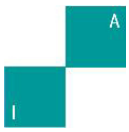
Inteligencia Artificial 48(2010), 3-14
doi: 10.4114/ia.v14i48.1600



INTELIGENCIA ARTIFICIAL

http://erevista.aepia.org/

Modelling autonomic dataspaces using answer sets

Gabriela Montiel-Moreno[1] and José Luis Zechinelli-Martini[1] and Genoveva Vargas-Solar[2]

[1] Universidad de las Américas, Puebla, CENTIA -LAFMIA
Sta. Catarina Mártir s/n, 72820, San Andrés Cholula, México.
gabriela.montielmo@udlap.mx, joseluis.zechinelli@udlap.mx

[2] French National Council of Scientific Research, LIG - LAFMIA
681 rue de la Passerelle, BP 72, Saint Martin d'Hères, France.
Genoveva.Vargas-Solar@imag.fr

Abstract This paper presents an approach for managing an *autonomic dataspace*, able to automatically define views that fulfill the requirements of a set of users, and adjust them as the dataspace evolves. An autonomic dataspace deals with incomplete knowledge to manage itself because of the heterogeneity and the lack of metadata related to the resources it integrates. Our approach exploits the expressiveness of stable models and the K action language for expressing the dataspace management functions It is based on a model for specifying an *autonomic dataspace* expressed using answer set programming (ASP). ASP is a type of declarative logic programming particularly useful in knowledge-intensive applications. It is based on the stable semantics (answer sets), which allows negation as failure and applies the ideas of auto-epistemic logic to distinguish between what is true and what is believed.

Keywords: Data integration, knowledge management, views, answer set planning.

1 Introduction

The massive production and spread of data and applications around the Web introduces new challenges related to their management and exploitation. Users continuously produce and exploit resources implemented in different formats and accessible through different application programming interfaces (API). With the emergence of the semantic Web, certain resources may be tagged[1] with descriptions of their data structures, APIs and content. Users exploit a certain set of these resources and develop new knowledge that is sometimes integrated again in the Web. For example, consider the personal information managed by Alice, consisting of images, documents, web pages, applications, etc. Suppose Alice wants to write the state of art of her dissertation on `Dataspaces management`. Therefore, Alice must explore and analyze every resource stored in the servers she accesses in order to define the bibliography she needs. This task is long and complex because not all resources (html pages, blogs, pdf's, doc's, figures) are tagged, or tags may describe them using different vocabularies than the one used by Alice. Thus, the integration of information is rather manual and demands an important implication of Alice so that she can filter relevant resources and exploit their content.

[1] A tag is meta-data associated to a resource. Meta-data can respect a model, they can belong to a predefined vocabulary or they can be just keywords associated to a resource.

ISSN: 1988-3064(on-line)

The notion of *dataspace* [5, 7] arises as a possible solution for giving homogeneous access to heterogeneous resources by defining a dynamic virtual environment publishing, accessing, and managing a set of heterogeneous and distributed resources (data or services) shared by different communities of users. A dataspace is fed when users publish new resources, and when new information is generated as a result of the exploitation of resources. A dataspace has an associated platform called the DataSpaces Support Platform (DSSP) that provides services for managing heterogeneous resources having different models and querying them in a coordinated and controlled way [5, 7]. A DSSP can deal with incomplete knowledge about resources for answering queries. [10, 11] propose the definition of a global logic view of information contained in the dataspace through the definition of relevant objects and associations between these objects. [11] defines and enriches approximate meta-data and semantic relations between resources as the dataspace evolves, as queries are executed and as users qualify them over time. [10] maintains a set of schemas modelling resources respect to a specific domain and store them in information repositories organized by topic.

However, given the dynamicity and autonomy of the resources of a dataspace, the users have to continually analyze the dataspace in order to verify the availability and the subscription of new resources potentially useful to their activities. To illustrate this situation, consider that Alice has already defined her bibliography after analysing her dataspace. Suppose that her advisor uploads a new document relevant to Alice after this process. In order to incorporate this resource, Alice must analyze after certain period of time if new resources have been published and, if any is relevant to her state of art. This task is hard and difficult to achieve if we consider the continuous evolution of her dataspace. Additionally, if Alice modifies her requirement she must query again the dataspace to retrieve the pertinent information.

These reasons motivate us to build a dataspace that must adapt itself respect to its evolution, by automatically defining and executing strategies that optimize its behaviour. An autonomic dataspace auto-manages itself by defining automatically views, and adjusting them as the dataspace evolves. As in relational databases, a view represents a subset of the resources contained in the dataspace, that is expressed according to the terms shared by a group of users. To achieve this, a dataspace must provide a set of services satisfying the following autonomic properties [8, 9]:

- **Auto-configuration.** The DSSP configures its components automatically according to its evolution. This configuration must respect a set of policies describing the correct behaviour of a dataspace for ensuring its consistency.
- **Auto-optimization.** An autonomic dataspace analyses continuously the behaviour of its components to make decisions about the execution of operations, and to optimize the access and exploitation of its resources.
- **Auto-healing.** The DSSP automatically detects failed resources and implements recovery strategies avoiding the interruption of its services.
- **Auto-protection.** A DSSP detects, identifies and protects itself automatically from attacks. It avoids access to resources prohibited according to access control constraints.

The problem of specifying of an autonomic dataspace is stated as follows. Given a set of resources tagged with technical (e.g., the description of an access protocol) and non-technical data (e.g., the description of a content topic) and given users communities that express information requirements through queries using different vocabularies, the challenge is to model (i) views expressed as semantic relations that can couple users' queries with resources' tags; (ii) events that represent the evolution of the dataspace as new resources are added or deleted, and as new views are defined; and (iii) actions that can update, create, merge, and split views according to the events produced in the dataspace.

We propose an autonomic dataspace as a system that provides transparent access to resources stored in a virtual integrated space by automatically managing views. Our approach consists in formally defining the static components of a dataspace, namely, participants, resources and views, its dynamic aspect through event- reaction expressions and then extending the DSSP management platform. The components of the dataspace are modelled using stable model semantics [6]. The K action language [4] is used for representing the actions and events related to the strategies used for automatically managing the views of the dataspace. The DSSP platform that implements the formal specification and that provides a set of services:

1. A *resource indexation structure* [3, 11, 10, 16] that organizes resources according to the terms used in their annotations and defines mappings between those terms.

2. An *autonomic view manager* that automatically triggers actions to configure views respect to the presence of events over the resources and the participants.

The remainder of the paper is organized as follows. Section 2 and 3 describes our model to characterize a dataspace, views over dataspaces and their operations using Answer Set Prolog [1] and the K action language [13]. Section 4 specifies the auto-configuration functions for a dataspace. Section 5 presents an overview of our current implementation. Finally, Section 6 concludes this paper and describes our future work.

2 Autonomic dataspace

A dataspace is a virtual space that consists of a (i) set of heterogeneous distributed resources, and a (ii) set of participants publishing and consuming resources. An autonomic dataspace automatically adapts itself to the requirements of groups of participants as new resources are added or deleted, and new requirements are specified. The participants of a dataspace generally exploit a sub-group of relevant resources in the dataspace satisfying a requirement. Requirements are expressed using a set of terms belonging to the vocabulary of their community and express the semantic conditions that a resource must verify to be relevant for a given requirement. Given a requirement, the DSSP analyzes each subscribed resource using its associated metadata and generates a view over the dataspace. A view represents the set of relevant resources fulfilling the requirements of a group of participants.

In our approach, an autonomic dataspace is represented as a tuple `dataspace (Participants, Resources, Events)` where `Resources` represents a set of resources of the dataspace, `Participants` characterizes the set of participants publishing or exploiting resources in the dataspace, and `Events` represent a set of events characterizing the evolution of the dataspace's components over time. We model the components of the dataspace (resources and participants) as predicates in Answer Set Prolog [1]. Events are modeled as fluents in the K action language [4] [2].

2.1 Participants

Participants are defined by the predicate `participant(ParticipantName)` for representing individuals providing or consuming resources to generate new information or execute a particular activity. Participants in a dataspace can be organized into communities.
Community represents a group of participants sharing similar interests about certain knowledge domains. Communities are formally defined with the predicate `community(Community)` where `Community` represents the name of the community. The members of a certain community are specified using a set of predicates `belongsTo(ParticipantName, Community)`.

(2.1) ← `community(Community),`
`♯count{Participant : belongsTo(Participant, Community)}< 1.`

(2.2) ← `participant(Participant),`
`♯count{Community : belongsTo(Participant, Community)}< 1.`

A community must have at least one participant (2.1) and every participant must belong to at least one community (2.2). According to the communities he/she belongs, a participant has access to a certain sub-space of resources over which he/she can express requirements.

Every community has at least one associated vocabulary used for specifying queries over the dataspace. This association is formally defined with the predicate `hasVocabulary(Community, Vocabulary)`.
Vocabulary represents a set of terms belonging to a specific knowledge domain, e.g. Computer Science. A vocabulary is formally defined as a set of predicates `vocabulary(Vocabulary, Domain, Term)`, where

[2] Fluents express a property of an object in a world and form part of the Resource of states of the world. Fluents keep their truth values during time unless they are explicitly affected by an action

`Vocabulary` represents the identifier of the specific vocabulary, `Domain` its knowledge domain, and `Term` a specific term belonging to the vocabulary. Communities must have at least one associated vocabulary (2.3).

$$\leftarrow \quad \texttt{community(CommunityName)}, \quad (2.3)$$
$$\sharp\texttt{count}\{\texttt{Vocabulary : hasVocabulary(Community, Vocabulary)}\} < 1.$$

Participants in the dataspace exploit their resources according to a set of requirements expressed using terms of the vocabularies. This association is defined using a fluent `hasRequirement(ParticipantName, RequirementID)` where `ParticipantName` represents the name of the participant defining the requirement `RequirementID`.

Requirement is represented as a set of fluents of type `requirement(Requirement, Term)`, where each predicate states that a `Term` describes a specific requirement `Requirement` used for defining a query on the dataspace.

2.2 Resource

A resource in a dataspace can represent a data source, an application, or a Web service. A resource is defined using the predicate `resource(ResourceName, URI, Type)` that characterizes it with a name `ResourceName`, an universal resource identifier (`URI`) specifying the way the resource can be accessed, and a `Type`, e.g. document, image, or application. According to its type, a resource has associated metadata. For instance, a document can be described with the attributes: title, description, words number, related topics, and format (pdf, word, latex, and plain text) using a predicate `document(Resource, Title, Description, TotalWords, Topic)`. Additionally, a resource have associated data related to its producer, the operations it provides, and its content.

Producer represents participants providing a resource specified using the fluent `hasProducer(Resource, Producer)`, where `Producer` represents the name of the participant providing the `Resource`.

$$\leftarrow \quad \texttt{hasProducer(Resource, Producer)}, \quad (2.4)$$
$$\texttt{not resource(Resource, URI, Type).}$$
$$\texttt{uri(URI), format(Type).}$$

$$\leftarrow \quad \texttt{hasProducer(Resource, Producer)}, \quad (2.5)$$
$$\texttt{not participant(Producer).}$$

The association between a resource and a producer cannot exist if a resource is not defined (2.4). The association between a resource and a producer cannot exist if the producer is not defined as a participant (2.5).

According to its type, a resource may provide a set of operations over its data. An operation is modelled with the predicate `operation(OperationName, Type)`. Every operation in the dataspace must be associated with at least one resource within the dataspace using the predicate `hasOperation(ResourceName, OperationName)`.

Additionally, an operation can be defined by a set of input and output parameters. Input and Output parameters are modelled through predicates of the form `parameter(ParameterName, DataType)`, where a `ParameterName` denotes the name of the parameter and `DataType` represents the abstraction of basic data types, i.e. boolean, real, integer, string, or double. An operation is associated with its input and output parameters using a set of predicates: `hasInput/Output(Operation, Parameter, Order)`, where `Order` specifies the position of the parameter within the operation.

Annotations represent the description of a resource using a set of terms from a specific vocabulary domain, i.e. Computer Science. An annotation is expressed using a predicate `annotation(Annotation, Term)` stating that a term `Term` from a specific vocabulary describes a specific `Annotation`.

An annotation cannot be defined by a specific positive or negative literal (2.6) and it cannot be expressed using two terms `TermA` and `TermB` that contradict themselves, i.e. good and bad (2.7).

(2.6) ← `annotation(AnnotationID, Term),`
`not annotation(AnnotationID, Term), term(Term).`

(2.7) ← `annotation(AnnotationID, TermA),`
`annotation(AnnotationID, TermB),`
`contradicts(TermA, TermB).`

A resource can have several annotations defined under different vocabularies and they can be inconsistent between each other. The association of an annotation with a specific resource is represented using the predicate `hasAnnotation (Resource, Author, Annotation)`.

(2.8) ← `hasAnnotation(Resource, Author, Annotation),`
`not resource(ResourceName, URI, Type).`
`uri(URI), format(Type).`

(2.9) ← `hasAnnotation(Resource, Author, Annotation),`
`not annotation(AnnotationID, Term), term(Term).`

An annotation can be only associated with resources that have been previously defined in the dataspace (2.8). Resources can be associated with an annotation if and only if it has been previously defined and it is composed by at least one term (2.9).

2.3 Events

Events in the dataspace represent changes over the dataspace at a given instant. Events are modelled as fluents using the K action language [4]. An event is defined as `eventName(Timestamp, Producer, Delta)`, where `Timestamp` represents the time in which the event was produced [15]. Additionally, an event is characterized by specifying its producer and information describing the conditions in which the event was produced, expressed by a set `Delta`, e.g. the terms added and deleted from an annotation when it has been updated. We classify the events of the dataspace according to their producer: *resource events* defined as fluents `eventName(Timestamp, ResourceName, Delta)`, where `ResourceName` represents the resource name that produced the event; and *participant events* represented as fluents `eventName(Timestamp, ParticipantName, Delta)`, where `ParticipantName` represents the participant name producing the event).

3 View

Resources in a dataspace can be organized into sub-spaces named *views*. A *view* represents a set of resources satisfying a participant's requirement. A view has two main parts: a semantic and an extension. The semantic represents the content of the view expressed through a set of terms possibly belonging to multiple vocabularies. Every term must be mapped (equivalence, generalization, etc.) to at least one term in the requirement of the view. The extension represents the set of resources relevant to the view.

A resource is relevant to the view if it is indexed by a subset of terms (or related terms) from the requirements associated with the view. A specific resource `Resource` using the term `Term` can be part of the extension of the view if and only if the resource has an annotation described by this term, and the term is related to the view's requirements. Since a view changes automatically in response to the events of the dataspace, it is modelled using the fluent `view(ViewID, Term, Resource)` that states that a view with an identifier `ViewID` uses a resource `Resource` under the semantic defined by `Term`. A view must be associated with at least one requirement defined by a participant in certain period of time through the fluent `respondsTo(View, Requirement)`. The requirement and the view must be previously defined. The association between a participant and a specific view is modelled with the fluent `hasView (Participant, Requirement, View)`. This association can be defined if and only if the participant is connected in the dataspace and the participant has been previously associated with the view's requirement.

Inspired in relational algebra and set theory, we propose a family of operations over views: projection, union, intersection, and difference. We also defined operations for managing views: insert and remove

a view, and associate a vocabulary to a view. Because operations produce effects over the current state of views, we model them as actions in the K action language [4]. The following sections define these operations.

3.1 View Projection - filter(Vocabulary, ViewA, ViewB)

This operation produces a new view composed of all the elements from a view (terms and associated resources) that are related to at least one term of the vocabulary.

Requirements: (i) The view `ViewA` must have been previously defined, (ii) the vocabulary must have been defined and composed by at least one term, and the view `ViewB` must not have been defined.

```
filter(Vocabulary, ViewA, ViewB)
requires   view(ViewA, TermA, ResourceA),              (i)
           vocabulary(Vocabulary, Domain, ViewTerm),   (ii)
           not view(ViewB, TermB, ResourceB).          (iii)
```

Execution conditions: There exists a `VocTerm` in the vocabulary that is related to a `TermA` in `viewA`'s requirements.

```
executable  filter(Vocabulary, ViewA, ViewB)
if          respondsTo(ViewA, ReqA),
            requirement(ReqA, TermA),
            vocabulary(Vocabulary, Domain, VocTerm),
            mapping(VocTerm, TermA).
```

Effects: A `ViewB` is created by defining a set of fluents of the form `view(ViewB, TermA, ResourceA)`. This fluent states that the `ResourceA` described with `TermA` is related to a `VocTerm` of the vocabulary. `ViewB`'s requirement is composed by all `VocTerm` of the vocabulary related to at least one term of `viewA`'s requirement.

```
caused  view(ViewB, TermA, ResourceA)
if      view(ViewA, TermA, ResourceA),
        vocabulary(Vocabulary, VocTerm),
        mapping(VocTerm, TermA)
after   filter(Vocabulary, ViewA, ViewB).
```

3.2 Union - union(ViewA, ViewB, ViewC)

Produces a new `ViewC` including all the elements from `ViewA` and `ViewB`.

Requirements: (i) The views `ViewA` and `ViewB` must have been previously defined, and (iii) the view `ViewC` must not have been defined.

```
union(ViewA, ViewB, ViewC)
requires   view(ViewA, TermA, ResourceA),       (i)
           view(ViewB, TermB, ResourceB),
           not view(ViewC, TermC, ResourceC).   (ii)
```

Execution conditions: This operation has no execution conditions.

Effects: A `ViewC` is created by defining a set of fluents of the form `view(ViewC, Term, Resource)` where `Resource` is an element of `ViewA` or `ViewB` described with `Term`. `ViewC`'s requirement is composed by all `TermA` and `TermB` from the requirements of `viewA` and `viewB` respectively.

```
caused  view(ViewC, Term, Resource)
if      view(ViewA, Term, Resource)
after   union(ViewA, ViewB, ViewC).

caused  view(ViewC, Term, Resource)
if      view(ViewB, Term, Resource)
after   union(ViewA, ViewB, ViewC).
```

3.3 Intersection - intersection(ViewA, ViewB, ViewC)

Produces a new `ViewC` including all elements from `ViewA` that are related to at least one element of the `ViewB` and vice versa.

Requirements: (i) The views `ViewA` and `ViewB` must have been previously defined, and (iii) the view `ViewC` must not have been defined.

```
intersection(ViewA, ViewB, ViewC)
requires  view(ViewA, TermA, ResourceA),        (i)
          view(ViewB, TermB, ResourceB),
          not view(ViewC, TermC, ResourceC).    (ii)
```

Execution conditions: There exists a `ReqATerm` in `viewA`'s requirements that is related to a `ReqBTerm` in `viewB`'s requirements.

```
executable  intersection(ViewA, ViewB, ViewC)
if          respondsTo(ViewA, ReqA),
            requirement(ReqA, ReqATerm),
            respondsTo(ViewB, ReqB),
            requirement(ReqB, ReqBTerm),
            mapping(ReqATerm, ReqBTerm).
```

Effects: A `ViewC` is created by defining a set of fluents of the form `view(ViewC, Term, Resource)` where `Resource` is an element of `ViewA` and `ViewB` described with `Term`. `ViewC`'s requirement is composed by all `TermA` from the requirements of `ViewA` related to at least one term in `ViewB`'s requirements and vice versa.

```
caused  view(ViewC, TermA, ResourceA),
        view(ViewC, TermB, ResourceB)
if      view(ViewA, TermA, ResourceA),
        view(ViewB, TermB, ResourceB),
        mapping(TermA, TermB)
after   intersection(ViewA, ViewB, ViewC).
```

3.4 Difference - difference(ViewA, ViewB, ViewC)

Produces a new `ViewC` including all the elements from `ViewA` that are not related to any element in the `ViewB`.

Requirements: (i) The views `ViewA` and `ViewB` must have been previously defined, and (iii) the view `ViewC` must not have been defined.

```
difference(ViewA, ViewB, ViewC)
requires    view(ViewA, TermA, ResourceA),       (i)
            view(ViewB, TermB, ResourceB),
            not view(ViewC, TermC, ResourceC).   (ii)
```

Execution conditions: There exists a `ReqATerm` in viewA's requirements that is not related to any term in `viewB`'s requirements.

```
executable    difference(ViewA, ViewB, ViewC)
if            respondsTo(ViewA, ReqA),
              requirement(ReqA, ReqATerm),
              not inR(ReqATerm, ViewB),
              not view(ViewC, TermC, ResourceC).
```

Effects: A `ViewC` is created by defining a set of fluents of the form `view(ViewC, Term, Resource)` where `Resource` is an element of `ViewA` and not of `ViewB` and is described with `Term`. ViewC's requirement is composed by all `TermA` from the requirements of `ViewA` that are not related to any term in `ViewB`'s requirements.

```
caused    view(ViewC, TermA, ResourceA)
if        view(ViewA, TermA, ResourceA),
          not in(TermA, ViewB)
after     difference(ViewA, ViewB, ViewC).
```

3.5 Resource insertion/removal - insertR/deleteR(Resource, View)

These operations update a `View` by adding (or removing) a `Resource` and its associations with terms related to the requirements of the view.

Requirements: (i) The view must have been previously defined, and (ii) the resource to be inserted (or removed) must be defined in the dataspace.

```
insertR/deleteR(Resource, View)
requires  view(View, ViewTerm, ViewResource),   (i)
          resource(Resource, URI, Type).        (ii)
```

Execution conditions: The resource to be inserted (or removed) has been previously defined and indexed using at least one term related to the view's requirements.

```
executable  insertR/deleteR(Resource, View)
if          respondsTo(View, ReqID),
            requirement(ReqID, ReqTerm),
            isIndexed(Resource, IndexTerm),
            mapping(IndexTerm, ReqTerm).
```

Effects: A `View` is updated by adding (or negating) a set of fluents `view(View, IndexTerm, Resource)`. This fluent states that the `Resource` indexed with `IndexTerm` forms part of the view and satisfies the view's requirements expressed with `Resource`.

```
caused   view(View, IndexTerm, Resource)/
         -view(View, IndexTerm, Resource)
if       isIndexed(Resource, IndexTerm),
         responds(View, ReqID),
         requirement(ReqID, ReqTerm),
         mapping(IndexTerm, ReqTerm)
after    insertR/deleteR(Resource, View).
```

3.6 Vocabulary insertion/removal - insertV/deleteV(Vocabulary, View)

This operation updates a `View` by adding (or removing) the terms of a `Vocabulary` and their indexed resources as elements of the view.

Requirements: (i) The view must have been previously defined, and (ii) the vocabulary to be inserted must have been defined and composed by at least one term.

```
insertV/deleteV(Vocabulary, View)
requires   view(View, Term, Resource),                (i)
           vocabulary(Vocabulary, Domain, VocTerm).   (ii)
```

Execution conditions: There exists at least one term in the vocabulary that is related to one from the view's requirements.

```
executable   insertV/deleteV(Vocabulary, View)
if           responds(View, ReqID),
             requirement(ReqID, ReqTerm),
             vocabulary(Vocabulary, Domain, VocTerm),
             mapping(VocTerm, ReqTerm).
```

Effects: The `View` is updated by adding (or negating) a set of fluents `view(View, VocTerm, Resource)`. This fluent states that the `Resource` indexed with `VocTerm` belonging to the `Vocabulary` to be inserted forms part of the `View` and satisfies the view's requirements expressed with `Resource`.

```
caused   view(View, VocTerm, Resource)/
         -view(View, VocTerm, Resource)
if       responds(View, ReqID),
         requirement(ReqID, ReqTerm),
         vocabulary(Vocabulary, Domain, VocTerm),
         mapping(VocTerm, ReqTerm),
         resource(Resource, URI, Type),
         isIndexed(Resource, VocTerm)
after    insertV/deleteV(Vocabulary, View).
```

4 Auto-configuration of views

The auto-configuration of views involves the creation and execution of view management plans over views when events are notified. Thereby it is possible to automatically define new views when a participant specifies new requirements, update views when resources are subscribed or removed, and delete views when they are no longer required. The auto-configuration of views is achieved through the definition of a set of management strategies. Our auto-configuration strategies are organized in three categories:

Auto-definition This category refers to all the strategies related to the definition of views within the dataspace when an event `RequirementAdded` is detected. In this case, according to the participant producing this event and the requirement that has been inserted a new view is defined. The new view is computed by using previously defined views by projecting or integrating existing views; or by directly defining one by searching resources in the dataspace that fulfill the new requirement.

Auto-update The main objective of this category of strategies is to automatically update the set of views when events are produced by the resources and participants. The events can represent resource subscription, annotation insertion, update or removal, resource removal and update of the requirements from a participant. For doing so, it is necessary to identify all the views $v_1, \ldots, v_n$ that are partially or totally described by the terms contained in the annotation of the resource producing the event or the modified requirements.

Auto-removal This category refers to the strategies related to the elimination of views within the dataspace when an event `RequirementDeleted` is detected. In this case, according to the participant $\mathtt{p}_1$ producing this event and the requirement that has been removed the corresponding view is deleted from the dataspace. If another participant $\mathtt{p}_2$ is associated to the view, only the association between the participant $\mathtt{p}_1$ and the view is deleted.

5 Implementation issues

In order to validate our approach we implemented an autonomic view manager oriented to personal dataspaces. A personal dataspace integrates heterogeneous and distributed resources produced by an individual to be exploited by her/him and her co-workers [14]. An autonomic view manager executes the auto-configuration task and is composed of three main components: a monitor, an evaluator, and an executer [8, 9, 12]. The monitor observes continuously the evolution of the dataspace looking up the presence of events over the resources or the requirements specified by the participants. When a new event is detected, it is notified to the evaluator. The evaluator identifies the views that are potentially affected by the event and the strategy that is associated with the event. With these data, the evaluator determines the actions that have to be executed over the views and generates a view management plan. This plan contains all the information about the actions that have to be executed and their order. Finally, this plan is sent to the executer, who is in charge to execute the operations over the views and validate the consistency of the new state of the dataspace.

We implemented a background knowledge base containing the components and structure of Alice's personal dataspace using the datalog programming system DLV [2]. Events, views, operations and strategies were implemented as a planning program in DLV-K planning system [13]. The strategies defined over the auto-configuration of views were modelled as a set of Event-Condition-Action rules in the planning program. By expressing the strategies as Event-Condition-Action rules, we could exploit the capabilities of planning programming to determine the sequence of actions and the state of the dataspace under the presence of events in a period of time.

We have currently validated the correct execution of strategies related to the auto-update and auto-removal in the autonomic and requirement-based personal dataspace, as well as the operations over views. We are currently implementing the auto-definition strategies in DLV-K and defining an approximation algorithm to optimize the definition of views based on predefined views under the JAVA platform version 1.5.0.

6 Conclusions

This paper presented an autonomic dataspace specified using answer sets programming. An autonomic dataspace is a system that auto-manages itself according to the evolution of its resources and participants. Thanks to an autonomic dataspace, users can integrate heterogeneous resources (data and applications) and exploit them by defining views representing sub-spaces of resources that fulfill users' requirements. Our approach exploits the expressiveness of stable models semantics [6] to model incomplete knowledge within its components and the K action language [4] to represent the actions and events related to the auto-management strategies in the dataspace. The action logic-based language K is used for modelling operations over views as actions and representing the view management strategies as sequences of actions triggered given a set of events [1, 4]. Our work defines the control strategies for triggering view management strategies whenever the dataspace evolves. Existing view management techniques are encapsulated in the auto-management actions and they are specified using the operations on views that we specified.

ASP enables the definition of mappings between the terms used for tagging resources and for expressing requirements. As in classic query rewriting and query unfolding approaches of distributed databases, a view is an expression that uses mappings for denoting the set of resources that will fulfill a specific requirement. Different from those approaches, in autonomic dataspaces such views are dynamic. The K language provides the concept of fluent that enables the definition of a view and its dynamic aspect.

Currently, we have implemented a first version of the management platform in the area of personal management. This version was implemented using DLV-K [13], an extension of DLV datalog programming system for planning based on answer sets. Future work relies on the definition of strategies for the auto-optimization, auto-protection and auto-repair in the dataspace. Also, it is necessary to develop strategies to fulfill the autonomic properties to the index structures so they are automatically configured respect to the evolution of the environment. Once we complete these implementations, we will validate the performance of our system with a highly dynamic environment having multiple predefined views. Through these experiments we will be able to determine the efficiency of our solution and the cost related to achieving autonomy in such a variable and increasing environment.

References

[1] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*, chapter Reasoning about actions and planning in AnsProlog, pages 1–30. Cambridge University Press, New York, NY, USA, 2003.

[2] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. *DLV User Manual*. The DLV Project, April 2009.

[3] Xin Dong and Alon Halevy. Indexing dataspaces. In *2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, pages 43–54, New York, NY, USA, 2007. ACM Press.

[4] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. In *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 807–821, New York, NY, USA, 2000. Springer Verlag.

[5] Michael J. Franklin, Alon Y. Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *ACM SIGMOD Records*, 34(4):27–33, 2005.

[6] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming (LPAR 1988)*, pages 1070–1080, 1988.

[7] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '06)*, pages 1–9, New York, NY, USA, 2006. ACM Press.

[8] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporation, Riverton, NJ, USA, 2001.

[9] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):1–28, 2008.

[10] X. Dong A. Y. Halevy S. R. Jeffery D. Ko J. Madhavan, S. Cohen and C. Yu. Web-scale data integration: You can afford to pay as you go. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, pages 342–350, 2007.

[11] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, pages 847–860, New York, NY, USA, 2008. ACM Press.

[12] Mohammad Reza Nami and Mohsen Sharifi. A survey of autonomic computing systems. In *Proceedings of the Intelligent Information Processing*, volume 228, pages 101–110, New York, NY, USA, 2007. Springer Verlag.

[13] Axel Polleres. The dlvk system for planning with incomplete knowledge. Master's thesis, Institut fÃ¼r Informationssysteme, Technische UniversitÃ¤t Wien, Vienna, Austria, February 2001.

[14] Marcos A. Vaz Salles, Jens P. Dittrich, Shant K. Karakashian, Olivier R. Girard, and Lukas Blunschi. itrails: Pay-as-you-go information integration in dataspaces. In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*, pages 663–674. VLDB Endowment, 2007.

[15] Genoveva Vargas-Solar and Christine Collet. Adees: An adaptable and extensible event based infrastructure. In *13th International Conference on Database and Expert Systems Applications (DEXA 2002)*, volume 2453 of *Lecture Notes in Computer Science*, pages 423–432, New York, NY, USA, 2002. Springer.

[16] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(2):1008–1019, 2008.