Answer Set Application Programming: a Case Study on Tetris∗

PETER SCHÜLLER

Department of Computer Engineering, Faculty of Engineering, Marmara University
(*e-mail:* peter.schuller@marmara.edu.tr)

ANTONIUS WEINZIERL

Institute of Information Systems, TU Wien
(*e-mail:* weinzierl@kr.tuwien.ac.at)

submitted 29 April 2015; accepted 5 June 2015

Abstract

Answer-Set Programming (ASP) is a successful branch of the logic programming paradigm with many applications in modelling and solving of NP-hard problems. Combinatorial problems are the main application domain of ASP and it seems unsuitable for serving as a programming language for interactive applications. However, we conjecture that there is no theoretical obstacle for using ASP to that end. As witnessed by functional programming, it can be useful to use a declarative paradigm for creating applications. In this work we explore possibilities, benefits, and drawbacks, of programming an interactive application in ASP. We find that this is hard mainly for the following reasons: managing change over time, interaction with the user, generating output that is ordered (i.e., not a set), handling persistence of certain data, and ensuring efficiency. ASP and related fields provide powerful techniques for representing actions and change, executing programs with respect to external environments, and processing external events. Even if the full power of these techniques is not required to build an interactive application, combining them is necessary, and putting together these concepts in a practical framework is challenging. We realize such an integration in a framework we call Answer Set Application Programming framework which is based on the HEX language and features syntactic shortcuts to make application programming more intuitive. We describe design decisions and discuss alternative possibilities. Our sample application is a playable version of Tetris which demonstrates that ASP can be used as a general-purpose programming-language.

KEYWORDS: Answer-Set Programming, Programming Techniques, Knowledge Representation, Software Engineering, Nonmonotonic Reasoning

∗ This work has been supported by the Austrian Science Fund (FWF) Project P27730 and the Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E430.

Fig. 1. ASPeTris screenshots: game and main menu (Highscores entry is selected).

1 Introduction

Answer-Set Programming (ASP) is a successful approach for modelling and solving NP-hard problems. We propose a framework for general-purpose application-programming based on ASP and realize Tetris as a case study.

Using ASP instead of imperative languages like Java would likely be met with skepticism. We think this is mainly due to the following reasons: firstly, ASP is lacking features such as input/output facilities and programming libraries; secondly, Prolog aimed for the same but it set a negative example since it forced programmers to think in terms of the evaluation underlying Prolog (e.g., using red cuts). The resulting application programs cannot be understood in a declarative way since their (temporal) behavior is based on the Prolog search/evaluation algorithm.

We therefore consider it paramount to maintain declarativity when aiming to broaden the scope of ASP. In this work we realize a small application, the game Tetris, to investigate from a practicioner's perspective how far ASP can serve as a basis for application programming. We identified several critical points: the interaction of real time, logical time (steps) and state of a program, external effects that influence ASP reasoning, and causing change in the outside world.

We present a theoretical framework to achieve the above and we report on an implementation of this framework. The feasibility of our approach is demonstrated by ASPeTris, an interactive and playable implementation of Tetris which features editing the player name, playing the game, interrupting and continuing a game later, and a persistent high score list. Figure 1 shows two screenshots of ASPeTris.

Our Answer-Set Application-Programming framework (ASAP) allows declarative reasoning between two logical time points: the closest time point in the past (called `prev`) and the closest time point in the future (called `next`). ASAP enables interaction with the real world through external atoms and actions, similar as in `acthex` (Basol et al. 2010). Program states that change over time are represented by fluents similar as in ASP planning (Gelfond and Lifschitz 1993). Logical time

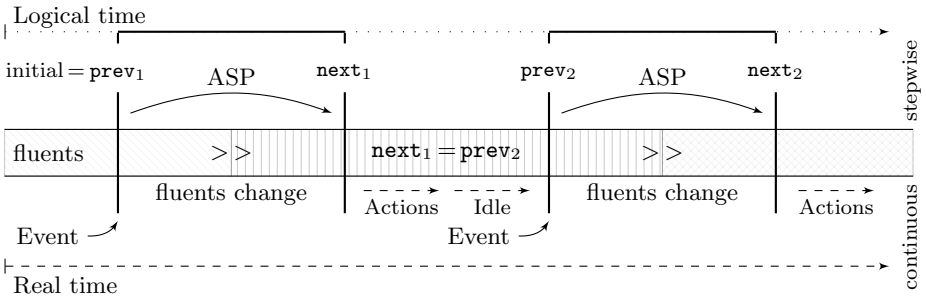

Fig. 2. Real time and declarative reasoning in the ASAP framework.

and real time are coupled via events. An event, such as keyboard activity or a timer, triggers evaluation of ASP semantics to determine the effect of the event. An ASAP program defines the reaction of the application to an event in terms of new fluent values and actions to be executed as a consequence of the event, such as drawing the user interface or starting a timer. Actions have an execution order and ASAP provides specific syntax for comfortably defining that order. After executing actions, the framework waits for arrival of the next event which triggers evaluation of ASP semantics anew. Fluent inertia is realized using the encoding of Lee et al. (2012) which yields a declarative management of changing program state over time. ASP evaluation requires time itself, therefore we cannot guarantee that events are handled immediately. However ASAP guarantees that events are handled in exactly the order they occurred and as soon as possible after all previous events have been evaluated, similar as in the Event Calculus (Kowalski and Sergot 1986).

The conceptual view of time in ASAP is shown in Figure 2. Real time and logical time are synchronized via the order of events. Reasoning is based on the previous logical time point, and actions are executed after computing the next logical time point. (Events that occur in the meantime are enqueued and processed later.)

This paper contributes a framework for ASP application programming which features the following concrete solutions: (i) inertial fluents for declarative reasoning about program state; (ii) event- and action-based interaction with the outside world; (iii) a syntax to specify the execution order of actions; (iv) out-of-the box persistence for selected fluents; (v) a working prototype implementation of the framework in HEX and a playable version of Tetris.[1] We found that Tetris is a demanding application scenario which points to necessary improvements of ASP syntax, software engineering tools, and solvers. At the same time our realization of Tetris shows that the declarative nature of ASP can be exploited as a strength in application programming. We believe that this work can be beneficial both for application programming and for ASP.

We give preliminaries in Section 2, discuss fluents and time in Section 3, events in Section 4, actions in Section 5, persistence and the overall framework in Section 6. We discuss related work in Section 7 and conclude with Section 8.

[1] Source code is available at `https://bitbucket.org/peterschueller/asap-aspetris` .

2 Preliminaries

Answer Set Programming (ASP). We assume familiarity with ASP (Gelfond and Lifschitz 1988; Lifschitz 2008; Gebser et al. 2012) and give only brief preminaries of logic programs with (uninterpreted) function symbols and aggregates and a brief overview of HEX. A logic program consists of rules of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \textbf{not } \beta_{n+1}, \ldots, \textbf{not } \beta_m$$

where α_i and β_i are head and body atoms, respectively. Atoms can contain constants, variables, and function terms, and a program must allow for a finite instantiation. Details of these restrictions and syntax/semantics of ASP are described in the ASP-Core-2 standard (Calimeri et al. 2012) of which we will mainly use aggregates. An aggregate literal can occur in the body of a rule; it accumulates truth values from a set of atoms, e.g., `2 = #count { X : p(X) }` is true in an answer set candidate I iff the extension of p in I contains exactly 2 elements. HEX (Eiter et al. 2006; Eiter et al. 2015) extends ASP with external atoms of the form $\&atom[p_1, \ldots, p_n](q_1, \ldots, q_m)$ where $atom$ is the name of the external atom, and p_i and q_j are lists of input and output terms. We denote by $\mathbf{AS}(P, \{f, f'\})$ the set of answer sets of a program P which is defined using its Herbrand universe, ground instantiation, the FLP-reduct (Faber et al. 2011) which intuitively reduces a program using assumptions in an answer set candidate, and oracle functions f, f' of form $f_{\&atom}(I, p_1, \ldots, p_n, q_1, \ldots, q_m) \in \{0, 1\}$ to represent semantics of external atoms. For example $\&setminus[a, b](X)$ could have its oracle function defined as $f_{\&setminus}(I, a, b, X) = 1$ for all X such that $a(X) \in I$ and $b(X) \notin I$.

Tetris. Tetris is played in a grid that is initially empty. A piece is a set of four stones arranged such that they are connected by adjacency. Pieces are created randomly and appear on the top center of the grid. They fall down with a certain speed that increases with the score. The user can rotate pieces by 90 degree, or move them left or right by one stone. If a piece hits the bottom or another stone then it becomes fixed and the next random piece starts to fall. If a grid row becomes filled, these stones are removed, stones above move down by one row and the score increases. The game ends when the new random piece hits stones before falling down.

3 Application State and Time: Fluents

Applications that interact with users are usually driven by the input from the user. If the user clicks the menu button, the menu should appear, if the user clicks on exit, the program stops, if the user gives no input, then the application should remain in its current state (or the state evolves in a pre-defined manner). Altogether, the behaviour of an application is driven not only by a sequence of inputs but also by the passage of time. Hence managing the change of state over time is one of the most important concepts that distinguishes classical usage of ASP for solving combinatorial problems from its usage for programming applications. Fortunately, ASP has been applied to perform reasoning with time, in particular (i) in planning (e.g., (Gelfond and Lifschitz 1993)), where we reason about a hypothetical sequence

of possible future worlds; and (ii) in stream reasoning (e.g., (Gebser et al. 2012)), where we reason about a possibly infinite sequence of data.

In an interactive application, reasoning is done on-line while the application is running and while future input is not yet known, which is similar to stream reasoning. At the same time, applications mostly have one deterministic future, and *running* an application means to *determine what changes* from the present to the future. This is similar to executing an action in planning and it also requires to solve the frame problem, i.e., the problem of what does *not* change.

We next combine notions of planning and stream reasoning to obtain a declarative treatment of time and change in ASP applications. We use inertial fluents (McCarthy and Hayes 1969) for modelling the state of the application, and we limit the number of situations of interest to just two: the closest point in the past, and the closest point in the future. We call these two time points *previous situation* and *next situation*. In the (real) time that passes between the previous and next situation, we evaluate ASP semantics to determine what happens next. Events that happened during evaluation are enqueued for later processing (see next section).

To work with fluents we define *fluent atoms* that are of the form

```
fluent=v@prev   or   fluent=v@next                    (1)
```

where `fluent` is the name of the fluent, `v` is its value, and `prev`/`next` refer to the situation of interest. Both `v` and `fluent` may contain terms with uninterpreted functions, constants, and variables. Fluent atoms can be used in the head or the body of rules. A fluent and its initial value is declared using a statement of the form

```
#initially fluent=v                                   (2)
```

in the head of a rule, with `fluent` and `v` as above.

Example 1
The Tetris game area is defined using a rule with an `#initially` head:

```
#initially area(X,Y)=empty :- xcoo(X), ycoo(Y).
```

where `xcoo` and `ycoo` represent x and y coordinates of the game area.

The stones in a falling piece are represented in an atom `fallingStone(X,Y,S)` with coordinates `X` and `Y` and stone of type `S`. If a piece cannot fall further, `fixPiece` becomes true and the game area is changed to contain the stones of the falling piece:

```
area(X,Y)=S@next :- fixPiece, fallingStone(X,Y,S).
```

□

All fluents are inertial: if the value is not changed by any rule, it stays at the value from the previous situation. This is realized by a rewriting to ordinary ASP atoms and an inertia encoding inspired by existing ASP planning systems. A fluent `fluent=v@prev` is transformed into an atom `val(fluent,v,1)`, similarly `fluent=v@next` becomes `val(fluent,v,0)`. The reserved predicate `val` is used throughout the program to represent the values of fluents, 1 denotes the previous situation and 0 the next. We decided to use 0 and 1 in this way to allow a straightforward extension for programs that need to look further into the past with increasing integers, e.g., 2 would be the situation before 1.

An initialization atom of form `#initially fluent=v` can only occur in the head

of a rule and is rewritten to `ifluentinit(fluent,v)`. Initialization and inertia of fluents is handled by the following rules, inspired by (Lee 2012).

```
sit(0).  sit(1).  prevsit(0).                                          (3)
ifluent(F) :- ifluentinit(F,_).                                        (4)
val(F,V,S+1) :- &event[first], ifluentinit(F,V), prevsit(S).           (5)
val(F,V,S+1) :- not &event[first], &prevval[F](V,S),
                prevsit(S), sit(S+1), ifluent(F).                      (6)
val(F,V,0) :- not nval(F,V,0), val(F,V,1).                             (7)
nval(F,V,0) :- not val(F,V,0), val(F,V,1).                             (8)
:- 2 <= #count { V : val(F,V,S) }, ifluent(F), sit(S).                 (9)
:- 0 = #count { V : val(F,V,S) }, ifluent(F), sit(S).                  (10)
```

Facts (3) define situations of interest: we can access the previous situation's fluent values with time point `S=1` and in every evaluation of the program we consider two situations: `prev=1` and `next=0` (see also Fig. 2). Rule (4) represents which fluents exist, (5) manages initialization where `&event[first]` is true in the first evaluation of semantics (see Algorithm 1). External atom `&prevval[F](V,S)` provides access to the value of fluent F in the answer set of the previous evaluation of the program. If an evaluation yields a fluent value at time point `S`, in the next evaluation this fluent value is provided as `&prevval[F](V,S)` and shifted to time point `S+1` using (6). Inertia is realized using (7) and (8): a value in the past possibly remains like that in the future, moreover (9) and (10) ensure that each fluents has a single unique value per situation. This encoding has the advantage that it does not require to define a fluent domain, which is particularly important for fluents with a practically infinite domain such as player names and highscores. The encoding relies on the `&prevval[.](.,.)` external atom to 'remember' the history of fluent values, i.e, the values `f=v@prev`. Note that determining the next state `f=v'@next` with the help of inertia is realized purely in ASP (without any outside machinery).

Note that the above rules can handle an arbitrary amount of previous situations by adding situations to the extension of `sit` and `prevsit` predicates in (3). In ASPeTris we did not encounter a need for more than one previous situation.

A natural question is whether *all* atoms in an ASP application should be fluents. Similar to the notion of the *volatile* part of a program in incremental answer set programming (Gebser et al. 2008) we decided to allow non-fluent atoms that can be used in reasoning but are not preserved among situations.

In an interactive application we must forget past information, since every point in time adds new atoms, meaning that eventually atoms with information about the past occupy all available memory. This is mainly a practical issue which we solve by using a fixed window of size $k = 2$. Several more involved aspects of such reasoning have been analyzed in stream-based reasoning (Gebser et al. 2012; Beck et al. 2015) and these could be useful for applications, e.g., for deducing a maximum required window size for certain atoms, or giving each predicate its own fixed windows size.

Regarding the future values of fluents, one can model just the next point in time,

or several future steps. The latter may allow easier modelling but it is not strictly necessary, since anything deduced for k steps in the future may be deduced at $k-1$ steps in the future by looking back $k-1$ steps into the past. We here consider a minimal setting with a single future time point.

4 External Input: Events

The ASP semantics of the application program are evaluated repeatedly. But how often should the ASP program be evaluated? In principle, one could evaluate immediately after evaluation of the previous time step finished. This realizes a busy-loop which is easy to implement but wastes resources. We therefore decided to evaluate the program only if some event occurred, e.g., a key press or an expired timer.

Conceptually, the only thing happening between the previous and the next situation is the evaluation of an ASP program to determine the future. This takes time in practice, therefore a thread collects all events that happen during this time in an *event queue* which preserves the order of events.

Events are exposed to the application program through external atoms, so their occurrence and the properties of an event can be obtained through HEX as truth values. For example, `&event["keyboard","special"]("Esc")` becomes true if the 'Esc' key is pressed on the keyboard. Multiple events may be treated by the same rules, for example `&event["keyboard","normal"](Key)` becomes true if any character or number key is pressed and the key is instantiated in the variable `Key`.

Example 2
Displaying the name of the player in fluent `player` and editing that name is achieved by the following short set of ASP rules,

```
#initially player="New Player".                                    (11)
@drawTextCentered(P) :- playerName=P@next.                         (12)
player=New@next :- &event["keyboard","normal"](Key),
                   player=Old@prev, &concat[Old,Key](New).         (13)
player=New@next :- &event["keyboard","special"]("Backspace"),
                   player=Old@prev, &chop[Old](New).               (14)
```

where (11) declares the fluent, (12) displays it, (13) appends a pressed key to the fluent's value, and (14) removes the last character if Backspace is pressed. The required string operations are realized in external atoms `&concat` and `&chop`. □

An important design issue regarding events is the question whether one iteration should handle a single or multiple events. Handling multiple events at once requires to process them in a particular order or as if they happened simultaneously. The former strategy means to represent multiple steps in one evaluation, which would make application programs more complex. Moreover, we already handle multiple time steps using repeated evaluation, hence this strategy would introduce redundancy in the framework. The latter possibility creates additional complexity as

several events potentially compete for what should happen in the program, e.g., to move the Tetris piece to the right and to the left simultaneously.

While realizing ASPeTris we observed that handling only a single event at a time greatly simplifies the application program and helps to avoid bugs. Moreover, in case an event depends on complex conditions, one may be tempted to check these conditions in the framework and outside of ASP. However, this makes the interface between ASP and the outside world more complex and duplicates reasoning facilities. We decided that the ASAP framework should provide only basic events, whereas complex conditions are supposed to be represented in ASP.

5 Executing Actions

The syntax of actions in our framework is inspired by acthex: an action is an atom preceeded by `@` and contains as the last argument a number which is called the action's *priority*. For example, `@exit(27,0)` states that the program execution shall end with return code 27 and that the action has priority 0, i.e., it shall be executed as one of the first actions. Multiple actions in an answer-set are executed in ascending order of their last argument (ties are broken arbitrarily).

Action ordering is especially important for drawing user interfaces: it makes a big difference whether a blue rectangle is drawn and then some red text, or if the red text is drawn first and then some blue rectangle (erasing the text).

We observe from imperative languages that establishing the order in which statements are executed is no problem at all: the execution order is given straight by the order of statements in the input program. This implicit ordering based on the sequence of statements in the code is missing in ASP, because of the guiding principle of declarativity: to treat programs and many aspects of semantics as (unordered) sets instead of lists. Because the order of actions is relevant in an application, having the possibility of ordering actions comfortably makes programs more maintainable and readable. In functional programming, Monads (Wadler 1995) were introduced to achieve an ordering of events within a declarative formalism. To the best of our knowledge, such a mechanism does not exist in ASP.

In the ASAP framework we therefore propose a minor syntactic addition, namely code blocks enclosed by `[` and `]` brackets. Actions in these blocks are automatically ordered in the same order as they appear in the source program, without the need to explicitly specify their priorities. We consider this syntactic sugar for specifying an order, although one might say this weakens declarativity. For example, the program

```
[
  @drawTextCentered("ASPeTris Game Menu") :- draw_main_menu.
  @drawTextCentered("\n") :- draw_main_menu.
  @drawTextCentered("New Game") :- draw_main_menu.
]
```

draws a main menu consisting of the headline "ASPeTris Game Menu", followed by a newline, followed by the menu item "New Game".

In practice we realize this implicit action priority using a rewriter which transforms this into an ordinary HEX program by removing the '`@`' symbol and adding to

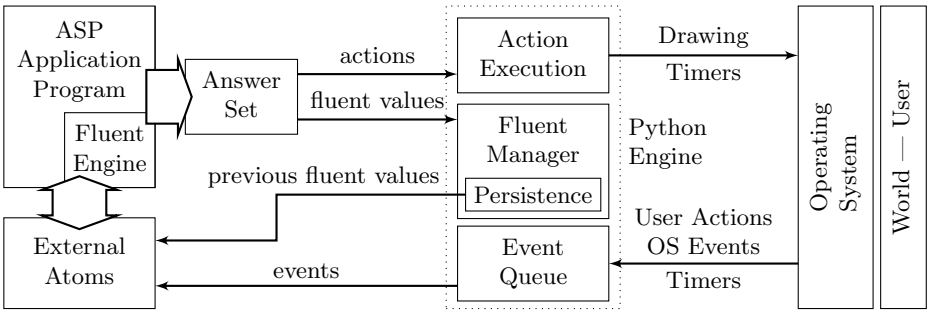

Fig. 3. Components and data flow in the ASAP framework.

each atom a priority argument. Actions that are not inside square brackets receive priority 0, and if multiple square brackets occur, then the actions inside are ordered, but no order is enforced across both brackets, e.g., `[ @aa. @bb. ] [ @cc. @dd. ]` is rewritten to `aa(0). bb(1). cc(0). dd(1).` This guarantees that `aa` is executed before `bb` as well as that `cc` is executed before `dd`, but enforces no further ordering.

Timers and Soft Events. In Tetris, pieces are falling from the top to the bottom without any user interaction, so a timer is required to trigger evaluation of a new situation. Timers are started using the action `@startTimer(name,delay)` where `name` indicates an identifier for the timer and `delay` is an integer in milliseconds. Timers produce events of the form `&event[timer](name)` which trigger evaluation of a new situation and can be handled like other events.

In ASPeTris, when the falling piece stops and is fixed in the game area we also need to handle the elimination of full rows. This intuitively happens sequentially. To perform both, fixing the piece and eliminating full rows in one time step, requires an extra copy of the gaming area, however. As an alternative, we introduce *soft events* which create a new time step and cause immediate reevaluation of the ASP program (bypassing all queued events). In our case this allows some separation of concerns and makes the program easier to write and read (at the potential cost of losing some declarativity).

6 The Application Execution Framework

An overview of the components of the ASAP framework is given in Figure 3, where arrows indicate information flow. The ASPeTris game is implemented purely as an ASP application program (top left component); the ASAP framework itself is mainly implemented as a Python program (center components) that controls the ASP evaluation and communicates with the operating system (right).

In addition to the above features, the ASAP framework also provides facilities for persistence, since this is an important topic for applications. Information is persistent if it remains known to the application across runs.

As we use fluents in our framework, we chose to realize a selective persistence for fluents. A declaration of the form '`#persistent name.`' in the program causes the code that takes care of `&prevval[.](.,.)` to store those fluents that match

Algorithm 1: Core ASAP Algorithm.

input: Answer Set Application Program P (after rewriting, including fluent engine)

1 $A \leftarrow \{\texttt{val(}Flu\texttt{,}Val\texttt{,0)} \mid (Flu, Val)$ in persistent storage on disk $\}$ `// can be` $\emptyset$

2 $f_{\&event} \leftarrow \{(\texttt{first}) \mapsto 1\}$ `// initial event`

repeat

4 $\quad f_{\&prevval} \leftarrow \{(Flu, Val, 0) \mapsto 1 \mid \texttt{val(}Flu\texttt{,}Val\texttt{,0)} \in A\}$

5 $\quad S \leftarrow \mathbf{AS}(P, \{f_{\&prevval}, f_{\&event}\})$ `// evaluate ASP application program`

$\quad$ **if** $S \neq \{A'\}$ **then return** "Error: zero or more than one answer set found."

$\quad$ **if** `@exit` $\in A'$ **then return** "Application finished."

$\quad$ **Execute** *actions in* A' *ordered by their last argument.*

$\quad$ **Store** *to disk:* $\{(Flu, Val) \mid \texttt{val(}Flu\texttt{,}Val\texttt{,0)} \in A'$ *and* Flu *is* `#persistent` $\}$

10 $\quad Ev \leftarrow$ next event in queue `// waits for next event if queue is empty`

11 $\quad f_{\&event} \leftarrow \{(Ev) \mapsto 1\}$

12 $\quad A \leftarrow A'$

forever

`name` to disk. The name matches if it matches the complete fluent name, or if the fluent name is a function term and the name matches the outermost function (for example '`#persistent area.`' makes all `area(X,Y)` fluents persistent).

To restore a serialized state, the ASAP framework initializes `&prevval[.](.,.)` with the stored fluent values. In the first situation fluents are then initialized using these previous values and if none is available, the `#initial` value of the fluent is taken. The following rules achieve this when we use them instead of (5).

```
prevFluent(F) :- &event[first], ifluent(F), &prevval[F](V,0).
val(F,V,1) :- &event[first], prevFluent(F), &prevval[F](V,0).
val(F,V,1) :- &event[first], not prevFluent(F),
              ifluentinit(F,V), ifluent(F).
```
(15)

In ASPeTris we found it useful to make only selected fluents persistent. For example the highscore list and the player name are persistently stored, however, the currently active menu item is not persistent (otherwise starting the game would always select 'exit' from the previous run).

This notion of persistence makes it significantly easier to return to previous states of the application compared with imperative programming. In ASPeTris, simply by adding persistence of the game field and the score and falling stones, we realized a 'Continue Game' functionality. Note that we do not require explicit reading or writing of persistence, although using actions to that end might improve efficiency.

Evaluation Algorithm. The core behaviour of the ASAP framework is given by Algorithm 1. Line 1 loads persistent fluent values from disk. Line 2 prepares the initial event which causes loading persistence or falling back to initialization values in the first iteration of the loop using encoding (15).

In the loop, line 4 configures `&prevval[.](.,.)` to provide fluent values in A to the next evaluation of P. Line 5 evaluates the program with respect to the prepared oracle functions for external atoms. We ensure that there is a single answer set (this check could be replaced by mechanisms for selecting a preferred single answer set as described in (Fink et al. 2013)). After executing actions from answer set A' and

storing persistent fluent values to disk, line 10 obtains the next event from the queue. If no event is in the queue, this step waits for an event to become available. Line 11 prepares the oracle function of `&event[.](.)` for the next iteration and line 12 stores the answer set as the basis for the next iteration where it will be communicated to the HEX program via line 4. The algorithm loops indefinitely until the ASP application program yields an `@exit` action or an error occurs. Note that other external atoms (such as `&concat`) can be used but we do not show them as they are not specific to the ASAP framework.

7 Related Work

This work is related to several important papers in the areas of planning and interacting with an (infinitely) changing world. Therefore we will only give a brief discussion of some important related work.

Our framework manages program state as fluents and situations as already described by McCarthy and Hayes (1969). Planning with ASP was analyzed in several projects, for example (Gelfond and Lifschitz 1993; Eiter et al. 2004; Lifschitz 2002; Lee 2012) (we realize fluent inertia based on the last of these papers). Our framework preserves the order of events but not their absolute time, which is similar to one of the guiding principles of the Event Calculus (Kowalski and Sergot 1986).

The acthex formalism (Basol et al. 2010; Fink et al. 2013) extends HEX with actions and an environment. Actions are extracted from answer sets similar as in our framework, however acthex immediately repeats semantic evaluation after executing actions, until no more actions are generated. Actions in our framework can have implicit execution order using the `[ ]` notation, while in acthex an explicit assignment of priorities is required. The acthex formalism is designed for programs that act in the world, driven by their own actions. Different from that, our framework is driven by outside events and can remain idle between subsequent semantic evaluations. Another difference is that our framework includes mechanisms for managing inertia and persistence (which requires additional work in acthex).

The oClingo framework and reactive ASP (Gebser et al. 2011) deal with ASP in the context of reacting to external events. To that end an incremental solving approach was defined which has a performance advantage over repeated complete re-evaluation (as done in our framework). Partial incremental evaluation comes at the price of a modularity condition (Oikarinen and Janhunen 2006) that makes programming cumbersome. As a result, Online Agent Logic Programming (Cerexhe et al. 2014) was defined to satisfy modularity automatically at the cost of restricting use of negation. In ASPeTris we consider efficiency to be future work and concentrate on ease of use. Clingo-4 (Gebser et al. 2014) allows to control grounding and solving using Python. We decided to allow Python and nondeclarative code only *within the boundaries* of our framework, i.e., to realize actions and external atoms. We believe this helps to maintain declarativity and provides sufficient flexibility for creating applications. Realizing ASAP in Clingo-4 is possible, but requires to change the program at each time point (e.g., realizing (11)-(14) for editing the player name in Clingo-4 is challenging). HEX as a foundation has the advantage

that the same program is used for all time points, only oracle functions of external atoms change (see Alg. 1).

Stream reasoning (Gebser et al. 2012; Beck et al. 2015) focuses on processing continuous data streams with ASP; the main issue is to limit the amount of data being processed. The ASAP framework discards all historical information beyond the previous state, while stream reasoning uses more sophisticated techniques.

Dedalus (Alvaro et al. 2011) is a Datalog framework for modeling distributed, asynchronous systems, where each atom obtains an additional time argument. Rules can define truth within a timepoint or one timepoint into the future.

Different from ASPeTris, most ASP research related to games focuses on combinatorial problems of generating new game instances (Smith and Mateas 2011) or solving existing instances (Denecker et al. 2009; Calimeri et al. 2014).

8 Conclusion and Future Work

In this paper we showed that ASP can be used for application programming, an area with many potential applications. We took inspirations from several existing areas of ASP and related areas and developed the ASAP framework which captures application programming and time-dependent interaction in a declarative way.

To demonstrate the feasibility of our approach, we implemented ASAP using the Python extension of the dlvhex solver, and within ASAP we realized ASPeTris, an implementation of the well-known game of Tetris. ASPeTris is a fully playable game: interactive input from the user is processed, the game is drawn, a list of highscores is persistenly stored, and the player name can be edited interactively. Our key contribution is a framework for interactive and declarative problem solving.

We furthermore proposed syntactic shortcuts that ease application programming: ordering actions implicitly by using their order of appearance in the input program, and an engine for fluents to reason between logical points in time.

The work shown here is only a case study from which we have learned that the following important issues need to be addressed in the future: (i) the full ASP code is instantiated and evaluated over and over, which makes ASPeTris slow (2 frames per second) and could be avoided if the solver would automatically disregard large sets of irrelevant rules for evaluation; (ii) a declarative concept of modularity should be integrated into the syntax to ease programming and simultaneously facilitate automatic optimizations in the solver; (iii) a declarative method for error handling to catch and treat exceptional cases within ASP itself would be useful; (iv) enabling usage of nondeterminism within ASAP programs is an open issue; finally (v) a notion of library functions to avoid 'reinventing the wheel' over and over again (e.g., (Ianni et al. 2004)) and also to integrate existing functionalities written in imperative programming languages, possibly through the use of external atoms.

Acknowledgements

We thank the anonymous reviewers and Yuliya Lierler for their feedback.

References

Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. 2011. Dedalus: Datalog in Time and Space. In *Datalog 2010 Workshop*. Vol. 6702 LNCS. 262–281.

Basol, S., Fink, M., Erdem, O., and Ianni, G. 2010. HEX Programs with Action Atoms. In *International Conference on Logic Programming (ICLP), Technical Communications*. 24–33.

Beck, H., Dao-Tran, M., Eiter, T., and Fink, M. 2015. LARS: A Logic-based Framework for Analyzing Reasoning over Streams. In *AAAI Conference on Artificial Intelligence*.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. 2012. ASP-Core-2 Input language format. Tech. rep.

Calimeri, F., Ianni, G., and Ricca, F. 2014. The third open answer set programming competition. *Theory and Practice of Logic Programming 14,* 1, 117–135.

Cerexhe, T., Gebser, M., and Thielscher, M. 2014. Online Agent Logic Programming with oClingo. In *Pacific Rim International Conference on Artificial Intelligence (PRICAI)*. 945–957.

Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. 2009. The second answer set programming competition. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Vol. 5753 LNAI. 637–654.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. 2004. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic 5,* 2, 206–263.

Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., and Schüller, P. 2015. A Model Building Framework for Answer Set Programming with External Computations. *Theory and Practice of Logic Programming*. To appear, arXiv:1507.01451.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. 2006. Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In *European Semantic Web Conference (ESWC)*. 273–287.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298.

Fink, M., Germano, S., Ianni, G., Redl, C., and Schüller, P. 2013. ActHEX: Implementing HEX Programs with Action Atoms. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 317–322.

Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., and Schaub, T. 2012. Stream Reasoning with Answer Set Programming: Preliminary Report. *Principles of Knowledge Representation and Reasoning (KR)*, 613–617.

Gebser, M., Grote, T., Kaminski, R., and Schaub, T. 2011. Reactive Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning*. 54–66.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. 2008. Engineering an incremental ASP solver. In *International Conference on Logic Programming*. Vol. 5366 LNCS. 190–205.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan Claypool.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. Tech. rep.

Gelfond, M. and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *International Conference and Symposium on Logic Programming (ICLP/SLP)*. 1070–1080.

Gelfond, M. and Lifschitz, V. 1993. Representing action and change by logic programs. *The Journal of Logic Programming 17*, 301–321.

Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C., and Calimeri, F. 2004. Enhancing Answer Set Programming with Templates. In *Workshop on Nonmonotonic Reasoning (NMR)*. 233–239.

Kowalski, R. and Sergot, M. 1986. A logic-based calculus of events. *New generation computing 4,* June 1975.

Lee, J. 2012. Reformulating Action Language C+ in Answer Set Programming. In *Correct Reasoning*. Number 7265 of LNCS. 405–421.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1-2, 39–54.

Lifschitz, V. 2008. What Is Answer Set Programming ? In *AAAI Conference on Artificial Intelligence*. 1594–1597.

McCarthy, J. and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, 463–502.

Oikarinen, E. and Janhunen, T. 2006. Modular Equivalence for Normal Logic Programs. *European Conference on Artificial Intelligence (ECAI)*, 412–416.

Smith, A. M. and Mateas, M. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games 3,* 3, 187–200.

Wadler, P. 1995. Monads for functional programming. In *Advanced Functional Programming*. 24–52.