Multi-shot stream reasoning in answer set programming: Preliminary report

Philipp Obermeier, Javier Romero, and Torsten Schaub

University of Potsdam

Abstract. In the past, we presented a first approach for stream reasoning using Answer Set Programming (ASP). At the time, we implemented an exhaustive wrapper for our underlying ASP system, *clingo*, to enable reasoning over continuous data streams. Nowadays, *clingo* natively supports *multi-shot solving*: a technique for processing continuously changing logic programs. In the context of stream reasoning, this allows us to directly implement seamless sliding-window-based reasoning over emerging data. In this paper, we hence present an exhaustive update to our stream reasoning approach that leverages multi-shot solving. We describe the implementation of the stream reasoner's architecture, and illustrate its workflow via job shop scheduling as a running example.

1 Introduction

Stream Reasoning [1, 2] has become a major subject of research in the last years. Some approaches rely on Answer Set Programming (ASP; [3, 4]), a well established paradigm to *declarative problem solving*. Rather than solving a problem by telling a computer *how to solve the problem*, the idea is to simply describe *what the problem is* and leave its solution to the computer.

Several years ago, we proposed one of the first stream reasoning approaches [5–7] with ASP. At that time, we had to implement an exhaustive wrapper for our underlying ASP system, *clingo* [8–10], to enable reasoning over continuous data streams. Nowadays, *clingo* natively offers *multi-shot solving* [11]: a set of functionalities to support operative processing of streams. With that, instead of restarting the solver whenever an update arrives, we reuse the running *clingo* process by directly updating its internal knowledgebase accordingly. Furthermore, users can define custom stream reasoning workflows with minimal effort via *clingo*'s API, available for programming languages C, Lua and Python.

In this paper, we will detail a barebones approach to leverage *clingo*'s multi-shot solving capabilities for stream reasoning. Particularly, we will give a brief introduction to multi-shot solving; devise a basic stream reasoner based on multi-shot solving using *clingo*'s Python API; and showcase our stream reasoner's workflow via job shop scheduling as a running example.

2 Job Shop Scheduling with Answer Set Programming

For better illustration, we introduce a running example, *job shop scheduling* (JSP) that we solve first with regular (single-shot) ASP. We consider a variant of JSP

where jobs are pre-assigned to machines. The problem is the following: given m jobs each with a specific duration and pre-assigned to one of n available machines, what is a suitable schedule? Typically, we also assume $m > n$ to eliminate trivial problem instances.

The basic idea of ASP is to declaratively describe a problem by a logic program and feed it into an ASP solver to retrieve the program's stable models, which correspond to the solutions of the original problem. For example, consider a JSP instance consisting of jobs 1, 2 and 3 with respective durations of 4, 2, and 2 minutes, assigned to machines 1, 2 and 3, respectively. In ASP, this can be represented by the facts

```
job(1,1,4). job(2,2,2). job(3,2,2).
```

where the first argument of `job/3` is the job's ID, the second its assigned machine, and the third its duration. The general JSP problem can represented as follows[1]:

```
1 #const horizon = 4.
3 { start(J,T) : T=1..horizon+1-D } = 1 :- job(J,_,D).
5 occupies(J,M,  D,T) :- start(J,T), job(J,M,D).
7 occupies(J,M,D-1,T) :- occupies( J,M,D,T-1), D> 1.
9 :- occupies(J1,M,_,T), occupies(J2,M,_,T), J1<J2.
```

Line 1 sets the constant for the planning horizon to 4. The next rule (Line 3) chooses for each job `J` a single starting time point `T` stated as `start(J,T)`. Line 5 and 7 define when a machine is occupied by a job: when a job is started on a machine (Line 5), and when it is still running (Line 7). The constraint in the last line makes sure that at no point two jobs occupy the same machine. When we evaluate this encoding with the instance above, we retrieve two stable models: the first model, {`start(1,1),start(2,3),start(3,1)`}, schedules both job 1 and 3 at time point 1 and job 2 at 3; the second model, {`start(1,1),start(2,1),start(3,3)`}, schedules both job 1 and 2 at time point 1 and job 3 at 3.

3 Multi-Shot Stream Reasoning

In this section, we first give a brief conceptual overview of our stream reasoning approach. Afterwards, we informally explain how to leverage multi-shot solving in this context. Eventually, we give a technical description of our software architecture and its workflow.

In essence, our approach continuously evaluates an ASP problem over a *stream*, a finite sequence of sets of ASP atoms. The domain of the sequence, i.e., natural numbers $1, 2, 3, 4, \ldots$ represent time steps. We employ a *tuple-based sliding window* [12] of fixed size to limit the scope of data to consider. To implement this mechanism in ASP, we leverage multi-shot solving [8, 13, 14] to seamlessly add new and remove outdated information from the window over time. Formally, this

[1] Full source code available at `https://github.com/potassco/aspStream/blob/master/aspStream/job.lp`.

portrays an alternative operational characterization for solving *time-decaying logic programs*[5] with fixed expiration time. We leave the proof of this claim for the full paper.

Since our approach uses ASP multi-shot solving [8] as semantic foundation, we subsequently give an informal introduction to the features we used, mostly borrowed from the material in [13, 14]. In general, *clingo* offers several high-level constructs to realize reasoning processes that tolerate evolving problem specifications to enable the dynamic addition and removal of rules and data over time. This type of continuous solving interleaved with data manipulation is also known as Multi-Shot Solving. For our implementation, we heavily rely on the `#external` directive that allows to declare input atoms that may be instantiated by rules added later on. To this end, a directive like '`#external p(X,Y) : q(X,Z), r(Z,Y).`' is treated similar to a rule '`p(X,Y) :- q(X,Z), r(Z,Y).`' during grounding. However, the head atoms of the resulting ground instances are merely collected as inputs, whereas the ground rules as such are discarded. Once grounded, the truth value of external atoms can be changed via *clingo*'s API. By default, the initial truth value of external atoms is set to false. Then, for example, with *clingo*'s Python API, function call `assign_external(clingo.parse_term('p(a,b)'),True)` can be used to set the truth value of the external atom `p(a,b)` to true.

With the `#external` directive, we can also declare input atoms to resemble a sliding window, e.g. via

```
#external window(A,WT) : WT=1-wsize..0, watom(A).
```

Intuitively, the atoms in the sliding window are represented as external atoms over predicate `window/2` where `A` is an atom occurring in the stream at the current time step offset by `WT`. Further, `wsize` denotes the window size and `WT=1-wsize..0` states the domain of `WT` ranging from `1-wsize` to `0`, i.e, the time steps covered by the window relative to the current time step. To advance the window by one time step, we can toggle the truth values of the `window/2` atoms via *clingo*'s python API accordingly, as detailed above.

At its core, our technical implementation[2] relies on a *controller* module for the overall workflow shown in Listing 1.1. Its `main` function in Line 30-45 expects a *streamed program* as input derived from abstract class `AbstractStreamedProgram` in Line 20-27. A streamed program is custom to a problem and constitutes 1) the ASP problem encoding to apply, 2) the sliding window size, and 3) a function `next` that returns and afterwards removes the first element in the stream. With this setup, the `main` function is able to evaluate the problem encoding on the stream in a sliding window fashion. More precisely, in Line 33, the clingo control object gets initialized with the window size passed as a constant. Afterwards, in Line 34-35, the problem encoding provided by the streamed program and the declaration of external `window/2` in line 6 are grounded. As described previously, `window/2` represents the atoms of the current window and will be used in sequel to dynamically update those according to the current time step. Line 40-45 facilitate a loop that iteratively solves the problem for the atoms in the current window

[2] Source code available at `https://github.com/potassco/aspStream`

until the end of the stream. That is, Line 41-42 update the current window's atoms by calling function `set_externals` to set newly added and expired atoms of external `window/2` to true or false, respectively. With the updated ground program, the solve method of clingo is called in line 44 utilizing our custom `on_model` method in line 9-11 to print the solve result to the standard output. Eventually, the sliding window is advanced by one time step by calling `next` in Line 45. In general, this loop can also be re-designed to advance the window more than one time step per cycle by adding the result of multiple `sp.next()` calls to the currently considered window atoms.

```
5  externals = """
6  #external window(A,WT) : WT=1-wsize..0, watom(A).
7  """

9  def on_model(m):
10     show = "␣".join([str(i) for i in m.symbols(shown=True)])
11     print("Answer:\n{}".format(show))

13 def set_externals(ctl, window):
14     for idx, item in enumerate(window):
15         for atom, value in item:
16             watom = clingo.parse_term("window("+atom+",-"+str(idx)+")")
17             ctl.assign_external(watom, value)

20 class AbstractStreamedProgram:

22     def __init__(self):
23         self.base = ""
24         self.wsize = 1

26     def next(self):
27         return None

30 def main(sp):

32     # ground base
33     ctl = clingo.Control(["-c␣wsize={}".format(sp.wsize)])
34     ctl.add("base",[],sp.base + externals)
35     ctl.ground([("base",[])])

37     # solve until end of the stream
38     window = [[]]*sp.wsize
39     item = sp.next()
40     while item is not None:
41         window = [item] + window[:-1]
42         set_externals(ctl, window)
43         print("\nSolving...")
44         ctl.solve(on_model=on_model)
45         item = sp.next()
```

Listing 1.1. Controller module of the stream reasoner written in Python[3].

At this juncture, we apply our devised concepts to the job shop scheduling problem from Section 2. Since we intend to reuse our existing JSP encoding, we introduce a new predicate `request/3` that has the same signature as `jobs/3` but captures the notion of dynamically arriving job requests from the stream. For instance, consider a stream of length 2 where at time step 1 two job requests arrive: request 1 for machine 1 of duration 4, and request 2 for machine 2 of duration 2. Further, at time step 2 a single request arrives: request 1 for machine 2 of duration 3. With `request/3`, we can express this as a stream of ASP atoms as follows

[3] Full source code available at `https://github.com/potassco/aspStream/blob/master/aspStream/stream_controller.py`

time step	window atoms
1	`request(1,1,4). request(2,2,2).`
2	`request(1,2,3).`

where {`request(1,1,4), request(2,2,2)`} and {`request(1,2,3)`} are the set of arriving ASP atoms for time steps 1 and 2, respectively. Then, processed by the controller function `set_externals` (Listing 1.1, Line 13-17) at time point 1, the following atoms of the sliding window external `window` would hold:

```
window(request(1,1,4),0), window(request(2,2,2),0)
```

Processed again for time point 2 and assuming that the window size is at least 2 time steps, `set_externals` would change the true atoms of external `window` to

```
window(request(1,1,4),-1), window(request(2,2,2),-1),
window(request(1,2,3),0)
```

To effectively support the scheduling of streamed requests, we still have to supplement our encoding in Section 2, e.g. with the following rules[4]:

```
1  #const jobs=4. #const requests=2. #const machines=2.
3  { assign((R,WT),J) : J=1..jobs } = 1 :- window(request(R,_,_),WT).
5  :- assign((R1,WT1),J), assign((R2,WT2),J), (R1,WT1) < (R2,WT2).
7  job(J,M,D) :- assign((R,WT),J), window(request(R,M,D),WT).
9  watom(request(R,M,D)) :- R=1..requests, M=1..machines, D=1..horizon.
11 #show wstart(request(R,M,D),WT,T) : window(request(R,M,D),WT),
12                                     assign((R,WT),J), start(J,T).
```

The overall idea of this addition is the mapping of arriving `request/3` atoms to `job/3` atoms such that our previous encoding can be reused for solving. Specifically, in Line 1 constants `jobs` and `requests` state the maximum number of streamed requests and processing jobs per time step, respectively, and `machines` states the number of available machines. The assignment of requests to jobs is accomplished by Line 3-7: the cardinality rule in Line 3 assigns each request `R` within the scope of the sliding window at relative time point `WT` to a job expressed by the implied `assign/2` atom. To ensure that different requests cannot be assigned to the same job, we add the constraint in Line 5. For every assigned request, we yield the respective `job/3` atom in Line 7. Recalling our declaration of external `window/2` from above, we must also define the domain of possible atoms occurring in the sliding window via `watom/1` in Line 9. The `#show` statement in Line 11-12 formats the solution as `wstart/4` terms which state for each request `request(R,M,D)` in the window at relative time point `WT` its absolute starting time point `T`. By setting up a custom streamed program[5] with window size of at least 2 and our extended encoding, we can use the controller to schedule the request in the example stream above: after the first time step we retrieved

[4] Full source code available at `https://github.com/potassco/aspStream/blob/master/aspStream/stream.lp`

[5] Full source code available at `https://github.com/potassco/aspStream/blob/master/aspStream/streamed_program_job.py`

stable model {`wstart(request(1,1,4),0,1)`,`wstart(request(2,2,2),0,3)`} which suggest to schedule request 1 and 2 at time point 1 and 3, respectively. After solving again at time point 2, we get stable model {`wstart(request(1,1,4),-1,1)`, `wstart(request(2,2,2),-1,1)`, `wstart(request(1,2,2),0,3)`} which suggest to schedule request 1 and 2 arrived at relative time point -1 both to absolute time point 1, and request 1 arrived at relative time point 0 to absolute time point 3. A notable drawback of this approach is that we reschedule all previously submitted jobs. Specifically, jobs that were started before the current timepoint may be reassigned to another start time. As a remedy, we developed an extended variant that considers already running jobs and occupied machines. For reasons of space, we skip a detailed explanation here and refer to our encoding[6].

4 Discussion

There have been a host of logic and ASP-based approaches in recent years. For ontological knowledgebases, [15] proposes a stream reasoner with time-annotated RDF data that may expire. To bridge the gap between purely event-driven systems and rule-based reasoning, the first-order language ETALIS [16] was developed. The ASP-based language LARS [17] adds language primitives for time and data-based window operators as wells as temporal modalities to compare streams and windows. An implementation of LARS based on *clingo* is presented in [18]. In a broader sense, *clingo* by itself can be seen as a general toolkit to facilitate dynamic workflows based on external updates. Our work in [8–10] presented a first concept for incremental stream reasoning with ASP.

In conclusion, we laid out an pragmatic approach to leverage *clingo*'s multi-shot solving capabilites for stream reasoning. To this end, we presented a basic implementation of a stream reasoner and showcased how to realize a sliding window using *clingo*'s Python API. Moreover, we demonstrated our devised concepts by applying them to an online-version of job shop scheduling. For the future, we aim to elaborate more advanced design patterns from a implementation point of view. In terms of theoretical conception, we plan to particularize the link to time-decay logic programs and potentially expand their semantics.

References

1. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems **24**(6) (2009) 83–89
2. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: a survey and outlook: A summary of ten years of research and a vision for the next decade. Data Science Journal **1** (2017)

[6] The extended encoding and an example instance can be found at `https://github.com/potassco/aspStream/blob/master/aspStream/stream_extended.lp` and `https://github.com/potassco/aspStream/blob/master/aspStream/examples/stream_instance3.lp`, respectively.

3. Lifschitz, V.: Answer set planning. In de Schreye, D., ed.: Proceedings of the International Conference on Logic Programming (ICLP'99), MIT Press (1999) 23–37
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
5. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In Brewka, G., Eiter, T., McIlraith, S., eds.: Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12), AAAI Press (2012) 613–617
6. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Extended version. Available at `http://www.cs.uni-potsdam.de/wv/publications/` (2012).
7. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Answer set programming for stream reasoning. In Fink, M., Lierler, Y., eds.: Proceedings of the Fifth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'12). (2012)
8. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Extended report. Technical report, Universität Potsdam (2014)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Leuschel, M., Schrijvers, T., eds.: Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14). Volume 14(4-5) of Theory and Practice of Logic Programming, Online Supplement. (2014) Available at `http://arxiv.org/abs/1405.3694v1`.
10. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In Carro, M., King, A., eds.: Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16). Volume 52., Open Access Series in Informatics (OASIcs) (2016) 2:1–2:15
11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming (2018) To appear.
12. Golab, L., özsu, M.: Data Stream Management. Synthesis Lectures on Data Management. Morgan and Claypool Publishers (2010)
13. Gebser, M., Kaminski, R., Obermeier, P., Schaub, T.: Ricochet robots reloaded: A case-study in multi-shot ASP solving. In Eiter, T., Strass, H., Truszczyński, M., Woltran, S., eds.: Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation: Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday. Volume 9060 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2015) 17–32
14. Kaminski, R., Schaub, T., Wanko, P.: A tutorial on hybrid answer set solving with clingo. In Ianni, G., Lembo, D., Bertossi, L., Faber, W., Glimm, B., Gottlob, G., Staab, S., eds.: Proceedings of the Thirteenth International Summer School of the Reasoning Web. Volume 10370 of Lecture Notes in Computer Science., Springer-Verlag (2017) 167–203
15. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In Aroyo, L., Antoniou, G., Hyvönen, E., Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T., eds.: Proceedings of the Seventh Extended Semantic Web Conference (ESWC'10). Volume 6088 of Lecture Notes in Computer Science., Springer-Verlag (2010) 1–15

16. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A rule-based language for complex event processing and reasoning. In Hitzler, P., Lukasiewicz, T., eds.: Proceedings of the Fourth International Conference on Web Reasoning and Rule Systems (RR'10). Volume 6333 of Lecture Notes in Computer Science., Springer-Verlag (2010) 42–57
17. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: A logic-based framework for analyzing reasoning over streams. 1431–1438
18. Beck, H., Eiter, T., Folie, C.: Ticker: A system for incremental asp-based stream reasoning. Theory and Practice of Logic Programming **17**(5-6) (2017) 744–763