Stream Packing for Asynchronous Multi-Context Systems using ASP*

Stefan Ellmauthaler and Jörg Pührer

Institute of Computer Science,
Leipzig University, Germany,
{ellmauthaler,puehrer}@informatik.uni-leipzig.de

Abstract. When a processing unit relies on data from external streams, we may face the problem that the stream data needs to be rearranged in a way that allows the unit to perform its task(s). On arrival of new data, we must decide whether there is sufficient information available to start processing or whether to wait for more data. Furthermore, we need to ensure that the data meets the input specification of the processing step. In the case of multiple input streams it is also necessary to coordinate which data from which incoming stream should form the input of the next process instantiation. In this work, we propose a declarative approach as an interface between multiple streams and a processing unit. The idea is to specify via answer-set programming how to arrange incoming data in packages that are suitable as input for subsequent processing. Our approach is intended for use in *asynchronous multi-context systems* (aMCSs), a recently proposed framework for loose coupling of knowledge representation formalisms that allows for online reasoning in a dynamic environment. Contexts in aMCSs process data streams from external sources and other contexts.

Keywords: multi-context systems, stream reasoning, answer-set programming

1 Introduction

The omnipresence of smart devices and recent advancements towards a Semantic Web and the Internet of Things has increased the interest in reasoning over streaming data (e.g., [6, 9, 14, 17, 2]). In this work, we take a closer look at *asynchronous multi-context systems* (aMCSs) [9]. The aMCS framework allows for loose coupling of knowledge representation formalisms and, at the same time, for online reasoning in a dynamic environment. Unlike other recent proposals for online multi-context systems [6, 14], contexts of an aMCS run independently of each other, i.e., there are no synchronised time steps in which all contexts exchange information. Instead, after each evaluation of a context, it sends its results to other contexts and, from the perspective of the receiving side, a context can continuously receive data from other contexts or the outside world in an unpredictable order. As the context needs some time for each evaluation and since there might not always be sufficient or the right kind of data in order to start another

* This work has been partially supported by the German Research Foundation (DFG) under grants BR-1817/7-1 and FOR 1513.

evaluation, the incoming data needs to be buffered. In this work we deal with the problem of how and when to take information from the buffer to hand it over to the context for processing.

In a running example we deal with an aMCS with a context that can assign ambulances to an emergency case. The input buffer of this context can contain information about multiple unassigned cases and multiple available ambulances. We need to ensure that for every evaluation we only pass on information about a single emergency case (e.g., the one with the highest priority) but all information about available rescue units, in order to meet the input requirements of the knowledge base of the context. On the other hand, if no information about a case or no ambulance is available yet, the context cannot start, so we need to wait for more data. We propose to use answer-set programming (ASP) [19, 21, 1] to declaratively specify when a context may start processing and how to combine the arrived information into logical packages suitable for the context. Decisions are based on meta information such as the amount of available data or the source of the information. Furthermore, by tagging incoming data, e.g., also information about time of arrival or the content of the data can be taken into account. With this approach we give the context to possibility to filter the data it receives. We define a range of dedicated ASP atoms that can be used in answer-set programs to specify packages and manipulate data in the input buffer. In addition, we discuss creating multiple packages at once and how we can exploit ASP optimisation features.

The remainder of this work is structured as follows: Next, we will give some background on aMCS in form of a short overview and introduce the example scenario. Then, the main approach is introduced in Section 3 and discussed along various illustrative examples. Finally, we will conclude in Section 4.

2 Asynchronous Multi-Context Systems

We next introduce the aMCS framework, focusing on aspects relevant to this work. In particular, we describe the asynchronous semantics of aMCSs from the perspective of a single context, and refer the interested reader to [9] for a precise characterisation of the semantics of an aMCS as a whole. Every context in an aMCS is associated with a *logic suite* [5] which can be seen as an abstraction of different KR formalisms. A logic suite is a triple $\mathcal{LS} = \langle \mathcal{KB}, \mathcal{BS}, \mathcal{ACC} \rangle$, where $\mathcal{KB}$ is the set of admissible knowledge bases (KBs) of $\mathcal{LS}$. $\mathcal{BS}$ is the set of possible belief sets of $\mathcal{LS}$, whose elements are *beliefs*. $\mathcal{ACC}$ is a set of semantics for $\mathcal{LS}$: a semantics for $\mathcal{LS}$ is a function $\mathrm{ACC} : \mathcal{KB} \to 2^{\mathcal{BS}}$ assigning to each KB a set of acceptable belief sets. Using a semantics with potentially more than one acceptable belief set allows for modelling non-determinism, where each belief set corresponds to an alternative solution. We assume a set $\mathcal{N}$ of *names* that serve as labels for sensors, contexts, and output streams. A *context* is a pair $C = \langle \mathsf{n}, \mathcal{LS} \rangle$ where $\mathsf{n} \in \mathcal{N}$ is the name of the context and $\mathcal{LS}$ is a logic suite.

Definition 1 ([9]). *An aMCS (of length n with m output streams) is a pair $M = \langle \mathsf{C}, \mathsf{O} \rangle$, where $\mathsf{C} = \langle C_1, \ldots, C_n \rangle$ is an n-tuple of contexts and $\mathsf{O} = \langle \mathsf{o}_1, \ldots, \mathsf{o}_m \rangle$ with $\mathsf{o}_j \in \mathcal{N}$ for each $1 \leq j \leq m$ is a tuple containing the names of the output streams of M.*

A context in an aMCS communicates with other contexts and the outside world by means of streams of data. In particular, every context has an input stream on which information

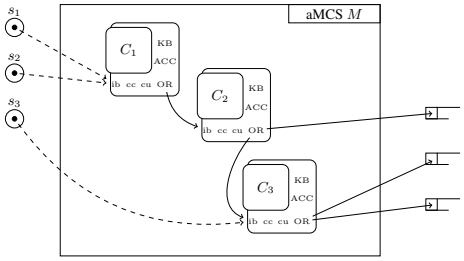

Fig. 1. An aMCS with three contexts, three sensors on the left side, and three output streams on the right side. A solid line represents a flow of information from a context to its stakeholder streams, whereas a dashed line indicates sensor data written to the input buffer of a context.

can be written from both external sources (called sensors) and internal sources (i.e., other contexts). For the data in the communication streams we assume a communication language $\mathcal{IL}$ where every $\mathsf{i} \in \mathcal{IL}$ is an abstract *piece of information*. Both, the data in the input stream of a context and the data in output streams are modelled by *information buffers* that are defined in the following.

Definition 2 ([9]). *A* (source-tagged) data set *is a pair* $d = \langle \mathsf{n}, I \rangle$*, where* $\mathsf{n} \in \mathcal{N}$ *is either a context name or a sensor name, stating the* source *of* d*, and* $I \subseteq \mathcal{IL}$ *is a set of pieces of information. An* information buffer *is a sequence of data sets*[1].

We continue with a big picture on the mode of operation of an aMCS. Every context in an aMCS asynchronously decides whether the data that is currently available on its input information buffer ib is sufficient to start a computation using the current knowledge base KB of the context. That is done by the *computation controller* cc of the context, formally a binary relation that holds all pairs of input buffers and knowledge bases for which a computation should begin. When the computation controller decides to start a computation, first, the context is updated using the *context update function* cu: it maps the input buffer ib (as it was when the computation has started) and the current knowledge base KB of a context C to a new *configuration* of C, where a configuration of C comprises the knowledge base KB, the semantics ACC, and the whole *context management* of C, that is, the computation controller cc, set of *output rules* OR (see below), and the context update function cu itself. That means, except for its unique name and the underlying logic suite, a context can change completely after every computation. Then, one-by-one, the acceptable belief sets of the updated knowledge base are computed according to the updated semantics ACC. Each of the computed belief sets is matched against the set of output rules of the context. A belief set can activate a rule and thereby determine which information should be sent to which stakeholders of the context (these can be other contexts or output streams of the aMCS). A computation finishes after the final acceptable belief set has been computed. Then, notifications about the end of the computation are sent to the stakeholders. Figure 1 depicts an aMCS with three contexts.

Preparing the data in the input information buffer of a context for processing is the focus of this paper. Therefore, we now have a closer look at this data. As stated

[1] Note that data sets are called *data packages* in [9]. We changed the name to avoid confusion with packages as introduced later in the paper.

in Definition 2 it consists of data sets. These packages can come from two different sources: a data set $d = \langle \mathsf{n}, I \rangle$ can origin from a sensor named n, in which case the aMCS framework simply assumes the package appears at some time point in the buffer and I contains the corresponding sensor readings; in the other case, the package is sent from another context. Then, n is the name of this context C and I contains either the dedicated symbol $\mathrm{EOC} \in \mathcal{IL}$ that notifies that C has finished a computation or pieces of information generated by the output rules of C: an *output rule* r for context $C = \langle \mathsf{n}, \mathcal{LS} \rangle$ is an expression of the form

$$\langle \mathsf{n}', \mathsf{i} \rangle \leftarrow b_1, \ldots, b_j, \mathrm{not}\ b_{j+1}, \ldots, \mathrm{not}\ b_m, \tag{1}$$

such that $\mathsf{n}' \in \mathcal{N}$ is the name of a context or an output stream, $\mathsf{i} \in \mathcal{IL}$ is a piece of information, and every b_ℓ ($1 \leq \ell \leq m$) is a belief for C, i.e., $b_\ell \in S$ for some $S \in \mathcal{BS}$ where $\mathcal{BS}$ is the set of possible belief sets of $\mathcal{LS}$. We call n' the *stakeholder* of r, $\langle \mathsf{n}', \mathsf{i} \rangle$ the head of r denoted by $hd(r)$, and $b_1, \ldots, b_j, \mathrm{not}\ b_{j+1}, \ldots, \mathrm{not}\ b_m$ the body $\mathrm{bd}(r)$ of r. Moreover, we say that r is active under S, denoted by $S \models \mathrm{bd}(r)$, if $\{b_1, \ldots, b_j\} \subseteq S$ and $\{b_{j+1}, \ldots, b_m\} \cap S = \emptyset$. Intuitively, the stakeholder is a reference to the addressee of information i and when an output rule becomes active, the information is sent to the stakeholder, i.e., the input stream of another context or an output stream of the aMCS. The *output* of C with respect to a set OR of output rules for C under a belief set $S \in \mathcal{BS}$ relevant for n' is the data set

$$d_C(S, \mathrm{OR}, \mathsf{n}') = \langle \mathsf{n}, \{i \mid r \in \mathrm{OR}, hd(r) = \langle \mathsf{n}', \mathsf{i} \rangle, S \models \mathrm{bd}(r)\} \rangle.$$

So, intuitively, if n' refers to some context C' and C has computed belief set S, then $d_C(S, \mathrm{OR}, \mathsf{n}')$ will be put on the information buffer that represents the input stream of C'. Note that the first component of the data set is the name n, serving as reference to context C as the source of information. To summarise, the head of an output rule determines the name n' of target and the name n in a data set points to the source of the information.

As a running example, we reuse a scenario from previous work [9] dealing with the coordination and handling of assignments of medical rescue units. An aMCS consisting of five context is used to model the scenario. It is depicted in Figure 2. The system gives assistance during the rescue call, helps in assigning priorities and rescue units to a case, and assists in the necessary communication among all involved parties.

In our examples we will deal with the following contexts:

Case Analyser This context implements a computer-aided call handling system which assists an emergency response employee during answering an emergency call. It chooses which questions need to be asked based on previous answers and checks whether answers are inconsistent (e.g., amniotic sac bursts when the gender is male). For these purposes the case analyser context may also consult a medical ontology represented by another context. The communication with the ER employee is represented, on the one hand, as a sensor that reads the input of the employee and, on the other hand, by an output stream which prints the questions and results on a computer screen. During the collection of all the important facts for this emergency case, the analyser computes the priority of the case and passes it to the task planner.

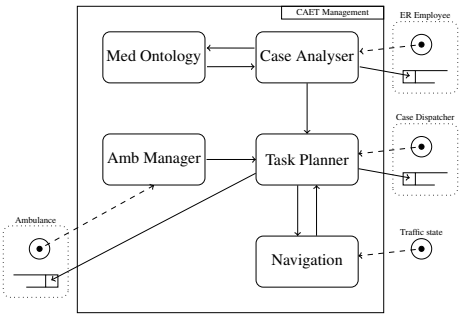

Fig. 2. An aMCS for Computer-Aided Emergency Team Management

Task Planner This context keeps track of emergency cases. Based on the priority and age of a case and the availability and position of ambulances it suggests an efficient plan of action for the ambulances to the (human) case dispatcher. The dispatcher may approve some of the suggestions or all of them. If the dispatcher has no faith in the given plan of action, she can also alter it at will. These decisions are reported back to the planning system such that it can react to the alterations and provide further suggestions. Based on the final plan, the task planner informs the ambulance about their new mission.

Amb Manager The ambulance manager is a database, which keeps track of the status and location of ambulance units. Each ambulance team reports its status (e.g., to be on duty, waiting for new mission, ...) to the database (modelled by the sensor "Ambulance"). Additionally, the car periodically sends GPS-coordinates to the database. These updates will be pushed to the task planner.

3 Declarative Packing of Data Sets

Unlike previous proposals for heterogeneous multi-context systems, in the static as well as the dynamic setting [4–6, 14], aMCSs do not use a synchronised equilibria semantics. For applications that do not require the tight semantic integration offered by equilibria, aMCSs have the advantage that each context can run at its own pace and that the overall computational complexity is in general significantly lower. A natural consequence of asynchronicity is that we cannot use bridge rules that are typically used in multi-context systems because they depend on synchronised belief sets (so-called belief states). Instead, output rules are used that are defined by the context that sends information rather than the context that receives it. As a consequence, the receiving context is not in control of what data it receives on its input buffer and in what order the data arrives. We propose to declaratively specify *packages of data sets* that the context accepts as input. The goal is to allow the system to automatically pack incoming data in a form suitable as input for the logic of the context. At the same time, the method gives the context control over the data it receives.

As means of specification we propose to rely on answer-set programming. It offers a rich language for representing complicated problems and efficient ASP solvers are

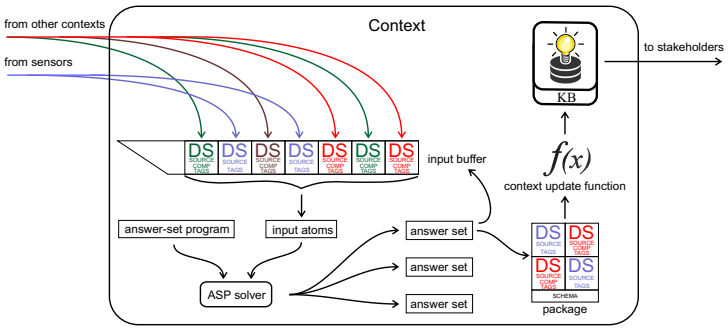

Fig. 3. Overview of the approach. Data sets from other contexts and sensors are written into the input buffer of a context. Different colours of data sets mean that they are from a different source. The data in the input buffer is represented as input atoms to an answer-set program. One of the resulting answer sets is chosen. It contains directives on how to change the data on the input buffer as well as how to pack data sets. Resulting packages are associate with a package schema. Based on the schema and the chosen data sets, the context update functions determines changes of the knowledge base, the semantics to use, as well as changes in the output rules of a context. After evaluation, the context sends it output to its stakeholders.

available. By using ASP, we profit from its elaboration tolerance (small changes in the requirements typically require only minimal changes of the program) and its elegant and concise declarative specifications. Regarding computational complexity, the packing and filtering tasks we deal with will in many cases not require the full expressive power of ASP from a complexity theoretic point of view. Often, very efficient fragments of ASP (such as programs with stratified negation) will suffice. Nevertheless, there can be cases where it is handy to exploit the capability of ASP to solve problems from NP and beyond, e.g., when an aMCS combines pre-existing contexts that are not expressive enough to solve a required task. Additionally, ASP offers different optimisation frameworks that we also want to exploit in our approach.

An answer-set program is a set of logic-programming rules of the form

$$a \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \tag{2}$$

where a, and every b_i for $1 \leq i \leq n$ are first order atoms and not denotes *default negation*. The intuition is that if one knows that all atoms $b_1, \ldots, b_m$ are true and there is no evidence that some $b_{m+1}, \ldots, b_n$ is true, than one can derive that a is true. The overall semantics of an answer-set program is given by its answer sets: dedicated models of the program that satisfy the implications of the program rules and adhere to certain minimality conditions. For a formal account of (different) ASP semantics we refer to the vast body of literature on ASP (starting points are [1, 18]). We assume familiarity with the ASP language of the `clingo` solver [10, 12] that supports aggregates, function symbols, user-defined functions, and further popular language extensions[2].

[2] We will make uses of `prolog` style lists which are not part of the `clingo` language but can easily be simulated by function symbols. As the simulation relies on nested functions it leads to heavy use of bracketing which we avoid for the sake of readability.

The idea is that the aMCS engine provides information about the data sets on a context's input stream in form of ASP facts. They are the input of an answer-set program that decides

- if the data in the buffer is sufficient for passing it on for processing (i.e., to the update function and subsequently to the knowledge base);
- which data sets should form a package that is passed on for processing; and
- which data sets should be deleted from or remain on the input buffer.

These decisions are provided by directives encoded in atoms in the resulting answer sets. When and how often and the ASP evaluation should take place depends on the needs of the application. One could, e.g., re-evaluate whenever new data arrives on the input buffer or use fixed time intervals to reduce computation efforts. Regarding the components of an aMCS, the answer-set program can be seen as an implementation of the computation controller cc of a context as it decides when to start a computation. Moreover, it partially implements the context update function cu, as it can remove from or leave data set on the input buffer. The packages of data sets generated by the answer-set program are input to the components of the aMCS engine that implements the remaining functionality of the update function. The overall method is illustrated in Figure 3.

Example 1. Consider an aMCS for the emergency team management scenario as in Figure 2. The task planner context receives information about available ambulances from the ambulance manager context and emergency cases to assign from the case analyser. For now, we assume that every data set from the case manager represents one case that needs assignment. A sensible input for the task planner's knowledge base consists of exactly one case that needs assignment (e.g., the one with the highest priority—cf. Section 3.2) and all information about currently available ambulances. A representation of a possible input buffer could contain the following facts

```
ds_avail(ca_ds11). ds_avail(ca_ds12).
ds_avail(am_ds54). ds_avail(am_ds55). ds_avail(am_ds56).
source(ca_ds11,ctxt_case_anl). source(ca_ds12,ctxt_case_anl).
source(am_ds54,ctxt_amb_mng). source(am_ds55,ctxt_amb_mng).
source(am_ds56,ctxt_amb_mng).
```

stating which data sets are available and what their source is, e.g., `ds_avail(ca_ds11)` encodes that a data set identified by the constant `ca_ds11` is available in the buffer and `source(ca_ds11,ctxt_case_anl)` that the data set stems from the case analyser context.

The following answer-set program can be used for packing.

```
1  aux_case_avail :- ds_avail(DS),source(DS,ctxt_case_anl).
2  aux_ambulance_avail :- ds_avail(DS),source(DS,ctxt_amb_mng).
3  process_as_schema(sch1) :- aux_case_avail,
       aux_ambulance_avail.
4  in_pack(DS) :- ds_avail(DS), source(DS,ctxt_amb_mng).
5  1 {aux_case_in_pack(DS) : ds_avail(DS), source(DS,
       ctxt_case_anl)} 1.
6  in_pack(DS) :- aux_case_in_pack(DS).
7  rm_pack.
```

The first two rules derive auxiliary atoms that indicate whether data sets from the case analyser, respectively, the ambulance manager are available. If both is the case the rule in line 3 derives the directive `process_as_schema(sch1)` which means that the selected data sets form a package of schema `sch1` and should be passed on for processing. A schema can be seen as the type of the package that may influence how the context processes the package. The selection of data sets is expressed by atoms of the unary predicate `in_pack`: the fourth rule adds all data sets that come from the ambulance manager to the package; the rule in line 5 is a fact enforcing a cardinality constraint. It expresses that exactly one data set from the case analyser, thus the information about a single open case, should be part of the package. The rule in line 3 derives a corresponding `in_pack` directive atom for the `aux_case_in_pack` atom. Finally, the directive `rm_pack` removes all data sets from the input buffer that are part of the chosen package.

Together with the input facts the program has two answer sets, one containing the directives

```
in_pack(ca_ds12), in_pack(am_ds54), in_pack(am_ds55),
in_pack(am_ds56), process_as_schema(sch1), rm_pack
```

and the other one the same but `in_pack(ca_ds11)` instead of `in_pack(ca_ds12)`.

An overview over all input and directive atoms is given in Table 1. As illustrated by Example 1, we propose to represent data sets on the object level of the ASP language. The same we can do with computations, where the constants that identify individual data sets and computations are generated at runtime by the aMCS engine. The identifiers for contexts or input streams are their names from $\mathcal{N}$.

Example 2. Remember that the case analyser is a context responsible for the collection of data about an emergency case during a phone call between someone in need and a human employee. This collection can be a lengthy process and the data available about the case can evolve over time. Hence, let us now assume that the case analyser provides not only a single data set for one case but multiple ones such that the whole handling of a case is considered a computation that has multiple acceptable belief states representing different states of refinement of a case. Thus, we can have multiple data sets for the same computation of the case analyser in the input buffer of the task planner. Moreover, due to the asynchronous nature of aMCSs, it can happen that there are data sets of multiple computations available at the same time. Therefore, we need to be able to distinguish to which computation a data set belongs to. An input for packing could then consist of the facts given next.

```
ds_avail(ca_ds21). ds_avail(ca_ds22).
ds_avail(ca_ds24). ds_avail(ca_ds25).
ds_comp(ca_ds21,ca_comp35). ds_comp(ca_ds22,ca_comp35).
ds_comp(ca_ds24,ca_comp36). ds_comp(ca_ds25,ca_comp36).
ds_avail(am_ds54). ds_comp(am_ds54,am_comp61).
eoc(ca_comp35). eoc(am_comp61).
source(ca_ds21,ctxt_case_anl). source(ca_ds22,ctxt_case_anl).
source(ca_ds24,ctxt_case_anl). source(ca_ds25,ctxt_case_anl).
source(ca_comp35,ctxt_case_anl). source(ca_comp36,ctxt_case_anl).
```

Atom	Type	Description
`ds_avail(ds)`	Input	Data set `ds` is available on the input buffer
`ds_comp(ds,comp)`	Input	Data set `ds` belongs to computation `comp`
`source(comp,ctxt)`	Input	Computation `comp` is a computation of context `ctxt`
`source(ds,ctxt)`	Input	Data set `ds` originates from context `ctxt`
`source(ds,therm)`	Input	Data set `ds` originates from sensor `therm`
`eoc(comp)`	Input	Computations `comp` has ended.
`tag(comp,solves(probl1))`	Input	Computations `comp` is tagged with function `solves(probl1)`
`tag(ds,"optimum")`	Input	Data set `ds` is tagged with string `"optimum"`
`time(1000)`	Input	An external clock provides `1000` as current time
`ignore(comp)`	Directive	Ignore future data sets of computation `comp`
`add_tag(comp,best(3))`	Directive	Tag computation `comp` with function `best(3)`
`rm_tag(comp,"trusted")`	Directive	Remove the tag `"trusted"` from computation `comp`
`rm(comp)`	Directive	Remove all data sets of computation `comp` from the input buffer
`rm(ds)`	Directive	Remove data set `ds` from the input buffer
`rm_pack`	Directive	Remove all data sets of processed packages from the input buffer
Variant: one package per answer set		
`in_pack(ds)`	Directive	Data set `ds` is considered part of the package
`process_as_schema(sch)`	Directive	The data sets defined by `in_pack/1` atoms form a package of schema `sch` and are passed on for processing
Variant: multiple packages per answer set		
`process(sch,[ds1,ds3,ds7])`	Directive	The data sets in the list `[ds1,ds3,ds7]` form a package of schema `sch` and are passed on for processing

Table 1. Input and directive atoms for package specifications. Atoms of other predicates can be used as needed and are considered auxiliary.

Compared to Example 1, the additional binary predicate `ds_comp` states to which computation a data set belongs and the use of `source` atoms is extended to computations. The `eoc` atoms state which computations have finished (this is useful if the context needs to wait for all data sets of a computation). In our setting, for every emergency case it is always the latest data set provided by the case analyser that we want to consider in the task planner. For identifying these in the answer-set program we can use a comparison relation that indicates the order of arrival of data sets. In this example, we assume that the constants representing data sets are assigned by system in a way that the internal term comparison relation of `clingo` respects the arrival order. Alternatively, the system could provide the arrival order by further input facts. We can adapt the previous packing program as follows.

```
1  aux_case_avail :- ds_avail(DS),source(DS,ctxt_case_anl).
2  aux_ambulance_avail :- ds_avail(DS),source(DS,ctxt_amb_mng).
3  process_as_schema(sch1) :- aux_case_avail,
       aux_ambulance_avail.
4  in_pack(DS) :- ds_avail(DS), source(DS,ctxt_amb_mng).
5  1 {aux_selected_case_comp(CO) : source(CO,ctxt_case_anl),
       ds_comp(DS,CO)} 1.
6  in_pack(MDS) :- MDS = #max{DS : ds_avail(DS), ds_comp(DS,CO),
        aux_selected_case_comp(CO)}.
7  rm_pack.
```

The rule in line 5 selects one of the computations from the case analyser context for which a data set is available. The next rule selects the latest data set from this computation (that with the maximum identifier according to the term comparison relation of `clingo`) to be part of the package. We get two answer sets, one including `in_pack(ca_ds22)` representing the latest state of the case covered in computation `ca_comp35` and the other one directive `in_pack(ca_ds25)` selecting the latest data set `ca_ds25` of computation `ca_comp36`.

3.1 Tagging

Both examples showed how we can select a data set to be part of a package based on meta information such as the number of available data sets or their order. That is, we did not use any information about the content or purpose of the data set itself. To gain flexible control for the selection we propose to use a *tagging system* for data sets and computations. Every data set and every computation is associated with a set of arbitrary (ground) `clingo` terms that serve as tags.

Example 3. In Example 2, the latest data set of a computation that is available in the buffer is considered to describe the corresponding emergency case. This was based on the assumption that the consecutive data packages of a computation matches the evolution of data available for the case. By design of aMCSs, one context may only have one computation at a time. That means that, if we modelled the aMCS as in the previous example, if we first deal with a case a and then a new case b is handled by the case analyser, the previous computation has finished and we could not continue to refine data on case a. In order to allow such late refinements, we drop the assumption that a single computation of the case analyser contains all data sets to a case. Instead, we tag every data set with an uninterpreted function symbol `case(id,i)`, where `id` is an identifier for the case and `i` is the index of the current revision of the case data. In the new setting we could have the following input facts.

```
ds_avail(ca_ds26). ds_avail(ca_ds27). ds_avail(ca_ds28).
ds_comp(ca_ds26,comp37). ds_comp(ca_ds27,comp38).
ds_comp(ca_ds28,comp39).
source(ca_ds26,ctxt_case_anl). source(ca_ds27,ctxt_case_anl).
source(ca_ds28,ctxt_case_anl). source(comp37,ctxt_case_anl).
source(comp38,ctxt_case_anl). source(comp39,ctxt_case_anl).
tag(ca_ds26,case(c1,1)).tag(ca_ds27,case(c2,1)).
tag(ca_ds28,case(c1,2)).
```

There are three data sets from the case analyser, each being the result of a different computation. Yet, data sets `ca_ds26` and `ca_ds28` deal with the same emergency case as indicated by the tags. As `ca_ds28` has the higher index, this data set should be selected if this case is processed. We can also use tags for differentiating types of data sets. The following packing program distinguishes between data sets from the ambulance manager that state that an ambulance is available and data sets indicating a broken ambulance.

```
 1  aux_case_avail :- ds_avail(DS),source(DS,ctxt_case_anl).
 2  aux_ambulance_avail :- ds_avail(DS),source(DS,ctxt_amb_mng),
        tag(DS,"available"), not aux_some_amb_broken.
 3  aux_some_amb_broken :- ds_avail(DS),source(DS,ctxt_amb_mng),
        tag(DS,"broken").
 4  process_as_schema(sch1) :- aux_case_avail,
        aux_ambulance_avail, not aux_some_amb_broken.
 5  process_as_schema(sch2) :- aux_some_amb_broken.
 6  in_pack(DS) :- ds_avail(DS),source(DS,ctxt_amb_mng), not
        aux_some_amb_broken.
 7  in_pack(DS) :- ds_avail(DS),source(DS,ctxt_amb_mng), tag(DS,"
        broken").
 8  1 {aux_selected_case(C) : tag(DS,case(C,I)),ds_avail(DS),
        source(DS,ctxt_case_anl)} 1 :- not aux_some_amb_broken.
 9  in_pack(MDS) :- MI = #max{I:tag(DS,case(C,I)),ds_avail(DS),
        source(DS,ctxt_case_anl)}, tag(MDS,case(C,MI)), ds_avail(
        MDS), aux_selected_case(C), source(MDS,ctxt_case_anl).
10  rm_pack.
11  rm(DS) :- ds_avail(DS), source(DS,ctxt_case_anl),
        aux_selected_case(C), tag(DS,case(C,I)), not in_pack(DS).
```

The rule in line 8 derives one auxiliary atom indicating which case the answer set deals with. The next rule then adds the data set dealing with this case that has the highest index to the package.

Unlike the previous examples, the program defines packages of two different schemata: schema `sch1` contains one data set defining a case and all data sets about available rescue units. But as soon as some ambulance is broken, only packages of schema `sch2` are processed. These contain all data sets about broken ambulances but none about cases.

Finally, the example illustrates another feature for removing data sets from the buffer: the final rule ensures that data sets dealing with the chosen case that have a non-maximal index should be removed. Note that with the given encoding they are even removed if no package is processed: for the given input fact, the program does not derive the `process_as_schema` directive, since this time no ambulance is available. Still, for case `c1` the directive `rm(ca_ds26)` for removing data set `ca_ds26` is derived.

The example illustrated how we can use tags in the answer-set program but so far we did not address how these are assigned. We see multiple sensible mechanisms for doing so:

1. Tags can be assigned by the sending context. Formally, a function is added to the context that, given the current configuration of the context, assigns tags to the data sets and computations it produces. By taking the current configuration of the context

into account, the tags can also provide relevant meta data. For example, when a context solves an optimisation problem, a tag can indicate whether a data set is derived from an optimal accepted belief set.
2. It also makes sense when the receiving context can generate tags on reception of data sets in a similar way. This additional variant is useful, e.g., in a distributed setting when we are only in control of the receiving context or when multiple contexts are receiving the same data sets but require different tagging.
3. An aMCS engine could itself provide tags giving access to system parameters such as the time when a data set was created. The latter is interesting when we also have access to the current time in the answer-set program by a further input fact stating the current time. This way we can, e.g., wait for a minute after receiving a data set with a non-optimal solution whether a better solution is provided. In case it is not, the suboptimal solution will be passed on by the first evaluation of the packing program after the time-out.
4. The answer-set program itself can be extended to modify tags for computations and data sets that remain on the input buffer. Thus, by means of tags, the packing mechanism becomes a stateful system that can remember or forget information. We propose two directives `add_tag` and `rm_tag` for this purpose. Applications include strategies to avoid starvation of data sets and marking computations as trusted or untrusted based on the data sets they provide over time. In case that an ongoing computation is identified as not of interest to the context, we additionally introduce the directive `ignore(comp)` that permanently bans further packages of computation `ignore(comp)` from entering the input buffer.

3.2 Multiple Answer Sets and Multiple Packages

An important principle of answer-set programming is that a program encodes a problem whose solutions correspond to the answer sets of the problem. This is also a valid point of view for our packing programs: the answer sets we get correspond to solutions of a packing problem. As a consequence, when a problem has multiple solutions, we also deal with multiple answer sets and, indeed, all the programs in Examples 1-3 have multiple answer sets. Our proposal to deal with multiple answer sets is to let the aMCS engine compute at most one of them in the first place. The underlying assumption is that every answer set is equally well suited as all of them are considered a solution. Therefore, a situation where a program has answer sets that are considered inferior to others requires modifying the program such that only wanted answer sets remain. ASP offers many excellent features for specifying preferences and finding optimal solutions (see [1, 3, 7]).

Example 4. Remember that the packing program in Example 1 has two answer sets and that the output of the case analyser context contains information about the priority of an emergency case. So far we did not make use of this information. In order to do so we assume that the case analyser assigns tags specifying the case priority to its data sets. Then, in addition to the input given in Example 1, we could have

```
tag(ca_ds11,3). tag(ca_ds12,7).
```

as input atoms that hold the numeric priority values of the cases handled by the respective data sets. In order to get only one answer set dealing with case `ca_ds12` that has higher priority it suffices to add the following maximize statement.

```
1  #maximize{P:aux_case_in_pack(DS),tag(DS,P)}.
```

Another option for handling multiple answer sets is to consider them all, i.e., multiple packages would be passed on for processing and all the modifications of the input buffer appearing in some answer sets would be applied. This variant however would require very careful modelling. First, inconsistencies might occur, e.g., tags could be assigned and removed at the same time. Moreover, overlapping of answer sets might often be a problem: both answer sets in Example 1 contain the same data sets about available ambulances. That is, the same ambulance could subsequently be assigned for both cases. As ASP offers little means for reasoning across answer sets (an exception are optimisation features as in Example 4) such situations are hard to avoid or lead to complicated encodings.

If one wants to process multiple packages at a time, we suggest to encode all desired packages in one answer set using lists (or function symbols as stated in footnote 2). To this end, we propose to replace directives `in_pack` and `process_as_schema` by a single new one. The binary directive `process` takes as first argument the schema, similar as the argument of `process_as_schema`. The second argument is a list containing the data sets that form one of the packages to be processed. Deriving multiple `process` atoms we can create multiple packages.

Example 5. The following code divides the data sets about available ambulances in three disjunct packages.

```
1  aux_min_amb_ds(MDS) :- MDS = #min{DS : ds_avail(DS),source(DS
       ,ctxt_amb_mng)}.
2  aux_amb_in_between(DS1,DS2) :- ds_avail(DS1),source(DS1,
       ctxt_amb_mng), ds_avail(DS),source(DS,ctxt_amb_mng),
       ds_avail(DS2),source(DS2,ctxt_amb_mng), DS1<DS, DS<DS2.
3  aux_amb_ds_nr(MDS,0) :- aux_min_amb_ds(MDS).
4  aux_amb_ds_nr(DS2,I+1) :- aux_amb_ds_nr(DS1,I), ds_avail(DS2)
       ,source(DS2,ctxt_amb_mng), DS1<DS2, not
       aux_amb_in_between(DS1,DS2).
5  aux_build_package(I,I,DS) :- I=0..2,aux_amb_ds_nr(DS,I).
6  aux_build_package(I\3,I,[DS|T]) :- aux_build_package(I\3,I-3,
       T),aux_amb_ds_nr(DS,I).
7  package(sch,L) :- aux_build_package(M,MI,L),M=0..2,MI=#max{I:
       aux_build_package(M,I,_)}.
```

The first four rules assign every data set an index according to `clingo`'s built in order using the auxiliary binary predicate `aux_amb_ds_nr`. The next two rules create partial lists, where the first argument of the `aux_build_package` atoms is the number of the final list (always 0,1, or 2), the second argument is the index of the data set added to the list, and the third argument is the partial list. Note that `I\3` stands for `I` modulo `3`. Finally, the last rule derives the `package` directive with the maximum partial list with first argument 0,1, and 2, respectively. Consider the following input facts.

```
ds_avail(ca_ds26). ds_avail(ca_ds27). ds_avail(ca_ds28).
ds_comp(ca_ds26,comp37). ds_comp(ca_ds27,comp38).
ds_comp(ca_ds28,comp39).
source(ca_ds26,ctxt_case_anl). source(ca_ds27,ctxt_case_anl).
```

```
source(ca_ds28,ctxt_case_anl). source(comp37,ctxt_case_anl).
source(comp38,ctxt_case_anl). source(comp39,ctxt_case_anl).
tag(ca_ds26,case(c1,1)). tag(ca_ds27,case(c2,1)).
tag(ca_ds28,case(c1,2)).
```

As a result, we get an answer set with the following directives with the desired packages.

```
package(sch, [am_ds56, am_ds49, am_ds34]),
package(sch, [am_ds74, am_ds53, am_ds45]),
package(sch, [am_ds84, am_ds55, am_ds46, am_ds24]).
```

4 Conclusion

While the method presented in this paper was developed for use in the aMCS framework, it addresses a general problem that will often occur when dealing with stream data: deciding when and what part of buffered incoming data to process or what information to ignore. It was our goal in this paper to show in a range of examples that declarative specifications using ASP can be of great help here. The rich language of ASP allows for concisely expressing how data should be bundled in packages that meet the input requirements for further processing which can be seen as a combined classification and configuration task. Using ASP for configuration purposes has a long tradition [22, 13]. We use ASP for pre-processing and structuring of stream data which can itself be seen as an instance of stream reasoning using ASP [8, 11, 20]. An interesting question is to what extent reasoning should take place in the packing layer or in the contexts. The answer seems to depend on the application: reasoning should take place in the contexts if the context formalism is expressive enough and the reasoning task conceptually belongs to the context; otherwise we can exploit ASP on the packing layer to solve problems that are too complex for the contexts or that do not fit the conceptual scope of any context. Our method does not fully inspect the contents of data sets but relies on clearly structured meta information and tags provided by the system or the source of information. This is a difference to data stream mining in the areas of data mining and machine learning, dealing with approximate classification and clustering of stream data [16, 15].

In ongoing work, we are developing an aMCS engine that implements the features we described. In order to guarantee good interoperability the software will make use of the efficient ASP solver `clingo` as a library.

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York, NY, USA (2003)
2. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: A logic-based framework for analyzing reasoning over streams. In: Proc. 29th AAAI Conference on Artificial Intelligence (AAAI 2015). pp. 1431–1438. AAAI Press (2015)
3. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: asprin: Customizing answer set preferences without a headache. In: Proc. 29th AAAI Conference on Artificial Intelligence (AAAI 2015). pp. 1467–1474. AAAI Press (2015)

4. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI'07. pp. 385–390 (2007)
5. Brewka, G., Eiter, T., Fink, M., Weinzierl, A.: Managed multi-context systems. In: IJCAI'11. pp. 786–791 (2011)
6. Brewka, G., Ellmauthaler, S., Pührer, J.: Multi-context systems for reactive reasoning in dynamic environments. In: Proc. ECAI'14. pp. 159–164 (2014)
7. Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence 20(2), 308–334 (2004)
8. Do, T.M., Loke, S.W., Liu, F.: Answer set programming for stream reasoning. In: Proc. 24th Canadian Conference on Artificial Intelligence (Canadian AI 2011). LNCS, vol. 6657, pp. 104–109. Springer (2011)
9. Ellmauthaler, S., Pührer, J.: Asynchronous multi-context systems. In: Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation. LNCS, vol. 9060, pp. 141–156. Springer (2015)
10. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection 24(2), 107–124 (2011)
11. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In: Proc. 13th Int. Conference on Principles of Knowledge Representation and Reasoning (KR 2012). AAAI Press (2012)
12. Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., Schaub, T.: Progress in clasp series 3. In: Proc. 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015). LNCS, vol. 9345, pp. 368–383. Springer (2015)
13. Gebser, M., Kaminski, R., Schaub, T.: aspcud: A linux package configuration tool based on answer set programming. In: Proc. 2nd Workshop on Logics for Component Configuration (LoCoCo 2011). EPTCS, vol. 65, pp. 12–25 (2011)
14. Gonçalves, R., Knorr, M., Leite, J.: Evolving multi-context systems. In: Proc. ECAI'14. pp. 375–380 (2014)
15. Hahsler, M., Dunham, M.H.: Temporal structure learning for clustering massive data streams in real-time. In: Proc. 11th SIAM Int. Conference on Data Mining (SDM 2011). pp. 664–675. SIAM / Omnipress (2011)
16. Huang, H., Yoo, S., Kasiviswanathan, S.P.: Unsupervised feature selection on data streams. In: Proc. 24th ACM International on Conference on Information and Knowledge Management (CIKM 2015). pp. 1031–1040. ACM (2015)
17. Le-Phuoc, D., Parreira, J.X., Hauswirth, M.: Linked stream data processing. In: Proc. RW'12. pp. 245–289 (2012)
18. Lifschitz, V.: Twelve definitions of a stable model. In: Proc. 24th Int. Conference on Logic Programming (ICLP 2008). LNCS, vol. 5366, pp. 37–51. Springer (2008)
19. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: In The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer (1999)
20. Nickles, M., Mileo, A.: Web stream reasoning using probabilistic answer set programming. In: Proc. 8th Int. Conference on Web Reasoning and Rule Systems (RR 2014). LNCS, vol. 8741, pp. 197–205. Springer (2014)
21. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)
22. Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Representing configuration knowledge with weight constraint rules. In: Proc. 1st Int. Workshop on Answer Set Programming (ASP 2001), Towards Efficient and Scalable Knowledge Representation and Reasoning (2001)