Reactive Answer Set Programming

Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub*

Institut für Informatik, Universität Potsdam

Abstract. We introduce the first approach to Reactive Answer Set Programming, aiming at reasoning about real-time dynamic systems running online in changing environments. We start by laying the theoretical foundations by appeal to module theory. With this, we elaborate upon the composition of the various offline and online programs in order to pave the way for stream-driven grounding and solving. Finally, we describe the implementation of a reactive ASP solver, *oclingo*.

1 Introduction

Answer Set Programming (ASP; [1]) has become a popular declarative problem solving paradigm, facing a growing number of increasingly complex applications. So far, however, ASP systems are designed for offline usage, lacking any online capacities. We address this shortcoming and propose a reactive approach to ASP that allows us to implement real-time dynamic systems running online in changing environments. This new technology paves the way for applying ASP in many new challenging areas, dealing with agents, (ro)bots, policies, sensors, etc. The common ground of these areas is reasoning about dynamic systems incorporating online data streams.

For capturing dynamic systems, we take advantage of *incremental logic programs* [2], consisting of a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter t ranging over the natural numbers. In view of this, we sometimes denote P and Q by $P[t]$ and $Q[t]$. The base program B is meant to describe static knowledge, independent of parameter t. The role of P is to capture knowledge accumulating with increasing t, whereas Q is specific for each value of t. Roughly speaking, we are interested in finding an answer set of the program $B \cup \bigcup_{1 \leq j \leq i} P[t/j] \cup Q[t/i]$ for some (minimum) integer $i \geq 1$.

As a motivating example, consider a very simple elevator controller accepting requests to go to a certain floor whenever it is not already at this floor. At each step, the elevator moves either up or down by one floor. If it reaches a floor for which a request exists, it serves the request automatically until its goal to serve all requests is fulfilled. This functionality is specified by the incremental logic program $(B, P[t], Q[t])$ in Fig. 1.[1] The answer set of the program $B \cup P[t/1] \cup Q[t/1]$ is $B \cup \{atFloor(2,1), goal(1)\}$.[2] The elevator moves one floor and sees its goal fulfilled because there were no requests.

Observe that atoms of the form $request(F, t)$, representing incoming requests, are not defined by $P[t]$; that is, they do not occur in the head of any rule in $P[t]$. In fact,

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

[1] We use shorthands '1{...}1', '..', ';', and ':' following the syntax of *gringo* (cf. [3]).

[2] For simplicity, we identify facts with atoms.

$$B = \left\{ \begin{array}{c} floor(1..3) \leftarrow \\ atFloor(1,0) \leftarrow \end{array} \right\}$$

$$P[t] = \left\{ \begin{array}{c} 1\,\{atFloor(F{-}1;F{+}1,t)\}\,1 \leftarrow atFloor(F,t{-}1), floor(F) \\ \leftarrow atFloor(F,t), not\ floor(F) \\ requested(F,t) \leftarrow request(F,t), floor(F), not\ atFloor(F,t) \\ requested(F,t) \leftarrow requested(F,t{-}1), floor(F), not\ atFloor(F,t) \\ goal(t) \leftarrow not\ requested(F,t) : floor(F) \end{array} \right\}$$

$$Q[t] = \left\{ \leftarrow not\ goal(t) \right\}\ .$$

Fig. 1. Incremental logic program for elevator control

requests are coming from outside the system, and their occurrences cannot be foreseen within an incremental program. Assume we get the request

$$E[1] = \{request(3,1) \leftarrow\} \tag{1}$$

telling our controller that a request to serve floor 3 occurred at time 1. While adding $E[1]$ to the above program yields no answer set, we get one from program $B \cup P[t/1] \cup P[t/2] \cup Q[t/2] \cup E[1]$, in which the elevator takes two steps to move to the third floor:[2]

$$B \cup E[1] \cup \{requested(3,1), atFloor(2,1), atFloor(3,2), goal(2)\}\ . \tag{2}$$

In fact, reasoning is driven by successively arriving events. No matter when a request arrives, its logical time step is aligned with the ones used in the incremental program. In this way, an event like (1) complements the domain description in Fig. 1 and initiates the subsequent search for an answer set as in (2). The next answer set computation is started by the following event, that is, upon the next request. As a particular feature of this methodology, observe that some rules in an encoding like $P[t]$ in Fig. 1 stay inactive until they get triggered by an event as in (1) adding $request(3,1)$ as a fact.

Grounding and solving in view of possible yet unknown future events constitutes a major technical challenge. For guaranteeing redundancy-freeness, the continuous integration of new program parts has to be accomplished without reprocessing previously treated programs. Also, simplifications related to events must be suspended until they become decided. Once this is settled, our approach leaves room for various application scenarios. While the above example is inspired by Cognitive Robotics [5], our approach may just as well serve as a platform for Autonomous Agent Architectures [6], Policy reasoning [7], or Sensor Fusion in Ambient Artificial Intelligence [8]. All in all, our approach thus serves as a domain-independent framework providing a sort of middle-ware for specific application areas rather than proposing a domain-specific solution.

2 Background

This section provides a brief introduction of answer sets of logic programs with choice rules and integrity constraints (see [1, 9] for details). A *rule* is an expression of the form

$$h \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{3}$$

where a_i, for $1 \leq m \leq n$, is an *atom* of the form $p(t_1, \ldots, t_k)$, and $t_1, \ldots, t_k$ are terms, viz., constants, variables, or functions. For a rule r as in (3), the *head* h of r is either an atom, a *cardinality constraint* of the form $l\,\{h_1, \ldots, h_k\}\,u$ in which l, u are integers and $h_1, \ldots, h_k$ are atoms, or the special symbol $\bot$. If h is a cardinality constraint, we call r a *choice rule*, and an *integrity constraint* if $h = \bot$. We denote the atoms occurring in h by $head(r)$, i.e., $head(r) = \{h\}$ if h is an atom, $head(r) = \{h_1, \ldots, h_k\}$ if $h = l\,\{h_1, \ldots, h_k\}\,u$, and $head(r) = \emptyset$ if $h = \bot$. In the following, we sometimes write h_r to refer to the head h of r, and we skip $\bot$ when writing an integrity constraint. The atoms occurring positively and negatively, respectively, in the *body* of r are denoted by $body(r)^+ = \{a_1, \ldots, a_m\}$ and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$.

A *logic program* R is a set of rules of the form (3). By $atom(R)$, we denote the set of all atoms occurring in R, and $head(R) = \bigcup_{r \in R} head(r)$ is the collection of head atoms in R. The set of all ground terms constructible from the constants and function symbols that occur in R (if there is no constant, just add an arbitrary one) forms the *Herbrand universe* of R. The *ground instance* of R, denoted by $grd(R)$, is the set of all ground rules constructible from rules $r \in R$ by substituting every variable in r with some element of the Herbrand universe of R.[3] A set X of ground atoms satisfies a ground rule r of the form (3) if $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$ imply that $h \in X$ or $h = l\,\{h_1, \ldots, h_k\}\,u$ and $l \leq |\{h_1, \ldots, h_k\} \cap X| \leq u$. We call X a *model* of R if X satisfies every rule $r \in grd(R)$. The *reduct* of R relative to X is $R^X = \{a \leftarrow body(r)^+ \mid r \in grd(R), a \in head(r) \cap X, body(r)^- \cap X = \emptyset\}$; X is an *answer set* of R if X is a model of R such that no proper subset of X is a model of R^X.

3 Reactive Answer Set Programming

In this section, we augment the concept of an incremental logic program with asynchronous information, refining the statically available knowledge. To this end, we characterize the constituents of the combined logic program including schematic as well as online parts, below called online progression.

An *online progression* represents a stream of events and inquiries. While entire event streams are made available for reasoning, inquiries act as punctual queries.

Definition 1. *We define an online progression $(E_i[e_i], F_i[f_i])_{i \geq 1}$ as a sequence of pairs of logic programs E_i, F_i with associated positive integers e_i, f_i.*

An online progression is asynchronous in distinguishing stream positions like i from (logical) time stamps. Hence, each event E_i and inquiry F_i includes a particular time stamp e_i or f_i, respectively, indicated by writing $E_i[e_i]$ and $F_i[f_i]$. Such time stamps are essential for synchronization with parameters in the underlying (incremental) logic programs. Note that different events and/or inquiries may refer to the same time stamp.

Definition 2. *Let $(E_i[e_i], F_i[f_i])_{1 \leq i \leq j}$ be a finite online progression and $(B, P[t], Q[t])$ be an incremental logic program. We define*

[3] We also assume that built-ins of grounders like *lparse* and *gringo* (cf. [4, 3]), such as arithmetic functions, are evaluated upon instantiation.

1. *the k-expanded logic program of* $(E_i[e_i], F_i[f_i])_{1 \leq i \leq j}$ *wrt* $(B, P[t], Q[t])$ *as*

$$R_{j,k} = B \cup \bigcup_{1 \leq i \leq k} P[t/i] \cup Q[t/k] \cup \bigcup_{1 \leq i \leq j} E_i[e_i] \cup F_j[f_j] \tag{4}$$

   for each k such that $1 \leq e_1, \ldots, e_j, f_j \leq k$, and
2. *a reactive answer set of* $(E_i[e_i], F_i[f_i])_{1 \leq i \leq j}$ *wrt* $(B, P[t], Q[t])$ *as an answer set of a k-expanded logic program $R_{j,k}$ of* $(E_i[e_i], F_i[f_i])_{1 \leq i \leq j}$ *for a (minimum) $k \geq 1$.*

The incremental program constitutes the offline counterpart of an online progression; it is meant to provide a general (schematic) description of an underlying dynamic system. The parameter k represents a valid horizon accommodating all occurring events and inquiries. Thus, it is bound from below by the time stamps occurring in the online progression. The goal is then to find a (minimum) horizon k such that $R_{j,k}$ has an answer set, often in view of satisfying the global query $Q[t/k]$. In addition, inquiries, specific to each j, can be used for guiding answer set search. Unlike this, the whole stream $(E_i[e_i])_{1 \leq i \leq j}$ of events is taken into account. Observe that the number j of events is independent of the horizon k. Finally, it is important to note that the above definition of an expanded program is static because its parameters are fixed. The next section is dedicated to the online evolution of reactive logic programs, characterizing the transitions from $R_{j,k}$ to $R_{j+1,k}$ and $R_{j,k}$ to $R_{j,k+1}$ in terms of constituent programs.

4 Reactive Modularity

For providing a clear interface between the various programs and guaranteeing their compositionality, we build upon the concept of a *module* [10], $\mathbb{P}$, being a triple (P, I, O) consisting of a (ground) program P and sets I, O of ground atoms such that $I \cap O = \emptyset$, $atom(P) \subseteq I \cup O$, and $head(P) \subseteq O$. The elements of I and O are called *input* and *output* atoms, also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively; similarly, we refer to P by $P(\mathbb{P})$. The *join* of two modules $\mathbb{P}$ and $\mathbb{Q}$, denoted by $\mathbb{P} \sqcup \mathbb{Q}$, is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q})\,,\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P}))\,,\ O(\mathbb{P}) \cup O(\mathbb{Q})\ )\,,$$

provided that $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and there is no strongly connected component in the positive dependency graph of $P(\mathbb{P}) \cup P(\mathbb{Q})$, i.e., $(atom(P(\mathbb{P}) \cup P(\mathbb{Q})), \{(a, b) \mid r \in P(\mathbb{P}) \cup P(\mathbb{Q}), a \in head(r), b \in body(r)^+\})$, that shares atoms with both $O(\mathbb{P})$ and $O(\mathbb{Q})$. A set X of atoms is an *answer set* of a module $\mathbb{P} = (P, I, O)$ if X is a (standard) answer set of $P \cup \{a \leftarrow \mid a \in I \cap X\}$; we denote the set of all answer sets of $\mathbb{P}$ by $AS(\mathbb{P})$. For two modules $\mathbb{P}$ and $\mathbb{Q}$, the *composition* of their answer sets is $AS(\mathbb{P}) \bowtie AS(\mathbb{Q}) = \{X_\mathbb{P} \cup X_\mathbb{Q} \mid X_\mathbb{P} \in AS(\mathbb{P}), X_\mathbb{Q} \in AS(\mathbb{Q}), X_\mathbb{P} \cap (I(\mathbb{Q}) \cup O(\mathbb{Q})) = X_\mathbb{Q} \cap (I(\mathbb{P}) \cup O(\mathbb{P}))\}$. The module theorem [10] shows that the semantics of $\mathbb{P}$ and $\mathbb{Q}$ is *compositional* if their join is defined, i.e., if $\mathbb{P} \sqcup \mathbb{Q}$ is well-defined, then $AS(\mathbb{P} \sqcup \mathbb{Q}) = AS(\mathbb{P}) \bowtie AS(\mathbb{Q})$.

For turning programs into modules, we follow [2] and associate in Definition 3 a (non-ground) program P and a set I of (ground) input atoms with a module, denoted by $\mathbb{P}(I)$, imposing certain restrictions on the ground program induced by P. To this end, for a ground program P and a set X of ground atoms, define $P|_X$ as

$$\{h_r \leftarrow body(r)^+ \cup L \mid r \in P, body(r)^+ \subseteq X, L = \{not\ c \mid c \in body(r)^- \cap X\}\}\,.$$

Note that $P|_X$ projects the bodies of rules in P to the atoms of X. If a body contains an atom outside X, either the corresponding rule or literal is removed, depending on whether the atom occurs positively or negatively. This allows us to associate (non-ground) programs with (ground) modules, as proposed in [2].

Definition 3. *Let P be a logic program and I be a set of ground atoms. We define $\mathbb{P}(I)$ as the module $(grd(P)|_Y, I, head(grd(P)|_X))$, where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.*

The full ground instantiation $grd(P)$ of P is projected onto inputs and atoms defined in $grd(P)$. The head atoms of this projection, viz., $head(grd(P)|_{I \cup head(grd(P))})$, serve as output atoms and are used to simplify $grd(P)$, sparing only input and output atoms.

Unlike offline incremental ASP [2], its online counterpart deals with external knowledge acquired asynchronously. When constructing a ground module, we can thus no longer expect all of its atoms to be defined by the (ground) rules inspected so far. Rather, atoms may be defined by an online progression later on. To accommodate this, potential additions need to be reflected and exempted from program simplifications, as usually applied wrt (yet) undefined atoms. To this end, we assume in the following each (non-ground) program P to come along with some set of explicit ground input atoms (cf. the `#external` declaration described in Section 5), referred to by I_P. Such atoms provide "hooks" for online progressions to later incorporate new knowledge into an existing program part. Note that we could simply let $I_P = \emptyset$ for all program slices P to resemble offline incremental ASP.

We make use of the join to formalize the compositionality of instantiated modules induced by the respective programs in Definition 2.

Definition 4. *We define an online progression $(E_i[e_i], F_i[f_i])_{i \geq 1}$ as modular wrt an incremental logic program $(B, P[t], Q[t])$, if the modules*

$$\begin{aligned}
\mathbb{P}_0 &= \mathbb{B}(I_B) & \mathbb{P}_n &= \mathbb{P}_{n-1} \sqcup \mathbb{P}[t/n](O(\mathbb{P}_{n-1}) \cup I_{P[t/n]}) \\
\mathbb{E}_0 &= (\emptyset, \emptyset, \emptyset) & \mathbb{E}_n &= \mathbb{E}_{n-1} \sqcup \mathbb{E}_n[e_n](O(\mathbb{P}_{e_n}) \cup O(\mathbb{E}_{n-1}) \cup I_{E_n[e_n]}) \\
\mathbb{R}_{j,k} &= \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup \mathbb{Q}[t/k](O(\mathbb{P}_k) \cup I_{Q[t/k]}) \sqcup \mathbb{F}_j[f_j](O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]})
\end{aligned}$$

are well-defined for all $j, k \geq 1$ such that $e_1, \ldots, e_j, f_j \leq k$.

In detail, this definition inspects the joins $\mathbb{P}_n$ and $\mathbb{E}_n$ of instantiated cumulative modules obtained from $P[t/n]$ and $E_n[e_n]$, respectively, for all $n \geq 0$. The former takes the instantiation of the static program B as its base case, where the input I_B related to B is considered by the instantiation. A module $\mathbb{P}_{n-1}$ obtained in this way is then joined with the instantiation of $P[t/n]$ relative to the atoms defined by preceding cumulative program slices, viz., $O(\mathbb{P}_{n-1})$, and the specific inputs $I_{P[t/n]}$, thus obtaining the next combined module $\mathbb{P}_n$. Observe that this join is independent of an online progression, yet the inputs collected over successive join operations provide an interface for online progressions to refine the available knowledge.

The join of the instantiations of online progressions' cumulative parts $E_n[e_n]$ starts from the empty module $\mathbb{E}_0$, given that the first event is provided for $n = 1$. Then, $\mathbb{E}_{n-1}$ is joined with the instantiation of $E_n[e_n]$ relative to the defined atoms requested via e_n, viz., $O(\mathbb{P}_{e_n})$, the atoms defined by preceding cumulative parts of the online progression,

i.e., $O(\mathbb{E}_{n-1})$, and finally the particular inputs $I_{E_n[e_n]}$. As before, the latter provide means to refine the information gathered in the combined module $\mathbb{E}_n$.

A full module $\mathbb{R}_{j,k}$ incorporates all information accumulated in $\mathbb{P}_k$ and $\mathbb{E}_j$ as well as the volatile (query) parts $Q[t/k]$ and $F_j[f_j]$ of the incremental logic program and the online progression, respectively. Their instantiations consider all atoms defined by cumulative incremental program slices up to k or f_j, i.e., $O(\mathbb{P}_k)$ or $O(\mathbb{P}_{f_j})$, respectively, the specific inputs $I_{Q[t/k]}$ and $I_{F_j[f_j]}$, and in the case of $F_j[f_j]$ also the atoms $O(\mathbb{E}_j)$ defined by events of the online progression. Note that $O(\mathbb{E}_j)$ is not used as input for instantiating $Q[t/k]$, which reflects its role of belonging to an incremental logic program that is independent of and also invariant under particular online progressions. However, the explicit inputs $I_{Q[t/k]}$ (and $I_{F_j[f_j]}$) still admit passing information between (volatile parts of) the incremental logic program and an online progression.

The condition characterizing modularity of incremental programs and online progressions is that $\mathbb{R}_{j,k}$ must be well-defined for all $j, k \geq 1$, viz., each instantiation must yield a module and each join must be defined, where the requirement $e_1, \ldots, e_j, f_j \leq k$ makes sure that the slices of an incremental program requested in an online progression contribute to $\mathbb{R}_{j,k}$. However, note that k is not bound to be $\max\{e_1, \ldots, e_j, f_j\}$; rather, it can be increased beyond that as needed (for obtaining an answer set).

As an example, let us instantiate the incremental logic program in Fig. 1. While its static and query part, B and $Q[t]$, respectively, do not make use of particular inputs ($I_B = I_{Q[t]} = \emptyset$), the incremental part $P[t]$ relies on atoms of the form $request(F, t)$, which are not defined by $P[t]$. Unlike offline incremental ASP, where undefined atoms do not belong to the instantiation of a program slice, they must now be preserved to react to asynchronous requests. Accordingly, we let $I_{P[t]} = \{request(1, t), request(2, t), request(3, t)\}$; here, the first argument of an input atom is a floor and the second is the incremental parameter. Given the described inputs, the following ground modules are derived from the incremental program in Fig. 1 and contribute to $\mathbb{R}_{1,2}$:

$$\mathbb{P}_0 = \mathbb{B}(\emptyset) = (\, B, \emptyset, head(B)\,)$$
$$\text{where } B = \{\, floor(1) \leftarrow \quad floor(2) \leftarrow \quad floor(3) \leftarrow \quad atFloor(1,0) \leftarrow \,\}$$
$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{P}[t/1](O(\mathbb{P}_0) \cup I_{P[t/1]}) = \big(\, P(\mathbb{P}_0) \cup P_1, I_{P[t/1]}, O(\mathbb{P}_0) \cup head(P_1)\,\big)$$
$$\text{where } P_1 = \left\{ \begin{array}{rl} & 1\,\{atFloor(0,1), atFloor(2,1)\}\,1 \leftarrow atFloor(1,0), floor(1) \\ & \leftarrow atFloor(0,1) \\ & \leftarrow atFloor(2,1), not\ floor(2) \\ requested(1,1) & \leftarrow request(1,1), floor(1) \\ requested(2,1) & \leftarrow request(2,1), floor(2), not\ atFloor(2,1) \\ requested(3,1) & \leftarrow request(3,1), floor(3) \\ goal(1) & \leftarrow not\ requested(1,1), \\ & \quad not\ requested(2,1), \\ & \quad not\ requested(3,1) \end{array} \right\}$$

Note that the program in $\mathbb{P}_1$, viz. $P_1 = grd(P[t/1])|_{head(grd(P[t/1])\cup B)\cup I_{P[t/1]}}$, is obtained by simplifying $grd(P[t/1])$ relative to the output of the preceding module, thereby, sparing the inputs $I_{P[t/1]}$ from simplifications (cf. Definition 3 and 4). For instance, input atoms of the form $request(F, 1)$ are not eliminated from P_1, while ground

rules including undefined non-input atoms of the form $requested(F, 0)$ in their positive bodies do not contribute to P_1. The same considerations apply to program P_2 of $\mathbb{P}_2$ below. That is, $P_2 = grd(P[t/2])|_{head(grd(P[t/2])\cup P_1 \cup B)\cup I_{P[t/2]}}$ is obtained by simplifying $grd(P[t/2])$ relative to the preceding outputs, while sparing the inputs $I_{P[t/2]}$:

$$\begin{aligned}\mathbb{P}_2 &= \mathbb{P}_1 \sqcup \mathbb{P}[t/2](O(\mathbb{P}_1) \cup I_{P[t/2]}) \\ &= \big(\, P(\mathbb{P}_1) \cup P_2, I(\mathbb{P}_1) \cup I_{P[t/2]}, O(\mathbb{P}_1) \cup head(P_2) \,\big)\end{aligned}$$

where

$$P_2 = \left\{ \begin{array}{r} 1\,\{atFloor(1,2), atFloor(3,2)\}\,1 \leftarrow atFloor(2,1), floor(2) \\ \leftarrow atFloor(1,2), not\ floor(1) \\ \leftarrow atFloor(3,2), not\ floor(3) \\ requested(1,2) \leftarrow request(1,2), floor(1), not\ atFloor(1,2) \\ requested(2,2) \leftarrow request(2,2), floor(2) \\ requested(3,2) \leftarrow request(3,2), floor(3), not\ atFloor(3,2) \\ requested(1,2) \leftarrow requested(1,1), floor(1), not\ atFloor(1,2) \\ requested(2,2) \leftarrow requested(2,1), floor(2) \\ requested(3,2) \leftarrow requested(3,1), floor(3), not\ atFloor(3,2) \\ goal(2) \leftarrow not\ requested(1,2), \\ not\ requested(2,2), \\ not\ requested(3,2) \end{array} \right\}$$

$$\mathbb{Q}[t/2](O(\mathbb{P}_2)) = (\,\{\leftarrow not\ goal(2)\}, O(\mathbb{P}_2), \emptyset\,)$$

To complete $\mathbb{R}_{1,2}$, we further join $\mathbb{P}_2$ and $\mathbb{Q}[t/2](O(\mathbb{P}_2))$ with the module $\mathbb{E}_1 = (\,\{request(3,1)\ \leftarrow\}, \emptyset, \{request(3,1)\}\,)$ stemming from the online progression $(\{request(3,1)\ \leftarrow\}, \emptyset)$, capturing the request $E[1]$ in (1). In view of its five input atoms, $request(1,1)$, $request(2,1)$, $request(1,2)$, $request(2,2)$, and $request(3,2)$, the full module $\mathbb{R}_{1,2}$ has four answer sets, obtained by augmenting the answer set shown in (2) with an arbitrary subset of $\{request(2,1), request(3,2)\}$. (That is, fictitious requests along the way of the elevator do not preclude it from serving floor 3, as required in view of $request(3,1)\ \leftarrow$.) However, note that the answer set in (2) is the only one that does not assume any of the residual input atoms of $\mathbb{R}_{1,2}$ to hold.

Regarding the formal properties of (modular) incremental logic programs and online progressions, we have that the module theorem [10] applies to instantiated modules contributing to $\mathbb{R}_{j,k}$.

Proposition 1 (Compositionality). *Let $(B, P[t], Q[t])$ be an incremental logic program, $(E_i[e_i], F_i[f_i])_{i\geq 1}$ be an online progression, $j, k \geq 1$ be such that $e_1, \ldots, e_j, f_j \leq k$, and $\mathbb{R}_{j,k}$ as well as $\mathbb{P}_n, \mathbb{E}_n$, for $n \geq 0$, be as in Definition 4.*

If $(E_i[e_i], F_i[f_i])_{i\geq 1}$ is modular wrt $(B, P[t], Q[t])$, then we have that

$$\begin{aligned}AS(\mathbb{R}_{j,k}) \;=\; & AS(\mathbb{P}_0) \bowtie AS(\mathbb{P}[t/1](O(\mathbb{P}_0) \cup I_{P[t/1]})) \bowtie \cdots \bowtie \\ & AS(\mathbb{P}[t/k](O(\mathbb{P}_{k-1}) \cup I_{P[t/k]})) \bowtie \\ & AS(\mathbb{E}_0) \bowtie AS(\mathbb{E}_1[e_1](O(\mathbb{P}_{e_1}) \cup O(\mathbb{E}_0) \cup I_{E_1[e_1]})) \bowtie \cdots \bowtie \\ & AS(\mathbb{E}_j[e_j](O(\mathbb{P}_{e_j}) \cup O(\mathbb{E}_{j-1}) \cup I_{E_j[e_j]})) \bowtie \\ & AS(\mathbb{Q}[t/k](O(\mathbb{P}_k) \cup I_{Q[t/k]})) \bowtie \\ & AS(\mathbb{F}_j[f_j](O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]}))\,.\end{aligned}$$

Note that compositionality holds wrt instantiated modules obtained by passing information (output atoms) from one module to another as specified in Definition 4.

Another question of interest concerns conditions under which the answer sets of a module $\mathbb{R}_{j,k}$ match the ones of a k-expanded logic program $R_{j,k}$, being the union of incremental logic program slices and programs of an online progression (cf. Definition 2). The major difference between both constructions is that modules contributing to $\mathbb{R}_{j,k}$ are instantiated successively wrt an evolving Herbrand universe, while the (non-ground) programs of $R_{j,k}$ share a Herbrand universe. To this end, we next provide a sufficient condition under which incremental and single-pass grounding yield similar answer sets. The idea is to require that, in the successive construction of $\mathbb{R}_{j,k}$, atoms that can already be used before they become defined must be declared to be inputs.

We say that an incremental logic program $(B, P[t], Q[t])$ and an online progression $(E_i[e_i], F_i[f_i])_{i\geq 1}$ are *mutually revisable* if the following conditions hold for all $n \geq 1$:

1. $atom(grd(B)) \cap head(grd(\bigcup_{i\geq 1}(P[t/i] \cup Q[t/i] \cup E_i[e_i] \cup F_i[f_i]))) \subseteq I_B$,
2. $atom(grd(P[t/n])) \cap head(grd(\bigcup_{i>n} P[t/i] \cup \bigcup_{i\geq n} Q[t/i] \cup \bigcup_{i\geq 1}(E_i[e_i] \cup F_i[f_i]))) \subseteq I_{P[t/n]}$,
3. $atom(grd(Q[t/n])) \cap head(grd(\bigcup_{i\geq 1}(E_i[e_i] \cup F_i[f_i]))) \subseteq I_{Q[t/n]}$,
4. $atom(grd(E_n[e_n])) \cap head(grd(\bigcup_{i>e_n} P[t/i] \cup \bigcup_{i\geq e_n} Q[t/i] \cup \bigcup_{i>n} E_i[e_i] \cup \bigcup_{i\geq n} F_i[f_i])) \subseteq I_{E_n[e_n]}$, and
5. $atom(grd(F_n[f_n])) \cap head(grd(\bigcup_{i>f_n} P[t/i] \cup \bigcup_{i\geq f_n} Q[t/i])) \subseteq I_{F_n[f_n]}$.

Observe that atoms belonging to the ground instance of the static program B or a cumulative program slice $P[t/n]$ must be consumed as inputs, i.e., belong to I_B or $I_{P[t/n]}$, respectively, if they can be defined by subsequent cumulative or query programs, or by the online progression. The latter condition must likewise hold for a query program $Q[t/n]$, which can however ignore atoms defined by program slices $I_{P[t/i]}$, for $i > n$, because a different query program $Q[t/i]$ will then be used instead. For the programs $E_n[e_n]$ and $F_n[f_n]$ of the online progression, we similarly require in 4. and 5. that all atoms in their ground instances that can be defined by the incremental program in a step $i > e_n$ or $i > f_n$ (also $i = e_n$ or $i = f_n$ for $Q[t/i]$), respectively, must be contained in $I_{E_n[e_n]}$ or $I_{F_n[f_n]}$. The inputs of an event $E_n[e_n]$ also need to include atoms that can be defined later by the online progression, i.e., in $E_i[e_i]$ or $F_i[f_i]$ for $i > n$ (also $i = n$ for $F_i[f_i]$). In summary, if all requirements of mutual revisability are met, the instantiated modules in $\mathbb{R}_{j,k}$ are via their inputs susceptible to atoms defined subsequently, as in the case of instantiating the full collection $R_{j,k}$ of (non-ground) programs in a single pass.

The following result formalizes the correspondence between the answer sets of $R_{j,k}$ and the ones of $\mathbb{R}_{j,k}$ not including input atoms, provided that mutual revisability applies.

Proposition 2 (Instantiation). *Let $(B, P[t], Q[t])$ be an incremental logic program, $(E_i[e_i], F_i[f_i])_{i\geq 1}$ be a modular online progression wrt $(B, P[t], Q[t])$, $j, k \geq 1$ be such that $e_1, \ldots, e_j, f_j \leq k$, $R_{j,k}$ be the k-expanded logic program of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ wrt $(B, P[t], Q[t])$, and $\mathbb{R}_{j,k}$ be as in Definition 4.*

If $(B, P[t], Q[t])$ and $(E_i[e_i], F_i[f_i])_{i\geq 1}$ are mutually revisable, then we have that X is an answer set of $R_{j,k}$ iff X is an answer set of $\mathbb{R}_{j,k}$ such that $X \subseteq O(\mathbb{R}_{j,k})$.

Note that, by letting $I_{P[t]} = \{request(1, t), request(2, t), request(3, t)\}$, Proposition 2 applies to the incremental logic program in Fig. 1 along with the online progression $(\{request(3, 1) \leftarrow\}, \emptyset)$, capturing the request $E[1]$ in (1). In fact, atoms defined

```
#base.
floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).
1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),     floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.
:- not goal(t).
```

Table 1. `elevator.lp`

by $P[t/n]$ do not occur in B or $P[t/i]$ for any $1 \leq i < n$, so that $I_{P[t]}$ is sufficient to reflect facts representing asynchronously arriving requests in instantiated modules. Hence, the answer set in (2) is obtained both for $R_{1,2}$ and $\mathbb{R}_{1,2}$. In fact, it is the only answer set of $\mathbb{R}_{1,2}$ not including any of its residual input atoms, viz., $request(1,1)$, $request(2,1)$, $request(1,2)$, $request(2,2)$, and $request(3,2)$.

To see that incremental instantiation and single pass grounding yield, in general, different semantics, note that, if $I_{P[t]} = \emptyset$, the instantiated modules obtained from $P[t]$ in Fig. 1 do not include any rule containing an atom of the form $request(F,t)$ in the positive body. Then, the answer sets of $\mathbb{R}_{1,2}$ would not yield a schedule to satisfy a request given in an online progression, but merely provide possible moves of the elevator. Unlike this, a request like in (1) would still be served in an answer set of $R_{1,2}$.

5 The Reactive ASP Solver *oclingo*

We implemented a prototypical reactive ASP solver called *oclingo*, which is available at [11] and extends *iclingo* [2] with online functionalities. To this end, *oclingo* acts as a server listening on a port, configurable via its `--port` option upon start-up. Unlike *iclingo*, which terminates after computing an answer set of the incremental logic program it is run on, *oclingo* waits for client requests. To issue such requests, we implemented a separate controller program that sends online progressions to *oclingo* and displays answer sets received in return.

For illustrating the usage of *oclingo*, consider Table 1 displaying the source code representation (`elevator.lp`) of the incremental logic program in Fig. 1. Its three parts are distinguished via the declarations '`#base.`', '`#cumulative t.`', and '`#volatile t.`', respectively, where `t` serves as the parameter. Of particular interest is the declaration preceded by '`#external`', delineating the input to the cumulative part provided by future online progressions (cf. $I_{P[t/n]}$ in Definition 4). In fact, the

Algorithm 1: `osolve`

Input : An incremental logic program $(B, P[t], Q[t])$.
Internal : A grounder GROUNDER and a solver SOLVER.

```
1  i ← 0
2  i_old ← 0
3  j ← 0
4  P_0 ← GROUNDER.ground(B)
5  SOLVER.add(P_0)
6  loop
7      j ← j + 1
8      (E_j, F_j, m_j) ← getExternalKnowledge()
9      while i < m_j do
10         i ← i + 1
11         P_i ← GROUNDER.ground(P[t/i])
12         SOLVER.add(P_i)
13     O_j ← GROUNDER.ground(E_j ∪ F_j(β_j))
14     SOLVER.add(O_j ∪ {← β_{j-1}})
15     repeat
16         if i_old < i then
17             Q_i ← GROUNDER.ground(Q[t/i](α_i))
18             SOLVER.add(Q_i ∪ {← α_{i_old}})
19             i_old ← i
20         χ ← SOLVER.solve({α_i, β_j})
21         if χ = ∅ then
22             i ← i + 1
23             P_i ← GROUNDER.ground(P[t/i])
24             SOLVER.add(P_i)
25     until χ ≠ ∅
26     send({X \ {α_i, β_j} | X ∈ χ})
```

declaration instructs *oclingo* to not apply any simplifications in view of yet undefined instances of `request(F,t)`, where `F` is a floor.

After launching *oclingo* on file `elevator.lp`, it proceeds according to Algorithm 1, which is basically an extension of *iclingo*'s `isolve` algorithm [2]. The base part is grounded in Line 4 and added to the solver in Line 5. Then, the main loop starts by waiting for external knowledge (Line 8), passed to *oclingo* by a client. For instance, the external knowledge representing the online progression in (1) is provided as follows:

```
#step 1.      request(3,1).      #endstep.
```

Here '`#step 1.`' specifies the time stamp $m_1 = 1$, and E_1 is '`request(3,1).`', as signaled via '`#endstep.`' A program for F_1 could be provided by specifying rules af-

ter a '`#volatile.`' declaration, but this functionality is not yet supported by *oclingo*. If it were, note that m_1 is supposed to be the maximum of e_1 and f_1 (cf. Definition 1).

After receiving the external knowledge, since $i = 0 < m_1 = 1$ in Line 9, `osolve` proceeds by incrementing i and processing a first slice of the incremental program's cumulative part. This includes grounding $P[t/1]$ and adding the ground program to the solver. Similarly, in Line 13 and 14, E_1 (and $F_1(\beta_1) = \emptyset$) are grounded and added to the solver. The notation $F_j(\beta_j)$ indicates that a fresh atom β_j is inserted into the body of each rule in F_j, so that the inquiry can in step $j + 1$ be discarded in Line 14 via adding the integrity constraint $\leftarrow \beta_j$ to the solver (cf. [12, 2]). Note that *oclingo* currently supports ground external input only, so that the "grounding" in Line 13 merely maps textual input to an internal representation.

The repeat loop starting in Line 15 is concerned with unrolling the incremental program in view of satisfying $Q[t]$. In our example, $Q[t/1](\alpha_1)$ is grounded and then added to the solver (Line 17 and 18), where a fresh atom α_i is used to mark volatile rules to enable their discarding via adding an integrity constraint $\leftarrow \alpha_{i_{old}}$ later on. (Note that the step number m_j passed as external knowledge may cause jumps of i in query programs $Q[t/i]$, which are not possible with $P[t]$ in view of the loop starting in Line 9, and i_{old} is used to address a volatile part becoming obsolete.) The solving accomplished in Line 20 checks for an answer set in the presence of Q_1, stipulating the absence of pending requests for the elevator at time step 1. Note that α_1 and β_1 are passed as assumptions (cf. [12, 2]) to the solver, telling it that queries in Q_1 and F_1 must be fulfilled. On the other hand, *oclingo* makes sure that yet undefined input atoms, i.e., elevator requests that did not arrive, are not subject to "guessing." In this case, `request(1,1)` and `request(2,1)` must not belong to an answer set, as no such external knowledge has been provided.

Given the pending request for floor 3, no answer set is obtained in Line 20, i.e., $\chi = \emptyset$. Thus, the next cumulative program slice, $P[t/2]$, is grounded and added to the solver (Line 23 and 24). Then, the repeat loop is re-entered, where the query $Q[t/2](\alpha_2)$ is added and $Q[t/1](\alpha_1)$ discarded (Line 17 and 18). Afterwards, the answer set in (2) is found in Line 20. As mentioned above, it contains `request(3,1)` as the only externally provided atom: although $Q[t/2](\alpha_2)$ would stay satisfied if `request(2,1)` and/or `request(3,2)` were assumed to be true, *oclingo* eliminates these options by disallowing undefined input atoms to hold. Finally, the obtained answer set (without α_2 and β_1) is sent back to the client for further processing (Line 26) before *oclingo* waits for new external knowledge in Line 8. In practice, this process terminates when the client sends '`#stop.`' (rather than '`#step` m_j`. ... #endstep.`') to *oclingo*.

As already described, the current version of *oclingo* does not yet support non-ground or volatile external input. Furthermore, it includes no modularity checks for successively obtained ground program slices (cf. Definition 3). As a consequence, it is the responsibility of the user to make sure that all programs are modularly composable (cf. Definition 4) in order to guarantee that the answer sets computed wrt the jth online program and the incremental program up to step k match the ones of the combined module $\mathbb{R}_{j,k}$ that do not assume residual inputs to hold. (The successive grounding performed by *oclingo* yields answer sets of $\mathbb{R}_{j,k}$ rather than of $R_{j,k}$ (cf. Definition 2); see Proposition 2 for sufficient conditions guaranteeing their correspondence.) Note that

modularity between incremental and online programs is easiest achieved at the predicate level, primarily, by not using atoms over input predicates in the heads of rules in the incremental program; e.g., `elevator.lp` follows this methodology. Of course, one also ought to take the modularity of the incremental program, when it is unrolled, into account (cf. [2]).

Note that, in view of the incremental approach, the step counter i is never decreased within `osolve`. Hence, it does not admit a "step back in time" wrt successive online programs and can, in general, not guarantee the k in answer sets of $\mathbb{R}_{j,k}$ to be minimal. (The minimal k', k such that $\mathbb{R}_{j-1,k'}$ and $\mathbb{R}_{j,k}$ admit answer sets may be such that $k < k'$.) To support minimality, one could add optimization statements (like `#minimize` and `#maximize`) to incremental programs, which is a subject to future work.

The application-oriented features of *oclingo* also include declarations '`#forget t.`' in external knowledge to signal that yet undefined input atoms, declared at a step smaller or equal to `t` are no longer exempted from simplifications, so that they can be falsified irretrievably by the solver in order to compact its internal representation of accumulated incremental program slices (cf. [12, 2]). Furthermore, *oclingo* supports an asynchronous reception of input, i.e., the call in Line 8 of Algorithm 1 is processed also if solving in Line 20, relative to a previous online program, is still ongoing. If new input arrives before solving is finished, the running solving process is aborted, and the solver is relaunched wrt the new external knowledge.

6 Further Case Studies

The purpose of our reactive framework is to provide a middle-ware for various application areas. For the sake of utility and versatility, we have conducted a set of assorted case studies, all of which are available at [11].

First of all, we have experimented with a more complex elevator control than given above, involving opening and closing doors as well as more elaborated control knowledge. The strategy is to never change directions, as long as there is an active request in a current direction. Also, this use case adds an external input indicating the respective position of the elevator. A second case study is an extension of the well-known blocks-world example [13]. Our extension allows for new, falling blocks thwarting previously computed states and/or plans. This scenario aims at studying dynamic adaptions to new (unexpected) situations. Our third use case deals with position tracking by means of sensor networks. In contrast to the above planning tasks, this scenario looks for histories and is thus directed backward in time. It is inspired by [14], where a person moves in a home environment (given as a 2D grid) involving doors, rooms, obstacles, walls, etc. Interestingly, missing sensor information may lead to alternative histories. Moreover, these histories may change with the arrival of further sensor readings.

Our last two scenarios deal with simple games. The first one considers the well-known Wumpus world [15]. An agent moves on a grid and tries to find gold in the dark, while avoiding pits and the Wumpus. The moving Wumpus is externally controlled and the agent has to react to the bad Wumpus' smell. The latter is obtained through events within an online progression. This is a typical agent-oriented scenario in which an agent has to react in view of its changing environment. The second game-based

use case implements a simplistic TicTacToe player. This scenario is interesting from a technical point of view because it allows for having two ASP players compete with each other. Interestingly, each move has to be communicated to both players in order to keep the game history coherent.

7 Discussion

We introduced the first genuinely reactive approach to ASP. For this purpose, we developed a module theory guaranteeing an incremental composition of programs, while avoiding redundancy in grounding and solving. Unlike offline incremental ASP [2], reactive ASP includes dedicated support of the input/output interface from (ground) module theory [10]; in practice, inputs can be declared conveniently at the predicate level. Our approach has a general, domain-independent nature and may thus serve as a middle-ware opening up numerous new reactive applications areas to ASP. To this end, we have implemented the reactive ASP solver *oclingo* and conducted a variety of case studies demonstrating the utility and versatility of our approach. The implementation along with all case studies are freely available at [11].

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In ICLP'08, Springer (2008) 190–205
3. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. http://potassco.sourceforge.net
4. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
5. Reiter, R.: Knowledge in Action. The MIT Press (2001)
6. Balduccini, M., Gelfond, M.: The autonomous agent architecture. Newsletter ALP **23** (2010)
7. Son, T., Lobo, J.: Reasoning about policies using logic programs. In ASP'01, AAAI/The MIT Press (2001)
8. Mileo, A., Merico, D., Bisiani, R.: Non-monotonic reasoning supporting wireless sensor networks for intelligent monitoring. In LPNMR'09, Springer (2009) 585–590
9. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
10. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In ECAI'06, IOS Press (2006) 412–416
11. `http://www.cs.uni-potsdam.de/wv/oclingo`
12. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in TCS **89**(4) (2003)
13. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)
14. Mileo, A., Schaub, T., Merico, D., Bisiani, R.: Knowledge-based multi-criteria optimization to support indoor positioning. In RCRA'10, CEUR-WS.org (2010)
15. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson (2010)