**UNIVERSITÀ DEGLI STUDI DI PERUGIA**
DIPARTIMENTO DI MATEMATICA E INFORMATICA
06123 PERUGIA – VIA VANVITELLI 1

# Modeling Resource Production and Consumption in Answer Set Programming

STEFANIA COSTANTINI
Dipartimento di Informatica, Università di L'Aquila

ANDREA FORMISANO
Dipartimento di Matematica e Informatica, Università di Perugia

Rapporto Tecnico 2008/4

**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 specific atoms, to which we associate *quantities* that represent the available amount. The firing of a RASP-rule can both consume and produce resources. We define the semantics for RASP programs by extending the usual answer set semantics and we propose an implementation based on standard ASP-solvers.

# Contents



**Author Address:**
Dipartimento di Matematica e Informatica, Università degli Studi di Perugia
Via Vanvitelli 1, I-06123 Perugia, Italy.
E-mail: `formis@dipmat.unipg.it`
WWW: `http://www.dipmat.unipg.it/~formis`

Dipartimento di Informatica, Università degli Studi di L'Aquila
Via Vetoio, I-67010 L'Aquila, Italy.
E-mail: `stefcost@di.univaq.it`

# Modeling Resource Production and Consumption in Answer Set Programming*

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

## Introduction

Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [11], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [14, 15]. 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 [4, 19, 1] among many).

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 [12] and Description Logics [3]. In this direction, 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 $q$. 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.

We provide syntax of RASP programs (Section 1) by allowing amount-atoms to occur both in the head and in the body of rules. Amount-atoms in the body (resp., in the head) of rules indicate which resources are *consumed* (resp., *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 s(he) can cook a cake. Similarly, if s(he) has three eggs, two hundred grams of sugar and two hundred milliliters of milk, then s(he) can prepare an ice-cream. Depending on the number $n$ of available eggs, it might be the case that the ingredients are sufficient to prepare both, one, or none of the desserts. Program $P_1$ below could be a RASP program encoding such a situation:

*A preliminary version of this paper appeared in ASP07, workshop of ICLP07, Porto, Portugal September 2007.

```
γ1: have_cake :- egg:3, flour:3, sugar:3.
γ2: have_ice_cream :- egg:3, sugar:2, milk:2.
γ3: egg:n.
γ4: flour:8.
γ5: sugar:6.
γ6: milk:3.
```

where the amounts occurring in the body of a rule correspond to resources which are consumed whenever the rule is used. Facts $\gamma_3, \ldots, \gamma_6$ describe initially available 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 4).

Notice that the heads of the rules $\gamma_1$ and $\gamma_2$ do not involve resources, i.e., the two atoms `have_cake` and `have_ice_cream` are usual atoms that just 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.

Semantics for RASP programs is provided in Section 2, by combining usual answer set semantics with an interpretation of resource amounts, where different allocation choices correspond to different answer sets. We discuss extensions to the basic paradigm (Section 3) where, in the context of the same semantics, one can define both constraints on quantities and negative quantities. Also, 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. In Section 4 we discuss how to introduce a filter on the answer sets of a RASP program, so as to specify different *policies* for production and consumption of resources. Finally, an implementation of RASP based on standard ASP-solvers is proposed in Section 5.

## 1 Syntax of RASP-programs

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

The elements of $\mathcal{C}_R$ are said *amount-symbols*, while the elements of $\Pi_R$ are said *resource-symbols*. 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 $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. An *amount-atom* is a writing of the form $q{:}a$ where $q \in \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$ and $a$ is an amount-term. 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, naf, for short).[1] 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.

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

[1] 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.

**Definition 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.*

*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.*

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.

**Definition 2** *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 3** *A* RASP-rule *(*rule*, for short) $\gamma$ is either a program-rule or a resource-rule. An* r-program *is a finite set of RASP-rules.*

**Example 4** *Let us consider the r-program $P_1$ mentioned in the Introduction. All the rules are r-rules. We identify $\mathcal{C}_R$ with the set of integer numbers $\mathbb{Z}$ and we have $\Pi_R = \{\mathtt{egg}, \mathtt{flour}, \mathtt{sugar}, \mathtt{milk}\}$ and $\Pi_P = \{\mathtt{have\_cake}, \mathtt{have\_ice\_cream}\}$.*

**Remark 5** *As Definition 1 states, in general, we admit several amount-atoms in the head of a rule where the case in which a rule $\gamma$ has an empty head is admitted only if $\gamma$ is a program-rule (i.e., $\gamma$ is an ASP* constraint*).*

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

As customary, a term (atom, literal, rule, ...) is ground if no variable occurs in it. A ground program is composed of ground terms. The grounding of a program $P$ is the set of all ground instances of rules of $P$, obtained through ground substitutions over the Herbrand universe of $P$. As it is well-known, ASP solvers produce the grounding of the given program as a first step, as they are able to find the answer sets of ground programs only.[2] For clear pragmatical reasons, in view of the implementation of RASP (cf. Section 5), it is convenient to impose suitable constraints on r-programs so to ensure finiteness of the grounding process. To this aim, we impose usual restrictions on those variables not occurring as amount-terms (e.g., *range restriction*, cf. [4]). Moreover, we extend such requirement to all variables occurring as amount-terms (hence, all variables of a rule must occur (positively) in a program-atom of the body).

[2] Work is under way both theoretically and practically to overcome at least partially this limitation. However, at present all ASP solvers perform the grounding.

**Remark 6** *Notice that in any r-program only a finite number of resource-symbols of $\mathcal{C}_R$ occurs, also because all r-facts must be ground. Hence, the above-mentioned restriction on variables guarantees that 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.*

In what follows, let $\mathcal{B}_{\mathcal{H}}(X,Y)$ denote the collection of all ground atoms built up from predicate symbols in $X$ and terms in $Y$.

## 2 Semantics of RASP-programs

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. Then, in order to define semantics of r-programs, we have to fix an interpretation for amount-symbols. This is done by choosing a collection $Q$ of *quantities*, a natural choice being $Q = \mathbb{Z}$, and the operations to combine and compare quantities. To this aim, we consider given a mapping $\kappa : \mathcal{C}_R \to \mathbb{Z}$ that associate integers to amount-symbols. Positive (resp. negative) integers will be used to model produced (resp. consumed) amounts of resources.

**Remark 7** *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 addictive operation $\oplus_{\mathcal{Q}}$, and $\preccurlyeq_{\mathcal{Q}}$ is a translation invariant total order over $Q$.)*

**Example 8** *Consider the r-program $P_1$ used in the Introduction and in Example 4. By choosing $Q$ to be $\mathbb{Z}$ (with the usual sum and total order relation), since we have identified $\mathcal{C}_R$ with $\mathbb{Z}$, $\kappa$ is the identity function over $\mathbb{Z}$.*

As will be discussed in Section 5, a solver for RASP can be obtained by exploiting existing ASP-solvers through compilation into ASP. Once $\kappa$ has been chosen, it can be encoded in r-programs through program-predicates and usual ASP-rules. Alternatively, one can directly employ numerical terms and rely on the built-in features of the specific ASP-solver.

Before going on, we introduce some useful notation. Given two sets $X, Y$, let $\mathcal{FM}(X)$ denote the collection of all finite multisets 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.

We introduce now the notion of r-interpretation for r-programs, by considering the ground case in the first place. As we will see, for any fixed resource-symbol, an interpretation of a (ground) r-program $P$ must determine an allocation of amounts for all occurrences of such 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. Hence, we impose, from the very beginning, suitable restrictions on the admitted interpretations of amount-symbols. Namely, we restrain to those allocations having (for each resource) a non-negative global balance. The collection $\mathbb{S}_P$ of all potential allocations (for any single resource-symbol) is the following set 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 rational behind the definition of $\mathbb{S}_P$ is as follows: 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. All such $F$ must satisfy the requirement that, considering the entire $P$, the global sum of all the quantities $F$ assigns must be non-negative.

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

$$\mu : \Pi_R \to \mathbb{S}_P.$$

Such a function $\mu$ determines, for each resource-symbol $q \in \Pi_R$, 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, a quantity for each occurrence of an amount-atom $q{:}a$ in $\gamma$ (being $a \in \mathcal{C}_R$ an amount-symbol).

As it will be clear in the sequel, 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 in characterizing the notion of r-interpretation for an r-program we have to take into account of two kind of literals: resource-atoms and program literals. Namely, two aspect have to be considered: the truth of program literals and the allocation of resources. We have the following definition.

**Definition 9** *An* r-interpretation *for a (ground) r-program $P$ is a pair $\mathcal{I} = \langle I, \mu \rangle$, with $I \subseteq \mathcal{B}_{\mathcal{H}}(\Pi_P, \mathcal{C})$ and $\mu : \Pi_R \to \mathbb{S}_P$.*

**Example 10** *Let us consider again the r-program $P_1$ of Examples 4 and 8. By putting $n = 4$, an r-interpretation for $P_1$ is $\langle I, \mu \rangle$ with $I = \{\mathtt{have\_cake}\}$ and $\mu$ such that*

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

Clearly, 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 such rule. We reflect the fact that the satisfaction of an r-rule $\gamma$ depends on the truth of its program-literals by introducing a suitable fragment of ASP program $\widehat{\gamma}$. Let the r-rule $\gamma$, 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:

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

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

The following definition states that in order to be a model, an r-interpretation that allocates non-void amounts to the resource-symbols of $\gamma$ (i.e., $\gamma$ is *fired*, see below), has to model the ASP-rules in $\widehat{\gamma}$.

**Definition 11** *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 $\gamma \in P$*

$$\Big(\forall\, q \in \Pi_R\, \big(\mu(q)(\gamma) = \emptyset\big)\Big) \vee \Big(\forall\, q \in \Pi_R\, \big(\mu(q)(\gamma) = \mathcal{R}(q,\gamma)\big)\Big) \tag{2}$$

  *where $\mathcal{R}(q,\gamma)$ is the multiset so defined:*

$$\begin{aligned} \mathcal{R}(q,\gamma) \quad = \quad \{\!\!\{ v \in \mathbb{Z} \mid v = -\kappa(a) \textit{ for } q{:}a \textit{ in the body of } \gamma \\ \textit{or }\ v = \kappa(a) \textit{ for } q{:}a \textit{ in the head of } \gamma \}\!\!\} \end{aligned} \tag{3}$$

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

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

The two disjoints of formula (2) in Definition 11 correspond to the two cases: a) the rule $\gamma$ is not fired, so null amounts are allocated to all its amount-symbols; b) the rule $\gamma$ is actually fired and all needed amounts are allocated.

We now formally introduce the notions of *rule firing* and *resource balance*:

**Definition 12** *Let $\mathcal{I} = \langle I, \mu\rangle$ be an answer set for a (ground) r-program $P$. An r-rule $\gamma$ in $P$ is* fired *if $\exists\, q \in \Pi_R\, \big(\mu(q)(\gamma) \neq \emptyset\big)$ (i.e., through its satisfaction, some resources are consumed and/or produced). The* resource balance *for $P$, w.r.t. $\langle I, \mu\rangle$, is the mapping $\varphi : \Pi_R \to \mathbb{Z}$ defined as:*

$$\varphi(q) = \sum \Big( \Big\{ \sum \big(\mu(q)(\gamma)\big) \mid \gamma \in P \Big\} \Big)$$

*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 $P$ (cf. Remark 6).

**Example 13** *Consider the r-interpretation* $\langle I, \mu \rangle$ *of Example 10. 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* $\gamma_1$*. The resource balance* $\varphi$ *is such that:* $\varphi(\mathit{egg}) = 1$*,* $\varphi(\mathit{flour}) = 5$*,* $\varphi(\mathit{sugar}) = 3$*, and* $\varphi(\mathit{milk}) = 3$*.*

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 4. 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 13).

**Example 14** *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* $\gamma_1$ *and* $\gamma_2$*:*

$\gamma_1'$: `cake:1 :- egg:3, flour:3, sugar:3.`
$\gamma_2'$: `ice_cream:1 :- egg:3, sugar:2, milk:2.`

*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* $\mathcal{C}_R$ *with* $\mathbb{Z}$*, but now we have* $\Pi_R = \{\mathit{cake}, \mathit{ice\_cream}, \mathit{egg}, \mathit{flour}, \mathit{sugar}, \mathit{milk}\}$ *and* $\Pi_P = \emptyset$*. By putting* $n = 4$*, an r-interpretation for* $P_2$ *is* $\langle \emptyset, \mu' \rangle$ *with* $\mu'$ *such that*

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

*where* $\mu$ *is as in Example 10. The program* $\widehat{P_2} = \widehat{\gamma_1}'$ *is empty. Rule* $\gamma_1'$ *is fired (as for rule* $\gamma_1$ *in Example 13). In this case the resource balance* $\varphi'$ *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 15** *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* $a_1, a_2, a_3 \in \mathbb{Z}$*):*

```
warm :- fuel(Type):Q, load(Load),
        requiredFuel(Load,Type), needed(Type,Q).
load(low) :- temperature(Celsius), Celsius>10, is_winter.
load(high) :- temperature(Celsius), Celsius=<10, is_winter.
load(none) :- not is_winter.
requiredFuel(high,oil).
requiredFuel(low,electricity).
needed(oil, a2).
needed(electricity, a1).
fuel(oil):a1 :- money:a3.
fuel(electricity):a2 :- money:a3.
:- not warm.
```

*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:*

```
temperature(5).        money:a3.        is_winter.
```

*otherwise, for a mite winter we could assert the fact* `temperature(20)` *in place of* `temperature(5)`*. Finally, to obtain from a RASP-solver (see Section 5), both models as (alternative) expected solutions of a single r-program, we could exploit a cardinality constraint (cf. [17, 18]) as follows:*

```
1{temperature(5), temperature(20)}1.        money:a3.        is_winter.
```

*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 16** *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 [7]):*

**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:**

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

*The following rules encode such reactions:*

```
oxygen:A, atp:C, nadph:B, proton:B :- water:B, adp:C, nadp:B, pi:C,
                                      light, B=2*A, C=3*A, A=6*Moles,
                                      reagents1(Moles,A,B,C,D).

glucose:D, water:A, adp:C, nadp:B, pi:C :- nadph:B, proton:B, atp:C,
                                           oxygen:A, carbDioxide:A,
                                           A=6*D, B=2*A, C=3*A,
                                           reagents2(A,B,C,D).
reagents1(V,W,X,Y,Z) :- ...
reagents2(W,X,Y,Z) :- ...
```

*where we generically used the two predicates* `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.*

The reader may have noticed that in the semantics we do not have considered negation-as-failure applied to amount-atoms. The reason is not technical, as there might be several approaches to assess this semantics. The problem is that it is not clear to us which is the most intuitive meaning of saying that "an amount-atom $q{:}a$ cannot be assumed to be true". Should it mean that the resource $q$ is not produced at all? Or it means 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 (for instance, to impose that an r-rule does not produce a specific $a$ amount of a resource $q$) or 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 section.

# 3 Extending the basic framework

## 3.1 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 by admitting r-rules $\gamma$ of the form

$$[N_1, N_2] : \quad H \leftarrow B_1, \ldots, B_k \tag{4}$$

where $H$ and the $B_i$ are as before, and each $N_i$ is either a variable or a positive integer. The intended meaning is that, whenever $\gamma$ is fired, this can be done any number $n$ of times provided that $N_1 \leqslant n \leqslant N_2$ holds. For the sake of generality, let us assume that both $N_1$ and $N_2$ are variables. Then, after grounding, $N_1$ and $N_2$ have to be instantiated to positive integers (namely, in $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$). To this purpose, both variables $N_1$ and $N_2$ have to occur in program atoms of the body of the rule. Notice that, if in an instance of (4) both $N_1$ and $N_2$ are instantiated to 1, we obtain an r-rule as introduced in Section 1.

**Example 17** *The following rule $\gamma_1''$ specifies that we can produce from one to three cakes (or even zero, if the rule is not fired at all):*

```
γ1'':    [1,3]: cake:1 :- egg:3, flour:3, sugar:3.
```

The rule $\gamma_1''$ is an "iterable" version of rule $\gamma_1'$ of Example 14. 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.).

How may firings will be actually performed? We will introduce in Section 4 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 18** *Notice that, similarly to what done in Example 17 for $\gamma_1'$, it is possible to introduce an "iterable" version of the rule $\gamma_1$, as follows:*

```
γ1''':    [1,3]: have_cake :- egg:3, flour:3, 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 that necessary.*

To deal with multiple fireable rules in the semantics, it suffices to modify the definition of r-interpretation as follows (cf. Definitions 9 and 11): An r-interpretation for a program $P$ is a triple $\mathcal{I} = \langle I, \mu, \xi \rangle$, where $I$ and $\mu$ are as before, while $\xi$ is a mapping $\xi : P \to \mathbb{N}^+$ that assign to each (fired) rule the number of its 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 $N_1 \leqslant \xi(\gamma) \leqslant N_2$.

In view of this notion of r-interpretation, the definition of answer set is obtained from Definition 11 by replacing condition (2) with the following one:[3]

$$\Big(\forall\, q \in \Pi_R\big(\mu(q)(\gamma) = \emptyset\big)\Big) \vee \Big(\forall\, q \in \Pi_R\big(\mu(q)(\gamma) = \xi(\gamma)\cdot\mathcal{R}(q,\gamma)\big) \wedge \big(N_1 \leqslant \xi(\gamma) \leqslant N_2\big)\Big).$$

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

### 3.2 Negative amount-atoms.

It is possible to introduce *negative amount-atoms* of the form - $q{:}a$ in the body of an r-rule. When that rule is fired, a negative amount-atom implies the consumption of $-\kappa(a)$ units of resource $q$. This actually means that an amount $\kappa(a)$ of resource $q$ is produced. The 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.

Formally, a *negative* amount-atom is of the form - $q{:}a$, where $q{:}a$ is an amount-atom. The easiest manner of interpreting negative amount-atoms is through the inverse function on $\mathbb{Z}$. Formally, it suffices to modify the definition of $\mathcal{R}(q, \gamma)$ (cf. (3) in Definition 11) as follows:

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

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

### 3.3 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 $\gamma$ and that the firing of $\gamma$ should be avoided for particular amounts of $q$. This can be ensured by using program-literals in the body of $\gamma$ that inhibit the firing of $\gamma$ whenever $X$ assumes such values (as 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 $\mathit{leaveAtMost} : q{:}a$. In this case, in each answer set no more that $\kappa(a)$ units of $q$ can be left unused.

For a resource $q$, in order to fulfill the constraints $\mathit{leaveAtLeast} : q{:}a_q$ and $\mathit{leaveAtMost} : q{:}b_q$, we filter the potential r-interpretations for an r-program $P$ by refining the definition of the mapping $\mu$ (cf. (1), page 7). In particular, we restrain its codomain $\mathbb{S}_P$ as follows:

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

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

### 3.4 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 $\gamma$ 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 $\gamma$. If $C$ is a variable, then it has to occur in a program atom of the body of $\gamma$ and after grounding it has to be instantiated to an amount-symbol.

**Example 19** *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 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 a cardinality constraint (see [17, 18]).*

*Provided that 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.*

```
Time:[1,3]:  cake:1 :- egg:3, flour:3, sugar:3,
                       needed_time(Helpers,Time), helpers(Helpers).

             helpers(N) :- N { at_home(husband), at_home(son),
                               at_home(daughter), at_home(sister) } N,
                           num(N).

             needed_time(0,45).  % no one helps, I cook by myself!
             needed_time(N,Time) :- N>0, N<3, Time=45-(N*10).
             needed_time(N,25) :- N>2.  % too many cooks do not help!
             num(0..5).
             at_home(X) :- .....
```

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

```
Time:[1,3]:  cake:1 :- egg:3, flour:3, sugar:3, have_free_time(Time),
                       needed_time(Helpers,Time), helpers(Helpers).
```

*In this case the truth of a fact of the form* `have_free_time(·)` *(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 by smodels.

## 4 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 $\mu$ 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 the set $I$, more that one mapping might satisfy the conditions expressed in Definition 11. 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 $\gamma$ is fired only if this is forced. (For instance, because the truth of a program-atom occurring as head of $\gamma$ is required by effect of other parts of the program.)

**Prodigal.** Whenever an r-rule $\gamma$ 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 20** *Recall the program $P_2$ of Example 14, where we put $n = 7$ in $\gamma_3$. Four potential solutions are: make none, one, or both the 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: $P = P_{\mathbf{P}} \cup P_{\mathbf{T}} \cup P_{\mathbf{O}}$. Then, apply the policies Prodigal, Thrifty, and Optional, to the rules in $P_{\mathbf{P}}$, $P_{\mathbf{T}}$, and $P_{\mathbf{O}}$, respectively.

To characterize the Mixed policy (which has the others as particular cases), consider a partial order on the collection of mappings $\mu$. Such an order can be directly induced by a partial order $\sqsubseteq$ on $\mathcal{FM}(Q)$ simply defined as: $\forall\, m \in \mathcal{FM}(Q)\ (\emptyset \sqsubseteq m)$. Consequently, a partial order $\sqsubseteq'$ on $\mathbb{S}_P$ can be so defined:

$$\forall\, F_1, F_2 \in \mathbb{S}_P \left( F_1 \sqsubseteq' F_2 \quad\leftrightarrow\quad \Big(\big(\forall\,\gamma \in P_{\mathbf{P}}\ (F_1(\gamma) \sqsubseteq F_2(\gamma))\big) \wedge \big(\forall\,\gamma \in P_{\mathbf{T}}\ (F_2(\gamma) \sqsubseteq F_1(\gamma))\big)\Big)\right).$$

Finally, a partial order $\sqsubseteq''$ on the collection of mappings is so definable:

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

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 11, fixing $\sqsubseteq$ as above implies that the order $\sqsubseteq''$ does not distinguish among answer sets that fires 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 refined 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 21** *Consider a scenario*[4] *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 able to generate 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 that 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:*

```
        megaWatt:E :- sunnyDay, solar_irradiation(P), E=(10*P)/100.
[1,9]:  megaWatt:5, co2:1 :- oil:1.
[1,7]:  megaWatt:3, co2:2 :- coal:1.
        leaveAtMost:coal:15.
```

*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.*

```
oil:40.                          coal:20.
sunnyDay.                        solar_irradiation(80).
leaveAtLeast:megaWatt:40.         leaveAtMost:megaWatt:50.
```

## 5 Towards an implementation of ground RASP

In this section we explore a way of characterizing the semantics of r-programs by describing a translation of ground r-programs into plain ASP, so as to establish a correspondence between the answer sets of an r-program and the answer sets of a suitable ASP-program. (Needless to say that such a translation should be carried out by a compiler and then it would be transparent to the user.) In what follows we deal with ground r-programs only. We remark that the grounding process has to take care of multiple amount-atoms that could appear in a rule as effect of ground substitution. Multiple copies of the same amount-atom must not be deleted, since each occurrence of an amount-atom corresponds 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.

In implementing RASP one have to render the group $\mathbb{Z}$ (or $\mathbb{Q}$, see Remark 7), together with all needed operations and functions (namely, sum, inversion, comparison, etc.), as well as the mapping $\kappa$. 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 $\mathbb{Z}$ 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

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

the user-defined functions and the API of lparse [16], external evaluation in dlv [8, 5], or even the integration with other programming paradigms and tools [6].

In Section 5.1 we propose an encoding of r-rules in ASP. Each r-rule is translated into a fragment of ASP program. Notice that the framework for reasoning on resource in ASP does not involve just a pure syntactical rewriting of r-rules into ASP rules. The rendering of all r-rules must be joined with an ASP program (to be seen in Section 5.2) that acts as an inference engine and performs the concrete reasoning activity on resource allocations.

### 5.1 ASP encoding of r-rules.

In encoding resource handling in ASP, in order to use solvers such as smodels or dlv, we have to cope with a limitation of these tools relative to dealing with negative numbers. Some solvers (e.g. dlv [13]) explicitly restrain the integers occurring in a program to be non-negative. Others, namely those based on the front-end lparse [17], do not explicitly impose such a restriction. Nevertheless, some unexpected behaviors may be experimented, with the grounder lparse, when negative quantities enter into play in the search/construction of answer sets.[5] To overcome this limitation, we are forced to separate the treatment of negative and positive quantities. The adoption of such an approach has two immediate consequences. On the one hand, it emphasizes the different use of amounts by distinguishing, w.r.t. each rule, those resources that are consumed from those that are produced. On the other hand, it slightly complicates the translation of an r-program into an ASP program, that would be simpler if the ASP-solver at hand treated negative integers directly.

Let $\gamma$ be the a ground r-rule

$$p \leftarrow L_1, \ \ldots, \ L_n, \ q_1{:}a_1, \ \ldots, \ q_k{:}a_k$$

with $k, n \geqslant 0$, then its rendering in ASP code is made of these ASP rules:[6]

(1) $\texttt{rule(n}_\gamma\texttt{).}$
(2) $\texttt{res(q}_j\texttt{).}$ for each resource-symbol $\texttt{q}_j$
(3) $\texttt{p :- L}_1\texttt{, ..., L}_n\texttt{, fired(n}_\gamma\texttt{).}$
(4) $\texttt{:- }\overline{\texttt{L}_i}\texttt{, fired(n}_\gamma\texttt{).}$ for $j = 1, \ldots, n$
(5) $\texttt{fired(n}_\gamma\texttt{) :- use\_n(n}_\gamma\texttt{,1,q}_1\texttt{,a}_1\texttt{),...,use\_n(n}_\gamma\texttt{,k,q}_k\texttt{,a}_k\texttt{).}$
(6) $\texttt{cons(n}_\gamma\texttt{,j,q}_j\texttt{,a}_j\texttt{).}$ for $j = 1, \ldots, k$
(7) $\texttt{:- 1\{use\_n(n}_\gamma\texttt{,1,q}_1\texttt{,a}_1\texttt{),...,use\_n(n}_\gamma\texttt{,k,q}_k\texttt{,a}_k\texttt{)\}}k-1\texttt{.}$

where $\texttt{n}_\gamma$ is a new constant uniquely associated to $\gamma$. The cardinality constraint used in rule (7) (see, [16] for a detailed description), imposes that either none or all of the literals $\texttt{use\_n(n}_\gamma\texttt{,1,q}_1\texttt{,a}_1\texttt{),...,use\_n(n}_\gamma\texttt{,k,q}_k\texttt{,a}_k\texttt{)}$ are true. Notice the use of a specific argument of the predicate `cons` (namely, the second one) in order to distinguish among different occurrences of identical amount-atoms by indexing the amount-atoms of the rule; we assume defined a domain predicate $\texttt{idx(0..}m\texttt{)}$, where $m$ is the maximum number of amount-atoms in a rule. For a rule of the form

$$q_0{:}a_0 \leftarrow L_1, \ \ldots, \ L_n, \ q_1{:}a_1, \ \ldots, \ q_k{:}a_k$$

[5] Consider, for instance, the two variants of the program $P_i$: {`val(-5..8). g. g:-num(`$i$`). num(`$i$`):-g. nump(X):-num(X),val(X).`}, obtained for $i = 3$ and $i = -3$. `Lparse` (vers. 1.1.1) produces unexpected answers: `nump(`$i$`)` belongs to the answer set only for $i = 3$. (Similar behaviors have been reported in [9, Sec. 3.1].)

[6] Here we are encoding assuming lparse as front-end of the solver.

the encoding changes as follows:

(4') :- $\overline{\mathtt{L}_i}$, fired($\mathtt{n}_\gamma$). for $j = 1, \ldots, n$
(5') fired($\mathtt{n}_\gamma$) :- use_p($\mathtt{n}_\gamma$,0,$\mathtt{q}_0$,$\mathtt{a}_0$),
use_n($\mathtt{n}_\gamma$,1,$\mathtt{q}_1$,$\mathtt{a}_1$), ..., use_n($\mathtt{n}_\gamma$,k,$\mathtt{q}_k$,$\mathtt{a}_k$).
(6') prod($\mathtt{n}_\gamma$,0,$\mathtt{q}_0$,$\mathtt{a}_0$). cons($\mathtt{n}_\gamma$,j,$\mathtt{q}_j$,$\mathtt{a}_j$). for $j = 1, \ldots, k$
(7') :- 1{use_p($\mathtt{n}_\gamma$,0,$\mathtt{q}_0$,$\mathtt{a}_0$), use_n($\mathtt{n}_\gamma$,1,$\mathtt{q}_1$,$\mathtt{a}_1$),...,use_n($\mathtt{n}_\gamma$,k,$\mathtt{q}_k$,$\mathtt{a}_k$)}$k$.

(Where (5')-(7') should be suitably generalized in case of rules having more that one amount-atom in the head.) Notice the separation between positive/produced and negative/consumed amounts by means of the predicate symbols `use_p`/`prod` and `use_n`/`cons`. Negative amount-atoms - $q$:$a$ are treated by exchanging the use of `cons` and `prod`. Ordinary program-rules are left unchanged.

## 5.2 Ensuring correct usage of resources.

The following code imposes correct usage of resources in firing a rule. The ASP fragment made of rules (10)–(20) constitute the core of an inference engine capable of reasoning on resource allocations.

```
(10) use_p(G,I,R,Q) :- fired(G), prod(G,I,R,Q).
(11) use_n(G,I,R,Q) :- fired(G), cons(G,I,R,Q).
(12) :- use_p(G,I,R,N), N!=Q, val(N), prod(G,I,R,Q).
(13) :- use_n(G,I,R,N), N!=Q, val(N), cons(G,I,R,Q).
```

(where `val` encodes a domain predicate for quantities, needed by lparse for domain restriction during grounding). The balance for each resource is evaluated by the following fragment of code. Observe the rather counterintuitive programming trick exploited to evaluate sums (e.g., lines (15)-(17)) due to limitations of lparse's language.

```
(14) 1{notfired(G), fired(G)}1 :- rule(G).
(15) #weight use_n(X,Y,W,Z)=Z.     #weight use_p(X,Y,W,Z)=Z.
(16) consumed(R,N) :- N [use_n(G,I,R,Q):val(Q):idx(I):rule(G)] N,
                      res(R), val(N).
(17) 1 {consumed(T,N) : val(N)} 1 :- res(T).
(18) produced(R,N) :- N [use_p(G,I,R,Q):val(Q):idx(I):rule(G)] N,
                      res(R), val(N).
(19) 1 {produced(T,N) : val(N)} 1 :- res(T).
(20) :- consumed(R,N), produced(R,P), res(R), val(P;N), P<N.
```

Rules (15)-(19) evaluates the global consumed/produced amounts. Such an involved use of weight literals to compute sums could be avoided in systems providing aggregate functions (e.g., dlv). Rule (20) imposes a positive global balance for each resource (cf. (1), page 7). To impose other forms of global constraints on the balance of a specific resource, one has to introduce modified versions of rule (20).

## 5.3 Dealing with multiple firing.

Multiple firing of an r-rule $\gamma$ of the form (4) of Section 3, is rendered through this fragment of ASP code:

```
(21) firings(nγ,N1..N2).                                  for each rule [N1,N2] : γ
(22) 1 {count(G,N):firings(G,N)} 1 :- fired(G), rule(G).
(23) :- not fired(G), count(G,N), firings(G,N), rule(G).
```

where count encodes the mapping $\xi$ which associates a number $n$ of firings to each fired rule, subject to the restriction $N_1 \leqslant n \leqslant N_2$ (imposed by the predicate firings).

To take into account repeated firings in the resource balances, both the encoding of r-rules into ASP (i.e., rules (5),(7), as well as (5'),(7')) and the core inference engine (rules (10)-(13)) have to be slightly modified as follows:

```
(5) fired(nγ):- use_n(nγ,1,q1,C*a1), ..., use_n(nγ,k,qk,C*ak),
               count(nγ,C), firings(nγ,C).
(7) :- 1 {use_n(nγ,1,q1,C*a1), ..., use_n(nγ,k,qk,C*ak)} k−1,
       count(nγ,C), firings(nγ,C).
(10) use_p(G,I,R,C*Q) :- fired(G), count(G,C),
                         firings(G,C), prod(G,I,R,Q).
(11) use_n(G,I,R,C*Q) :- fired(G), count(G,C),
                         firings(G,C), cons(G,I,R,Q).
(12) :- use_p(G,I,R,N), N!=C*Q, val(N), count(G,C),
        firings(G,C), prod(G,I,R,Q).
(13) :- use_n(G,I,R,N), N!=C*Q, val(N), count(G,C),
        firings(G,C), cons(G,I,R,Q).
```

### 5.4 Rendering of budget policies.

The translation described so far realizes 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. This rules are added to the inference engine:

```
(30) balance(R,P-N) :- res(R), consumed(R,N), produced(R,P),
                       P>=N, val(P;N).
(31) #weight cons(X,Y,W,Z)=Z.       #weight prod(X,Y,W,Z)=Z.
(32) consumes(G,R,N) :- N [cons(G,Idx,R,Q) : val(Q) : idx(Idx)] N,
                        rule(G), res(R), val(N).
(33) 1 {consumes(G,T,N):val(N)} 1 :- rule(G), res(T).
(34) produces(G,R,N) :- N [prod(G,Idx,R,Q) : val(Q) : idx(Idx)] N,
                        rule(G), res(R), val(N).
(35) 1 {produces(G,T,N):val(N)} 1 :- rule(G), res(T).
(36) :- enabled(G), rule(G).
(37) enbld(G,R) :- balance(R,U), produces(G,R,P), consumes(G,R,N),
                  (U+P)>=N, val(U;N;P), rule(G), res(R).
```

Note that the predicate balance encodes the component $\varphi$ of the answer set of r-programs (see page 8). Rule (36) ensures that all fireable r-rule have been fired. Instances of (38) and (39) are added to the translation of each r-rule $\gamma$:

```
(38) enabled(nγ) :- not fired(nγ), L1, ..., Ln,
                  enbld(nγ,q1), ..., enbld(nγ,qk).
```

```
(39) enabled(nγ) :- count(nγ,C), firings(nγ,More), firings(nγ,C),
                    enbld(nγ,q1), ..., enbld(nγ,qk), L1, ..., Ln.
```

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 instances of the predicate `rule(·)`).

It is also easy to single-out an answer set corresponding to maximal numbers of firings by using the `maximize` instruction offered by smodels. For example (using `num` as auxiliary domain predicate):

```
#weight count(X,Y)=Y.
maximize[count(G,C):num(C):rule(G)].
```

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 the general case it suffices to multiply the number of firings of each r-rule by its cost.

The above-outlined translation from RASP into ASP, has been checked on several examples. For the time being, the translation is done by hand and the implementation of an automated compiler is part of our future works. We conclude this section by reporting on the output of smodels for a short example.

**Example 22** *Assembling different PCs require 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 PC that are assembled.*

```
        cpu:15.                   hd:25.                 fan:13.
        raid:4.                   mother:7.              ram:20.
[1,3]:  pc(server):1 :- cpu:2, hd:6, fan:3, raid:1, mother:1, ram:4.
[2,6]:  pc(desk):1 :- cpu:1, hd:2, fan:1, mother:1, ram:2.
```

*By translation into ASP, under the Prodigal policy, and by running the ASP solver, we obtain the models (the two rules are referred as g1 and g2, resp.):*

```
Answer 1:  balance(pc(server),1) balance(pc(desk),6) balance(cpu,7) balance(hd,7)
balance(fan,4) balance(raid,3) balance(mother,0) balance(ram,4) count(g1,1) count(g2,6)
fired(g2) fired(g1) ...
Answer 2:  balance(pc(server),3) balance(pc(desk),3) balance(cpu,6) balance(hd,1)
balance(fan,1) balance(raid,1) balance(mother,1) balance(ram,2) count(g1,3) count(g2,3)
fired(g2) fired(g1) ...
Answer 3:  balance(pc(server),2) balance(pc(desk),5) balance(cpu,6) balance(hd,3)
balance(fan,2) balance(raid,2) balance(mother,0) balance(ram,2) count(g1,2) count(g2,5)
fired(g2) fired(g1) ...
```

*In the second solution, for instance, three desktop 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.*

## 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.

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 7).

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. In this translation we have used specific features offered by the front-end lparse. However, the approach can be easily rephrased for other solvers [2] and other grounders could be exploited. As an example we mention the grounder GrinGo [10]. Being designed to represent a viable alternative to lparse, GrinGo can process lparse language. Moreover, it can plainly deal with negative integers and does not suffer, for instance, from the problems mentioned in Section 5.1 (cf, footnote 5). Preliminary tests run with a beta version of GrinGo (being developed at the time we are writing this article) demonstrated its potential efficiency. His availability will permit a simpler encoding of RASP programs in ASP code.

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.

We intend to reason about possible meanings of negation for amount-atoms. Actually, some forms of "negation" can be expressed by means of the constraints on global resource balance described in Section 3: 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 comparison with other frameworks suited for modeling production processes, resource management, and quantitative reasoning (such as Linear Logic, for instance) represents an interesting subject for future investigations.

## References


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

[2] 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`.

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

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

[5] 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.

[6] E. Corona and M. Osorio. The A-Pol system. In M. D. 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.

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

[8] 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.

[9] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2006.

[10] 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.

[11] 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.

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

[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

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

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

[16] 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.

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

[18] T. Syrjänen. Lparse 1.0 user's manual, 2000. http://www.tcs.hut.fi/Software/smodels.

[19] WASP–WP5 Report: Model applications and proofs-of-concept, 2005. Working Group on Answer Set Programming (WASP): Web site: http://www.kr.tuwien.ac.at/projects/WASP/report.html.