Modelling autonomic dataspaces using answer sets

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

[1] Research Center of Information and Automation Technologies
Universidad de las Américas, Puebla,
Sta. Catarina Mártir s/n, 72820, San Andrés Cholula, México
[2] French National Council of Scientific Research
Laboratory of Informatics of Grenoble
681 rue de la Passerelle, BP 72, Saint Martin d'Hères, France

Abstract. This paper presents an approach for managing an autonomic dataspace, able to automatically define views adapted for fulfilling 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 This paper proposes a model for specifying an *autonomic dataspace* using answer set programming. Answer set programming (ASP) is a type of declarative logic programming particularly useful in knowledge-intensive applications [1, 7, 8]. 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.

1 Introduction

The increase in the 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 in different formats defined according to the models used by their producers and their applications, and may provide different computing and search capabilities. Moreover, certain resources may be characterized with description of their structure and content.

Users around the Web exploit a certain set of resources and develop new knowledge. 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 analyse every resource stored in the servers she accesses in order to define the bibliography she needs. This task is long and complex because resources may not provide metadata, or may be described under a different vocabulary than the one used by Alice.

The notion of *dataspace* arises as a possible solution to this requirement 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 [6, 9]. A dataspace is fed when users publish new resources, and when new information is generated as a result of the exploitation of resources. Dataspaces have an associated platform (DSSP) that provides services for managing heterogeneous resources having different models and querying than in a coordinated and controlled way [6, 9].

However, the dynamicity and autonomy of the components of a dataspace causes the participants to continually analyse the dataspace in order to verify the availability and the subscription of new resources potentially useful to their activities. To illustrate this situation, reconsider the 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 analyse 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. To achieve this, a dataspace must provide a set of services satisfying the following autonomic properties [10, 11]:

- **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.

1.1 Related works

A DSSP can deal with incomplete knowledge about resources for answering queries. [14, 13] 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. [13] 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. [14] maintains a set of schemas

modelling resources respect to a specific domain and store them in information repositories organized by topic.

Our approach towards an autonomic dataspace is done by extending the DSSP management platform with a set of services:

1. A *resource indexation structure* [3, 13, 14, 20] 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.

1.2 Contribution and organization of the paper

We specify the main components of this dataspace and describe our approach towards the auto-configuration of views exploiting the expressiveness of stable model semantics [8]. Stable model semantics is used for modelling the knowledge of the dataspace and the K action language [5] is used for representing the actions and events related to the auto-management strategies in the dataspace.

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 language Answer Set Prolog [1] and the K action language [17]. Section 4 specifies the auto-configuration functions for a dataspace. Section 5 presents our current implementation. Finally, Section 6 concludes this paper and describes our future work.

2 Modelling an autonomic dataspace

In our approach, an autonomic dataspace is represented as a tuple `dataspace (Participants, Resources, Events)` where Resources represents a set of resources subscribed into 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 dataspaces components along time.

We model the components of the dataspace (resources, participants) as predicates in Answer Set Prolog language [1]. Events are modeled as fluents in the K action language [5]. 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.

2.1 Participants

Participants defined by the predicate `participant(ParticipantName)` represent individuals providing or consuming a set of 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 execute queries or operations.

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. They are formally defined as a set of predicates `vocabulary(Vocabulary, Domain, Term)`, where `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).

(2.3) ← `community(CommunityName)`,
`♯count{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 formally defined using a fluent `hasRequirement(ParticipantName, Requirement)` where `ParticipantName` represents the name of the participant defining the requirement `RequirementID`.

Requirement is represented as a set of fluents `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

Resources in a dataspace can represent a data source, an application or a Web service subscribed into the dataspace. Resources are defined using the predicate `resource(ResourceName, URI, Type)` which is characterized 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 meta-data describing its structure. For instance, a document resource can be described with the attributes: title, description, words number, related topics and format (pdf, word, latex, plain text) using a predicate `document(Resource, Title, Description, TotalWords, Topic).` Additionally, a resource may provide data related to its producer, the operations it provides, and its content.

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

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

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

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).

Operation 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 to 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` expresses 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 to 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. Annotations are expressed as a set of predicates `annotation(Annotation, Term)` stating that a term `Term` from a specific vocabulary describes a specific `Annotation`.

An annotation can not 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 (2.5).

(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 to resources that have been previously defined in the dataspace (2.8). Resources can be associated to a certain 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 in K action language [5] of the form `evenName(Timestamp, Producer, Delta)`, where `Timestamp` represents the time in which the event was produced [19]. Additionally, an events is characterized specifying its producer and additional information described 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 with respect to the entity over which they are produced: *resource* and *participant*.

2.4 Resources index

The resource index of an autonomic dataspace is composed of three layers: (i) physical that organizes resources into physical storage, (ii) logical that classifies the resources according to the terms used in their annotations, and (iii) external composed by a set of terms and relations between terms. The index is out of the scope of this paper. Interested readers can refer to [12].

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 [16]. Views

allow to have different perspectives from resources in the dataspace. As a view changes automatically in response to the events of the dataspace,it is modelled as fluents characterized with an integer identifier `viewID`. The fluent `view(ViewID, Term, Resource)` states that a view with an identifier `ViewID` uses a resource `Resource` under the semantic defined by `Term`.

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 belonging to multiple vocabularies. Every term in the semantic of the view 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 problem dealt within 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 to the view. A specific resource `Resource` using the term `Term` can be a component 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.

Because of the heterogeneity of resources in a dataspace, it can be possible that a resource provides an incorrect annotation of its content, or it does not provide any annotation at all. Views may not be complete with respect to the requirements of a participant, because resources can have imprecise or incorrect associated annotations.

A view must be associated to 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 to the view's requirement.

3.1 Operation on views

Inspired in relational algebra and set theory, we propose a family of operations over views. Because operations produce effects over the current state of views, we model them as actions in the K action language [5].

3.2 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 `ReqTerm`.

```
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).
```

3.3 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, IndexTerm, 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 `ReqTerm`.

```
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).
```

3.4 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 views A'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. View B's requirement is composed by all `VocTerm` of the vocabulary related to at least one term of view A's requirement.

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

3.5 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`. View C's requirement is composed by all `TermA` and `TermB` from the requirements of views A and B 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).
```

3.6 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 view A's requirements that is related to a `ReqBTerm` in views B'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`. View C's requirement is composed by all `TermA` from the requirements of view A related to at least one term in view B'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).
```

3.7 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 view A's requirements that is not related to any term in view B'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`. View C's requirement is composed by all `TermA` from the requirements of view A that are not related to any term in view B's requirements.

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

4 Auto-configuration of views

The auto-configuration of views involves the creation and execution of view management plans over views respect to the presence of events. Through this autonomic configuration 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.

An autonomic view manager executes the auto-configuration task and is composed of three main components: a monitor, an evaluator, and an executer [10, 11, 16]. 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 to 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 coherence of the new state of the dataspace.

The auto-configuration of views is achieved through the definition of a set of management strategies respect to the presence of events over resources and views. Our auto-configuration strategies are organized in three categories:

4.1 Auto-definition

This category refers to all the strategies related to the definition of views within the dataspace when an event of `RequirementAdded` is detected. In this case, the autonomic view manager must detect the participant producing this event and the requirement that has been inserted. The management platform must define a new view based on previous defined views or by executing a search using the index.

4.2 Auto-update

The main objective of this category of strategies is to automatically update the set of views respect to the presence of events produced over the resources and the participant like: resource subscription, annotation insertion, update or removal, resource removal and update of the requirements from a participant.

For doing so, it is necessary that the management platform identifies all the views $v_1, \ldots, v_n$ that are partially or totally described by the terms contained in the annotation of the resource involved in the event or the modified requirements.

4.3 Auto-removal

This category refers to the strategies related to the elimination of views within the dataspace when an event of `RequirementDeleted` is detected. In this case, the management platform must detect the participant producing this event and the requirement that has been removed. Using the requirement identifier, the management platform must identify the view associated to the requirement and delete the view from the dataspace. In case another participant is consuming the view, the management platform should delete only the association between the participant and the view.

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 [18].

We implemented a background knowledge base containing the components and structure of Alices personal dataspace using the datalog programming system DLV [4]. Events, views, operations and strategies were implemented as a planning program in DLV-K planning system [17].

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 our characterization of an autonomic and requirement-based dataspace using answer sets programming. An autonomic and requirement-based 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 through the definition of views representing sub-spaces of resources adapted respect to their requirements [16].

Our approach exploits the expressiveness of stable models semantics [8] to model incomplete knowledge within its components and the K action language [5] 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, 5].

Currently, we have implemented a first version of the management platform in the area of personal management. This version was implemented using DLV-K [17], 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 fulfil 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. Gennaro Bruno. *"ADEMS" a knowledge-based service for intelligent mediator configuration*. PhD thesis, Institut National Polytechnique de Grenoble, 2006.
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. Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. *DLV User Manual*. The DLV Project, April 2009.
5. 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.

6. 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.
7. Michael Gelfond. *Handbook of Knowledge Representation*, chapter Answer Sets.
8. 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.
9. 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.
10. Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporation, Riverton, NJ, USA, 2001.
11. 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.
12. Carlos-Manuel López-Enríquez, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, and Ofelia Cervantes. Indexing dataspaces: dealing with content and storage space. *RCS*, September 2009.
13. 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.
14. Jayant Madhavan, Shirley Cohen, Xin Luna Dong, Alon Y. Halevy, Shawn R. Jeffery, David Ko, and Cong Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, 2007.
15. Gabriela Montiel-Moreno, José-Luis Zechinelli-Martini, and Genoveva Vargas-Solar. Sisels: a mediation system for giving access to biology resources. *Journal Research in Computing Science Special Issue: Electronics and Biomedical Engineering, Computer Science and Informatics*, 35, 2008.
16. 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.
17. Axel Polleres. The dlvk system for planning with incomplete knowledge. Master's thesis, Institut fr Informationssysteme, Technische Universitt Wien, Vienna, Austria, February 2001.
18. 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.
19. 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.
20. 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.