is fired, its program literals are satisfied in $\mathcal{I}$. Let the r-rule γ, have $L_1, \ldots, L_k$ as program-literals and $R_1, \ldots, R_h$ as amount-atoms. The ASP-program $\widehat{\gamma}$ is so defined:

$$\widehat{\gamma} = \begin{cases}
\{\leftarrow \overline{L_1}.,\ldots,\leftarrow \overline{L_k}.\} & \text{if the head of } \gamma \text{ consists of amount-atoms} \\[1ex]
\begin{array}{r}\{\leftarrow \overline{L_1}.,\ldots,\leftarrow \overline{L_k}., \\ H \leftarrow L_1,\ldots,L_k.\ \}\end{array} & \begin{array}{l}\text{if } \gamma \text{ has the program-atom } H \text{ as head} \\ \quad \text{and } h > 0\end{array} \\[1ex]
\{\,\gamma\,\} & \text{otherwise.}
\end{cases}$$

Notice that if γ is a program-rule then $\widehat{\gamma} = \{\gamma\}$.

Let us introduce a notation to indicate, for each given resource symbol q and each r-rule γ, the multiset of amounts of q involved in the firing of γ (Negative quantities are associated to amount-atoms of the body of γ, as these resources are consumed.):

$$\begin{aligned} \mathit{Alloc}(q,\gamma) \;=\; & \{\!\![\, v \in \mathbb{Z} \mid v = -\kappa(a) \text{ for } q{:}a \text{ in the body of } \gamma \\ & \qquad\qquad \text{or } \; v = \kappa(a) \text{ for } q{:}a \text{ in the head of } \gamma \,]\!\!\} \end{aligned} \tag{2}$$

The following definition states that in order to be a model, an r-interpretation that allocates non-void amounts to the resource-symbols of γ, has to model the ASP-rules in $\widehat{\gamma}$.

Definition 3.2 *Let $\mathcal{I} = \langle I, \mu \rangle$ be an r-interpretation for a (ground) r-program P. $\mathcal{I}$ is an* answer set *for P if the following conditions hold:*

- *for all rules $\gamma \in P$*

$$\Big(\forall\, q \in \tau_{\mathcal{R}} \; \big(\mu(q)(\gamma) = \emptyset\big)\Big) \vee \Big(\forall\, q \in \tau_{\mathcal{R}} \; \big(\mu(q)(\gamma) = \mathit{Alloc}(q,\gamma)\big)\Big) \tag{3}$$

- *I is a stable model for the ASP-program $\widehat{P}$, so defined*

$$\widehat{P} = \bigcup \left\{ \; \widehat{\gamma} \; \left| \begin{array}{l} \gamma \textit{ is a program-rule in } P, \;\; \textit{or} \\ \gamma \textit{ is a resource-rule in } P \textit{ and } \exists\, q \in \tau_{\mathcal{R}} \; \big(\mu(q)(\gamma) \neq \emptyset\big) \end{array} \right. \right\}$$

The two disjoints of formula (3) in Definition 3.2 correspond to the two cases: a) the rule γ is not fired, so null amounts are allocated to all its amount-symbols; b) the rule γ is actually fired and all needed amounts are allocated.

We now formally introduce the notions of *resource balance*:

Definition 3.3 *Let $\mathcal{I} = \langle I, \mu \rangle$ be an answer set for a (ground) r-program P. The* resource balance *for P, w.r.t. $\langle I, \mu \rangle$, is the mapping $\varphi : \tau_{\mathcal{R}} \to Q$ defined as:*

$$\varphi(q) = \sum \left(\{\!\!\Big[\sum \big(\mu(q)(\gamma)\big) \mid \gamma \in P \Big]\!\!\} \right)$$

which summarizes consumptions and productions of all resources.

An r-interpretation $\mathcal{I}$ is an answer set of an r-program P if it is an answer set for the grounding of P.

Example 3.3 *Consider the r-interpretation $\langle I, \mu \rangle$ of Example 3.2. The program $\widehat{P_1}$ is made of the single fact have_cake (i.e., $\widehat{\gamma_1}$). Hence, I is a stable model for $\widehat{P_1}$. Notice that this r-interpretation fires the rule γ_1. The resource balance φ is such that: $\varphi(egg) = 1$, $\varphi(flour) = 5$, $\varphi(sugar) = 3$, and $\varphi(milk) = 3$.*

Another simple example involving r-rules that consume and produce the same resource.

Example 3.4 *Let us consider the following two r-rules:*

$$\begin{array}{ll} \gamma_a: & q{:}2 \leftarrow q{:}1. \\ \gamma_b: & q{:}1 \leftarrow q{:}2. \end{array}$$

Let P_a (resp., P_b) be the r-program made of the single r-rule γ_a (resp., γ_b) and let $P_{ab} = P_a \cup P_b$. For each of the three r-programs, given an r-interpretation, each r-rule might be fired at most once. In particular, for the r-program P_a there are two possible answer sets. Namely, $\mathcal{I}_1 = \langle \emptyset, \mu_1 \rangle$ and $\mathcal{I}_2 = \langle \emptyset, \mu_2 \rangle$, with $\mu_1(q)(\gamma_a) = \emptyset$ and $\mu_2(q)(\gamma_a) = \{\!\!\{ 2, -1 \}\!\!\}$, so that $\varphi(q) = 1$. (Notice that $\mathcal{I}_2$ and $\mathcal{I}_1$ correspond to the cases in which γ_a is fired or not, respectively.) For P_b there is only one possible answer set: $\mathcal{I}_3 = \langle \emptyset, \mu_3 \rangle$ with $\mu_3(q)(\gamma_b) = \emptyset$ and $\varphi(q) = 0$. (Actually, the r-rule cannot be fired because not enough resources are available). For the r-program P_{ab} there are several possibilities that correspond to these situations: (i) no r-rule is fired, (ii) only γ_a is fired, and (iii) both r-rules are fired. In the last case, the answer set is $\mathcal{I}_4 = \langle \emptyset, \mu_4 \rangle$ with $\mu_4(q)(\gamma_a) = \{\!\!\{ 2, -1 \}\!\!\}$ and $\mu_4(q)(\gamma_b) = \{\!\!\{ 1, -2 \}\!\!\}$ and $\varphi(q) = 0$.

The following example illustrates a crucial difference between resource-atoms and program atoms. We will use a slightly modified version P_2 of the r-program P_1 used in Example 2.2. In this case, the firing of a rule will produce some resource (i.e., one *cake*, see below) instead of making a program literal true (i.e., *have_cake* in Example 3.3).

Example 3.5 *Let us consider the r-program P_2 made of the rules $\gamma_3, \ldots, \gamma_6$ seen in the Introduction and the following two rules in place of γ_1 and γ_2:*

$$\begin{array}{ll} \gamma_1': & \mathit{cake}{:}1 \leftarrow \mathit{egg}{:}3,\ \mathit{flour}{:}3,\ \mathit{sugar}{:}3. \\ \gamma_2': & \mathit{ice\_cream}{:}1 \leftarrow \mathit{egg}{:}3,\ \mathit{sugar}{:}2,\ \mathit{milk}{:}2. \end{array}$$

The crucial difference between $\gamma'_{1,2}$ and $\gamma_{1,2}$ is that in $\gamma'_{1,2}$ we use amount-atoms as heads, instead of program atoms. As for the r-program P_1 we identify $\mathcal{C}_\mathcal{R}$ with $\mathbb{Z}$, but now we have $\Pi_\mathcal{R} = \{\mathit{cake}, \mathit{ice\_cream}, \mathit{egg}, \mathit{flour}, \mathit{sugar}, \mathit{milk}\}$ and $\Pi_\mathcal{P} = \emptyset$. An r-interpretation for P_2 is $\langle \emptyset, \mu' \rangle$ with μ' such that

$$\begin{array}{lcllcl} \mu'(\mathit{egg})(\gamma_1') & = & \{\!\!\{ -3 \}\!\!\} & \mu'(\mathit{egg})(\gamma_2') & = & \emptyset \\ \mu'(\mathit{flour})(\gamma_1') & = & \{\!\!\{ -3 \}\!\!\} & \mu'(\mathit{flour})(\gamma_2') & = & \emptyset \\ \mu'(\mathit{sugar})(\gamma_1') & = & \{\!\!\{ -3 \}\!\!\} & \mu'(\mathit{sugar})(\gamma_2') & = & \emptyset \\ \mu'(\mathit{milk})(\gamma_1') & = & \emptyset & \mu'(\mathit{milk})(\gamma_2') & = & \emptyset \\ \mu'(\mathit{cake})(\gamma_1') & = & \{\!\!\{ 1 \}\!\!\} & \mu'(\mathit{cake})(\gamma_2') & = & \emptyset \\ \mu'(\mathit{cake})(\gamma_i) & = & \emptyset \ \ \mathit{for}\ i \in \{3,4,5,6\} & & & \\ \mu'(\mathit{ice\_cream})(\gamma_i') & = & \emptyset \ \ \mathit{for}\ i \in \{1,2\} & & & \\ \mu'(\mathit{ice\_cream})(\gamma_i) & = & \emptyset \ \ \mathit{for}\ i \in \{3,4,5,6\} & & & \\ \mu'(q)(\gamma_i) & = & \mu(q)(\gamma_i) \ \ \mathit{otherwise} & & & \end{array}$$

where μ is as in Example 3.2. The program $\widehat{P_2} = \widehat{\gamma_1}'$ is empty. Rule γ_1' is fired (as for rule γ_1 in Example 3.3). In this case the resource balance φ' is: $\varphi'(\mathit{egg}) = 1$, $\varphi'(\mathit{flour}) = 5$, $\varphi'(\mathit{sugar}) = 3$, $\varphi'(\mathit{milk}) = 3$, $\varphi'(\mathit{cake}) = 1$, and $\varphi'(\mathit{ice\_cream}) = 0$.

The next example involves interactions between r-rules and ASP rules: through the use of variables, the values admitted for resource amounts can be controlled by means of program predicates; also, compound program atoms and terms can be used to define different ground instance of r-rules (cf., the first rule of the program below).

Example 3.6 *A simple example of a "toy heating system". It is a dual-fuel system and comprises two fuel sources: electricity as a primary system and alternate fuel such as gas or fuel oil as a secondary source. The control switches the electric heat off and the backup fuel on during peak load conditions, typically on the coldest days of the winter. We can represent this situation by separately modeling the knowledge base regarding the heating system, and the description of specific weather situation. The generic* domain knowledge *is modeled through this r-program (where we assume $a_1, a_2, a_3 \in \mathbb{Z}$):*

$$
\begin{array}{l}
\mathit{warm} \leftarrow \mathit{fuel}(\mathit{Type}){:}Q, \mathit{load}(\mathit{Load}),\\
\qquad\qquad \mathit{requiredFuel}(\mathit{Load}, \mathit{Type}), \mathit{needed}(\mathit{Type}, Q).\\
\mathit{load}(\mathit{low}) \leftarrow \mathit{temperature}(\mathit{Celsius}), \mathit{Celsius} > 10, \mathit{is\_winter}.\\
\mathit{load}(\mathit{high}) \leftarrow \mathit{temperature}(\mathit{Celsius}), \mathit{Celsius} =< 10, \mathit{is\_winter}.\\
\mathit{load}(\mathit{none}) \leftarrow \mathit{not}\ \mathit{is\_winter}.\\
\mathit{requiredFuel}(\mathit{high}, \mathit{oil}).\\
\mathit{requiredFuel}(\mathit{low}, \mathit{electricity}).\\
\mathit{needed}(\mathit{oil}, a_2).\\
\mathit{needed}(\mathit{electricity}, a_1).\\
\mathit{fuel}(\mathit{oil}){:}a_1 \leftarrow \mathit{money}{:}a_3.\\
\mathit{fuel}(\mathit{electricity}){:}a_2 \leftarrow \mathit{money}{:}a_3.\\
\leftarrow \mathit{not}\ \mathit{warm}.
\end{array}
$$

The last rule, an ASP constraint, represents the desired goal: to warm the whereabouts. This fragment of program can be seen as the inference engine to be joined with a specific instance of the "heating problem". Specific operating conditions can be stated by describing the current weather (as well as the availability of money). If it is a cold winter we write:

$$
\begin{array}{l}
\mathit{temperature}(5).\\
\mathit{money}{:}a_3.\\
\mathit{is\_winter}.
\end{array}
$$

otherwise, for a mild winter we could assert the fact temperature(20) in place of temperature(5). Finally, to obtain from a RASP-solver (see Section 8), both models as (alternative) expected solutions of a single r-program, we could use this fragment of program:

$$
\begin{array}{l}
\mathit{temperature}(5) \leftarrow \mathit{not}\ \mathit{temperature}(20).\\
\mathit{temperature}(20) \leftarrow \mathit{not}\ \mathit{temperature}(5).\\
\mathit{money}{:}a_3.\\
\mathit{is\_winter}.
\end{array}
$$

In this manner, one and only one among the atoms temperature(5) and temperature(20) has to be true in any model. In the former case we have a high peak load (load(high) is true) and oil is used. The other solution, when the temperature is 20 degrees (load(low) is true), involves using electricity.

The next example shows, in a simplified context, how the products of a rule can be used to sustain the firing of another rule.

Example 3.7 *We take advantage from having (multi)sets of amount-atoms as heads of rules, to model a chemical reaction. We consider a simplified description of the photosynthesis in green plants. Photosynthesis uses the energy of light to convert carbon dioxide (CO_2) and water (H_2O) into glucose ($C_6H_{12}O_6$) and oxygen (O_2). The process can be roughly described as two interleaved phases, called respectively* light-dependent reaction *(which converts solar energy into chemical energy stored in specific molecules, NADPH and ATP) and* carbon fixation *(which captures carbon dioxide and makes the precursors of glucose). Simple general equations that schematize the two phases are (adapted from [18]):*

light-dependent reaction:

$$\begin{array}{c} 12H_2O + \textit{ light energy } + \\ 12NADP^+ + 18ADP + 18Pi \end{array} \Rightarrow \begin{array}{c} 6O_2 + 18ATP + \\ 12NADPH + 12H^+ \end{array}$$

carbon fixation:

$$\begin{array}{c} 6CO_2 + 6O_2 + 18ATP + \\ 12NADPH + 12H^+ \end{array} \Rightarrow \begin{array}{c} C_6H_{12}O_6 + 6H_2O + \\ 12NADP^+ + 18ADP + 18Pi \end{array}$$

The following rules encode such reactions:

$$\begin{array}{lcl} \textit{oxygen}{:}A, \textit{atp}{:}C, \textit{nadph}{:}B, \textit{proton}{:}B & \leftarrow & \textit{water}{:}B, \textit{adp}{:}C, \textit{nadp}{:}B, \textit{pi}{:}C, \\ & & \textit{light}, B = 2 * A, C = 3 * A, \\ & & A = 6 * \textit{Moles}, \\ & & \textit{reagents}_1(\textit{Moles}, A, B, C, D). \\ \textit{glucose}{:}D, \textit{water}{:}A, \textit{adp}{:}C, \textit{nadp}{:}B, \textit{pi}{:}C & \leftarrow & \textit{nadph}{:}B, \textit{proton}{:}B, \textit{atp}{:}C, \\ & & \textit{oxygen}{:}A, \textit{carbDioxide}{:}A, \\ & & A = 6 * D, B = 2 * A, \\ & & C = 3 * A, \\ & & \textit{reagents}_2(A, B, C, D). \\ \textit{reagents}_1(V, W, X, Y, Z) \leftarrow \ldots & & \\ \textit{reagents}_2(W, X, Y, Z) \leftarrow \ldots & & \end{array}$$

where we generically used the two predicates $\textit{reagents}_{1,2}$ (assumed to be defined elsewhere in the program) to indicate that values for A, B, C, D, and Moles can be obtained/restrained through other parts of the r-program.

4 ASP Encoding of r-programs

In this section, we will describe an ASP-encoding of ground RASP programs. This will characterize an abstract implementation of RASP. We will proceed by providing a translation $\mathcal{T}$ from r-programs to ASP programs. By means of $\mathcal{T}$, any answer set of an r-program S determines an answer set for $\mathcal{T}(S)$ (which satisfies suitable constraints on resource balances), and vice versa.

Again, for the sake of simplicity, in what follows we will identify amount symbols of $\mathcal{C}_{\mathcal{R}}$ with their interpretations in $\mathbb{Z}$. (Consequently, the mapping κ will be left implicit.)

4.1 A translation from RASP into ASP

Let be given a set of fresh predicate symbols $\Pi_{\mathcal{T}}$ such that $\Pi_{\mathcal{T}} \cap \Pi = \emptyset$ and $\{\mathit{notfired}, \mathit{fired}, \mathit{use}, \mathit{r\_rule}, \mathit{a\_atom}\} \subseteq \Pi_{\mathcal{T}}$.

Some notation is needed. Given a ground r-program S, let $S_{\mathcal{R}} \subseteq S$ be the set of r-rules in S and let $S_{\mathcal{P}} = S \setminus S_{\mathcal{R}}$ (i.e., the program rules). For any set of ASP rules X, let $\mathit{atoms}(X)$ denote the set of all atoms occurring in X. For any ASP rule γ, let $\mathit{lits}^{+}(\gamma)$ (resp., $\mathit{lits}^{-}(\gamma)$) be the set of atoms occurring positively (resp., negatively) in the body of γ. Moreover, let $\mathit{head}(\gamma)$ be the set of atoms occurring in the head of γ.

The translation $\mathcal{T}$ is defined as follows. We start by univocally naming each r-rule γ in $S_{\mathcal{R}}$. This is done by introducing a fresh constant symbol r_γ (i.e., a constant not appearing elsewhere) and the fact:

$$\mathit{r\_rule}(r_\gamma). \tag{4}$$

Let the r-rule γ be of the form

$$q_0{:}a_0, \ldots, q_h{:}a_h \leftarrow q_{h+1}{:}a_{h+1}, \ldots, q_k{:}a_k,\ L_1, \ldots, L_n. \tag{5}$$

for some $0 \leqslant h \leqslant k$, $n \geqslant 0$, and $(k-h)+n > 0$, with $L_1, \ldots, L_n$ program-literals. For each amount-atom $q_i{:}a_i$ in γ we introduce the ASP fact:

$$\mathit{a\_atom}(r_\gamma, i, q_i, \hat{a_i}). \tag{6}$$

where $\hat{a_i} = a_i$ if $0 \leqslant i \leqslant h$ and $\hat{a_i} = -a_i$ if $h < i \leqslant k$.[5] These facts represent in the ASP translation the amount-atoms occurring in $S_{\mathcal{R}}$. The second argument of $\mathit{a\_atom}$ is needed in the ASP translation to distinguish among different occurrences of identical amount-atoms of the r-rule. Recall, in fact, that multiple copies of the same amount-atom must not be identified, since they correspond to a different amount of resource. Keeping track of multiple copies of amount-atoms reflects, in the translation into ASP code, the use of multisets in defining the semantics of r-programs.

[5]With a little abuse of notation due to the identification of $\mathcal{C}_{\mathcal{R}}$ and $\mathbb{Z}$.

As mentioned, the two disjoints of formula (3) in Definition 3.2 discriminate the two situations in which an r-rule γ is fired or not. These two situations are modeled in ASP through the two rules

$$\begin{array}{l} \mathit{fired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{notfired}(r_\gamma). \\ \mathit{notfired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{fired}(r_\gamma). \end{array} \tag{7}$$

Whenever an r-rule γ is fired, all of the resources mentioned in amount-atoms in γ are consumed/produced. We represent the fact that a certain amount a_i of a resource q_i (due to the amount-atom $q_i{:}a_i$ in γ), is actually used (i.e., consumed or produced), if and only if γ is fired, by introducing the ASP rules

$$\begin{array}{l} \mathit{use}(r_\gamma, i, q_i, \hat{a}_i) \leftarrow \mathit{fired}(r_\gamma), \mathit{a\_atom}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{fired}(r_\gamma) \leftarrow \mathit{use}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{notfired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{use}(r_\gamma, i, q_i, \hat{a}_i). \end{array} \tag{8}$$

for each $i \in \{0, \ldots, k\}$.
Finally, we impose that the firing of γ has to be enabled by the truth of the literals $L_1, \ldots, L_n$, through the ASP rules:[6]

$$\mathit{aux}_{L_i,\gamma} \leftarrow \mathit{not}\ \ \mathit{aux}_{L_i,\gamma}, \overline{L_i}, \mathit{fired}(r_\gamma). \tag{9}$$

for each $j \in \{1, \ldots, n\}$.

Summing up, the translation $\mathcal{T}(\gamma)$ of an r-rule γ of the form (5), is defined as the ASP fragment made of the rules (4) and (6)-(9):

$$\begin{array}{l} \mathit{r\_rule}(r_\gamma). \\ \mathit{a\_atom}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{fired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{notfired}(r_\gamma). \\ \mathit{notfired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{fired}(r_\gamma). \\ \mathit{use}(r_\gamma, i, q_i, \hat{a}_i) \leftarrow \mathit{fired}(r_\gamma), \mathit{a\_atom}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{fired}(r_\gamma) \leftarrow \mathit{use}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{notfired}(r_\gamma) \leftarrow \mathit{not}\ \ \mathit{use}(r_\gamma, i, q_i, \hat{a}_i). \\ \mathit{aux}_{L_i,\gamma} \leftarrow \mathit{not}\ \ \mathit{aux}_{L_i,\gamma}, \overline{L_i}, \mathit{fired}(r_\gamma). \end{array}$$

where i and j range over $\{0, \ldots, k\}$ and $\{1, \ldots, n\}$, respectively.

The above described translation has to be slightly modified in the case of r-rules having a program atom as head. If γ has the form

$$H \leftarrow q_1{:}a_1, \ldots, q_k{:}a_k, L_1, \ldots, L_n. \tag{10}$$

where $1 \leqslant k$, $n \geqslant 0$, $L_1, \ldots, L_n$ are program-literals and H is a program atom, then its translation is constituted by the set of ASP rules (4), (6)-(9), together with the following rule:

$$H \leftarrow \mathit{fired}(r_\gamma), L_1, \ldots, L_n. \tag{11}$$

[6]Here, to simplify the treatment in the following sections, we explicitly introduce the fresh auxiliary atoms $\mathit{aux}_{L_i,\gamma}$, instead of adopting the shorthand notation for ASP constraints mentioned in Section 1. (By using such a notation, (9) would be written as $\leftarrow \overline{L_i}, \mathit{fired}(r_\gamma).$).

We are left with the case in which γ is an r-fact. Facts are treated as r-rules that are supposed to be fired in each circumstance. Hence, if γ is the r-fact

$$q{:}a.$$

then $\mathcal{T}(\gamma)$ is the set of the following ASP facts:

$$\begin{array}{l} \mathit{r\_rule}(r_\gamma). \\ \mathit{a\_atom}(r_\gamma, 0, q, a). \\ \mathit{fired}(r_\gamma). \\ \mathit{use}(r_\gamma, 0, q, a). \end{array} \tag{12}$$

(where, as before, r_γ is a fresh constant symbol univocally associated to γ).

By defining $\mathcal{T}(\gamma) = \{\gamma\}$ for all program rules (namely, if $\gamma \in S \setminus S_{\mathcal{R}}$), the translation $\mathcal{T}(S)$ an r-program S is then defined as the ASP program

$$\mathcal{T}(S) = \bigcup_{\gamma \in S} \mathcal{T}(\gamma).$$

Assume that a set M of atoms is a model of the program $\mathcal{T}(S)$. For some resource symbols q, some of the atoms of the form $\mathit{use}(r_\gamma, i, q, a)$, occurring in $\mathcal{T}(S)$, is true in M. These atoms are intended to represent the amounts of resources involved in fired r-rules. To take into account the constraints on global balance of the allocated amounts, we introduce a condition $\mathit{pos}(M)$ so defined:

$$\mathit{pos}(M) \quad = \quad \forall q \in \tau_{\mathcal{R}} \left(\sum \{\!\!\{ a \mid \mathit{use}(r_\gamma, i, q, a) \in M \}\!\!\} \geqslant 0 \right) \tag{13}$$

This condition reflects the definition of $\mathbb{S}_P$ as introduced in Section 3.

Notice that the condition (13) can be imposed in the ASP encoding of RASP by means of an ASP constraint involving an aggregate sum, as follows:[7]

$$\leftarrow \mathit{sum}\{A : \mathit{use}(\mathit{Rule}, I, Q, A)\} < 0,\ \mathit{res\_symb}(Q).$$

and, for each resource symbol q occurring in $\mathcal{T}(P)$ a fact $\mathit{res\_symb}(q)$. is added to the ASP encoding.

Notice that we are introducing an aggregate literal in a constraint. Hence, no literal in $\mathcal{T}(S)$ is defined depending on such aggregate. This ensures that the resulting ASP program is aggregate stratified. Stable model semantics can be smoothly extended to such class of programs by plainly generalizing the notion of reduct of a program, see [36, 58, 23].

[7]Recall that variables Rule and I occurring in the aggregate literal are to be intended as existentially quantified (whereas the value of Q is determined by the atom $\mathit{res\_symb}(Q)$). Hence, for each q such that $\mathit{res\_symb}(q)$ holds, the aggregate function sum is applied to the multiset of those values A occurring in the facts $\mathit{use}(\mathit{Rule}, I, q, A)$ that are true in the interpretation at hand.

Often, real ASP-solvers require the use of domain predicates to restrain the set of values for these variables, as in:

$$\leftarrow \mathit{sum}\{A : \mathit{use}(\mathit{Rule}, I, Q, A), \mathit{idx}(I), \mathit{val}(A), \mathit{r\_rule}(\mathit{Rule})\} < 0,\ \mathit{res\_symb}(Q).$$

4.2 Completeness and soundness

Some notions (adapted from [42]) are needed. Consider two sets of atoms Z and X and an ASP program G. For a rule $\gamma \in G$ let γ' denote the rule obtained by removing those atoms belonging to Z from the body of γ. More precisely, γ' is such that $\mathit{head}(\gamma') = \mathit{head}(\gamma)$, $\mathit{lits}^+(\gamma') = \mathit{lits}^+(\gamma) \setminus Z$, and $\mathit{lits}^-(\gamma') = \mathit{lits}^-(\gamma) \setminus Z$.

Then, given G, the program $e_Z(G, X)$ is defined as the following set of rules $e_Z(G, X) = \{\gamma' \mid \gamma \in G,\ \mathit{lits}^+(\gamma) \cap Z \subseteq X,\ (\mathit{lits}^-(\gamma) \cap Z) \cap X = \emptyset\}$.

A *splitting set* Z for an ASP program G is a set of atoms such that for each rule $\gamma \in G$, if $\mathit{head}(\gamma) \cap Z \neq \emptyset$ then $\mathit{lits}^+(\gamma) \cup \mathit{lits}^-(\gamma) \subseteq Z$. The set of rules $b_Z(G) = \{\gamma \in G \mid \mathit{atoms}(\gamma) \subseteq Z\}$ is called the *bottom* of G w.r.t. Z.

Let Z be a splitting set for a program G. A *solution* to G w.r.t. Z is a pair $\langle X, Y\rangle$ of sets of atoms such that X is an answer set for $b_Z(G)$ and Y is an answer set for $e_Z(G \setminus b_Z(G), X)$.

Let $S = S_{\mathcal{P}} \cup S_{\mathcal{R}}$ be an r-program. Observe that $\mathit{atoms}(S_{\mathcal{P}}) \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C})$.[8] For simplicity, we assume that all rules in $S_{\mathcal{P}}$ have the form $H \leftarrow\ L_1, \ldots, L_m,\ \mathit{not}\ L_1, \ldots, \mathit{not}\ L_n$. (in particular, $S_{\mathcal{P}}$ does not contain ASP constraints, i.e., rules with empty head).

The ASP program $\mathcal{T}(S)$ can be partitioned as follows

$$\mathcal{T}(S) = S_{\mathcal{P}} \cup \check{S} \cup S_{\mathcal{T}}$$

where $S_{\mathcal{T}}$ is the set of all rules of the forms (4),(6),(7), and (8), while $\check{S}$ is the set of all rules of the forms (9) and (11), originating from the translation.

Let Π_{aux} be the set of all the predicate symbols $\mathit{aux}_{L_i,\gamma}$ introduced by the translation (cf., (9) in page 17). We have that

$$\mathit{atoms}(\check{S}) \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C}) \cup \mathcal{B}(\{\mathit{fired}\} \cup \Pi_{aux}, \{r_\gamma \mid \gamma \in S\})$$

and

$$\mathit{atoms}(S_{\mathcal{T}}) \subseteq \mathcal{B}(\Pi_{\mathcal{T}}, \{r_\gamma \mid \gamma \in S\} \cup \mathbb{Z} \cup \tau_{\mathcal{R}}).$$

We can see that the set

$$U_S = \mathcal{B}(\Pi_{\mathcal{T}}, \{r_\gamma \mid \gamma \in S\} \cup \mathbb{Z} \cup \tau_{\mathcal{R}})$$

is a *splitting set* for $\mathcal{T}(S)$ and the *bottom* of $\mathcal{T}(S)$ relative to U_S is

$$b_{U_S}(\mathcal{T}(S)) = \{\gamma \in \mathcal{T}(S) \mid \mathit{atoms}(\gamma) \subseteq U_S\} = S_{\mathcal{T}}.$$

This is because no atom in U_S occurs as head in rules of $S_{\mathcal{P}} \cup \check{S}$.

At this point, we can establish a relation between the answer sets of an r-program and those of its translation.

Let M be an answer set for $\mathcal{T}(S)$. By the *Splitting Set Theorem* [42], $M = X \cup Y$ for X and Y such that $\langle X, Y\rangle$ is a solution to $\mathcal{T}(S)$ with respect to U_S. Hence, by the definition of a solution, we have that

[8]Recall that $\mathcal{B}(X, Y)$ denotes the collection of all ground atoms built up from predicate symbols in X and terms in Y.

- X is an answer set for $b_{U_S}(\mathcal{T}(S)) = S_{\mathcal{T}}$, and
- Y is an answer set for $e_{U_S}(S_{\mathcal{P}} \cup \check{S}, X)$.

Notice that $Y \subseteq \mathit{atoms}(\mathcal{T}(S)) \setminus U_S \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C}) \cup \mathcal{B}(\{\mathit{fired}\} \cup \Pi_{aux}, \{r_\gamma \mid \gamma \in S\})$ and $X \subseteq \mathit{atoms}(\mathcal{T}(S)) \cap U_S$ encodes the information about which r-rules are fired in M.

Let us consider the r-interpretation $\mathcal{I}_M = \langle Y, \mu_M \rangle$, where μ_M is such that for all $q \in \tau_{\mathcal{R}}$ and $\gamma \in S$, $\mu_M(q)(\gamma) = \{\!\!\{ v \mid \mathit{use}(n_\gamma, i, q, v) \in X \}\!\!\}$.

The following result holds.

Theorem 4.1 *If M is an answer set for $\mathcal{T}(S)$ such that $\mathit{pos}(M)$ holds, then $\mathcal{I}_M$ is an answer set the r-program S.*

Proof. We have to show that $\mathcal{I}_M = \langle Y, \mu_M \rangle$ fulfills the requirements expressed in the two items in Definition 3.2. Recall that, by the Splitting Set Theorem, $M = X \cup Y$, with $\langle X, Y \rangle$ solution to $\mathcal{T}(S)$ with respect to U_S.

- Consider a rule $\gamma \in S$. Two cases are possible:
  - If $\gamma \in S_{\mathcal{P}}$ then $\mu_M(q)(\gamma) = \emptyset$ for all $q \in \tau_{\mathcal{R}}$ and condition (3) of Definition 3.2 is satisfied.
  - If $\gamma \in S_{\mathcal{R}}$ then let $q_0{:}a_0, \ldots, q_k{:}a_k$ be the amount-atoms in γ. There are two possibilities (i.e., γ is fired or not):
    - for all i, $\mathit{use}(n_\gamma, i, q_i, \hat{a}_i) \in X$. Then condition (3) of Definition 3.2 is satisfied because $\mathit{pos}(M)$ holds.
    - for each i there is no atom of the form $\mathit{use}(n_\gamma, i, q_i, \hat{a}_i)$ in X and $\mu_M(q)(\gamma) = \emptyset$ for all $\in \tau_{\mathcal{R}}$. Condition (3) Definition 3.2 is satisfied.
- Recall that Y is an answer set for $e_U(S_{\mathcal{P}} \cup \check{S}, X)$. Observe that the predicate symbols occurring in X do not occur in $S_{\mathcal{P}}$. Then $e_U(S_{\mathcal{P}} \cup \check{S}, X) = S_{\mathcal{P}} \cup e_U(\check{S}, X)$. Moreover, an atom of the form $\mathit{fired}(r_\gamma)$ belongs to X if and only if the r-rule γ produces/consumes the resources described by its amount-atoms (i.e., if and only if γ is fired, and by effect of rules (8), exactly all needed atoms $\mathit{use}(n_\gamma, i, q_i, \hat{a}_i)$ belong to M). On the other hand, U contains all atoms of the form $\mathit{fired}(r_\gamma)$. Let ρ be a rule $\check{S}$. By definition, ρ has the form (9) (or (11)) and originates from the translation of an r-rule γ. If such γ is fired, then $\mathit{fired}(r_\gamma)$ belongs to X and occurs in ρ. Let ρ' be obtained by removing the atom $\mathit{fired}(r_\gamma)$ from ρ. The rule ρ' is in $e_U(\check{S}, X)$ only if $\mathit{lits}^+(\rho) \cap U = \{\mathit{fired}(r_\gamma)\} \subseteq X$. This shows (because of the way μ_M has been defined, see above), that $S_{\mathcal{P}} \cup e_U(\check{S}, X) = \hat{S}$ and condition (2) of Definition 3.2 is satisfied.

$\square$

Theorem 4.1 states the soundness of the ASP embedding of r-programs. We now proceed by showing its completeness.

Let $\mathcal{I} = \langle I, \mu \rangle$ be an answer set of an r-program S, For each $\gamma \in S_{\mathcal{R}}$ one and only one of the disjuncts in condition (2) of Definition 3.2 is satisfied. We introduce these sets of atoms:

$$
\begin{array}{rcl}
S_{(1)} & = & \{r\_rule(r_\gamma) \mid \gamma \in S_{\mathcal{R}}\} \\
S_{(3)} & = & \{a\_atom(r_\gamma, i, q_i, \hat{a}_i) \mid q_i{:}a_i \text{ is an amount-atom in } \gamma \in S_{\mathcal{R}}\} \\
S_{(4)} & = & \{fired(r_\gamma) \mid \gamma \in S_{\mathcal{R}} \text{ and } \forall\, q \in \tau_{\mathcal{R}}\ (\mu(q)(\gamma) = Alloc(q, \gamma))\} \cup \\
 & & \{notfired(r_\gamma) \mid \gamma \in S_{\mathcal{R}} \text{ and } \forall\, q \in \tau_{\mathcal{R}}\ (\mu(q)(\gamma) = \emptyset)\} \\
S_{(5)} & = & \{use(r_\gamma, i, q_i, \hat{a}_i) \mid q_i{:}a_i \text{ is an amount-atom in } \gamma \\
 & & \text{ and } \forall\, q \in \tau_{\mathcal{R}}\ (\mu(q)(\gamma) = Alloc(q, \gamma))\} \\
X_S & = & S_{(1)} \cup S_{(3)} \cup S_{(4)} \cup S_{(5)}
\end{array}
$$

The following result states the completeness of the transformation.

Theorem 4.2 *Let $\mathcal{I} = \langle I, \mu \rangle$ be an answer set of an r-program S and X_S defined as described above. Then, $M = X_S \cup I$ is an answer set for $\mathcal{T}(S)$ and $pos(M)$ holds.*

Proof. Observe that I is an answer set for $\hat{S}$ and that X_S is an answer set for $b_{U_S}(\mathcal{T}(S))$. By an argument similar to the one applied in the proof of Theorem 4.1, we obtain that $e_{U_S}(\mathcal{T}(S) \backslash b_{U_S}(\mathcal{T}(S)), X_S) = e_{U_S}(S_{\mathcal{P}} \cup \check{S}, X_S) = \check{S}$. Hence, $\langle X_S, I \rangle$ is a solution of $\mathcal{T}(S)$ w.r.t. U_S. It follows, by the Splitting Set Theorem, that $X_S \cup I$ is an answer set for $\mathcal{T}(S)$. We can conclude the proof by observing that, by Definition 3.2 and by the definitions of $S_{(5)}$ and $\mathbb{S}_P$ (cf., (1) at page 10), $pos(M)$ holds. □

4.3 An inference engine for RASP

In Section 4.1 we outlined a translation from r-rules of an r-program S into fragments of ASP code. Something better could be done by observing that part of the rules in $\mathcal{T}(S)$, namely those of the forms (7) and (8), can be factorized by exploiting variables. This allows us to design the core of an ASP-based inference engine capable of reasoning on resource allocations. As we will see, such an engine will be successively extended in Section 8, in order to deal with more expressive constructs (to be introduced in Section 6).

The following code imposes correct usage of resources in firing each rule.

$$
\begin{array}{l}
fired(Rule) \leftarrow not\ \ notfired(Rule),\ r\_rule(Rule). \\
notfired(Rule) \leftarrow not\ \ fired(Rule),\ r\_rule(Rule). \\
use(Rule, I, Res, Amount) \leftarrow fired(Rule), \\
\qquad\qquad\qquad\qquad\qquad\qquad a\_atom(Rule, I, Res, Amount). \\
fired(Rule) \leftarrow use(Rule, I, Res, Amount). \\
notfired(Rule) \leftarrow not\ \ use(Rule, I, Res, Amount).
\end{array}
$$

The balance for each resource is evaluated by the following fragment of code. Observe the use of an ASP constraint involving an aggregate function to evaluate sums and to impose the condition (13):[9]

$$\begin{array}{l} \mathit{res\_symb}(\mathit{Res}) \leftarrow \mathit{a\_atom}(\mathit{Rule}, I, \mathit{Res}, \mathit{Amount}). \\ \leftarrow \mathit{sum}\{A : \mathit{use}(\mathit{Rule}, I, Q, A)\} < 0,\ \mathit{res\_symb}(Q). \end{array}$$

An ASP-based solver for RASP would act as follows: first each r-rule of the RASP program is translated as previously explained in Section 4.1 (recall that program rules are left unchanged); then, the rendering of all r-rules must be joined with the above ASP program that acts as an inference engine and performs the concrete reasoning activity on resource allocations; finally, the answer sets (if any) of the obtained ASP program are calculated by means of a standard ASP solver [5]. From each answer set M so computed, an answer set $\mathcal{I}_M$ of the original r-program can be extracted as described in Section 4.2.

5 Complexity

In order to determine the complexity of deciding about the existence of answer sets of a RASP program (and about the existence of answer sets containing a certain atom A), we will reduce the question to that of finding the answer sets (in the usual answer set semantics) of an ASP version of the given RASP program P, that we will call *adapted program*, and of establishing which of these answer sets are *admissible*, i.e., satisfy the constraint that resources can be consumed only if they have been actually produced. Deciding the existence of an answer set has been proved NP-complete in [45] and the same for deciding whether an atom is a member of some answer set (proved in [46]). In this section we will prove that the complexity of RASP remains the same as for ASP, by proving that the adapted program is not much larger than given program P, that admissibility of its answer sets can be checked in polynomial time and that a correspondence between admissible answer sets of the adapted program and answer sets of P can be established.

The two disjoints of formula (3) in Definition 3.2 show the two possible ways a rule γ can be treated: a) the rule γ is not fired, so null amounts are allocated to all its amount-symbols; b) the rule γ is actually fired and all needed amounts are allocated. Consider again the program P_1 in the introduction, that for the sake of clarity we report below.

$$\begin{array}{ll} \gamma_1 : & \mathit{have\_cake} \leftarrow \mathit{egg}{:}3,\ \mathit{flour}{:}3,\ \mathit{sugar}{:}3. \\ \gamma_2 : & \mathit{have\_ice\_cream} \leftarrow \mathit{egg}{:}3,\ \mathit{sugar}{:}2,\ \mathit{milk}{:}2. \\ \gamma_3 : & \mathit{egg}{:}4. \\ \gamma_4 : & \mathit{flour}{:}8. \\ \gamma_5 : & \mathit{sugar}{:}6. \\ \gamma_6 : & \mathit{milk}{:}3. \end{array}$$

[9]Alternatively, it would be possible to exploit *cardinality* and *weight constraints* [51] to surrogate the aggregate literals.

As only four eggs are available, it cannot be the case that both γ_1 and γ_2 are fired. In fact, only one of the two can be fired (in alternative) while leaving one egg unused.

The first problem with the above example is that the same atom *egg*:3 occurs in the body of two rules, which might produce ambiguity in the resource allocation. Therefore, it may be convenient to rewrite the program as follows:

γ_1 :	*have_cake* $\leftarrow$ *egg*^{1}:3, *flour*^{1}:3, *sugar*^{1}:3.
γ_2 :	*have_ice_cream* $\leftarrow$ *egg*^{2}:3, *sugar*^{2}:2, *milk*^{2}:2.
γ_3 :	*egg*:4.
γ_4 :	*flour*:8.
γ_5 :	*sugar*:6.
γ_6 :	*milk*:3.

I.e., resource symbols occurring in the body of rules are *standardized apart*, by substituting them by means of fresh resource symbols. This kind of elaboration is formalized in the following definition:

Definition 5.1 *Let P be a (ground) r-program, and let $\gamma_1, ..., \gamma_k$ be the rules in P containing amount-atoms in their body. The standardized-apart version P_s of P is obtained from P by renaming each amount-atom q:a in the body of γ_j, $j \leqslant k$ as q^j:a. The q^j's are called the standardized apart versions of q.*

Referring to the standardized-apart version of P_1, the resource allocations that may correspond to answer sets can be intuitively represented by means of the following two rewritings. In the first one, resources are allocated to the first r-rule by splitting each fact into two: one corresponding to the allocated quantity of resource, the other one to what is left.

γ_1 :	*have_cake* $\leftarrow$ *egg*^{1}:3, *flour*^{1}:3, *sugar*^{1}:3.
γ_2 :	*have_ice_cream* $\leftarrow$ *egg*^{2}:3, *sugar*^{2}:2, *milk*^{2}:2.
γ_3 :	*egg*^{1}:3.
γ_3' :	*egg*:1.
γ_4 :	*flour*^{1}:3.
γ_4' :	*flour*:5.
γ_5 :	*sugar*^{1}:3.
γ_5 :	*sugar*:3.
γ_6 :	*milk*:3.

By considering amount-atoms as plain atoms, we can now compute the (unique) answer set of this program, which is

$$\{\mathit{milk}{:}3,\ \mathit{sugar}{:}3,\ \mathit{sugar}^1{:}3,\ \mathit{flour}{:}5,\ \mathit{flour}^1{:}3,\ \mathit{egg}{:}1,\ \mathit{egg}^1{:}3,\ \mathit{have\_cake}\}.$$

This answer set is *admissible* as for each resource the total quantity which has been used does not exceed the available quantity. It can be reduced so as to refer to the language of P_1 by eliminating the standardized apart versions of amount-atoms. Thus, for each resource that has been to some extent consumed the answer set reports the remaining quantity. In particular, we obtain *milk*:3, *sugar*:3, *flour*:5, and *egg*:1, as unused amounts of resources. Analogously, the second version allocates the resources to the second rule.

$$
\begin{array}{ll}
\gamma_1 : & \mathit{have\_cake} \leftarrow \mathit{egg}^1{:}3,\ \mathit{flour}^1{:}3,\ \mathit{sugar}^1{:}3. \\
\gamma_2 : & \mathit{have\_ice\_cream} \leftarrow \mathit{egg}^2{:}3,\ \mathit{sugar}^2{:}2,\ \mathit{milk}^2{:}2. \\
\gamma_3 : & \mathit{egg}^2{:}3. \\
\gamma_3' : & \mathit{egg}{:}1. \\
\gamma_4 : & \mathit{flour}^1{:}3. \\
\gamma_4' : & \mathit{flour}{:}5. \\
\gamma_5 : & \mathit{sugar}^2{:}2. \\
\gamma_5 : & \mathit{sugar}{:}4. \\
\gamma_6 : & \mathit{milk}^2{:}2. \\
\gamma_6 : & \mathit{milk}{:}1.
\end{array}
$$

In this case, the answer set is

$$\{\mathit{milk}{:}1,\ \mathit{milk}^2{:}2,\ \mathit{sugar}{:}4,\ \mathit{sugar}^2{:}2,\ \mathit{flour}{:}8,\ \mathit{egg}{:}1,\ \mathit{egg}^2{:}3,\ \mathit{have\_ice\_cream}\}$$

which determines these unused amounts: *milk*:1, *sugar*:4, *flour*:8, and *egg*:1.

By considering ground amount-atoms as plain atoms, we can now "simulate" the rewriting shown above by adding, for each standardized-apart version of an amount-atom, an even cycle which simulates this resource to be allocated to the r-rule where it occurs or not. In order to save space, we will add such an even cycle only if that resource can potentially be produced by some other r-rule.

Definition 5.2 *Let P be a (ground) r-program, and let P_s be the standardized-apart version of P. The* adapted program *P' for P is obtained by adding to P_s for each $q^j{:}a$ occurring in the body of some r-rule of P_s and such that $q{:}b$ (for some b) occurs in the head of some r-rule of P_s the following pair of rules (where $no\_q^j{:}a$ is a fresh atom):*

$$
\begin{array}{l}
q^j{:}a \leftarrow \ not\ \ no\_q^j{:}a. \\
not\ \ no\_q^j{:}a \leftarrow \ not\ \ q^j{:}a.
\end{array}
$$

Theorem 5.1 *Let P be a (ground) r-program, and let P_s be the standardized-apart version of P. The size of adapted program P' for P grows linearly with respect to the size of P.*

Proof. Notice that P and P_s have the same size, and the same rules (apart from the renaming). Then, for each r-rule in P_s, P' contains at most $2A_B$ new rules, where A_B is the number of resource symbols which occur both in the body of an r-rule and in the head of some other r-rule of P_s (and then of P). The worst case corresponds to a situation where all rules of P are r-rules and resource symbols occurring in r-rule heads are distinct and all of them also occur in some r-rule body. In this case, A_B is equal to the number of the rules of P. Therefore, P'

may in the worst case be composed of $A_B + 2A_B = 3A_B$ rules, i.e., three times the number of rules of P. $\Box$

In the rest of this section, with some abuse of notation, for the sake of conciseness when mentioning "RASP answer sets" of an r-program P we mean answer sets in terms of the RASP semantics, while when mentioning "classical answer sets" of the adapted program P' we refer to answer sets in the usual answer set semantics obtained by considering amount-atoms as plain atoms.

Below we state that a classical answer set M of the adapted program is *admissible* if the sum of the resources that have been used does not exceed the sum of the resources that were available. Notice that for each resource predicate q, the available quantity t_a is obtained by summing all amounts of atoms of the form $q{:}a$ (that in the adapted program are found in the head of rules), while the total consumed quantity is obtained by summing all amounts of their standardized apart versions (that in the adapted program are found in the body of rules).

Definition 5.3 *Let P be a (ground) r-program and P' the adapted program P' for P. A classical answer set M of P' is* admissible *if*

$$\forall q \in \tau_{\mathcal{R}} \ \left(\sum \{\!\!\{ a \mid q{:}a \in M \}\!\!\}\right) - \left(\sum \{\!\!\{ a \mid q^j{:}a \in M \, for\, some\, j \}\!\!\}\right) \geqslant 0$$

I.e., if the resource balance relative to M *is non-negative.*

Theorem 5.2 *Let P be a (ground) r-program and P' the adapted program P' for P. Given a set M of atoms, checking whether M is an admissible answer set of P' can be done in polynomial time.*

In fact, as checking whether M is an answer set of P' can be done in polynomial time and implies examining all atoms in P': then, the resource balance can be computed and checked with the same time complexity.

We now have to formally state the relationship between RASP answer sets of program P and classical answer sets of the admissible program P' for P. As seen in Definition 3.1 and Definition 3.2, an interpretation and an answer set of a RASP program is defined as a couple $\langle I, \mu \rangle$ where I takes care of program atoms, and $\mu : \tau_{\mathcal{R}} \rightarrow \mathbb{S}_P$ is a function which assigns quantities to all occurrences of resource symbols.

Therefore, given an admissible answer set M of the adapted program P' for RASP ground program P, we have to identify in M the two components. The particular set I obtained from M is identified as the subset of M including the program-atoms only, obtained by removing from M all the amount-atoms.

Definition 5.4 *Let M be an admissible answer set of the adapted program P' for RASP ground program P. Let H be the set of amount-atoms occurring in P'. We let $\mathcal{P}(M) = M \setminus H$.*

The particular quantities-assignment function obtained from M is identified by collecting the quantities associated to amount-atoms occurring in rules which are fired in M, i.e., such that the head is in M and the body literals are satisfied in M. By the definition of $\mathbb{S}_P$ (cf., (1), page 10), the global sum (considering the entire P) of all the quantities assigned to a resource symbol must be non-negative: notice that this is ensured by the definition of admissible answer set.

Definition 5.5 *Let M be a classical answer set of an admissible adapted program P' for RASP ground program P. Let $\gamma_1, ..., \gamma_k$ be the r-rules in P' which are satisfied in M (i.e., all literals in both their head and their body are true w.r.t. M). We construct $\nu(M) : \tau_{\mathcal{R}} \rightarrow \mathbb{S}_P$ as follows: for each resource symbol q and $1 \leqslant r \leqslant k$ we let $\nu(M)(q)(\gamma_r) = \{\!\!\{ \kappa(a), -\kappa(a_1), \ldots, -\kappa(a_s) \}\!\!\}$ where $q{:}a$ is the occurrence of q in the head of γ_r and any of the $q^r{:}a_w$'s, $w \leqslant s$ is the s-th occurrence of one of its standardized-apart versions in the body, with and $s \geqslant 0$. For all other r-rules, for any resource symbol q we let $\nu(M)(q)(\gamma_v) = \emptyset$*

In the above definition of $\nu(M)(q)(\gamma_r)$, $\kappa(a)$ is (possibly) missing if q does not occur in the head of γ_r, while the $-\kappa(a_w)$'s are (possibly) missing if there is no occurrence of a standardized apart version of q in the body of γ_r.

We are now able to state the desired result.

Theorem 5.3 *$\langle I, \mu \rangle$ is a RASP answer set of a ground r-program P iff $I = \mathcal{P}(M)$ and $\mu = \nu(M)$ where M is an admissible answer set of the adapted program P' for P.*

Proof. Let M be an admissible answer set of the adapted program P' for P. As M is a classical answer set of P', the first condition of the definition of a RASP answer set given in Definition 3.2 is fulfilled by the construction of $\mu = \nu(M)$ where for any r-rule γ_r $\nu(M)(q)(\gamma_r)$ is non-empty only if this r-rule is satisfied w.r.t. M. Then, a rule which is satisfied w.r.t. M will be fired w.r.t. I. As M is a classical answer set of P', it satisfies all the rules in $\widehat{\gamma}$ by definition, as on the one hand it satisfies all literals occurring in r-rules that are satisfied, i.e., fired, and on the other hand it satisfies all program rules in P', which are the same as in P. Therefore, the second condition of the definition of a RASP answer set given in Definition 3.2 is also fulfilled and hence $\langle I, \mu \rangle$ is a RASP answer set of P.

Let $\langle I, \mu \rangle$ be a RASP answer set of a ground r-program P. Notice that, by the definition of a RASP answer set, μ will allocate to each rule γ in P either the empty set (i.e., no resources) or exactly the required quantities (the ones that occur in the rule, as P here is a ground program. By the second item of Definition 3.2, I is an answer set of the subprogram of P including program rules only, and satisfies (as expressed in the definition of $\widehat{\gamma}$) the program literals of those r-rules that are fired, i.e., those where μ allocates the required quantities. We can now construct M by first letting $M = I$. Then, we add to M all amount-atoms obtained by reverting μ. I.e., for each fired rule γ_j in P with an amount-atom in the head, we add to M this amount-atom $q{:}a$ and for each fired rule γ_j in P with an amount-atom in the body, we add to M the standardized apart amount-atom $q^j{:}a$. Then, by construction, M is an answer set of the subprogram of P' consisting of the program rules and it satisfies the program literals occurring in r-rules which are fired in the given RASP answer set. It also includes the (amount) atom in their head and all the amount-atoms occurring in their body. Therefore, M is a classical answer set of P'. □

From the above results we can conclude the following, similarly to plain ASP:

Theorem 5.4 *The problem of deciding whether there exists an answer set of a ground r-program P is NP-complete.*

In fact, one has to guess a set of atoms among those occurring in the adapted program P' for P, and then check whether it is an admissible answer set for P'. (Notice that, by definition, an ASP program is a RASP program.) Consequently, again similarly to plain ASP, we can state the following:

Theorem 5.5 *Given atom A occurring in an r-program P, the problem of deciding whether there exists an answer set of P containing A is NP-complete.*

6 Extending the basic framework

6.1 Multiple firing of resource-rules.

Let us consider an r-rule that produces a resource (in a certain amount). If a sufficient quantity of the resources required by that r-rule are available, the resulting amount of resource can be in principle produced several times. We introduce the possibility of declaratively specifying whether an r-rule can (or must) be "fired" more than once, and how many times.

To this aim, let us enrich the RASP language.

Definition 6.1 *Multiply-fireable r-rules γ are of the form*

$$\mathit{Idx}: \quad H \leftarrow B_1, \ldots, B_k. \tag{14}$$

where H and the B_i are as before (cf., Definition 2.1), and Idx is of the form $[N_{1,1}\text{-}N_{1,2}, \ldots, N_{h,1}\text{-}N_{h,2}]$, with $h \geqslant 1$, and each $N_{j,\ell}$ is either a variable or a positive integer number.

Intuitively, in any ground instance of (14) when all the $N_{j,\ell}$s are integers, *Idx* denotes the union of h (possibly void) intervals in $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. It is intended to restrain the (finite) number of times the r-rule can be used: such a number must be in *Idx* or the r-rule cannot be used at all. Definition 6.1 admits that each $N_{j,\ell}$ is a variable for non-ground r-rules. Then, after grounding, each $N_{j,\ell}$ has to be instantiated to a positive integer. Notice that, if in an instance of (14) *Idx* becomes instantiated to [1-1], we obtain an r-rule as introduced in Section 2. (As usual, a pair $N_{j,1}\text{-}N_{j,2}$ denotes a void interval whenever, possibly after instantiation, either $N_{j,1}$ or $N_{j,2}$ is not a positive integer, or it holds that $N_{j,2} < N_{j,1}$.)

Example 6.1 *The following r-rule γ_1'' specifies that we can produce a number n of cakes such that either $2 \leqslant n \leqslant 4$ or $7 \leqslant n \leqslant 7$ (or no cakes, if the r-rule is not fired at all):*

$$\gamma_1'' : \qquad [2\text{-}4, 7\text{-}7]\colon\ \mathit{cake}{:}1 \ \leftarrow\ \mathit{egg}{:}3, \mathit{flour}{:}3, \mathit{sugar}{:}3.$$

The rule γ_1'' is an "iterable" version of rule γ_1' of Example 3.5. Each firing of a rule may consume/produce resources in the indicated amounts. Clearly, repeated firings can occur provided enough resources are available to sustain all firings. In the example we can produce more cakes, but only if sufficient quantities of the ingredients are available (for instance, two firings produce two cakes and consume 6 eggs, etc.).

A piece of notation: we will not write the list *Idx* when all $N_{j,\ell}$ s are intended to be the constant 1 (meaning that at most one use of the rule is admitted). Without loss of generality, for simplicity, in what follows we always assume $h = 1$ in rules of the form (14). The treatment of the general case is essentially the same.

How many firings will be actually performed? We will introduce in Section 7 a manner of controlling the number of firings by imposing preferences/policies in resource consumption. As regards the previous example, we might wish to produce as many cakes as possible. Or else, we might prefer to produce only the minimum number of cakes that we are actually forced to prepare because they are explicitly required (for instance, by r-rules in other parts of the program). In absence of a specific policy, a RASP-solver will propose all admissible possibilities as alternative answer sets of the r-program (i.e., corresponding to different resource allocations).

Remark 6.1 *Notice that, similarly to what was done in Example 6.1 for γ_1', it is possible to introduce an "iterable" version of the rule γ_1, as follows:*

$$\gamma_1''' : \qquad [1\text{-}3]\colon\ \mathit{have\_cake} \ \leftarrow\ \mathit{egg}{:}3, \mathit{flour}{:}3, \mathit{sugar}{:}3.$$

In this case, since the head of the rule does not contain amount-atoms, multiple firings will result in making true the program atom have_cake —just as a single firing would do— and in repeatedly consuming more resources than needed.

To deal with multiple fireable rules in the semantics, it suffices to modify the definition of r-interpretation as follows (cf., Definition 3.1 and 3.2): an r-interpretation for a program P is a triple

$$\mathcal{I} = \langle I, \mu, \xi \rangle$$

where I and μ are as before, while ξ is a mapping $\xi : P \to \mathbb{N}$ that assigns to each (fired) rule the number of its firings. By little abuse of notation, we consider ξ to be defined also for program-rules and r-facts. For these kinds of r-rules we assume the interval $[N_1\text{-}N_2] = [1\text{-}1]$ as implicitly specified in the rule definition, as constraint on the number of firings.

Clearly, a necessary condition for an r-interpretation to be an answer set, is that for each fired rule $\gamma \in P$ we have $N_{1,1} \leqslant \xi(\gamma) \leqslant N_{1,2}$.

In view of this notion of r-interpretation, the definition of answer set is obtained from Definition 3.2 by replacing condition (3) with the following one:[10]

$$\Big(\forall\, q \in \tau_{\mathcal{R}}\big(\mu(q)(\gamma) = \emptyset\big)\Big) \quad \vee \quad \Big(\forall\, q \in \tau_{\mathcal{R}}\big(\mu(q)(\gamma) = \xi(\gamma){\cdot}\mathit{Alloc}(q,\gamma)\big) \;\wedge\; \big(N_{1,1} \leqslant \xi(\gamma) \leqslant N_{1,2}\big)\Big).$$

where, as for (3) in Definition 3.2, the two disjunct correspond to the cases: a) γ is not fired; and b) γ is fired $\xi(\gamma)$ times. In the latter case, for each resource symbol q, each firing uses the amounts specified by $\mathit{Alloc}(q,\gamma)$.

Let us consider a sligthly modified version of the r-program P_{ab} of Example 3.4.

Example 6.2 *Let P'_{ab} be constituted of these r-rules:*

$$\begin{array}{lrl} \gamma'_a: & \mathit{Idx}_a: & q{:}2 \leftarrow q{:}1. \\ \gamma'_b: & \mathit{Idx}_b: & q{:}1 \leftarrow q{:}2. \end{array}$$

where Idx_a and Idx_b denote sets of integers as explained earlier. I.e., the r-rule γ'_a can be repeatedly fired any (finite) number n_a of times, provided $n_a \in \mathit{Idx}_a \cup \{0\}$ (and similarly for γ'_b, provided some further amount of the resource q is available, cf., Example 3.4). Hence, the answer sets of P'_{ab} are of the form $\mathcal{I} = \langle I, \mu, \xi\rangle$ with $I = \emptyset$, $\mu(q)(\gamma_a) = \xi(\gamma'_a){\cdot}\{\!\!\{2,-1\}\!\!\}$, $\mu(q)(\gamma_b) = \xi(\gamma'_b){\cdot}\{\!\!\{1,-2\}\!\!\}$, $\xi(\gamma'_a) \in \mathit{Idx}_a \cup \{0\}$, $\xi(\gamma'_b) \in \mathit{Idx}_b \cup \{0\}$, and such that $\xi(\gamma'_a) - \xi(\gamma'_b) \geqslant 0$. The overall balance for the resource q is $\varphi(q) = (2-1)\xi(\gamma'_a) + (1-2)\xi(\gamma'_b)$.

6.2 Negative amount-atoms.

A *negative amount-atom* is of the form $\text{-}\,q{:}a$, where $q{:}a$ is an amount-atom. Negative amount-atoms can occur both in the head and in the body of an r-rule. When such a rule is fired, a negative amount-atom occurring in the body implies the consumption of $-\kappa(a)$ units of resource q. In general, as regards the global resource balance, a negative amount-atom $\text{-}\,q{:}a$ of the body (resp., head) can be interpreted as a (positive) amount-atom $q{:}a$ occurring in the head (resp., body). This actually means that an amount $\kappa(a)$ of resource q is produced instead of consumed. From the conceptual point of view, negative amount-atoms could help in emphasizing/expressing that some resources that are not intended to be the main product of a production process, but are seen as a secondary (possibly undesirable) effect of it. Hence, an intuitive reading can be that the amount $\kappa(a)$ of resource q is a *byproduct* of the application of the rule. A dual argument applies to negative amount-atoms in head of r-rules.

The easiest manner of interpreting negative amount-atoms is through the inverse function on $\mathbb{Z}$. Formally, it suffices to modify the definition of $\mathit{Alloc}(q,\gamma)$

[10]Given a multiset S and $n \in \mathbb{N}$, we are denoting by $n{\cdot}S$ the multiset $\{\!\!\{n{\cdot}s \mid s \in S\}\!\!\}$.

(cf., (2) in Definition 3.2) as follows:

$$\begin{array}{rcl} \mathit{Alloc}(q,\gamma) & = & \{\!\!\{ v \in \mathbb{Z} \mid v = -\kappa(a) \text{ for } q{:}a \text{ in the body of } \gamma \\ & & \qquad\qquad \text{or } v = \kappa(a) \text{ for } q{:}a \text{ in the head of } \gamma \\ & & \qquad\qquad \text{or } v = -\kappa(a) \text{ for } \texttt{-}\, q{:}a \text{ in the head of } \gamma \\ & & \qquad\qquad \text{or } v = \kappa(a) \text{ for } \texttt{-}\, q{:}a \text{ in the body of } \gamma \}\!\!\} \end{array}$$

This use of the inverse function imposes a duality between consumption and production of resources.

Another use of negative amount-atoms explicitly relies on the dual roles played by consumed and produced amounts. There might be situations in which an amount of resource may be either produced or consumed, depending on some condition to be evaluated, as the following simple toy-example shows.

Example 6.3 *Immagine a (very simplified) situation in which a trader believes that a stock's price will increase. Hence (s)he decides to buy the right to purchase the stock rather than simply buying the stock now. In other words, (s)he buys a* stock option *instead of real shares. In this way, (s)he has no obligation to buy the stock, but only the right to do so (until the expiration date of the option). In particular, if the stock's price at expiration date is above the exercise price by more than the premium paid (the price of the operation), then (s)he will profit. Otherwise, if the stock's price at expiration is lower than the exercise price, (s)he will let the call-contract expire worthless, and only lose the amount of the premium. The following r-rule encodes a situation in which* 100 *shares are involved (we omit the definition of the predicates stock_price, exercise_price, and premium, and assume all prices being expressed as integer numbers):*

$$\begin{array}{rcl} \mathit{dollar}{:}D & \leftarrow & \mathit{stock\_price}(S), \mathit{exercise\_price}(E), \mathit{premium}(R), \\ & & \mathit{share}{:}100,\ D = 100 * \mathit{max}(0, E - S) - R. \end{array}$$

Note that the number of dollars the trader obtains might be negative, representing a loss instead of a real profit.

6.3 Constraints on global resource balances.

In the framework introduced so far, one can easily express constraints on the consumption/production of amounts of resources. For instance, assume that an amount-atom $q{:}X$ (with $X \in \mathcal{V}$) occurs in an r-rule γ and that the firing of γ should be avoided for particular amounts of q. This can be ensured by using program-literals in the body of γ that inhibit the firing of γ whenever X assumes such values (as a consequence of grounding). Constraints of this kind are, in a sense, *local* to the specific rule being considered.

It is easy to extend the basic framework to allow the assertion of constraints on the global resource balance. We describe now two simple forms of such *global* constraints. The (qualified) fact $\mathit{leaveAtLeast}{:}q{:}a$ where $q{:}a$ is a ground amount-atom, can be used to weed out all models in which an amount of q smaller than $\kappa(a)$ is left unused. (Notice that, for a given resource-symbol q, it makes little sense to impose more than one constraint of this kind. It suffices, in fact, to consider the most restrictive one.) A similar constraint is imposed through the fact *leaveAtMost* : q:a. In this case, in each answer set no more than $\kappa(a)$ units of q can be left unused.

For a resource q, in order to fulfill the constraints *leaveAtLeast*:q:a_q and *leaveAtMost* : q:b_q, we filter the potential r-interpretations for an r-program P by refining the definition of the mapping μ (cf., (1), page 10). In particular, we restrain its codomain $\mathbb{S}_P$ as follows:

$$\mu \ : \ \tau_{\mathcal{R}} \ \rightarrow \ \Big\{F \in \big(\mathcal{FM}(\mathbb{Z})\big)^P \mid \kappa(a_q) \leqslant \sum \big(\bigcup_{\gamma \in P} F(\gamma)\big) \leqslant \kappa(b_q)\Big\}$$

Similarly, other constraint imposing different requirements on the global resource balance can be dealt with.

6.4 Costs for rule firings.

In modeling processes that consume/produce resources, it is sometimes useful to associate costs to rule firings. In RASP this can be easily done by introducing specific resource-symbols in the r-rules. Nevertheless, we propose here a simple extension of the language that avoids the use of such an '*ad hoc*' approach. If γ is an r-rule and C is an amount-symbol, then a writing of the form:

$$C : \ \gamma$$

states that C represents the cost of each firing of the rule γ. If C is a variable, then it has to occur in a program atom of the body of γ and after grounding it has to be instantiated to an amount-symbol.

Example 6.4 *The "cost" can be understood in different ways. In the following fragment of r-program each firing of the first rule has a cost consisting in the amount of time that we spend in the preparation of one cake. Such a cost is determined by some other rules. In this case, the atoms of the form* $needed\_time(Num, T)$ *determine the time needed to make a cake when Num persons help the cook. The second rule gives the number of helpers in the kitchen, by using an aggregate literal.*

Provided enough resources are available, the firing of the first rule can be iterated from one to three times. Consequently, we have to spend some time (how much depending on the number of helpers) for each cake we make.

$$\begin{array}{ll}
\mathit{Time}{:}[1\text{-}3]: & \mathit{cake}{:}1 \leftarrow \mathit{egg}{:}3, \mathit{flour}{:}3, \mathit{sugar}{:}3, \\
& \qquad\qquad \mathit{needed\_time}(\mathit{Helpers}, \mathit{Time}), \\
& \qquad\qquad \mathit{helpers}(\mathit{Helpers}). \\
& \mathit{helpers}(N) \leftarrow N = \mathit{count}\{\mathit{Person} : \mathit{at\_home}(\mathit{Person})\}, \\
& \qquad\qquad\qquad \mathit{number}(N). \\
& \mathit{needed\_time}(0, 45). \quad \textit{\% no one helps, I cook by myself!} \\
& \mathit{needed\_time}(N, \mathit{Time}) \leftarrow N > 0, N < 3, \\
& \qquad\qquad\qquad\qquad\qquad \mathit{Time} = 45 - (N * 10). \\
& \mathit{needed\_time}(N, 25) \leftarrow N > 2. \quad \textit{\% too many cooks do not help!} \\
& \mathit{number}(0). \ \ldots \ \mathit{number}(5). \\
& \mathit{at\_home}(X) \leftarrow \ldots
\end{array}$$

The following variant of the first rule illustrates how to enable/disable the firing of a rule, depending on the cost of its (single) firing.

$$\begin{array}{ll}
\mathit{Time}{:}[1\text{-}3]: & \mathit{cake}{:}1 \leftarrow \mathit{egg}{:}3, \mathit{flour}{:}3, \mathit{sugar}{:}3, \\
& \qquad\qquad \mathit{have\_free\_time}(\mathit{Time}), \\
& \qquad\qquad \mathit{needed\_time}(\mathit{Helpers}, \mathit{Time}), \\
& \qquad\qquad \mathit{helpers}(\mathit{Helpers}).
\end{array}$$

In this case the truth of a fact of the form $\mathit{have\_free\_time}(\cdot)$ *(to be defined in other parts of the program) establishes whether the cost can be paid or not.*

Depending on which rules are fired, we can associate costs to answer sets. Hence, it is possible to design specific cost-based criteria to impose preferences among answer sets. Selection of preferred solutions could, for instance, be implemented by exploiting optimization features such as those offered, for example, by smodels [55].

There is a relationship with the approach of [53] where, in the presence of explicit negation, there may be conflicting rules, and different answer sets correspond to one rule defeating the other one or vice versa. Defeating a rule may have a cost, and "preferred" answer sets are those which are minimal w.r.t. the sum of all costs paid. The sense is that defeating more "critical" rules has a higher cost. Then, while in our approach one pays a cost for *firing* a rule, in [53] one pays a cost for firing a rule while defeating, i.e. *not firing*, another one. Therefore, the two approaches are, in a sense, complementary: for each answer set, one might consider the couple of both cost evaluations and choose preferred ones according to some criterion, possibly tailored to the application at hand.

7 Budget Policies

As we have seen, an answer set for an r-program is constituted of a set of (true) program-atoms I and a mapping μ establishing the amounts for each consumption/production of resources. There might be the case that different answer sets agree on I but involve different mappings. This is, fixed set I, more than one mapping might satisfy the conditions expressed in Definition 3.2. In such a situation, it could be valuable to apply some criteria to filter the collection of all possible answer sets. In particular, different policies could be adopted in selecting a specific strategy in firing the r-rules (i.e., for the consumption/production of resources). We identify three basic possibilities, among many (for the sake of simplicity, let us focus on ground programs):

Thrifty. An r-rule γ is fired only if this is forced. (For instance, because the truth of a program-atom occurring as head of γ is required by effect of other parts of the program.)

Prodigal. Whenever an r-rule γ can be fired, it must be fired.

Optional. This is the most general policy and it admits all the solutions obtained by the previous policies. Different resource allocations can enable the firing of different rules, possibly in antithetic manners. Moreover, enabled rules might be not necessarily fired.

Example 7.1 *Recall the program P_2 of Example 3.5, and substitute the r-fact γ_3 with the r-fact egg:7. Four potential solutions are: make none, one, or both desserts. The Thrifty policy selects the answer set in which no dessert is made, since no firing of rules is forced. The Prodigal policy selects the answer set in which both desserts are made (this is possible because enough ingredients are available for both recipes). Using the Optional policy all of the four possibilities are admissible.*

The three policies can be combined by choosing one of them for each single rule of the program. We explicitly introduce now this mixed strategy. Since it encompasses all the other ones, it will be used to provide a model-based semantic characterization of all policies.

Mixed. Partition the program P in three mutually disjoint (possibly empty) sets of rules: $P = P_p \cup P_t \cup P_o$. Then, apply the policies Prodigal, Thrifty, and Optional, to the rules in P_p, P_t, and P_o, respectively.

To characterize the Mixed policy (which has the others as particular cases), consider a partial order on the collection of mappings μ. Such an order can be directly induced by a partial order $\sqsubseteq$ on $\mathcal{FM}(Q)$ simply defined as:

$$\forall\, m, m_1 \in \mathcal{FM}(Q)\ \big(m \sqsubseteq m_1 \leftrightarrow (m = \emptyset \vee m_1 \neq \emptyset)\big).$$

Consequently, a partial order $\sqsubseteq'$ on $\mathbb{S}_P$ can be so defined:

$$\forall\, F_1, F_2 \in \mathbb{S}_P \left(F_1 \sqsubseteq' F_2 \quad \leftrightarrow \quad \Big(\big(\forall\, \gamma \in P_p\ (F_1(\gamma) \sqsubseteq F_2(\gamma))\big) \wedge \big(\forall\, \gamma \in P_t\ (F_2(\gamma) \sqsubseteq F_1(\gamma))\big)\Big)\right).$$

Intuitively, by this definition, given two allocations F_1, F_2 of amounts to the rules in P, $F_1 \sqsubseteq' F_2$ holds if it is not the case that F_2 allocates some amount to a rule $\gamma \in P_p$ while F_1 does not (and vice versa for rules in P_t).
Finally, a partial order $\sqsubseteq''$ on the collection of mappings is definable as follows:

$$\forall\, \mu_1, \mu_2 \in (\mathbb{S}_P)^{\tau_{\mathcal{R}}} \; \big(\mu_1 \sqsubseteq'' \mu_2 \leftrightarrow \forall\, q \in \tau_{\mathcal{R}} \; (\mu_1(q) \sqsubseteq' \mu_2(q))\big).$$

Given an r-program P, consider the collection of those answer sets $\langle I, \mu \rangle$ of P which agree on I. The adoption of the policies corresponds to focusing on those answer sets that are maximal w.r.t. the order $\sqsubseteq''$.

Notice that, for a given (ground) r-program, by Definition 3.2, fixing $\sqsubseteq$ as above implies that the order $\sqsubseteq''$ does not distinguish among answer sets that fire the same set of rules a different number of times (i.e., $\sqsubseteq''$ only takes into account whether a rule is fired or not, regardless of the number of the firings). A different choice for the order $\sqsubseteq$, viable for instance in presence of multiple firings, could be: $\forall\, m_1, m_2 \in \mathcal{FM}(Q)\; (m_1 \sqsubseteq m_2 \leftrightarrow \sum(m_1) \leqslant \sum(m_2))$.

Example 7.2 *Consider a scenario*[11] *in which we have to produce an amount of electric power by means of different power stations. We can use a solar photovoltaic power plant capable of generating up to 10MW, depending on the percentage of solar irradiation that hits the solar cells. Alternatively, we can burn oil or coal in two thermoelectric power plants. One of them produces 5MW from 1 ton of oil; the other produces 3MW from 1 ton of coal. By structural limits of the plants, we cannot burn more than 9 tons of oil (resp., 7 tons of coal). Unfortunately, both fossil fueled power plants produce CO_2, which has to be limited as much as possible. Moreover, part of the coal (at most 15 tons) can be stored instead of burned. The situation can be represented by this program:*

$$\begin{array}{ll}
 & \mathit{megaWatt}{:}E \leftarrow \mathit{sunnyDay}, \mathit{solar\_irradiation}(P), \\
 & \qquad\qquad\qquad E = (10 * P)/100. \\
[1\text{-}9]: & \mathit{megaWatt}{:}5, \mathit{co2}{:}1 \leftarrow \mathit{oil}{:}1. \\
[1\text{-}7]: & \mathit{megaWatt}{:}3, \mathit{co2}{:}2 \leftarrow \mathit{coal}{:}1. \\
 & \mathit{leaveAtMost}{:}\mathit{coal}{:}15.
\end{array}$$

We want to apply the Prodigal policy for the first rule, because producing energy from sun irradiation does not produce CO_2, and the Thrifty policy for the other rules. Here is a possible instance of the problem to be solved, corresponding to a sunny day with the 80% of solar radiation reaching the solar plant. We also assume the availability of 40 tons of oil and 20 tons of coal. We have to produce between 40 and 50MW of power.

$$\begin{array}{ll}
\mathit{oil}{:}40. & \mathit{coal}{:}20. \\
\mathit{sunnyDay}. & \mathit{solar\_irradiation}(80). \\
\mathit{leaveAtLeast}{:}\mathit{megaWatt}{:}40. & \mathit{leaveAtMost}{:}\mathit{megaWatt}{:}50.
\end{array}$$

[11] Very simplified and with little intent of representing a realistic situation.

8 Implementing ground RASP

In this section, we refine the ASP encoding of ground r-programs described in Section 4 by taking into account all extensions of the RASP framework described in Section 6. We continue focusing on the ground case only. This constitute a first step towards the realization of a concrete implementation of RASP.

8.1 Multiple firings

In order to deal with multiply fireable r-rules (cf., Section 6.1), we have to slightly modify the encoding of r-rules described in Section 4.2.

In particular, given an r-rule γ of the form (14):

$$Idx: \quad H \leftarrow B_1, \ldots, B_k.$$

the following fragment of ASP code "declares" a counter used to denote the number of firings of γ. Clearly, its value is restrained considering the admitted values as specified by the collection of intervals Idx.

$firings(n_\gamma, i).$ for each integer i in Idx
$notcounter(n_\gamma, C) \leftarrow counter(n_\gamma, D),\ C \neq D.$
$counter(n_\gamma, C) \leftarrow firings(n_\gamma, C),\ fired(n_\gamma),\ not\ \ notcounter(n_\gamma, C).$
$\leftarrow not\ fired(n_\gamma),\ counter(n_\gamma, C).$

Notice that *counter* encodes the mapping ξ introduced as part of the r-interpretation (cf., Definition 3.1).

To take into account of repeated firings of r-rules in evaluating resource balances, the core inference engine has to be slightly modified as follows:

$fired(Rule) \leftarrow not\ \ notfired(Rule),\ r\_rule(Rule).$
$notfired(Rule) \leftarrow not\ \ fired(Rule),\ r\_rule(Rule).$
$use(Rule, I, Res, Count * Amount) \leftarrow fired(Rule),$
$\qquad a\_atom(Rule, I, Res, Amount),$
$\qquad counter(Rule, Count).$
$fired(Rule) \leftarrow use(Rule, I, Res, CA).$
$notfired(Rule) \leftarrow not\ \ use(Rule, I, Res, CA).$

$res\_symb(Res) \leftarrow a\_atom(Rule, I, Res, Amount).$
$balance(Q, N) \leftarrow N = sum\{A : use(Rule, I, Q, A)\},\ res\_symb(Q).$
$\leftarrow balance(Q, N),\ N < 0,\ res\_symb(Q).$

where the last three rules impose the constraint (13) on the global balance of resources. Note that the predicate *balance* encodes the component φ of the answer set of r-programs (see page 12).

8.2 Rendering of budget policies.

The translation described so far implements the Optional policy. For the Prodigal policy we add constraints that weed out the answer sets in which an r-rule could be fired but it is not. To this aim, these rules are added to the inference engine:

$$\begin{array}{rcl}
\mathit{extra\_need}(\mathit{Rule}, \mathit{Res}, \mathit{Need}) & \leftarrow & \mathit{r\_rule}(\mathit{Rule}),\ \mathit{res\_symb}(\mathit{Res}), \\
& & \quad \mathit{Need} = \mathit{sum}\{A : \mathit{a\_atom}(\mathit{Rule}, I, \mathit{Res}, A)\}. \\
\mathit{enbld}(\mathit{Rule}, \mathit{Res}) & \leftarrow & \mathit{balance}(\mathit{Res}, \mathit{Available}), \\
& & \mathit{extra\_need}(\mathit{Rule}, \mathit{Res}, \mathit{Need}), \\
& & \mathit{Available} + \mathit{Need} \geqslant 0.
\end{array}$$

where an atom $\mathit{enbld}(\mathit{Rule}, \mathit{Res})$ encodes the fact that the amount of the resource *Res* needed to fire the r-rule *Rule* is available in the answer set at hand.

To determine if a specific r-rule γ is fireable once more, we have to verify the truth of tha atom $\mathit{enbld}(n_\gamma, \mathit{Res})$ for each of the resources involved in γ. This condition is encoded through the atom $\mathit{enabled}(\mathit{Rule})$, defined as follows:

$$\begin{array}{rcl}
\mathit{enabled}(n_\gamma) & \leftarrow & \mathit{not}\ \mathit{fired}(n_\gamma),\ L_1,\ \ldots, L_n, \\
& & \ \mathit{enbld}(n_\gamma, q_1),\ \ldots,\ \mathit{enbld}(n_\gamma, q_k). \\
\mathit{enabled}(n_\gamma) & \leftarrow & \mathit{counter}(n_\gamma, C),\ \mathit{firings}(n_\gamma, \mathit{More}),\ \mathit{More} > C, \\
& & L_1,\ \ldots, L_n,\ \mathit{enbld}(n_\gamma, q_1),\ \ldots,\ \mathit{enbld}(n_\gamma, q_k).
\end{array}$$

where $L_1, \ldots, L_n$ are the program literals in the body of γ and $q_1{:}a_1, \ldots, q_k{:}a_k$ are all the amount-atoms in γ. Instances of these rules have to be added to the translation of each r-rule. In particular, the first rule handles the situation of a fireable r-rule that has not been fired at all. The second rule determines whether an r-rule, already fired C times, could have been fired once more.

The following rule (actually, an ASP constraint added to the inference engine) imposes that it is not the case that a fireable r-rule has not been fired. This enforces the Prodigal policy.

$$\leftarrow\ \mathit{enabled}(\mathit{Rule}).$$

An analogous treatment can be designed for other policies, such as the Thrifty policy, and hence for the Mixed one (to this aim it suffices to partition the r-rules of the program).

It is also easy to single-out an answer set corresponding to maximal numbers of firings by using the *maximize* statement commonly offered by ASP-solvers (such as, with sligthly different syntax, smodels and clasp). For example, in a syntax akin to the one accepted by the grounder gringo [25] (and by using *val* as auxiliary domain predicate), the following statement maximizes the sum of the values C occurring in atoms $\mathit{counter}(G, C)$, where G is any r-rule name:

$$\mathit{maximize}[\mathit{counter}(G, C) : \mathit{r\_rule}(G) : \mathit{val}(C) = C]$$

A solution of this kind also applies in dealing with costs of r-rules: in the above fragment of code each firing has unit cost. For a more general approach it suffices to multiply the number of firings of each r-rule by its cost.

8.3 A concrete implementation

In implementing RASP one has to render the underlying group (e.g., $\mathbb{Z}$, $\mathbb{Q}$,..., see Remark 3.1), in particular, all needed operations and functions (namely, sum, inversion, comparison, etc.), as well as the mapping κ. All these ingredients could be realized in at least two ways (not necessarily antithetic). On the one hand, since, as mentioned, a finite portion of Q enters into play in a ground program, we can encode it and the corresponding restrictions of all the needed operations as fragments of ASP code (e.g., by explicit tabulation). Alternatively, one could profit from some form of built-in or external source of computation. Examples of features supporting external evaluation in ASP-solvers are the user-defined functions and the API of lparse [52], external evaluation in dlv [22, 10], or even the integration with other programming paradigms and tools [13].

Once the underlying group has been instrumented, negative amount-atoms **-** q:a can be treated by exploiting the inverse function of the group (in conjunction with the rendering of κ, cf., Section 6.2).

The above-outlined translation from ground RASP programs into ASP has been implemented in a stand alone tool, named *raspberry*. Raspberry takes as input a file containing a ground RASP program S and the product $\mathcal{T}(S)$ of the translation can be processed by an ASP-solver. The answer sets found by the ASP solver encode the solutions of the RASP problem in the way explained in Section 4.2.

At the time of writing a first prototypical release of raspberry has been implemented and is available in `http://www.dipmat.unipg.it/~formis/raspberry`. Such a prototype is still under development, but covers many of the features described in this paper. Nevertheless, some of them, such as the handling of costs of r-rule firing and the management of constraint on global balance, are still to be implemented. Moreover, in this first release of raspberry, we fix the group Q to be the set $\mathbb{Z}$ of integer numbers. This is because commonly available ASP solvers offer operations on integers as built-in features. Refinements of the tool able to deal with other groups are a theme for future work.

Being raspberry in a preliminary and experimental stage of development, besides the translation described in this paper we implemented a few slightly different translations (in particular, the one described in [14], which differs in several points from the one treated in this paper), to provide output suitable to be processed by front-ends such as lparse and gringo (in turn, their output can be processed by smodels or clasp [5]). The translations mainly differ in the use of aggregate function and/or weight literals to implement some portions of the ASP encoding. Moreover, since some of the existing ASP solver do not treat negative integer values, raspberry provides a slightly more complex translation in which both consumed and produced amounts are represented by positive integers.

We conclude this section by reporting on the output produced by smodels for a short example.

Example 8.1 *Assembling different PCs requires different sets of components (motherboard, processor(s), ram modules, fan(s), hard disk(s), raid controller, etc.). In particular, the two r-rules below indicate the different requirements for a PC to be used as a server or as a simple desktop, resp. Moreover, they impose some bounds on the number of PCs that are assembled.*

$$\begin{array}{rl}
 & \mathit{cpu}{:}15. \\
 & \mathit{hd}{:}25. \\
 & \mathit{fan}{:}13. \\
 & \mathit{raid}{:}4. \\
 & \mathit{motherboard}{:}7. \\
 & \mathit{ram\_module}{:}20. \\
[1\text{-}3]: & \mathit{pc}(\mathit{server}){:}1 \leftarrow \mathit{cpu}{:}2, \mathit{hd}{:}6, \mathit{fan}{:}3, \mathit{raid}{:}1, \\
 & \qquad\qquad\qquad \mathit{motherboard}{:}1, \mathit{ram\_module}{:}4. \\
[2\text{-}6]: & \mathit{pc}(\mathit{desk}){:}1 \leftarrow \mathit{cpu}{:}1, \mathit{hd}{:}2, \mathit{fan}{:}1, \\
 & \qquad\qquad\qquad \mathit{motherboard}{:}1, \mathit{ram\_module}{:}2.
\end{array}$$

By translation into ASP, under the Prodigal policy, we obtain an ASP program having the following answer sets (the two r-rules are named $g1$ and $g2$, respectively):

$$\begin{array}{rcl}
S_1 & = & \{\mathit{balance}(\mathit{pc}(\mathit{server}), 1), \mathit{balance}(\mathit{pc}(\mathit{desk}), 6), \mathit{balance}(\mathit{cpu}, 7), \\
 & & \mathit{balance}(\mathit{hd}, 7), \mathit{balance}(\mathit{fan}, 4), \mathit{balance}(\mathit{raid}, 3), \\
 & & \mathit{balance}(\mathit{motherboard}, 0), \mathit{balance}(\mathit{ram\_module}, 4), \\
 & & \mathit{counter}(g1, 1), \mathit{counter}(g2, 6), \mathit{fired}(g2), \mathit{fired}(g1), \ldots\} \\
S_2 & = & \{\mathit{balance}(\mathit{pc}(\mathit{server}), 3), \mathit{balance}(\mathit{pc}(\mathit{desk}), 3), \mathit{balance}(\mathit{cpu}, 6), \\
 & & \mathit{balance}(\mathit{hd}, 1), \mathit{balance}(\mathit{fan}, 1), \mathit{balance}(\mathit{raid}, 1), \\
 & & \mathit{balance}(\mathit{motherboard}, 1), \mathit{balance}(\mathit{ram\_module}, 2), \\
 & & \mathit{counter}(g1, 3), \mathit{counter}(g2, 3), \mathit{fired}(g2), \mathit{fired}(g1), \ldots\} \\
S_3 & = & \{\mathit{balance}(\mathit{pc}(\mathit{server}), 2), \mathit{balance}(\mathit{pc}(\mathit{desk}), 5), \mathit{balance}(\mathit{cpu}, 6), \\
 & & \mathit{balance}(\mathit{hd}, 3), \mathit{balance}(\mathit{fan}, 2), \mathit{balance}(\mathit{raid}, 2), \\
 & & \mathit{balance}(\mathit{motherboard}, 0), \mathit{balance}(\mathit{ram\_module}, 2), \\
 & & \mathit{counter}(g1, 2), \mathit{counter}(g2, 5), \mathit{fired}(g2), \mathit{fired}(g1), \ldots\}
\end{array}$$

In the second solution, for instance, three desktops and three servers are produced. Even if we are not run out of any component, the remaining ones are not enough to compose another PC.

9 Related Work

A number of Prolog-like logic programming languages based on linear logic [48] have been proposed, such as, for instance, LO [2], LinLog [1], Lolli [32, 33], ACL [37, 38], Lygon [30, 31], Forum [34, 49], and Linear LF [11]. Most early works on linear logic programming had been based on Horn clauses and SLD-resolution (Prolog's execution model, [43]). However, extending this traditional procedural semantics to new logic programming languages based on richer logics rather than on Horn clauses has revealed to be a hard task: in fact, alternative proof-theoretical approaches have often been adopted. A widely-used design principle, called uniform proofs has been proposed by Miller et al. [50]. This design principle has required correspondent extensions to the standard Prolog abstract machine [59, 8, 7].

The programming style which is proper of these languages allows one to define and possibly generate a fact/rule as a resource that can be used only once. E.g., for finding a path in a graph the resources that need to be generated and then consumed are the vertices and edges of the graph. The reasoning is of the kind: "Assuming I have the following vertices and edges, look for a path".

The difference from RASP lies on the one hand on the underlying computational model: possible different uses of a resource and non-determinism in general are represented in RASP by different answer sets, rather than explored via backtracking. On the other hand, a resource in RASP is not a logical statement, but rather an atom which is defined via a logical statement. Thus, the program for finding a path in a graph in RASP would be written exactly like in standard ASP, unless one would like to find multiple paths: in this case, a fact like $edge(x, y)$:3 might state e.g., that the edge from x to y is allowed to occur in at most three paths, or more generally is allowed to be exploited by at most three procedures. A peculiar feature of RASP, which is not given a particular emphasis in linear logic programming languages, is that the quantity of a resource that is left unused by a certain process can be then exploited by some other process.

Another approach which exploits (a variant of) linear logic in order to equip logic programming with some notion of resources and reason about them is [54] which is meant to be a *resource programming language* (RPL). An operational semantics of RPL is given in terms of deduction rules: *storage operators* and *resource transformation rules* model availability and transformation of resources, respectively. The deduction proceeds by applying these rules in a Prolog-like goal-directed fashion. A notion of step-by-step evolution of the state of the world is implicit in the rule application mechanism and RPL is shown expressive enough to model Petri nets.

To deal with resources, [35] proposes a concurrent Prolog inference engine for clauses enriched with pre/post-conditions on resource availability. Resources are represented by multisets of atoms and terms (non-unit amounts of a resource are rendered through multiple copies of the same atom/term).

Both in [35] and [54] the operational semantics of the proposed frameworks can be given in terms of (refinements of) the SLD-procedure and (default) negation is not handled. Moreover, both programming languages of [54] and [35] offer little separation between the resource/amounts representation symbols and program symbols: resources and amounts are represented by program terms. The distinction is left to programmer's discipline.

A valuable feature of RPL is the presence of a temporal dimension (which is implicit in the succession of rule applications). This makes RPL closer to planners than RASP. The introduction of time in RASP, would, in principle, move the system towards the action description languages. This could represent an interesting topic for future work and comparison.

A form of resource treatment is described in [57, 56] to model product configuration problems. This framework is based on Weight Constraint Rules, which is a well-known construct encompassing default negation and disjunctive choices introduced in Answer Set Programming in [51]. Weight Constraint Rules have a wide applicability in many applications and are able to express costs and limits on costs, where however they do not express directly resource consumption/production.

Recently, [12] proposed the action description language $\mathcal{CARD}$. Resources are rendered through multi-valued fluents and the use of resources is implicitly modeled by the changes in fluents' values caused by actions' executions. The approach emphasizes the use of resources in planning problems and the semantics is given in terms of transition systems (in the spirit of [28]).

With respect to $\mathcal{CARD}$, in RASP there is a neater distinction between what is a resource and what is not. Moreover, the arithmetic of amounts is implicitly handled by RASP's inference engine. It seems that in $\mathcal{CARD}$ these aspects have to be encoded in the problem specification. However, since $\mathcal{CARD}$ is tailored to model action theories, time and state evolution are easily dealt with.

The approach of [44] is in the context of Horn Logic Programming. It assumes that a derivation may have a cost, and that in a chain of derivations the respective costs are accumulated. Thus, the approach is suitable for modeling production and use of items, where however it is assumed that each item can be produced/used as many times as needed and there is no concept of quantity. Then, the approach to costs of derivations is quite general and in principle might profitably be integrated with RASP.

The approach of [53] enhances ASP with explicit negation as it allows rules to be defeated by competing rules, where defeating a rule has a cost. Then, on the one hand answer sets exist even when contradictions are present by defeating one of the contradictory rules, on the other hand each of these answer sets corresponds to a global cost corresponding to the rules that have been defeated. The "preferred" answer sets are those that are minimal w.r.t. the cost that they pay. Also this approach is not in competition w.r.t. RASP, but rather can be seen as complementary.

Concluding remarks

In this work we have proposed RASP, an extension of ASP that offers the possibility of defining and reasoning about resources with their amounts. Resources can be produced and consumed by rules' firings, that can also be multiple, taking into account various kinds of *global* constraints on resource consumption/production, costs of rules, as well as policies to customize and filter resource allocation.

The semantics of resource-amounts relies upon an auxiliary algebraic structure which models quantities, operations, and relations among them. Plainly, the most natural choice for such a structure is $\mathbb{Z}$, but the framework easily generalizes to $\mathbb{Q}$ or to more general groups (cf., Remark 3.1).

The extension RASP can be useful to model easily and directly several kinds of production processes and planning problems, including configuration problems. Different allocations of resources correspond to different answer sets. We have outlined a compilation process that translates RASP specifications into plain ASP fragments of code. The resulting encoding is completed with a general inference engine, to obtain an ASP program that behaves according to RASP semantics. A prototypical translator from ground RASP into ASP has been designed. Such a tool implements the (core of the) translation described in this paper and can be used to produce ASP encodings to be processed by commonly available grounders, such as lparse and gringo.

An envisaged extension of the framework consists in allowing amounts in amount-atoms to be described explicitly as intervals or sets of values. Similarly, explicit expressions or compound resource-terms could be considered. First steps in this direction are reported in [24].

Interesting lines of research regard the extension of the RASP framework so to admit, within the r-rules, the specification of preferences on resource usage. Results in this stream of research can be found in [15, 16, 17].

We intend to investigate about possible reasonable semantics to for the negation for amount-atoms. Actually, some forms of "negation" can be expressed by means of the constraints on global resource balance described in Section 6: in fact, one can express for instance what *should not* be left. However, more work is needed in order to understand the possible uses of negation in this framework.

The extension of the ASP framework we described is, to the best of our knowledge, an original proposal. A practical comparison (on suitable case-studies) with other frameworks suited for modeling production processes, resource management, and quantitative reasoning represents an interesting subject for future investigations.

Acknowledgements

The authors would like to thank Wolfgang Faber, Georg Gottlob, Nicola Leone, and all the anonymous reviewers for useful suggestions on several aspects of this work. We also thank Torsten Schaub and Sven Thiele for the support and the advices about the grounder gringo. This research has been partially supported by GNCS.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[3] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See `http://asparagus.cs.uni-potsdam.de`.

[4] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *J. of Logic Programming*, 19/20:9–72, 1994.

[5] Web references for some ASP solvers. ASSAT: `http://assat.cs.ust.hk`; Ccalc: `http://www.cs.utexas.edu/users/tag/ccalc`; Clasp: `http://www.cs.uni-potsdam.de/clasp`; Cmodels: `http://www.cs.utexas.edu/users/tag/cmodels`; DeReS and aspps: `http://www.cs.uky.edu/ai/`; DLV: `http://www.dbai.tuwien.ac.at/proj/dlv`; Smodels: `http://www.tcs.hut.fi/Software/smodels`.

[6] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[7] M. Banbara. *Design and Implementation of Linear Logic Programming Languages*. PhD thesis, The Graduate School of Science and Technology of Kobe University, September 2002.

[8] M. Banbara and N. Tamura. Compiling resources in a linear logic programming language. In *In Proceedings of the Workshop on Parallelism and Implementation Technology for Logic Programming Languages*, pages 32–45, 1998.

[9] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[10] F. Calimeri and G. Ianni. External sources of computation for answer set solvers. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proc. of the 8th Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *LNCS*, pages 105–118. Springer, 2005.

[11] I. Cervesato and F. Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science–LICS'96*, pages 264–275. IEEE Computer Society Press, 1996.

[12] S. Chintabathina, M. Gelfond, and R. Watson. Defeasible laws, parallel actions, and reasoning about resources. In E. Amir, V. Lifschitz, and R. Miller, editors, *Logical Formalizations of Commonsense Reasoning: Proceedings of CommonSense'07*. AAAI Press, Menlo Park, 2007. Technical report SS-07-05.

[13] E. Corona and M. Osorio. The A-Pol system. In M. De Vos and A. Provetti, editors, *Answer Set Programming, Advances in Theory and Implementation, Proc. of the 2nd Intl. ASP'03*, volume 78 of *CEUR Workshop Proc.*, 2003.

[14] S. Costantini and A. Formisano. Modeling resource production and consumption in answer set programming. In *Proc. of ASP07*, 2007.

[15] S. Costantini and A. Formisano. Conditional preferences in P-RASP. In *Proc. of LANMR'08*, 2008.

[16] S. Costantini and A. Formisano. Modeling preferences on resource consumption and production in ASP. In *Proc. of ASPOCP'08*, 2008. Extended version as Report-09/2008 of Dip. di Matematica e Informatica, Univ. di Perugia: `www.dipmat.unipg.it/~formis/papers/report2008_09.ps.gz`.

[17] S. Costantini and A. Formisano. Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of Algorithms in Cognition, Informatics and Logic*, 64(1):3–15, 2009.

[18] M. Cox and D. Nelson. *Lehninger Principles of Biochemistry.* Freeman & Co., 2004.

[19] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In P. Hill and D. Warren, editors, *Logic Programming, 25st International Conference, ICLP 2009, Proceedings*, volume 5649 of *LNCS*. Springer, 2009.

[20] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[21] A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental and Theoretical Artificial Intelligence*, 21(2):79–121, 2009.

[22] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proc. of the 19th Intl. Joint Conference on Artificial Intelligence*, pages 90–96. Professional Book Center, 2005.

[23] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming*, 8:545–580, 2008.

[24] A. Formisano and D. Petturiti. Extending and implementing RASP. In M. Gavanelli and F. Riguzzi, editors, *Proc. of CILC'09*, 2009.

[25] M. Gebser, T. Schaub, and S. Thiele. GrinGo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Proc. of the 9th Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.

[26] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation, chapter 7*. Elsevier, 2007.

[27] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of the 5th Intl. Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[28] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16):193–210, 1998.

[29] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[30] J. Harland and D. Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

[31] J. Harland, D. Pym, and M. Winikoff. Programming in lygon: An overview. In M.Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 391–405. Springer-Verlag, 1996.

[32] J. S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[33] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[34] J. S. Hodas and J. Polakow. Forum as a logic programming language: Preliminary results and observations. In M. Okada, editor, *Proceedings of the Linear Logic'96 Meeting, volume 3*, Elsevier Electronic Notes in Theoretical Computer Science, 1996.

[35] J.-M. Jacquet and L. Monteiro. Towards resource handling in logic programming: The PPL framework and its semantics. *Computer Languages*, 22((2/3)):51–77, 1996.

[36] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Prog. of the 1991 International Logic Programming Symposium*, pages 387–401, 1991.

[37] N. Kobayashi and A. Yonezawa. Acl - a concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294. MIT Press, 1993.

[38] N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994.

[39] C. Lefèvre and P. Nicolas. A first order forward chaining approach for answer set computing. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR'09*, LNCS. Springer-Verlag, 2009.

[40] N. Leone. Logic programming and nonmonotonic reasoning: From theory to systems and applications. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*, page 1, 2007.

[41] V. Lifschitz. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, pages 23–37, 1999.

[42] V. Lifschitz and H. Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.

[43] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[44] V. W. Marek and M. Truszczynski. Logic programming with costs. Unpublished Note.

[45] V. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):587–618, 1991.

[46] V. W. Marek and M. Truszczyński. Computing intersection of autoepistemic expansions. In *Proceedings of the First International Workshop on Logic Programming and Non Monotonic Reasoning*, pages 35–70. The MIT Press, 1991.

[47] V. W. Marek and M. Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer, 1999.

[48] D. Miller. Linear logic programming references, 1995. Web site: `ftp://ftp.cis.upenn.edu/pub/papers/miller/ComputNet95/llsurvey.html`.

[49] D. Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[50] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[51] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR'99*, number 1730 in LNAI, pages 317–331. Springer-Verlag, 1999.

[52] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proc. of the 8th Workshop on Non-Monotonic Reasoning*, 2000.

[53] D. V. Nieuwenborgh, S. Heymans, and D. Vermeir. Weighted answer sets and applications in intelligence analysis. In *Proceedings LPAR 2004*, number 3452 in LNAI, pages 169–183. Springer-Verlag, 2005.

[54] Y. U. Ryu. A logic-based modeling of resource consumption and production. *Decision Support Systems*, 22(3):243–257, 1998.

[55] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[56] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, number 1551 in LNCS, pages 305–319. Springer-Verlag, 1999.

[57] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming (ASP'01): Towards Efficient and Scalable Knowledge.* AAAI Press, Menlo Park, 2001. Technical report SS-01-01.

[58] T. C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3), 2007.

[59] N. Tamura and Y. Kaneda. Extension of wam for a linear logic programming language, 1996.

[60] M. Truszczynski. Logic programming for knowledge representation. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*, pages 76–88, 2007.